دانشگاه آزاد اسلامی

واحد تهران جنوب

گروه نرم‌افزار

برنامه سازی پیشرفته

(Java)

به نام خدا

Contents

Chapter 1

The Java Language Specification, API, JDK, and IDE

Java syntax is defined in the Java language specification, and the Java library is defined in the Java API. The JDK is the software for developing and running Java programs. An IDE is an integrated development environment for rapidly developing programs.

Computer languages have strict rules of usage. If you do not follow the rules when writing a program, the computer will not be able to understand it. The Java language specification and the Java API define the Java standards. The *Java language specification* is a technical definition of the Java programming language's syntax and semantics. You can find the complete Java language specification at java.sun.com/docs/books/jls.

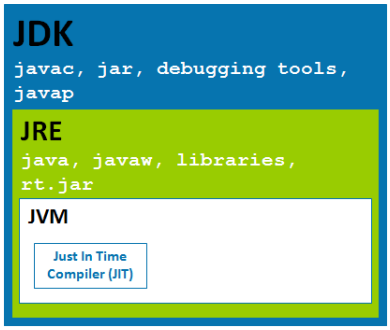
The *application program interface (API)*, also known as *library*, contains predefined classes and interfaces for developing Java programs. The API is still expanding. You can view and download the latest version of the Java API at:
www.oracle.com/technetwork/java/index.html.
Java is a full-fledged and powerful language that can be used in many ways. It comes in three editions:
- **Java *Standard Edition (Java SE)* to develop client-side standalone applications or applets.**
- ***Java Enterprise Edition (Java EE)* to develop server-side applications, such as Java servlets, JavaServer Pages (JSP), and JavaServer Faces (JSF). *Java EE* is geared toward developing large-scale, distributed networking applications and web-based applications.**
- ***Java Micro Edition (Java ME)* to develop applications for mobile devices, such as cell phones. *Java ME* is geared toward developing applications for small, memory-constrained devices, such as BlackBerry smartphones. Google's Android operating system—used on numerous smartphones, tablets (small, lightweight mobile computers with touch screens), e-readers and other devices—uses a customized version of Java not based on Java ME.**

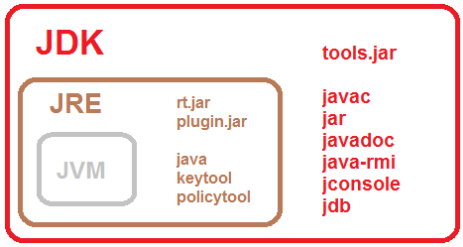
Java SE is the foundation upon which all other Java technology is based. There are many versions of Java SE. Oracle releases each version with a *Java Development Toolkit (JDK).* For Java SE 8, the Java Development Toolkit is called *JDK 1.8* (also known as *Java 8* or *JDK 8*).
The JDK consists of a set of separate programs, each invoked from a command line, for developing and testing Java programs. Instead of using the JDK, you can use a Java development tool (e.g., NetBeans, Eclipse, and TextPad) software that provides an *integrated development environment (IDE)* for developing Java programs quickly. Editing, compiling, building, debugging, and online help are integrated in one graphical user interface. You simply enter source code in one window or open an existing file in a window, and then click a button or menu item or press a function key to compile and run the program.

JRE (Java Runtime Environment)

Java Runtime Environment contains JVM, class libraries, and other supporting files. It does not contain any development tools such as compiler, debugger, etc. Actually JVM runs the program, and it uses the class libraries, and other supporting files provided in JRE. If you want to run any java program, you need to have JRE installed in the system. The Java Virtual Machine provides a platform-independent way of executing code; programmers can concentrate on writing software, without having to be concerned with how or where it will run.

- **/bin with executable programs like java and (for Windows) javaw, which are essentially the program that is the Java virtual machine;**
- **/lib with a large number of supporting files: Some jars, configuration files, property files, fonts, sounds, icons... all the "trimmings" of Java. Most important are rt.jar and a possibly a few of its siblings, which contain the "java API," i.e. the Java library code.**
- **Somewhere, possibly squirreled away by the installer to some directory specified by the operating system, are some .DLLs (for Windows) or .so's (Unix/Linux) with supporting, often system-specific native binary code.**

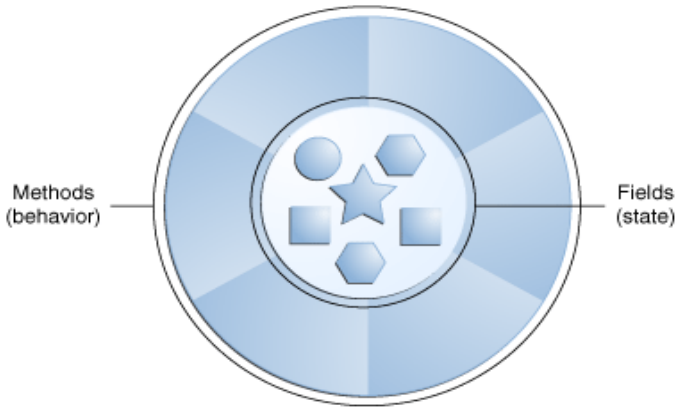




Java Class Libraries

You can create each class and method you need to form your Java programs. However, most Java programmers take advantage of the rich collections of existing classes and methods in the Java class libraries, which are also known as the Java APIs (Application Programming Interfaces).

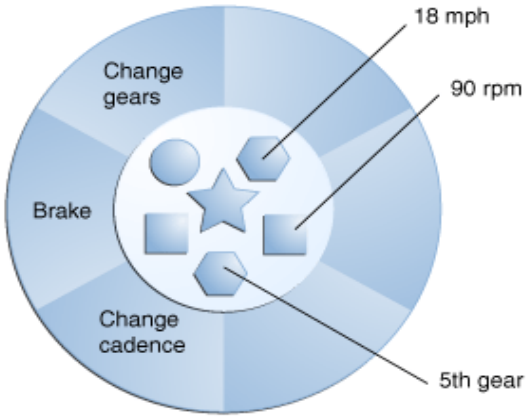
What Is an Object?



A software object.

Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in fields (variables in some programming languages) and exposes its behavior through methods (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication. Hiding internal state and requiring all interaction to be performed through an object's methods is known as data encapsulation — a fundamental principle of object-oriented programming.

Consider a bicycle, for example:



By attributing state (current speed, current pedal cadence, and current gear) and providing methods for changing that state, the object remains in control of how the outside world is allowed to use it. For example, if the bicycle only has 6 gears, a method to change gears could reject any value that is less than 1 or greater than 6.

Bundling code into individual software objects provides a number of benefits, including:

1. **Modularity: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.**
2. **Information-hiding: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.**
3. **Code re-use: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.**
4. **Pluggability and debugging ease: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace it, not the entire machine.**

What Is a Class, Inheritance and an Interface?

A class is the blueprint from which individual objects are created.

Object-oriented programming allows classes to inherit commonly used state and behavior from other classes. In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of subclasses. The syntax for creating a subclass is simple. At the beginning of your class declaration, use the `extends` keyword, followed by the name of the class to inherit from.

Objects define their interaction with the outside world through the methods that they expose. Methods form the object's interface with the outside world; the buttons on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing.

In its most common form, an interface is a group of related methods with empty bodies. To implement this interface, the name of your class would change (to a particular brand of bicycle, for example, such as `ACMEBicycle`), and you'd use the `implements` keyword in the class declaration:

```
class ACMEBicycle implements Bicycle {
```

Implementing an interface allows a class to become more formal about the behavior it promises to provide. Interfaces form a contract between the class and the outside world, and this contract is enforced at build time by the compiler. If your class claims to implement an interface, all methods defined by that interface must appear in its source code before the class will successfully compile.

What Is a Package?

A package is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another. Because software written in the Java programming language can be composed of hundreds or thousands of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.

The Java platform provides an enormous class library (a set of packages) suitable for use in your own applications. This library is known as the "Application Programming Interface", or "API" for short. Its packages represent the tasks most commonly associated with general-purpose programming. For example, a `String` object contains state and behavior for

character strings; a File object allows a programmer to easily create, delete, inspect, compare, or modify a file on the filesystem; a Socket object allows for the creation and use of network sockets; various GUI objects control buttons and checkboxes and anything else related to graphical user interfaces. There are literally thousands of classes to choose from. This allows you, the programmer, to focus on the design of your particular application, rather than the infrastructure required to make it work.

A Simple Java Program

***A Java program is executed from the* main *method in the class.* Let's begin with a simple Java program that displays the message Welcome to Java! on the console. (The word *console* is an old computer term that refers to the text entry and display device of a computer. *Console input* means to receive input from the keyboard, and *console output* means to display output on the monitor.)**

Welcome.java

```
1 public class Welcome {
2 public static void main(String[] args) {
3 // Display message Welcome to Java! on the console
4     System.out.println("Welcome to Java!");
5   }
6 }
```

output:

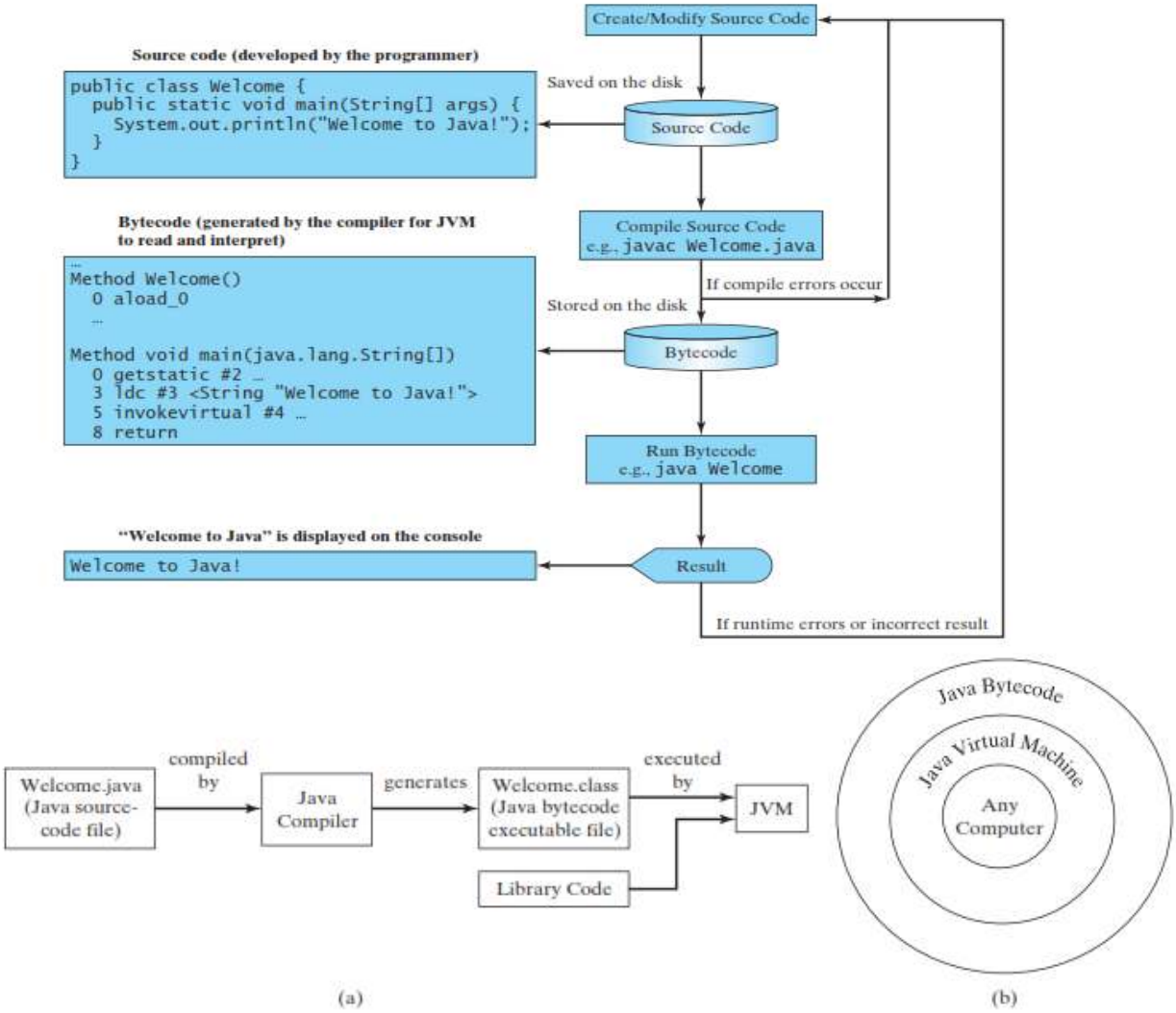
```
Welcome to Java!
```

Note that the line numbers are for reference purposes only; they are not part of the program. So, don't type line numbers in your program.

- **Line 1 defines a class. Every Java program must have at least one class. Each class has a name. By convention, class names start with an uppercase letter. In this example, the class name is Welcome.**
- **Line 2 defines the main method. The program is executed from the main method. A class may contain several methods. The main method is the entry point where the program begins execution.**
- **A method is a construct that contains statements. The main method in this program contains the System.out.println statement. This statement displays the string *Welcome to Java!* on the console (line 4). *String* is a programming term meaning a sequence of characters. A string must be enclosed in double quotation marks. Every statement in Java ends with a semicolon (;), known as the *statement terminator.***
- ***Reserved words*, or *keywords*, have a specific meaning to the compiler and cannot be used for other purposes in the program. For example, when the compiler sees the word class, it understands that the word after class is the name for the class. Other reserved words in this program are public, static, and void.**
- **Line 3 is a *comment* that documents what the program is and how it is constructed. Comments help programmers to communicate and understand the program. They are not programming statements and thus are ignored by the compiler. In Java, comments are preceded by two slashes (//) on a line, called a *line comment,* or enclosed between /* and */ on one or several lines, called a *block comment* or *paragraph comment*. When the compiler sees //, it ignores all text after // on the same line. When it sees /*, it scans for the next */ and ignores any text between /* and */.**
- **A pair of curly braces in a program forms a *block* that groups the program's components. In Java, each block begins with an opening brace ({) and ends with a closing brace (}). Every class has a *class block* that groups the data and methods of the class. Similarly, every method has a *method block* that groups the statements in the method. Blocks can be *nested*, meaning that one block can be placed within another, as shown in the following code.**

Caution

Java source programs are case sensitive. It would be wrong, for example, to replace main in the program with Main. You have seen several special characters (e.g.,{ }, //, ;) in the program. They areused in almost every program.

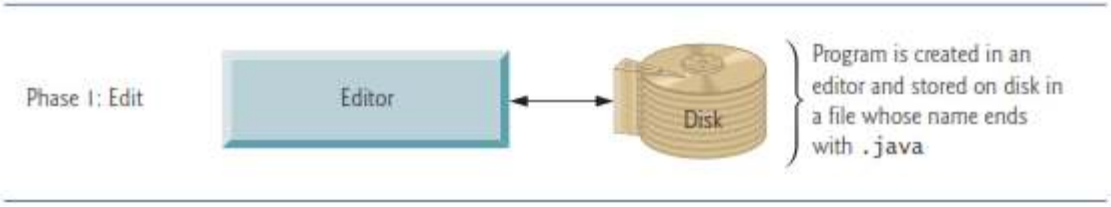


Creating, Compiling, and Executing a Java Program

You save a Java program in a .java file and compile it into a .class file. The .class file is executed by the Java Virtual Machine.

Note
You must first install, configure the JDK and set up the environment before you can compile and run programs.

Phase 1: Creating a Program
Phase 1 consists of editing a file with an *editor program*, normally known simply as an *editor*. You type a Java program (typically referred to as source code) using the editor, make any necessary corrections and save the program on a secondary storage device, such as your hard drive. A file name ending with the .java extension indicates that the file contains Java source code.
Note
The source file must end with the extension .java and must have the same exact name as the public class name.



Two editors widely used on Linux systems are vi and emacs. On Windows, Notepad will suffice.

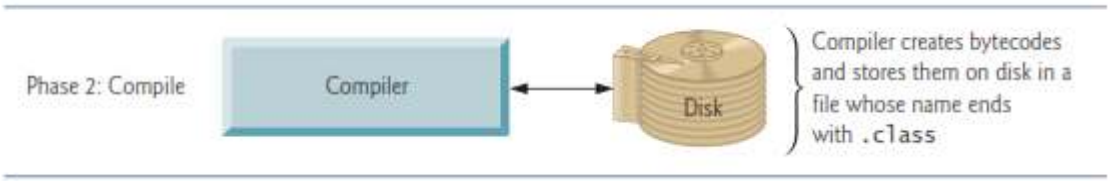
For organizations that develop substantial information systems, integrated development environments (IDEs) are available from many major software suppliers. IDEs provide tools that support the software development process, including editors for writing and editing programs and debuggers for locating logic errors—errors that cause programs to execute incorrectly. Popular IDEs include Eclipse (www.eclipse.org) and NetBeans (www.netbeans.org).

Phase 2: Compiling a Java Program into Bytecodes

In Phase 2, you use the command javac (the Java compiler) to compile a program. For example, to compile a program called Welcome.java, you'd type

javac Welcome.java

in the command window of your system (i.e., the Command Prompt in Windows, the *shell prompt* in Linux or the Terminal application in Mac OS X). If the program compiles, the compiler produces a .class file called Welcome.class that contains the compiled version of the program.



A Java compiler translates a Java source file into a Java bytecode file. If there aren't any syntax errors, the compiler generates a bytecode file with a .class extension. Thus, the preceding command generates a file named Welcome.class, The Java language is a high-level language, but Java bytecode is a low-level language. The bytecode is similar to machine instructions but is architecture neutral and can run on any platform that has a Java Virtual Machine (JVM). Rather than a physical machine, the virtual machine is a program that interprets Java bytecode. This is one of Java's primary advantages: Java bytecode can run on a variety of hardware platforms and operating systems.

Note:

A virtual machine (VM) is a software application that simulates a computer but hides the underlying operating system and hardware from the programs that interact with it. If the same VM is implemented on many computer platforms, applications that it executes can be used on all those platforms. The JVM is one of the most widely used virtual machines. Microsoft's .NET uses a similar virtual-machine architecture.

Unlike machine language, which is dependent on specific computer hardware, bytecodes are platform independent—they do not depend on a particular hardware platform. So, Java's bytecodes are portable without recompiling the source code, the same bytecodes can execute on any platform containing a JVM that understands the version of Java in which the bytecodes were compiled. The JVM is invoked by the java command. For example, to execute a Java application called Welcome, you'd type the command

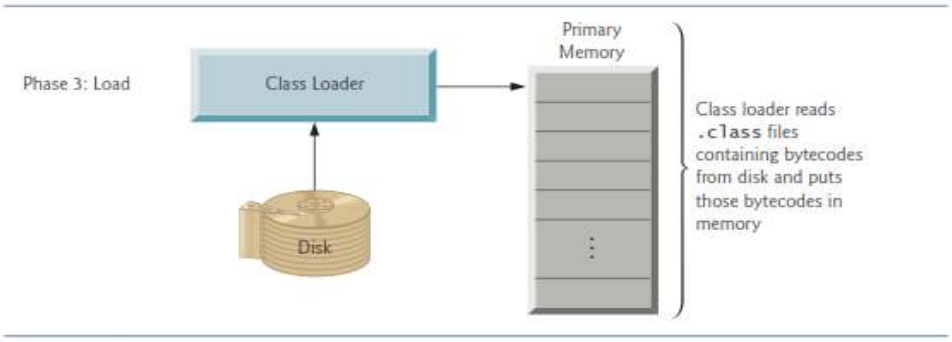
java Welcome

in a command window to invoke the JVM, which would then initiate the steps necessary to execute the application. This begins Phase 3.

Phase 3: Loading a Program into Memory

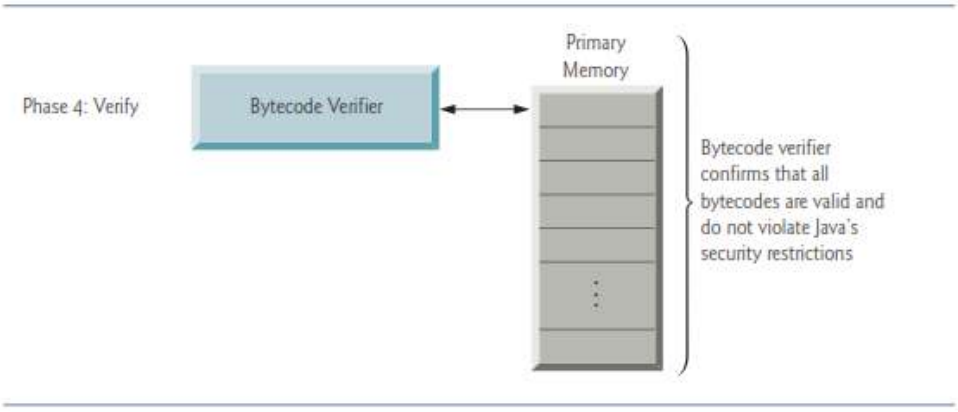
In Phase 3, the JVM places the program in memory to execute it—this is known as loading.The JVM's class loader takes the .class files containing the program's bytecodes and transfers them to primary memory. The class loader

also loads any of the .class files provided by Java that your program uses. The .class files can be loaded from a disk on your system or over a network (e.g., your local college or company network, or the Internet).



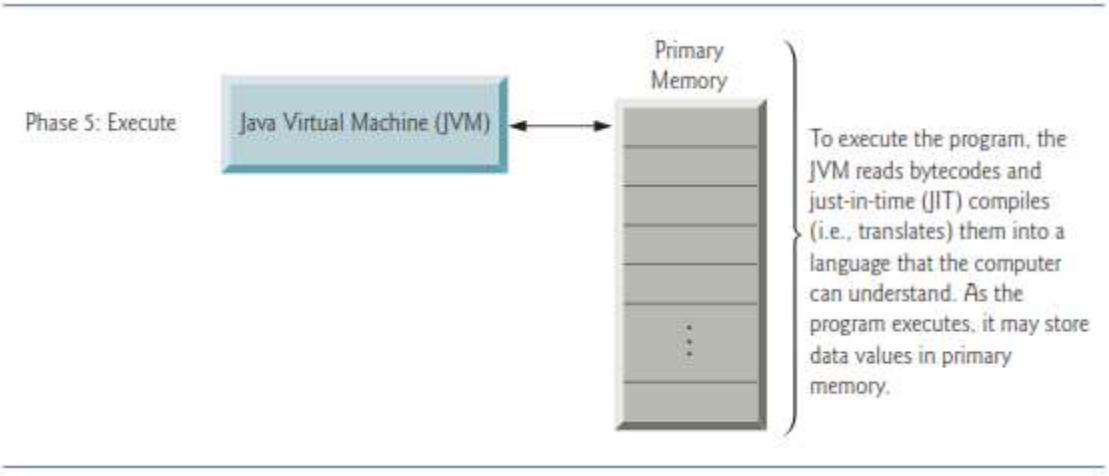
Phase 4: Bytecode Verification

In Phase 4, as the classes are loaded, the bytecode verifier examines their bytecodes to ensure that they're valid and do not violate Java's security restrictions. Java enforces strong security to make sure that Java programs arriving over the network do not damage your files or your system (as computer viruses and worms might).



Phase 5: Execution

In Phase 5, the JVM executes the program's bytecodes, thus performing the actions specified by the program. In early Java versions, the JVM was simply an interpreter for Java bytecodes. This caused most Java programs to execute slowly, because the JVM would interpret and execute one bytecode at a time. Some modern computer architectures can execute several instructions in parallel. Today's JVMs typically execute bytecodes using a combination of interpretation and so-called just-in-time (JIT) compilation. In this process, the JVM analyzes the bytecodes as they're interpreted, searching for *hot spots* (parts of the bytecodes that execute frequently). For these parts, a just-in-time (JIT) compiler (known as the Java HotSpot compiler) translates the bytecodes into the underlying computer's machine language. When the JVM encounters these compiled parts again, the faster machine-language code executes.



Thus Java programs actually go through *two* compilation phases—one in which source code is translated nto bytecodes (for portability across JVMs on different computer platforms) and a second in which, during execution, the bytecodes are translated into machine language for the actual computer on which the program executes.

Setting the PATH Environment Variable

The PATH environment variable on your computer designates which directories the computer searches when looking for applications, such as the applications that enable you to compile and run your Java applications (called javac and java, respectively). *Carefully follow the installation instructions for Java on your platform to ensure that you set the PATH environment variable correctly.*

If you do not set the PATH variable correctly, when you use the JDK's tools, you'll receive a message like: *'java' is not recognized as an internal or external command, operable program or batch file.*

If you've downloaded a newer version of the JDK, you may need to change the name of the JDK's installation directory in the PATH variable.

Setting the CLASSPATH Environment Variable

If you attempt to run a Java program and receive a message like:

Exception in thread "main" java.lang.NoClassDefFoundError: YourClass

then your system has a CLASSPATH environment variable that must be modified. To fix the preceding error, follow the steps in setting the PATH environment variable, to locate the CLASSPATH variable, then edit the variable's value to include the local directory (typically represented as a dot. On Windows add .; at the beginning of the CLASSPATH's value (with no spaces before or after these characters). On other platforms, replace the semicolon with the appropriate path separator characters (oftena colon).

Creating Your First Application (Microsoft Windows)

Your first application, HelloWorldApp, will simply display the greeting "Hello world!". To create this program, you will:

- **Create a source file**
  A source file contains code, written in the Java programming language, that you and other programmers can understand. You can use any text editor to create and edit source files.
- **Compile the source file into a .class file**
  The Java programming language *compiler* (javac) takes your source file and translates its text into instructions that the Java virtual machine can understand. The instructions contained within this file are known as *bytecodes*.
- **Run the program**
  The Java application *launcher tool* (java) uses the Java virtual machine to run your application.

Create a Source File

To create a source file, you have two options:

- **You can save the file HelloWorldApp.java on your computer and avoid a lot of typing. Then, you can go straight to Compile the Source File into a .class File.**
- **Or, you can use the following (longer) instructions.**

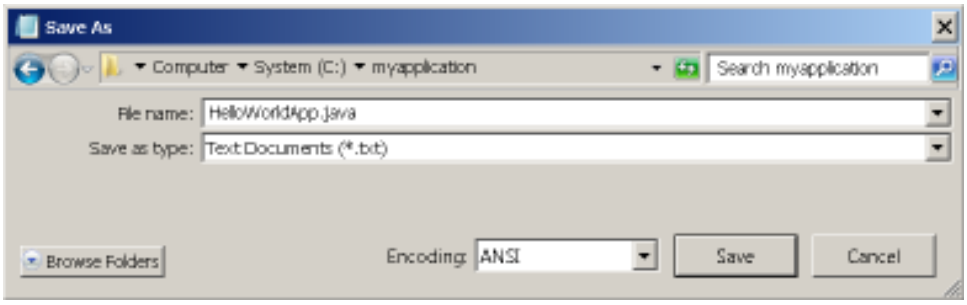
First, start your editor. You can launch the Notepad editor from the Start menu by selecting Programs > Accessories > Notepad. In a new document, type in the following code:

```
/**
 * The HelloWorldApp class implements an application that
 * simply prints "Hello World!" to standard output.
 */
class HelloWorldApp {
   public static void main(String[] args) {
      System.out.println("Hello World!"); // Display the string.
   }
}
```

Save the code in a file with the name HelloWorldApp.java. To do this in Notepad, first choose the File > Save As menu item. Then, in the Save As dialog box:

1. **Using the Save in combo box, specify the folder (directory) where you'll save your file. In this example, the directory is myapplication on the C drive.**
2. **In the File name text field, type "HelloWorldApp.java", including the quotation marks.**
3. **From the Save as type combo box, choose Text Documents (*.txt).**
4. **In the Encoding combo box, leave the encoding as ANSI.**

When you're finished, the dialog box should look like this.

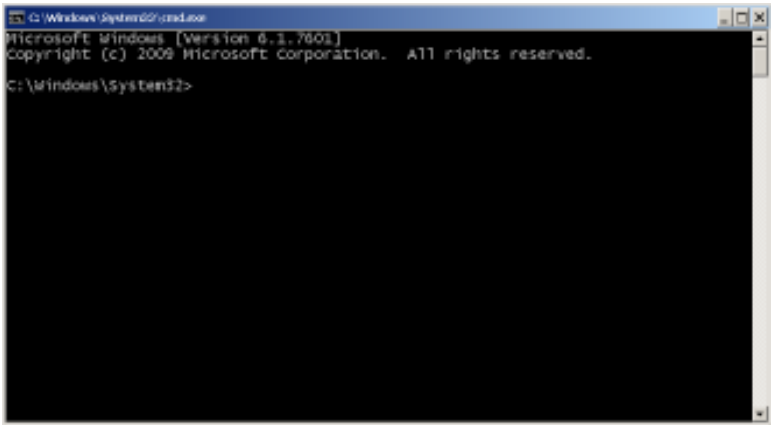


The Save As dialog just before you click Save.

Now click Save, and exit Notepad.

Compile the Source File into a .class File

Bring up a shell, or "command," window. You can do this from the Start menu by choosing Run... and then entering cmd. The shell window should look similar to the following figure.



A cmd window.

The prompt shows your *current directory*. When you bring up the prompt, your current directory is usually your home directory for Windows XP (as shown in the preceding figure.

To compile your source file, change your current directory to the directory where your file is located. For example, if your source directory is myapplication on the C drive, type the following command at the prompt and press Enter:

cd C:\myapplication

Now the prompt should change to C:\myapplication>.

Note:

To change to a directory on a different drive, you must type an extra command: the name of the drive. For example, to change to the myapplication directory on the D drive, you must enter D:, as follows:

```
C:\>D:

D:\>cd myapplication

D:\myapplication>
```

If you enter dir at the prompt, you should see your source file, as follows:

```
C:\>cd myapplication

C:\myapplication>dir
 Volume in drive C is System
 Volume Serial Number is F2E8-C8CC

 Directory of C:\myapplication

2014-04-24  01:34 PM    <DIR>          .
2014-04-24  01:34 PM    <DIR>          ..
2014-04-24  01:34 PM               267 HelloWorldApp.java
               1 File(s)            267 bytes
               2 Dir(s)  93,297,991,680 bytes free
```

```
C:\myapplication>
```

Now you are ready to compile. At the prompt, type the following command and press Enter.

javac HelloWorldApp.java

The compiler has generated a bytecode file, HelloWorldApp.class. At the prompt, type dir to see the new file that was generated as follows:

```
C:\myapplication>javac HelloWorldApp.java

C:\myapplication>dir
 Volume in drive C is System
 Volume Serial Number is F2E8-C8CC

 Directory of C:\myapplication

2014-04-24  02:07 PM    <DIR>          .
2014-04-24  02:07 PM    <DIR>          ..
2014-04-24  02:07 PM               432 HelloWorldApp.class
2014-04-24  01:34 PM               267 HelloWorldApp.java
               2 File(s)            699 bytes
               2 Dir(s)  93,298,032,640 bytes free

C:\myapplication>
```

Now that you have a .class file, you can run your program.

Run the Program

In the same directory, enter the following command at the prompt:

java -cp . HelloWorldApp

You should see the following on your screen:

```
C:\myapplication>java -cp . HelloWorldApp
Hello World!

C:\myapplication>
```

The HelloWorldApp Class Definition

The following bold text begins the class definition block for the "Hello World!" application:

```
/**
 * The HelloWorldApp class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorldApp {
   public static void main(String[] args) {
      System.out.println("Hello World!"); // Display the string.
```

```
    }
}
```

As shown above, the most basic form of a class definition is:

```
class name {
    . . .
}
```

The keyword class begins the class definition for a class named name, and the code for each class appears between the opening and closing curly braces marked in bold above. Chapter 2 provides an overview of classes in general, and Chapter 4 discusses classes in detail. For now it is enough to know that every application begins with a class definition.

The main Method

The following bold text begins the definition of the main method:

```
/**
 * The HelloWorldApp class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //Display the string.
    }
}
```

In the Java programming language, every application must contain a main method whose signature is:

public static void main(String[] args)

The modifiers public and static can be written in either order (public static or static public), but the convention is to use public static as shown above. You can name the argument anything you want, but most programmers choose "args" or "argv".

The main method is similar to the main function in C and C++; it's the entry point for your application and will subsequently invoke all the other methods required by your program.

The main method accepts a single argument: an array of elements of type String. This array is the mechanism through which the runtime system passes information to your application. For example:

java *MyApp arg1 arg2*

Each string in the array is called a *command-line argument*. Command-line arguments let users affect the operation of the application without recompiling it. For example, a sorting program might allow the user to specify that the data be sorted in descending order with this command-line argument:

-descending

The "Hello World!" application ignores its command-line arguments, but you should be aware of the fact that such arguments do exist.

Finally, the line:

System.out.println("Hello World!");

uses the System class from the core library to print the "Hello World!" message to standard output.

Chapter 2 Language Basics

Variables

The Java programming language defines the following kinds of variables:

- **Instance Variables (Non-Static Fields) Technically speaking, objects store their individual states in "non-static fields", that is, fields declared without the `static` keyword. Non-static fields are also known as *instance variables* because their values are unique to each *instance* of a class (to each object, in other words.**
- **Class Variables (Static Fields) A *class variable* is any field declared with the `static` modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated.**
- **Local Variables Similar to how an object stores its state in fields, a method will often store its temporary state in *local variables*. The syntax for declaring a local variable is similar to declaring a field (for example, `int count = 0;`). There is no special keyword designating a variable as local; that determination comes entirely from the location in which the variable is declared — which is between the opening and closing braces of a method. As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.**
- **Parameters Recall that the signature for the `main` method is `public static void main(String[] args)`. Here, the `args` variable is the parameter to this method. The important thing to remember is that parameters are always classified as "variables" not "fields.**

Naming

Variable names are case-sensitive. A variable's name can be any legal identifier — an unlimited-length sequence of Unicode letters and digits, beginning with a letter, the dollar sign "`$`", or the underscore character "`_`". The convention, however, is to always begin your variable names with a letter, not "`$`" or "`_`". Additionally, the dollar sign character, by convention, is never used at all. You may find some situations where auto-generated names will contain the dollar sign, but your variable names should always avoid using it. A similar convention exists for the underscore character; while it's technically legal to begin your variable's name with "`_`", this practice is discouraged. White space is not permitted. Also keep in mind that the name you choose must not be a keyword or reserved word:

Java Language Keywords

Here is a list of keywords in the Java programming language. You cannot use any of the following as identifiers in your programs. The keywords `const` and `goto` are reserved, even though they are not currently used. `true`, `false`, and `null` might seem like keywords, but they are actually literals; you cannot use them as identifiers in your programs.

`abstract`	`continue`	`for`	`new`	`switch`
`assert`***	`default`	`goto`*	`package`	`synchronized`
`boolean`	`do`	`if`	`private`	`this`
`break`	`double`	`implements`	`protected`	`throw`
`byte`	`else`	`import`	`public`	`throws`
`case`	`enum`****	`instanceof`	`return`	`transient`
`catch`	`extends`	`int`	`short`	`try`
`char`	`final`	`interface`	`static`	`void`
`class`	`finally`	`long`	`strictfp`**	`volatile`
`const`*	`float`	`native`	`super`	`while`

* **not used**

** **added in 1.2**

*** added in 1.4
**** added in 5.0

If the name you choose consists of only one word, spell that word in all lowercase letters. If it consists of more than one word, capitalize the first letter of each subsequent word. The names `gearRatio` and `currentGear` are prime examples of this convention. If your variable stores a constant value, such as `static final int NUM_GEARS = 6`, the convention changes slightly, capitalizing every letter and separating subsequent words with the underscore character. By convention, the underscore character is never used elsewhere.

Primitive Data Types

A primitive type is predefined by the language and is named by a reserved keyword. Primitive values do not share state with other primitive values. The eight primitive data types supported by the Java programming language are:

byte	8-bit signed two's complement integer	-128 to 127 (inclusive)
short	16-bit signed two's complement integer	-32,768 to 32,767 (inclusive)
int	32-bit signed two's complement integer	-2^{31} to $2^{31}-1$
long	64-bit two's complement integer	-2^{63} to $2^{63}-1$
float	a single-precision 32-bit IEEE 754 floating point	
double	a double-precision 64-bit IEEE 754 floating point	
boolean	The `boolean` data type has only two possible values: `true` and `false`	
char	a single 16-bit Unicode character	`'\u0000'` (or 0) to `'\uffff'` (or 65,535 inclusive).

In Java SE 8 and later, you can use the `int` data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of $2^{32}-1$. Static methods like `compareUnsigned`, `divideUnsigned` etc have been added to the `Integer` class to support the arithmetic operations for unsigned integers.
In Java SE 8 and later, you can use the `long` data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of $2^{64}-1$. Use this data type when you need a range of values wider than those provided by `int`. The `Long` class also contains methods like `compareUnsigned`, `divideUnsigned` etc to support arithmetic operations for unsigned long.
For decimal values, double is generally the default choice.
double and float data types should never be used for precise values, such as currency. For that, you will need to use the java.math.BigDecimal class instead
In addition to the eight primitive data types listed above, the Java programming language also provides special support for character strings via the java.lang.String class. Enclosing your character string within double quotes will automatically create a new `String` object. `String` objects are *immutable*, which means that once created, their values cannot be changed.

Literals

A *literal* is the source code representation of a fixed value; literals are represented directly in your code without requiring computation.

Integer Literals

An integer literal is of type `long` if it ends with the letter `L` or `l`; otherwise it is of type `int`. Values of the integral types `byte`, `short`, `int`, and `long` can be created from `int` literals. Integer literals can be expressed by these number systems:

- **Decimal: Base 10, whose digits consists of the numbers 0 through 9; this is the number system you use every day**
- **Hexadecimal: Base 16, whose digits consist of the numbers 0 through 9 and the letters A through F (0x….).**
- **Binary: Base 2, whose digits consists of the numbers 0 and 1 (you can create binary literals in Java SE 7 and later) (0b…).**

Floating-Point Literals

A floating-point literal is of type `float` if it ends with the letter `F` or `f`; otherwise its type is `double` and it can optionally end with the letter `D` or `d`.

The floating point types (`float` and `double`) can also be expressed using E or e (for scientific notation), F or f (32-bit float literal) and D or d (64-bit double literal; this is the default and by convention is omitted).

Character and String Literals

Literals of types `char` and `String` may contain any Unicode (UTF-16) characters. If your editor and file system allow it, you can use such characters directly in your code. If not, you can use a "Unicode escape" such as `'\u0108'` (capital C with circumflex), or `"S\u00ED Se\u00F1or"` (Sí Señor in Spanish). Always use 'single quotes' for `char` literals and "double quotes" for `String` literals. Unicode escape sequences may be used elsewhere in a program (such as in field names, for example), not just in `char` or `String` literals.

The Java programming language also supports a few special escape sequences for `char` and `String` literals: `\b` (backspace), `\t` (tab), `\n` (line feed), `\f` (form feed), `\r` (carriage return), `\"` (double quote), `\'` (single quote), and `\\` (backslash).

There's also a special `null` literal that can be used as a value for any reference type. `null` may be assigned to any variable, except variables of primitive types. There's little you can do with a `null` value beyond testing for its presence. Therefore, `null` is often used in programs as a marker to indicate that some object is unavailable.

Finally, there's also a special kind of literal called a *class literal*, formed by taking a type name and appending `".class"`; for example, `String.class`. This refers to the object (of type `Class`) that represents the type itself.

Using Underscore Characters in Numeric Literals

In Java SE 7 and later, any number of underscore characters (_) can appear anywhere between digits in a numerical literal. This feature enables you, for example. to separate groups of digits in numeric literals, which can improve the readability of your code.

You can place underscores only between digits; you cannot place underscores in the following places:

- **At the beginning or end of a number**
- **Adjacent to a decimal point in a floating point literal**
- **Prior to an `F` or `L` suffix**
- **In positions where a string of digits is expected**

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi =  3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

Operators

Operator Precedence	
Operators	**Precedence**
postfix	***expr*++ *expr*--**
unary	**++*expr* --*expr* +*expr* -*expr* ~ !**
multiplicative	*** / %**
additive	**+ -**
shift	**<< >> >>>**
relational	**< > <= >= instanceof (Type Comparison Operator)**
equality	**== !=**
bitwise AND	**&**
bitwise exclusive OR	**^**
bitwise inclusive OR	**|**
logical AND (*Conditional-AND* operation)	**&&**
logical OR (*Conditional-OR* operation)	**||**
ternary	**? :**
assignment	**= += -= *= /= %= &= ^= |= <<= >>= >>>=**

Expressions

An *expression* is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language, that evaluates to a single value.

Statements

Statements are roughly equivalent to sentences in natural languages. A *statement* forms a complete unit of execution. The following types of expressions can be made into a statement by terminating the expression with a semicolon (;).

- **Assignment expressions**
- **Any use of ++ or --**
- **Method invocations**
- **Object creation expressions**

Such statements are called *expression statements*.

In addition to expression statements, there are two other kinds of statements: *declaration statements* and *control flow statements*.

A *declaration statement* declares a variable.

Finally, *control flow statements* regulate the order in which statements get executed.

Blocks: A *block* is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed.

Control Flow Statements

- **The decision-making statements (`if-then`, `if-then-else`, `switch`)**
- **The looping statements (`for`, `while`, `do-while`)**
- **The branching statements (`break`, `continue`, `return`)**

A `switch` works with the `byte`, `short`, `char`, and `int` primitive data types. It also works with *enumerated types*, the `String` class, and a few special classes that wrap certain primitive types: `Character`, `Byte`, `Short`, and `Integer`.

The body of a `switch` statement is known as a *switch block*. A statement in the `switch` block can be labeled with one or more `case` or `default` labels. The `switch` statement evaluates its expression, then executes all statements that follow the matching `case` label.

Each `break` statement terminates the enclosing `switch` statement. Control flow continues with the first statement following the `switch` block. The `break` statements are necessary because without them, statements in `switch` blocks *fall through*: All statements after the matching `case` label are executed in sequence, regardless of the expression of subsequent `case` labels, until a `break` statement is encountered.

Technically, the final `break` is not required because flow falls out of the `switch` statement. Using a `break` is recommended so that modifying the code is easier and less error prone. The `default` section handles all values that are not explicitly handled by one of the `case` sections.

The `for` statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the `for` statement can be expressed as follows:

```
for (initialization; termination; increment) {
    statement(s)
}
```

Expressions in a for Statement's Header Are Optional
All three expressions in a `for` header are optional. If the loopContinuationCondition is omitted, Java assumes that the loop-continuation condition is always true, thus creating an infinite loop. You might omit the initialization expression if the program initializes the control variable before the loop. You might omit the increment expression if the program calculates the increment with statements in the loop's body or if no increment is needed. The increment expression in a `for` acts as if it were a standalone statement at the end of the `for's` body.

The `for` statement also has another form designed for iteration through Collections and arrays This form is sometimes referred to as the *enhanced for* statement, and can be used to make your loops more compact and easy to read. The following program, `EnhancedForDemo`, uses the enhanced `for` to loop through the array:

```
class EnhancedForDemo {
    public static void main(String[] args){
         int[] numbers =
             {1,2,3,4,5,6,7,8,9,10};
         for (int item : numbers) {
             System.out.println("Count is: " + item);
         }
    }
}
```

`Note:`
The `item` is a copy of an element (for example at location 'i') of `numbers` array, so assignment to that copy does not update the element at location 'i'.

```
int[] numbers == new int[5];
```

```
for(int x : numbers)
        x = 5;
for(int x : a)
        System.out.printf("%d\t", x);

Output: 0       0       0       0       0
```

Arrays

An *array* is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed. Each item in an array is called an *element*, and each element is accessed by its numerical *index.*

```
int[] anArray;
anArray = new int[10];
String[] s1 = new String[ 100 ], s2 = new String[ 27 ];
or:
String[] s1 = new String[ 100 ];
String[] s2 = new String[ 27 ];
```

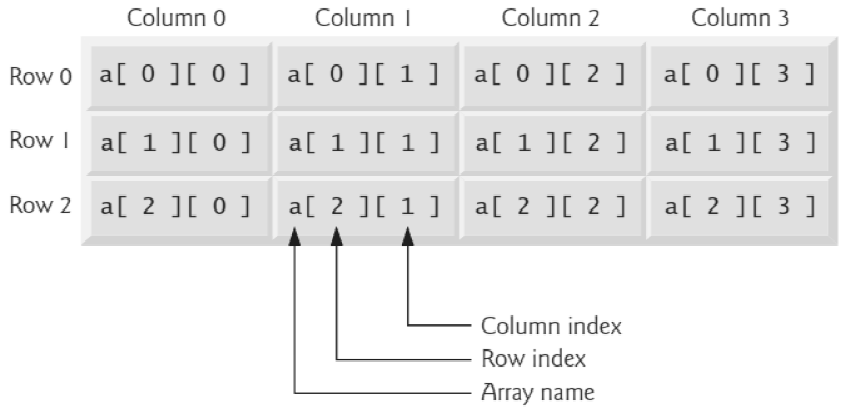
Note: When only one variable is declared in each declaration, the square brackets can be placed either after the type or after the array variable name, as in:

```
String s1[] = new String[ 100 ];
String s2[] = new String[ 27 ];
```

Note: Declaring multiple array variables in a single declaration can lead to subtle errors. Consider the declaration `int[] a, b, c;`. If a, b and c should be declared as array variables, then this declaration is correct; placing square brackets directly following the type indicates that all the identifiers in the declaration are array variables. However, if only `a` is intended to be an array variable, and `b` and `c` are intended to be individual int variables, then this declaration is incorrect; the declaration `int a[], b, c;` would achieve the desired result.

You can also declare an array of arrays (also known as a *multidimensional* array) by using two or more sets of brackets, such as `String[][] names`. Each element, therefore, must be accessed by a corresponding number of index values. In the Java programming language, a multidimensional array is an array whose components are themselves arrays. This is unlike arrays in C or Fortran.

```
String[][] names = {
            {"Mr. ", "Mrs. ", "Ms. "},
            {"Smith", "Jones"}
        };

int[][] a = new int[3][4];
```




```
int[][] b = new int[2][];
```

```
b[0] = new int[3];
b[1] = new int[4];
```

Example: Output rows and columns of a two-dimensional array

```
      public static void outputArray(int[][] array)
      {
          // loop through array's rows
          for ( int row = 0; row < array.length; row++ )
          {
              // loop through columns of current row
              for ( int column `= 0; column < array[ row ].length; column++ )
                  System.out.printf( "%d ", array[ row ][ column ] );

              System.out.println(); // start new line of output
          } // end outer for
      } // end method outputArray
```

Example: Find maximum element

```
public int getMaximum(int[][] array)
{
    // assume first element of array is largest
    int max = array [ 0 ][ 0 ];
    // loop through rows of array
    for ( int[] row : array )
    {
        // loop through columns of current row
        for ( int element : row )
        {
            if (element > max )
            max = element;
        } // end inner for
    } // end outer for
}
```

Variable-Length Argument List

With variable-length argument lists, you can create methods that receive an unspecified number of arguments. A type followed by an ellipsis (...) in a method's parameter list indicates that the method receives a variable number of arguments of that particular type. This use of the ellipsis can occur only once in a parameter list, and the ellipsis, together with its type, must be placed at the end of the parameter list. While you can use method overloading and array passing to accomplish much of what is accomplished with variable-length argument lists, using an ellipsis in a method's parameter list is more concise.

Example: Calculate average

```
public static double average(double... numbers)
{
    double total = 0.0;
    // calculate total using the enhanced for statement
    for ( double d : numbers )
        total += d;

    return total / numbers.length;
} // end method average

//
```

Java treats the variable-length argument list as an array whose elements are all of the same type. Hence, the method body can manipulate the parameter numbers as an array of doubles. respectively. Method average has a variable-length argument list, so it can average as many double arguments as the caller passes.

```
average( d1, d2 );
average( d1, d2, d3, d4 );
```

Chapter 3 Classes and Objects

Declaring Classes

```
class MyClass extends MySuperClass implements YourInterface {
    // field, constructor, and
    // method declarations
}
```

means that `MyClass` is a subclass of `MySuperClass` and that it implements the `YourInterface` interface.

In general, class declarations can include these components, in order:

1. **Modifiers such as *public*, *private*, and a number of others that you will encounter later.**
2. **The class name, with the initial letter capitalized by convention.**
3. **The name of the class's parent (superclass), if any, preceded by the keyword *extends*. A class can only *extend* (subclass) one parent.**
4. **A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword *implements*. A class can *implement* more than one interface.**
5. **The class body, surrounded by braces, {}.**

Declaring Member Variables

There are several kinds of variables:

- **Member variables in a class—these are called *fields*.**
- **Variables in a method or block of code—these are called *local variables*.**
- **Variables in method declarations—these are called *parameters*.**

Field declarations are composed of three components, in order:

1. **Zero or more modifiers, such as `public` or `private`.**
2. **The field's type.**
3. **The field's name.**

Defining Methods

More generally, method declarations have six components, in order:

1. **Modifiers—such as `public`, `private`, and others you will learn about later.**
2. **The return type—the data type of the value returned by the method, or `void` if the method does not return a value.**
3. **The method name—the rules for field names apply to method names as well, but the convention is a little different.**
4. **The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, `()`. If there are no parameters, you must use empty parentheses.**
5. **An exception list—to be discussed later.**
6. **The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.**

Definition: Two of the components of a method declaration comprise the *method signature*—the method's name and the parameter types.

By convention, method names should be a verb in lowercase or a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, etc. In multi-word names, the first letter of each of the second and following words should be capitalized.

Overloading Methods

The Java programming language supports *overloading* methods, and Java can distinguish between methods with different *method signatures*. This means that methods within a class can have the same name if they have different parameter lists. You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

Providing Constructors for Your Classes

A class contains constructors that are invoked to create objects from the class blueprint. Constructor declarations look like method declarations—except that they use the name of the class and have no return type.

You don't have to provide any constructors for your class, but you must be careful when doing this. The compiler automatically provides a no-argument, default constructor for any class without constructors. This default constructor will call the no-argument constructor of the superclass. In this situation, the compiler will complain if the superclass doesn't have a no-argument constructor so you must verify that it does. If your class has no explicit superclass, then it has an implicit superclass of `Object`, which *does* have a no-argument constructor.

You can use access modifiers in a constructor's declaration to control which other classes can call the constructor.

Passing Information to a Method or a Constructor

The declaration for a method or a constructor declares the number and the type of the arguments for that method or constructor.

Note: *Parameters* refers to the list of variables in a method declaration. *Arguments* are the actual values that are passed in when the method is invoked. When you invoke a method, the arguments used must match the declaration's parameters in type and order.

Note: If you want to pass a method into a method, then use a lambda expression or a method reference.

Arbitrary Number of Method's Arguments

You can use a construct called *varargs* to pass an arbitrary number of values to a method. You use varargs when you don't know how many of a particular type of argument will be passed to the method. It's a shortcut to creating an array manually.

To use varargs, you follow the type of the last parameter by an ellipsis (three dots, ...), then a space, and the parameter name. The method can then be called with any number of that parameter, including none.

```
public Polygon polygonFrom(Point... corners) {
    int numberOfSides = corners.length;
    double squareOfSide1, lengthOfSide1;
    squareOfSide1 = (corners[1].x - corners[0].x)
                    * (corners[1].x - corners[0].x)
                    + (corners[1].y - corners[0].y)
                    * (corners[1].y - corners[0].y);
    lengthOfSide1 = Math.sqrt(squareOfSide1);
```

```
    // more method body code follows that creates and returns a
    // polygon connecting the Points
}
```

You can see that, inside the method, `corners` is treated like an array. The method can be called either with an array or with a sequence of arguments. The code in the method body will treat the parameter as an array in either case.

You will most commonly see varargs with the printing methods; for example, this `printf` method:

```
public PrintStream printf(String format, Object... args)
```

allows you to print an arbitrary number of objects. It can be called like this:

```
System.out.printf("%s: %d, %s%n", name, idnum, address);
```

or like this

```
System.out.printf("%s: %d, %s, %s, %s%n", name, idnum, address, phone, email);
```

or with yet a different number of arguments.

Passing Primitive Data Type Arguments

Primitive arguments, such as an `int` or a `double`, are passed into methods *by value*. This means that any changes to the values of the parameters exist only within the scope of the method. When the method returns, the parameters are gone and any changes to them are lost.

Passing Reference Data Type Arguments

Reference data type parameters, such as objects, are also passed into methods *by value*. This means that when the method returns, the passed-in reference still references the same object as before. *However*, the values of the object's fields *can* be changed in the method, if they have the proper access level.

Returning a Value from a Method

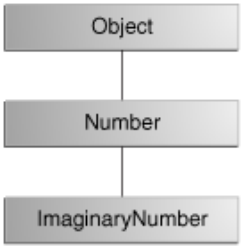
A method returns to the code that invoked it when it

- **completes all the statements in the method,**
- **reaches a `return` statement, or**
- **throws an exception (covered later),**

You declare a method's return type in its method declaration. Within the body of the method, you use the `return` statement to return the value.

Returning a Class or Interface

When a method uses a class name as its return type, such as `whosFastest` does, the class of the type of the returned object must be either a subclass of, or the exact class of, the return type. Suppose that you have a class hierarchy in which `ImaginaryNumber` is a subclass of `java.lang.Number`, which is in turn a subclass of `Object`, as illustrated in the following figure.



The class hierarchy for ImaginaryNumber

Note: You also can use interface names as return types. In this case, the object returned must implement the specified interface.

Objects

A typical Java program creates many objects, which as you know, interact by invoking methods. Through these object interactions, a program can carry out various tasks, such as implementing a GUI, running an animation, or sending and receiving information over a network. Once an object has completed the work for which it was created, its resources are recycled for use by other objects.

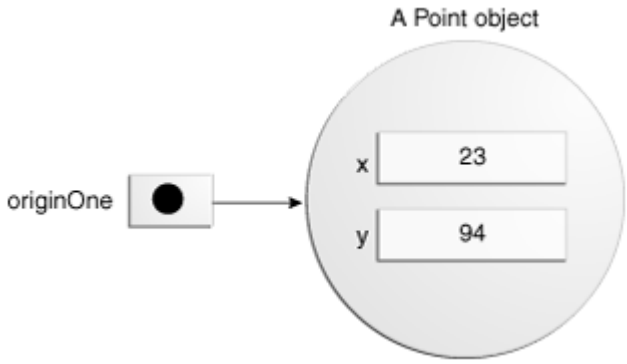
Creating Objects

As you know, a class provides the blueprint for objects; you create an object from a class. Each of create statements has three parts (discussed in detail below):

1. **Declaration: The code set in bold are all variable declarations that associate a variable name with an object type.**
2. **Instantiation: The `new` keyword is a Java operator that creates the object.**
3. **Initialization: The `new` operator is followed by a call to a constructor, which initializes the new object.**

The `new` operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The `new` operator also invokes the object constructor. The `new` operator requires a single, postfix argument: a call to a constructor. The name of the constructor provides the name of the class to instantiate.

Note: The phrase "instantiating a class" means the same thing as "creating an object." When you create an object, you are creating an "instance" of a class, therefore "instantiating" a class.



Using the this Keyword

Within an instance method or a constructor, this is a reference to the current object — the object whose method or constructor is being called. You can refer to any member of the current object from within an instance method or a constructor by using this.

- **Using this with a Field: The most common reason for using the this keyword is because a field is shadowed by a method or constructor parameter.**
- **Using this with a Constructor: From within a constructor, you can also use the this keyword to call another constructor in the same class. Doing so is called an explicit constructor invocation. If present, the invocation of another constructor must be the first line in the constructor.**

Controlling Access to Members of a Class

Access level modifiers determine whether other classes can use a particular field or invoke a particular method. There are two levels of access control:

- **At the top level—`public`, or *package-private* (no explicit modifier).**
- **At the member level—`public`, `private`, `protected`, or *package-private* (no explicit modifier).**

A class may be declared with the modifier `public`, in which case that class is visible to all classes everywhere. If a class has no modifier (the default, also known as *package-private*), it is visible only within its own package (packages are named groups of related classes).

At the member level, you can also use the `public` modifier or no modifier (*package-private*) just as with top-level classes, and with the same meaning. For members, there are two additional access modifiers: `private` and `protected`. The `private` modifier specifies that the member can only be accessed in its own class. The `protected` modifier specifies that the member can only be accessed within its own package (as with *package-private*) and, in addition, by a subclass of its class in another package.

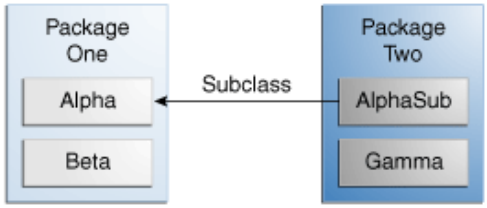
The following table shows the access to members permitted by each modifier.Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

The first data column indicates whether the class itself has access to the member defined by the access level. As you can see, a class always has access to its own members. The second column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member. The third column indicates whether subclasses of the class declared outside this package have access to the member. The fourth column indicates whether all classes have access to the member.

Access levels affect you in two ways. First, when you use classes that come from another source, such as the classes in the Java platform, access levels determine which members of those classes your own classes can use. Second, when you write a class, you need to decide what access level every member variable and every method in your class should have.

Let's look at a collection of classes and see how access levels affect visibility. The following figure shows the four classes in this example and how they are related.



Classes and Packages of the Example Used to Illustrate Access Levels

The following table shows where the members of the Alpha class are visible for each of the access modifiers that can be applied to them.Visibility

Modifier	Alpha	Beta	Alphasub	Gamma
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Tips on Choosing an Access Level:

If other programmers use your class, you want to ensure that errors from misuse cannot happen. Access levels can help you do this.

- **Use the most restrictive access level that makes sense for a particular member. Use `private` unless you have a good reason not to.**
- **Avoid `public` fields except for constants. (Many of the examples in the tutorial use public fields. This may help to illustrate some points concisely, but is not recommended for production code.) Public fields tend to link you to a particular implementation and limit your flexibility in changing your code.**

Class Members

We can use the `static` keyword to create fields and methods that belong to the class, rather than to an instance of the class.

Class Variables

When a number of objects are created from the same class blueprint, they each have their own distinct copies of *instance variables*. Each object has its own values for these variables, stored in different memory locations.

Sometimes, you want to have variables that are common to all objects. This is accomplished with the `static` modifier. Fields that have the `static` modifier in their declaration are called *static fields* or *class variables*. They are associated with the class, rather than with any object. Every instance of the class shares a class variable, which is in one fixed location in memory. Any object can change the value of a class variable, but class variables can also be manipulated without creating an instance of the class.

Class variables are referenced by the class name itself.

Note: You can also refer to static fields with an object reference like
`myBike.numberOfBicycles`

but this is discouraged because it does not make it clear that they are class variables.

Class Methods

The Java programming language supports static methods as well as static variables. Static methods, which have the `static` modifier in their declarations, should be invoked with the class name, without the need for creating an instance of the class, as in

```
ClassName.methodName(args)
```

You can also refer to static methods with an object reference like
`instanceName.methodName(args)`

but this is discouraged because it does not make it clear that they are class methods.

A common use for static methods is to access static fields.

Not all combinations of instance and class variables and methods are allowed:

- **Instance methods can access instance variables and instance methods directly.**
- **Instance methods can access class variables and class methods directly.**
- **Class methods can access class variables and class methods directly.**
- **Class methods *cannot* access instance variables or instance methods directly—they must use an object reference. Also, class methods cannot use the `this` keyword as there is no instance for `this` to refer to.**

Constants

The `static` modifier, in combination with the `final` modifier, is also used to define constants. The `final` modifier indicates that the value of this field cannot change.

For example, the following variable declaration defines a constant named `PI`, whose value is an approximation of pi (the ratio of the circumference of a circle to its diameter):

```
static final double PI = 3.141592653589793;
```

Constants defined in this way cannot be reassigned, and it is a compile-time error if your program tries to do so. By convention, the names of constant values are spelled in uppercase letters. If the name is composed of more than one word, the words are separated by an underscore (_).

Note: If a primitive type or a string is defined as a constant and the value is known at compile time, the compiler replaces the constant name everywhere in the code with its value. This is called a *compile-time constant*. If the value of the constant in the outside world changes (for example, if it is legislated that pi actually should be 3.975), you will need to recompile any classes that use this constant to get the current value.

Initializing Fields

As you have seen, you can often provide an initial value for a field in its declaration:

```
public class BedAndBreakfast {

    // initialize to 10
    public static int capacity = 10;

    // initialize to false
    private boolean full = false;
}
```

This works well when the initialization value is available and the initialization can be put on one line. However, this form of initialization has limitations because of its simplicity. If initialization requires some logic (for example, error handling

or a `for` loop to fill a complex array), simple assignment is inadequate. Instance variables can be initialized in constructors, where error handling or other logic can be used. To provide the same capability for class variables, the Java programming language includes *static initialization blocks*.

Note: It is not necessary to declare fields at the beginning of the class definition, although this is the most common practice. It is only necessary that they be declared and initialized before they are used.

Static Initialization Blocks

A *static initialization block* is a normal block of code enclosed in braces, `{ }`, and preceded by the `static` keyword. Here is an example:

```
static {
    // whatever code is needed for initialization goes here
}
```

A class can have any number of static initialization blocks, and they can appear anywhere in the class body. The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code.

There is an alternative to static blocks — you can write a private static method:

```
class Whatever {
    public static varType myVar = initializeClassVariable();

    private static varType initializeClassVariable() {

        // initialization code goes here
    }
}
```

The advantage of private static methods is that they can be reused later if you need to reinitialize the class variable.

Initializing Instance Members

Normally, you would put code to initialize an instance variable in a constructor. There are two alternatives to using a constructor to initialize instance variables: initializer blocks and final methods.

Initializer blocks for instance variables look just like static initializer blocks, but without the `static` keyword:

```
{
    // whatever code is needed for initialization goes here
}
```

The Java compiler copies initializer blocks into every constructor. Therefore, this approach can be used to share a block of code between multiple constructors.

A *final method* cannot be overridden in a subclass. This is discussed in the lesson on interfaces and inheritance. Here is an example of using a final method for initializing an instance variable:

```
class Whatever {
    private varType myVar = initializeInstanceVariable();

    protected final varType initializeInstanceVariable() {

        // initialization code goes here
    }
```

```
}
```

This is especially useful if subclasses might want to reuse the initialization method. The method is final because calling non-final methods during instance initialization can cause problems.

Garbage Collection and Method finalize

Every class in Java has the methods of class Object (package java.lang), one of which is the finalize method. This method is rarely used because it can cause performance problems and there's some uncertainty as to whether it will get called. Nevertheless, because finalize is part of every class.
The complete details of the finalize method are beyond the scope of this text, and most programmers should not use it.
Every object uses system resources, such as memory. We need a disciplined way to give resources back to the system when they're no longer needed; otherwise, "resource leaks" might occur that would prevent them from being reused by your program or possibly by other programs. The JVM performs automatic garbage collection to reclaim the memory occupied by objects that are no longer used. When there are no more references to an object, the object is eligible to be collected. This typically occurs when the JVM executes its garbage collector. So, memory leaks that are common in other languages like C and C++ (because memory is not automatically reclaimed in those languages) are less likely in Java, but some can still happen in subtle ways. Other types of resource leaks can occur.
For example, an application may open a file on disk to modify its contents. If it does not close the file, the application must terminate before any other application can use it.
The finalize method is called by the garbage collector to perform termination housekeeping on an object just before the garbage collector reclaims the object's memory. Method finalize does not take parameters and has return type void. A problem with method finalize is that the garbage collector is not guaranteed to execute at a specified time. In fact, the garbage collector may never execute before a program terminates. Thus, it's unclear whether, or when, method finalize will be called. For this reason, most programmers should avoid method finalize.
Note:
A class that uses system resources, such as files on disk, should provide a method that programmers can call to release resources when they're no longer needed in a program. Many Java API classes provide close or dispose methods for this purpose. For example, class Scanner has a close method.

Final Keyword

You can declare some or all of a class's methods final. You use the final keyword in a method declaration to indicate that the method cannot be overridden by subclasses. The Object class does this—a number of its methods are final.

You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object. For example, you might want to make the getFirstPlayer method in this ChessAlgorithm class final:

```
class ChessAlgorithm {
    enum ChessPlayer { WHITE, BLACK }
    ...
    final ChessPlayer getFirstPlayer() {
        return ChessPlayer.WHITE;
    }
    ...
}
```

Methods called from constructors should generally be declared final. If a constructor calls a non-final method, a subclass may redefine that method with surprising or undesirable results.

Note that you can also declare an entire class final. A class that is declared final cannot be subclassed. This is particularly useful, for example, when creating an immutable class like the String class.

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. **Variable: Stop value change**
2. **Method: Stop method overriding**
3. **Class: Stop inheritance**

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only.

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//end of class
```

Output: Compile Time Error

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}

   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

Output: Compile Time Error

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
final class Bike{}

class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}
```

```
  public static void main(String args[]){
  Honda1 honda= new Honda();
  honda.run();
  }
}
```

Output: Compile Time Error

Notes:

- **final method is inherited but you cannot override it. For Example:**

```
class Bike{
  final void run(){System.out.println("running...");}
}
class Honda2 extends Bike{
   public static void main(String args[]){
    new Honda2().run();
   }
}
```

Output: running...

- **blank or uninitialized final variable: A final variable that is not initialized at the time of declaration is known as blank final variable.**

  If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. It can be initialized only in constructor.

```
class Bike10{
  final int speedlimit;//blank final variable

  Bike10(){
  speedlimit=70;
  System.out.println(speedlimit);
  }


  public static void main(String args[]){
    new Bike10();
 }
}
```

Output:70

- **static blank final variable**

  A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

  Example of static blank final variable

```
class A{
  static final int data;//static blank final variable
  static{ data=50;}
  public static void main(String args[]){
    System.out.println(A.data);
```

```
 }
}
```

- **final parameter**

  If you declare any parameter as final, you cannot change the value of it.

```
class Bike11{
  int cube(final int n){
   n=n+2;//can't be changed as n is final
   n*n*n;
  }
  public static void main(String args[]){
    Bike11 b=new Bike11();
    b.cube(5);
 }
}
```

  Output: Compile Time Error

- **We cannot declare a constructor final, because constructor is never inherited.**
- **A final local variable is a constant.**

Here is what the different pieces of System.out.println() actually look like:

```
//the System class belongs to java.lang package
class System {
  public static final PrintStream out;
  //...
}

//the Prinstream class belongs to java.io package
class PrintStream{
public void println();
//...
}
```

Enum Types

An *enum type* is a special data type that enables for a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it. Common examples include compass directions (values of NORTH, SOUTH, EAST, and WEST) and the days of the week.

Because they are constants, the names of an enum type's fields are in uppercase letters.

In the Java programming language, you define an enum type by using the `enum` keyword. For example, you would specify a days-of-the-week enum type as:

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

***All* enums implicitly extend `java.lang.Enum`. Java programming language enum types are much more powerful than their counterparts in other languages. The `enum` declaration defines a *class* (called an *enum type*). The enum class**

body can include methods and other fields. The compiler automatically adds some special methods when it creates an enum. For example, they have a static `values` method that returns an array containing all of the values of the enum in the order they are declared. This method is commonly used in combination with the for-each construct to iterate over the values of an enum type. For example, this code from the `Planet` class example below iterates over all the planets in the solar system.

```
for (Planet p : Planet.values()) {
    System.out.printf("Your weight on %s is %f%n",
                      p, p.surfaceWeight(mass));
```

In the following example, `Planet` is an enum type that represents the planets in the solar system. They are defined with constant mass and radius properties.

Each enum constant is declared with values for the mass and radius parameters. These values are passed to the constructor when the constant is created. Java requires that the constants be defined first, prior to any fields or methods. Also, when there are fields and methods, the list of enum constants must end with a semicolon.

Note: The constructor for an enum type must be package-private or private access. It automatically creates the constants that are defined at the beginning of the enum body. You cannot invoke an enum constructor yourself.

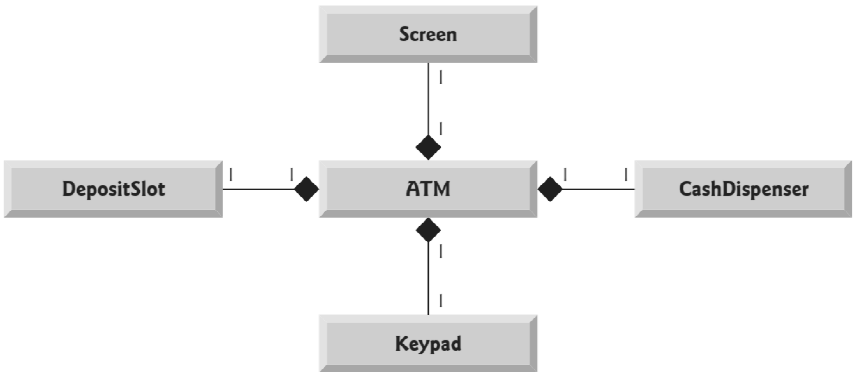
```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7);

    private final double mass;   // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    private double mass() { return mass; }
    private double radius() { return radius; }

    // universal gravitational constant  (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Usage: java Planet <earth_weight>");
            System.exit(-1);
        }
        double earthWeight = Double.parseDouble(args[0]);
        double mass = earthWeight/EARTH.surfaceGravity();
        for (Planet p : Planet.values())
           System.out.printf("Your weight on %s is %f%n",
                             p, p.surfaceWeight(mass));
    }
}
```

Composition and aggregation

In UML class diagrams, the solid diamonds attached to a class's association lines indicate that the class has a composition relationship with counterpart class. Composition implies a whole/part relationship. The class that has the composition symbol (the solid diamond) on its end of the association line is the whole, and the classes on the other end of the association lines are the parts.
According to the UML specification (www.omg.org/technology/documents/formal/uml.htm), composition relationships have the following properties:

1. **Only one class in the relationship can represent the whole (i.e., the diamond can be placed on only one end of the association line). For example, either the screen is part of the ATM or the ATM is part of the screen, but the screen and the ATM cannot both represent the whole in the relationship.**
2. **The parts in the composition relationship exist only as long as the whole does, and the whole is responsible for the creation and destruction of its parts. For example, the act of constructing an ATM includes manufacturing its parts. Also, if the ATM is destroyed, its screen, keypad, cash dispenser and deposit slot are also destroyed.**
3. **A part may belong to only one whole at a time, although it may be removed and attached to another whole, which then assumes responsibility for the part.**

The solid diamonds in our class diagrams indicate composition relationships that fulfill these properties. If a has-a relationship does not satisfy one or more of these criteria, the UML specifies that hollow diamonds be attached to the ends of association lines to indicate aggregation—a weaker form of composition. For example, a personal computer and a computer monitor participate in an aggregation relationship—the computer has a monitor, but the two parts can exist independently, and the same monitor can be attached to multiple computers at once, thus violating composition's second and third properties.



Class Arrays and Class ArrayList

For your convenience, Java SE provides several methods for performing array manipulations (common tasks, such as copying, sorting and searching arrays) in the `java.util.Arrays` class. Class Arrays helps you avoid reinventing the wheel by providing static methods for common array manipulations. These methods include sort for sorting an array (i.e., arranging elements into increasing order), binarySearch for searching an array (i.e., determining whether an array contains a specific value and, if so, where the value is located), equals for comparing arrays and fill for placing values into an array. These methods are overloaded for primitive-type arrays and for arrays of objects.

`static int binarySearch(double[] a, double key)`	**This method searches the specified array of doubles for the specified value using the binary search algorithm.**
`static int binarySearch(double[] a, int fromIndex, int toIndex, double key)`	**This method searches a range of the specified array of doubles for the specified value using the binary search algorithm.**

`static int binarySearch(Object[] a, int fromIndex, int toIndex, Object key)`	**This method searches a range of the specified array for the specified object using the binary search algorithm.**
`static int binarySearch(Object[] a, Object key)`	**This method searches the specified array for the specified object using the binary search algorithm.**
`static double[] copyOf(double[] original, int newLength)`	**This method copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.**
`static <T> T[] copyOf(T[] original, int newLength)`	**This method copies the specified array, truncating or padding with nulls (if necessary) so the copy has the specified length.**
`static boolean equals(double[] a, double[] a2)`	**This method returns true if the two specified arrays of doubles are equal to one another.**
`static boolean equals(Object[] a, Object[] a2)`	**This method returns true if the two specified arrays of Objects are equal to one another.**
`static void fill(double[] a, double val)`	**This method assigns the specified double value to each element of the specified array of doubles.**
`static void fill(double[] a, int fromIndex, int toIndex, double val)`	**This method assigns the specified double value to each element of the specified range of the specified array of doubles.**
`static void fill(Object[] a, int fromIndex, int toIndex, Object val)`	**This method assigns the specified Object reference to each element of the specified range of the specified array of Objects.**
`static void fill(Object[] a, Object val)`	**This method assigns the specified Object reference to each element of the specified array of Objects.**
`static void sort(double[] a)`	**This method sorts the specified array of doubles into ascending numerical order.**
`static void sort(double[] a, int fromIndex, int toIndex)`	**This method sorts the specified range of the specified array of doubles into ascending numerical order.**
`static String toString(double[] a)`	**This method returns a string representation of the contents of the specified array of doubles.**
`static String toString(Object[] a)`	**This method returns a string representation of the contents of the specified array of objects.**

Note: Class System's static arraycopy method for copy arrays.

ArrayList

The Java API provides several predefined data structures, called collections, used to store groups of related objects. These classes provide efficient methods that organize, store and retrieve your data without requiring knowledge of how the data is being stored. This reduces application-development time.

You've used arrays to store sequences of objects. Arrays do not automatically change their size at execution time to accommodate additional elements. The collection class ArrayList<T> (from package java.util) provides a convenient

solution to this problem, it can dynamically change its size to accommodate more elements. The T (by convention) is a placeholder, when declaring a new ArrayList, replace it with the type of elements that you want the ArrayList to hold. This is similar to specifying the type when declaring an array, except that only nonprimitive types can be used with these collection classes. For example,

```
ArrayList< String > list;
```

Declares list as an ArrayList collection that can store only Strings. Classes with this kind of placeholder that can be used with any type are called generic classes.

Some common methods of class ArrayList<T>:

`add(E e)`	**Adds an element to the end of the ArrayList.**
`add(int index, E e)`	**Adds an element to the end of the ArrayList.**
`clear()`	**Removes all the elements from the ArrayList.**
`contains(object o)`	**Returns true if the ArrayList contains the specified element; otherwise, returns false.**
`get(int index)`	**Returns the element at the specified index.**
`set(int index, E element)`	**Replaces the element at the specified position in the ArrayList with the specified element.**
`indexOf(object o)`	**Returns the index of the first occurrence of the specified element in the ArrayList, or -1 if the list does not contain the element.**
`lastIndexOf(Object o)`	**Returns the index of the last occurrence of the specified element in the ArrayList, or -1 if the list does not contain the element.**
`isEmpty()`	**Returns true if the list contains no elements.**
`remove(object o)`	**Removes the first occurrence of the specified element from the ArrayList (if it is present) and returns a boolean value.**
`remove(int index)`	**Removes the element at the specified position in the ArrayList and returns that element.**
`size()`	**Returns the number of elements stored in the ArrayList.**
`trimToSize()`	**Trims the capacity of the ArrayList to current number of elements.**

Chapter 4 Inheritance

Definitions: **A class that is derived from another class is called a subclass (also a derived class, extended class, or child class). The class from which the subclass is derived is called a superclass (also a base class or a parent class).**

Excepting Object, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object.

Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, Object. Such a class is said to be descended from all the classes in the inheritance chain stretching back to Object.

What You Can Do in a Subclass

A subclass inherits all of the public and protected members of its parent, no matter what package the subclass is in. If the subclass is in the same package as its parent, it also inherits the package-private members of the parent. You can use the inherited members as is, replace them, hide them, or supplement them with new members:

- **The inherited fields can be used directly, just like any other fields.**
- **You can declare a field in the subclass with the same name as the one in the superclass, thus hiding it (not recommended).**
- **You can declare new fields in the subclass that are not in the superclass.**
- **The inherited methods can be used directly as they are.**
- **You can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it.**
- **You can write a new static method in the subclass that has the same signature as the one in the superclass, thus hiding it.**
- **You can declare new methods in the subclass that are not in the superclass.**
- **You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.**

A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

A nested class has access to all the private members of its enclosing class—both fields and methods. Therefore, a public or protected nested class inherited by a subclass has indirect access to all of the private members of the superclass.

Note: You can make a logical test as to the type of a particular object using the instanceof operator. This can save you from a runtime error owing to an improper cast. For example:

```
if (obj instanceof MountainBike) {
    MountainBike myBike = (MountainBike)obj;
}
```

Here the instanceof operator verifies that obj refers to a MountainBike so that we can make the cast with knowledge that there will be no runtime exception thrown.

Multiple Inheritance of State, Implementation, and Type

One significant difference between classes and interfaces is that classes can have fields whereas interfaces cannot. In addition, you can instantiate a class to create an object, which you cannot do with interfaces. As explained in the

section What Is an Object?, an object stores its state in fields, which are defined in classes. One reason why the Java programming language does not permit you to extend more than one class is to avoid the issues of *multiple inheritance of state*, which is the ability to inherit fields from multiple classes. For example, suppose that you are able to define a new class that extends multiple classes. When you create an object by instantiating that class, that object will inherit fields from all of the class's superclasses. What if methods or constructors from different superclasses instantiate the same field? Which method or constructor will take precedence? Because interfaces do not contain fields, you do not have to worry about problems that result from multiple inheritance of state.

Multiple inheritance of implementation **is the ability to inherit method definitions from multiple classes. Problems arise with this type of multiple inheritance, such as name conflicts and ambiguity. When compilers of programming languages that support this type of multiple inheritance encounter superclasses that contain methods with the same name, they sometimes cannot determine which member or method to access or invoke. In addition, a programmer can unwittingly introduce a name conflict by adding a new method to a superclass. Default methods introduce one form of multiple inheritance of implementation. A class can implement more than one interface, which can contain default methods that have the same name. The Java compiler provides some rules to determine which default method a particular class uses.**

The Java programming language supports *multiple inheritance of type*, which is the ability of a class to implement more than one interface. An object can have multiple types: the type of its own class and the types of all the interfaces that the class implements. This means that if a variable is declared to be the type of an interface, then its value can reference any object that is instantiated from any class that implements the interface. This is discussed in the section Using an Interface as a Type.

As with multiple inheritance of implementation, a class can inherit different implementations of a method defined (as default or static) in the interfaces that it extends. In this case, the compiler or the user must decide which one to use.

Overriding and Hiding Methods

Instance Methods

An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass overrides the superclass's method.

The ability of a subclass to override a method allows a class to inherit from a superclass whose behavior is "close enough" and then to modify behavior as needed. The overriding method has the same name, number and type of parameters, and return type as the method that it overrides. An overriding method can also return a subtype of the type returned by the overridden method. This subtype is called a covariant return type.

When overriding a method, you might want to use the @Override annotation that instructs the compiler that you intend to override a method in the superclass. If, for some reason, the compiler detects that the method does not exist in one of the superclasses, then it will generate an error.

Static Methods

If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass hides the one in the superclass.

The distinction between hiding a static method and overriding an instance method has important implications:

- **The version of the overridden instance method that gets invoked is the one in the subclass.**
- **The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.**

Modifiers

The access specifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the superclass can be made public, but not private, in the subclass.

You will get a compile-time error if you attempt to change an instance method in the superclass to a static method in the subclass, and vice versa.

Summary

The following table summarizes what happens when you define a method with the same signature as a method in a superclass.

Defining a Method with the Same Signature as a Superclass's Method		
	Superclass Instance Method	**Superclass Static Method**
Subclass Instance Method	Overrides	Generates a compile-time error
Subclass Static Method	Generates a compile-time error	Hides

Note: **In a subclass, you can overload the methods inherited from the superclass. Such overloaded methods neither hide nor override the superclass instance methods—they are new methods, unique to the subclass.**

Rules for method overriding

- **A method can only be written in Subclass, not in same class.**
- **The argument list should be exactly the same as that of the overridden method.**
- **The return type should be the same or a subtype of the return type declared in the original overridden method in the super class.**
- **The access level cannot be more restrictive than the overridden method's access level. For example: if the super class method is declared public then the overridding method in the sub class cannot be either private or protected.**
- **Instance methods can be overridden only if they are inherited by the subclass.**
- **A method declared final cannot be overridden.**
- **A method declared static cannot be overridden but can be re-declared.**
- **If a method cannot be inherited then it cannot be overridden.**
- **A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.**
- **A subclass in a different package can only override the non-final methods declared public or protected.**
- **An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.**
  Exception thrown by the method in the derived class should be compatible with the overridden method.
- **This can happen if the base method is declared to throw no exceptions at all, or e.g. java.io.IOException (which is a subclass of java.lang.Exception your method is trying to throw here). Clients of the base class/interface expect its instances to adhere to the contract declared by the base method, so throwing Exception from an implementation of that method would break the contract.**
- **Constructors cannot be overridden.**

Polymorphism

The dictionary definition of *polymorphism* refers to a principle in biology in which an organism or species can have many different forms or stages. This principle can also be applied to object-oriented programming and languages like the Java language. Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

Polymorphism can be demonstrated with a minor modification to the Bicycle class. For example, a printDescription method could be added to the class that displays all the data currently stored in an instance.

```
public class TestBikes {
  public static void main(String[] args){
    Bicycle bike01, bike02, bike03;

    bike01 = new Bicycle(20, 10, 1);
    bike02 = new MountainBike(20, 10, 5, "Dual");
    bike03 = new RoadBike(40, 20, 8, 23);

    bike01.printDescription();
    bike02.printDescription();
    bike03.printDescription();
  }
}
```

The Java virtual machine (JVM) calls the appropriate method for the object that is referred to in each variable. It does not call the method that is defined by the variable's type. This behavior is referred to as *virtual method invocation* and demonstrates an aspect of the important polymorphism features in the Java language.

Hiding Fields

Within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if their types are different. Within the subclass, the field in the superclass cannot be referenced by its simple name. Instead, the field must be accessed through super.

Using the Keyword super

Accessing Superclass Members

If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword super. You can also use super to refer to a hidden field (although hiding fields is discouraged).

Subclass Constructors

Invocation of a superclass constructor must be the first line in the subclass constructor.

The syntax for calling a superclass constructor is

```
super();
```
or:
```
super(parameter list);
```

With super(), the superclass no-argument constructor is called. With super(parameter list), the superclass constructor with a matching parameter list is called.

Note: **If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the super class does not have a no-argument constructor, you will get a compile-time error. Object does have such a constructor, so if Object is the only superclass, there is no problem.**

If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that there will be a whole chain of constructors called, all the way back to the constructor of Object. In fact, this is the case. It is called constructor chaining, and you need to be aware of it when there is a long line of class descent.

Object as a Superclass

The Object class, in the java.lang package, sits at the top of the class hierarchy tree. Every class is a descendant, direct or indirect, of the Object class. Every class you use or write inherits the instance methods of Object. You need not use any of these methods, but, if you choose to do so, you may need to override them with code that is specific to your class. The methods inherited from Object that are discussed in this section are:

- `protected Object clone() throws CloneNotSupportedException`
  Creates and returns a copy of this object.
- `public boolean equals(Object obj)`
  Indicates whether some other object is "equal to" this one.
- `protected void finalize() throws Throwable`
  Called by the garbage collector on an object when garbage collection determines that there are no more references to the object
- `public final Class getClass()`
  Returns the runtime class of an object.
- `public int hashCode()`
  Returns a hash code value for the object.
- `public String toString()`
  Returns a string representation of the object.

The notify, notifyAll, and wait methods of Object all play a part in synchronizing the activities of independently running threads in a program, which is discussed in a later lesson and won't be covered here. There are five of these methods:

- `public final void notify()`
- `public final void notifyAll()`
- `public final void wait()`
- `public final void wait(long timeout)`
- `public final void wait(long timeout, int nanos)`

The clone() Method

If a class, or one of its superclasses, implements the Cloneable interface, you can use the clone() method to create a copy from an existing object. To create a clone, you write:

```
aCloneableObject.clone();
```

Object's implementation of this method checks to see whether the object on which clone() was invoked implements the Cloneable interface. If the object does not, the method throws a CloneNotSupportedException exception. For the moment, you need to know that clone() must be declared as

```
protected Object clone() throws CloneNotSupportedException
```

or:

```
public Object clone() throws CloneNotSupportedException
```

if you are going to write a clone() method to override the one in Object.

The equals() Method

```
public class Book {
    ...
```

```
    public boolean equals(Object obj) {
        if (obj instanceof Book)
            return ISBN.equals((Book)obj.getISBN());
        else
            return false;
    }
}
```

You should always override the equals() method if the identity operator is not appropriate for your class.

Note: **If you override equals(), you must override hashCode() as well.**

The finalize() Method

The Object class provides a callback method, finalize(), that may be invoked on an object when it becomes garbage. Object's implementation of finalize() does nothing—you can override finalize() to do cleanup, such as freeing resources.

The finalize() method *may be* called automatically by the system, but when it is called, or even if it is called, is uncertain. Therefore, you should not rely on this method to do your cleanup for you. For example, if you don't close file descriptors in your code after performing I/O and you expect finalize() to close them for you, you may run out of file descriptors.

The getClass() Method

You cannot override getClass.

The getClass() method returns a Class object, which has methods you can use to get information about the class, such as its name (getSimpleName()), its superclass (getSuperclass()), and the interfaces it implements (getInterfaces()). For example, the following method gets and displays the class name of an object:

```
void printClassName(Object obj) {
    System.out.println("The object's" + " class is " +
        obj.getClass().getSimpleName());
}
```

The Class class, in the java.lang package, has a large number of methods (more than 50). For example, you can test to see if the class is an annotation (isAnnotation()), an interface (isInterface()), or an enumeration (isEnum()). You can see what the object's fields are (getFields()) or what its methods are (getMethods()), and so on.

The hashCode() Method

The value returned by hashCode() is the object's hash code, which is the object's memory address in hexadecimal.

By definition, if two objects are equal, their hash code must also be equal. If you override the equals() method, you change the way two objects are equated and Object's implementation of hashCode() is no longer valid. Therefore, if you override the equals() method, you must also override the hashCode() method as well.

The toString() Method

You should always consider overriding the toString() method in your classes.

The Object's toString() method returns a String representation of the object, which is very useful for debugging. The String representation for an object depends entirely on the object, which is why you need to override toString() in your classes.

You can use toString() along with System.out.println() to display a text representation of an object

Final Classes and Methods

You can declare some or all of a class's methods final. You use the final keyword in a method declaration to indicate that the method cannot be overridden by subclasses. The Object class does this—a number of its methods are final.

You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object.

Methods called from constructors should generally be declared final. If a constructor calls a non-final method, a subclass may redefine that method with surprising or undesirable results.

Note that you can also declare an entire class final. A class that is declared final cannot be subclassed. This is particularly useful, for example, when creating an immutable class like the String class.

Abstract Classes and Methods

An abstract class is a class that is declared abstract; it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

If a class includes abstract methods, then the class itself must be declared abstract, as in:

```
public abstract class GraphicObject {
   // declare fields
   // declare nonabstract methods
   abstract void draw();
}
```

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.

Class Members

An abstract class may have static fields and static methods. You can use these static members with a class reference (for example, AbstractClass.staticMethod()) as you would with any other class.

Chapter 5 Interfaces

There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a "contract" that spells out how their software interacts. Each group should be able to write their code without any knowledge of how the other group's code is written. Generally speaking, interfaces are such contracts.

In the Java programming language, an interface is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods. Interfaces cannot be instantiated—they can only be implemented by classes or extended by other interfaces.

Note that the method signatures have no braces and are terminated with a semicolon.

To use an interface, you write a class that implements the interface. When an instantiable class implements an interface, it provides a method body for each of the methods declared in the interface.

If you do not specify that the interface is public, then your interface is accessible only to classes defined in the same package as the interface.

The interface body can contain abstract methods, default methods, and static methods. An abstract method within an interface is followed by a semicolon, but no braces (an abstract method does not contain an implementation). Default methods are defined with the default modifier, and static methods with the static keyword. All abstract, default, and static methods in an interface are implicitly public, so you can omit the public modifier.

In addition, an interface can contain constant declarations. All constant values defined in an interface are implicitly public, static, and final. Once again, you can omit these modifiers.

When you define a new interface, you are defining a new reference data type. You can use interface names anywhere you can use any other data type name. If you define a reference variable whose type is an interface, any object you assign to it must be an instance of a class that implements the interface.

Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

You specify that a method definition in an interface is a default method with the default keyword at the beginning of the method signature. All method declarations in an interface, including default methods, are implicitly public, so you can omit the public modifier.

When you extend an interface that contains a default method, you can do the following:

- **Not mention the default method at all, which lets your extended interface inherit the default method.**
- **Redeclare the default method, which makes it abstract.**
- **Redefine the default method, which overrides it.**

```
import java.time.*;

public interface TimeClient {
    void setTime(int hour, int minute, int second);
    void setDate(int day, int month, int year);
    void setDateAndTime(int day, int month, int year,
                        int hour, int minute, int second);
    LocalDateTime getLocalDateTime();
```

```
    static ZoneId getZoneId (String zoneString) {
        try {
            return ZoneId.of(zoneString);
        } catch (DateTimeException e) {
            System.err.println("Invalid time zone: " + zoneString +
                "; using default time zone instead.");
            return ZoneId.systemDefault();
        }
    }

    default ZonedDateTime getZonedDateTime(String zoneString) {
        return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));
    }
}
```

In addition to default methods, you can define static methods in interfaces. (A static method is a method that is associated with the class in which it is defined rather than with any object. Every instance of the class shares its static methods.) This makes it easier for you to organize helper methods in your libraries; you can keep static methods specific to an interface in the same interface rather than in a separate class.

Integrating Default Methods into Existing Libraries

Default methods enable you to add new functionality to existing interfaces and ensure binary compatibility with code written for older versions of those interfaces. In particular, default methods enable you to add methods that accept lambda expressions as parameters to existing interfaces.

Overriding and Hiding Interface Methods

Default methods and abstract methods in interfaces are inherited like instance methods. However, when the supertypes of a class or interface provide multiple default methods with the same signature, the Java compiler follows inheritance rules to resolve the name conflict. These rules are driven by the following two principles:

- **Instance methods are preferred over interface default methods.**
- **Methods that are already overridden by other candidates are ignored. This circumstance can arise when supertypes share a common ancestor.**

If two or more independently defined default methods conflict, or a default method conflicts with an abstract method, then the Java compiler produces a compiler error. You must explicitly override the supertype methods.

Consider the example about computer-controlled cars that can now fly. You have two interfaces (OperateCar and FlyCar) that provide default implementations for the same method, (startEngine):

```
public interface OperateCar {
    // ...
    default public int startEngine(EncryptedKey key) {
        // Implementation
    }
}
public interface FlyCar {
    // ...
    default public int startEngine(EncryptedKey key) {
        // Implementation
    }
}
```

A class that implements both OperateCar and FlyCar must override the method startEngine. You could invoke any of the of the default implementations with the super keyword.

```
public class FlyingCar implements OperateCar, FlyCar {
    // ...
    public int startEngine(EncryptedKey key) {
        FlyCar.super.startEngine(key);
        OperateCar.super.startEngine(key);
    }
}
```

The name preceding super (in this example, FlyCar or OperateCar) must refer to a direct superinterface that defines or inherits a default for the invoked method. This form of method invocation is not restricted to differentiating between multiple implemented interfaces that contain default methods with the same signature. You can use the super keyword to invoke a default method in both classes and interfaces.

Note: Inherited instance methods from classes can override abstract interface methods

Note: Static methods in interfaces are never inherited.

Note: Methods in an interface that are not declared as default or static are implicitly abstract, so the abstract modifier is not used with interface methods. (It can be used, but it is unnecessary.)

Abstract Classes Compared to Interfaces

Abstract classes are similar to interfaces. You cannot instantiate them, and they may contain a mix of methods declared with or without an implementation. However, with abstract classes, you can declare fields that are not static and final, and define public, protected, and private concrete methods. With interfaces, all fields are automatically public, static, and final, and all methods that you declare or define (as default methods) are public. In addition, you can extend only one class, whether or not it is abstract, whereas you can implement any number of interfaces.

Which should you use, abstract classes or interfaces?

- **Consider using abstract classes if any of these statements apply to your situation:**
  - **You want to share code among several closely related classes.**
  - **You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).**
  - **You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.**
- **Consider using interfaces if any of these statements apply to your situation:**
  - **You expect that unrelated classes would implement your interface. For example, the interfaces Comparable and Cloneable are implemented by many unrelated classes.**
  - **You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.**
  - **You want to take advantage of multiple inheritance of type.**

When an Abstract Class Implements an Interface

In the section on Interfaces, it was noted that a class that implements an interface must implement all of the interface's methods. It is possible, however, to define a class that does not implement all of the interface's methods, provided that the class is declared to be abstract. For example,

```
abstract class X implements Y {
  // implements all but one method of Y
}

class XX extends X {
  // implements the remaining method in Y
}
```

In this case, class X must be abstract because it does not fully implement Y, but class XX does, in fact, implement Y.

Chapter 6 Linked Lists

This chapter shows how to build dynamic data structures that grow and shrink at execution time. Linked lists are collections of data items "linked up in a chain"; insertions and deletions can be made anywhere in a linked list. Stacks are important in compilers and operating systems; insertions and deletions are made only at one end of a stack; its top. Queues represent waiting lines; insertions are made at the back (also referred to as the tail) of a queue and deletions are made from the front (also referred to as the head). Binary trees facilitate high-speed searching and sorting of data, eliminating duplicate data items efficiently, representing file-system directories, compiling expressions into machine language and many other interesting applications. We use classes, inheritance and composition to create and package them for reusability and maintainability.
A self-referential class contains an instance variable that refers to another object of the same class type. For example, the ListNode class declaration has two private instance variables, data (of Object type) and ListNode variable nextNode. Variable nextNode references a ListNode object, an object of the same class being declared here, hence the term "self-referential class." Field nextNode is a link, it "links" an object of type ListNode to another object of the same type. Type ListNode also has five methods: a constructor that receives a value to initialize data, a setData method to set the value of data, a getData method to return the value of data, a setNext method to set the value of nextNode and a getNext method to return a reference to the next node.
Programs can link self-referential objects together to form such useful data structures as lists, queues, stacks and trees.

Dynamic Memory Allocation

Creating and maintaining dynamic data structures requires dynamic memory allocation, allowing a program to obtain more memory space at execution time to hold new nodes and to release space no longer needed. Remember that Java programs do not explicitly release dynamically allocated memory. Rather, Java performs automatic garbage collection of objects that are no longer referenced in a program.
The limit for dynamic memory allocation can be as large as the amount of available physical memory in the computer or the amount of available disk space in a virtual memory system. Often the limits are much smaller, because the computer's available memory must be shared among many applications.

Linked Lists

A linked list is a linear collection (i.e., a sequence) of self-referential-class objects, called nodes, connected by reference links, hence, the term "linked" list. Typically, a program accesses a linked list via a reference to its first node. The program accesses each subsequent node via the link reference stored in the previous node. By convention, the link reference in the last node of the list is set to null. Data is stored in a linked list dynamically, the program creates each node as necessary. Stacks and queues are also linear data structures and are constrained versions of linked lists. Trees are nonlinear data structures.

Lists of data can be stored in arrays, but linked lists provide several advantages. A linked list is appropriate when the number of data elements to be represented in the data structure is unpredictable. Linked lists are dynamic, so the length of a list can increase or decrease as necessary, whereas the size of a "conventional" Java array cannot be altered, it's fixed when the program creates the array. "Conventional" arrays can become full. Linked lists become full only when the system has insufficient memory to satisfy dynamic storage allocation requests. Package java.util contains class LinkedList for implementing and manipulating linked lists that grow and shrink during program execution.

Note:

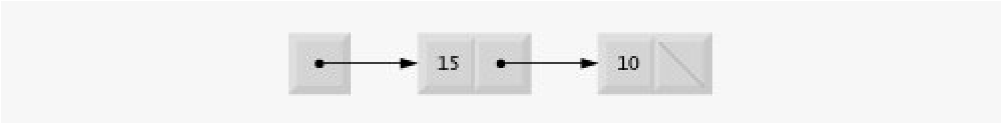
An array can be declared to contain more elements than the number of items expected, but this wastes memory. In these situations, linked lists provide better memory utilization, allowing the program to adapt to storage needs at runtime.
Insertion into a linked list is fast, only two references have to be modified (after locating the insertion point). All existing node objects remain at their current locations in memory. Linked lists can be maintained in sorted order simply by inserting each new element at the proper point in the list. (It does, of course, take time to locate the proper insertion point.) Existing list elements do not need to be moved.
Insertion and deletion in a sorted array can be time consuming, all the elements following the inserted or deleted element must be shifted appropriately.

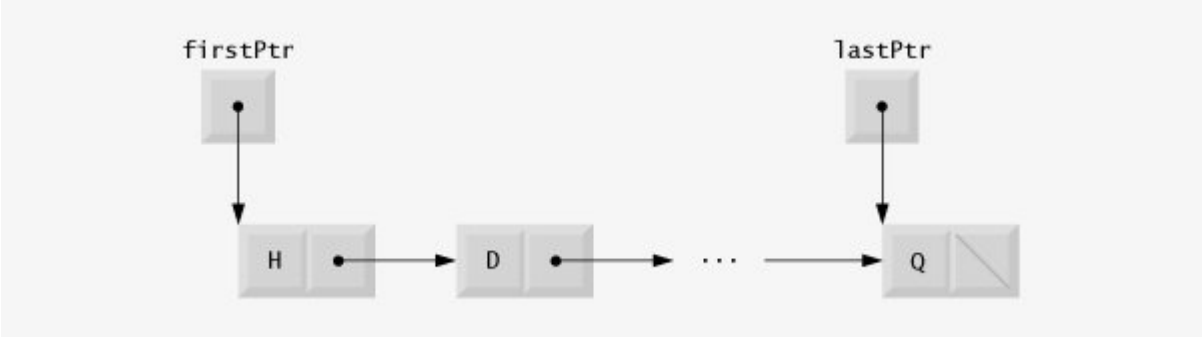
Singly Linked Lists

Linked list nodes normally are not stored contiguously in memory. Rather, they're logically contiguous. In a singly linked list, each node contains one reference to the next node in the list. Often, linked lists are implemented as doubly linked lists, each node contains a reference to the next node in the list and a reference to the preceding one.

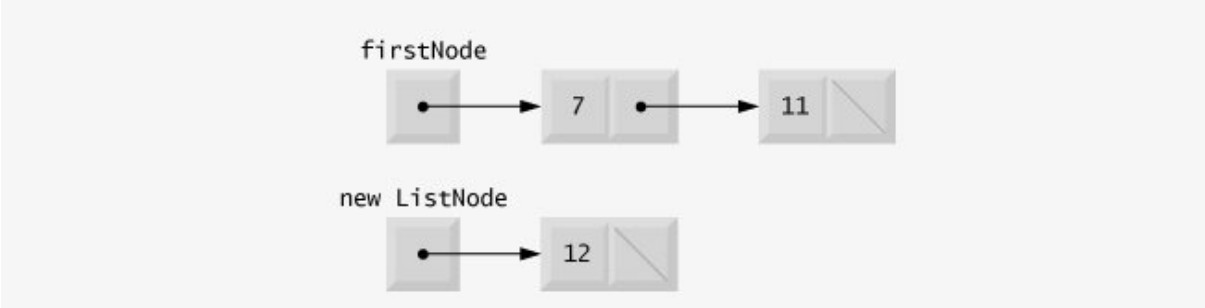
Note:

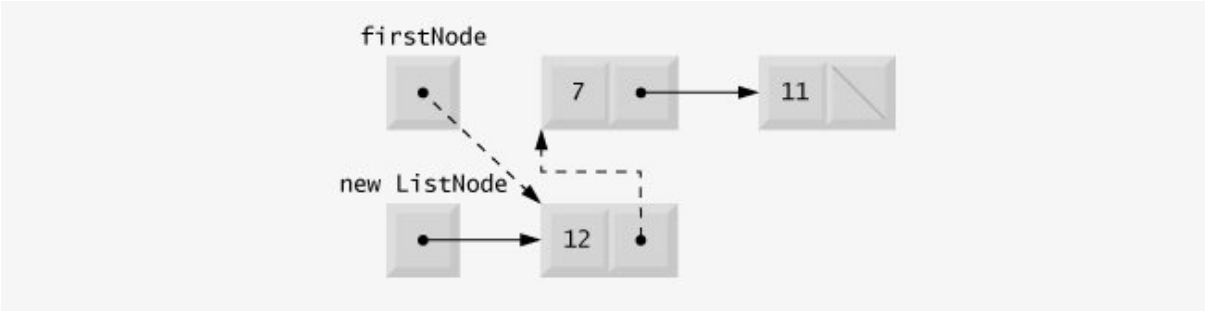
Normally, the elements of an array are contiguous in memory. This allows immediate access to any array element, because its address can be calculated directly as its offset from the beginning of the array. Linked lists do not afford such immediate access, an element can be accessed only by traversing the list from the front (or the back in a doubly linked list).



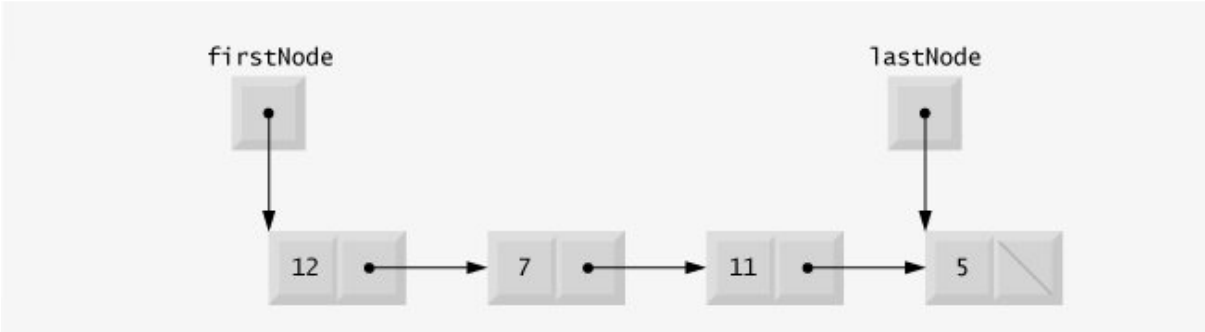


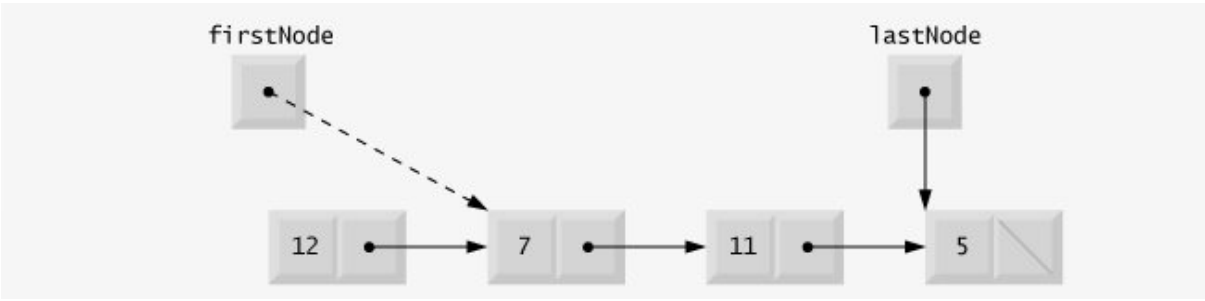
InsertAtFront



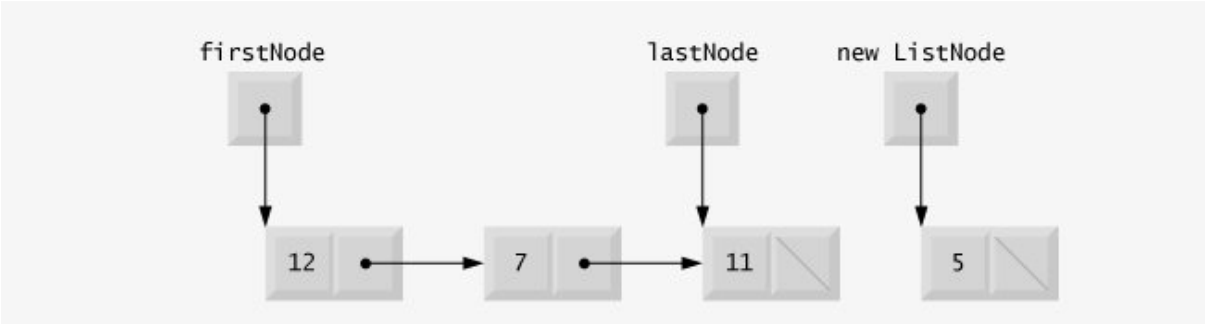


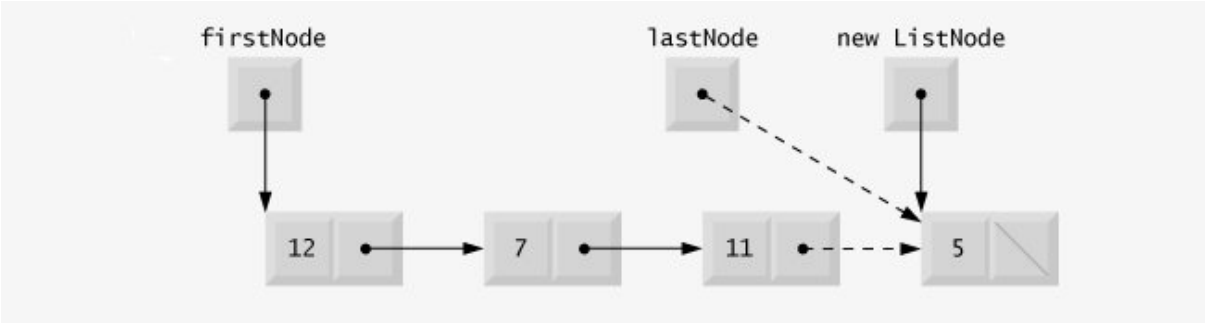
RemoveAtFront



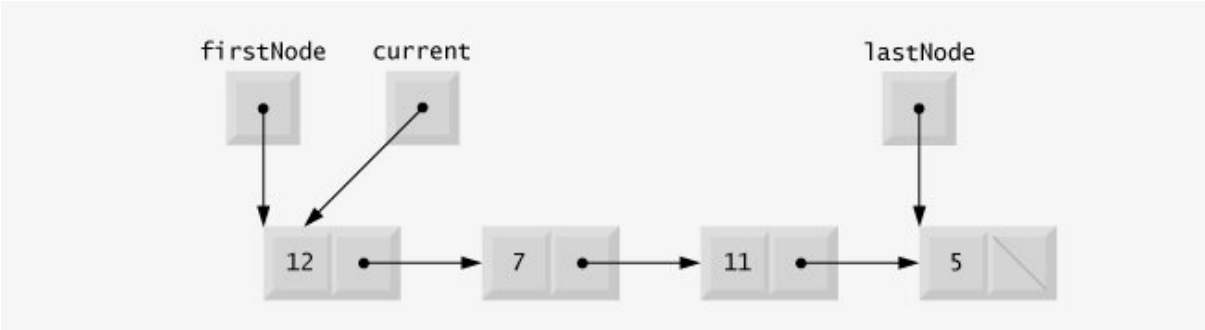


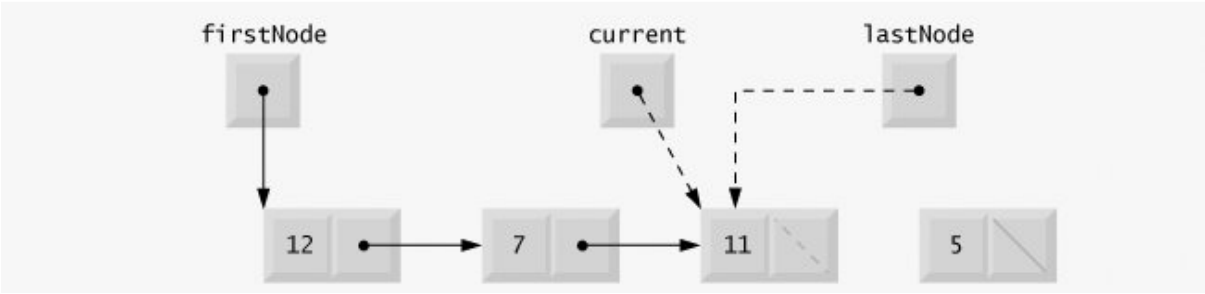
InsertAtEnd





RemoveFromEnd





Ordered Linked Lists

Introduction to Stacks, Queues and Trees

Stacks

A stack is a constrained version of a list, new nodes can be added to and removed from a stack only at the top. For this reason, a stack is referred to as a last-in, first-out (LIFO) data structure. The link member in the bottom node is set to null to indicate the bottom of the stack. A stack is not required to be implemented as a linked list, it can also be implemented using an array.

The primary methods for manipulating a stack are push and pop, which add a new node to the top of the stack and remove a node from the top of the stack, respectively. Method pop also returns the data from the popped node.

Stacks have many interesting applications. For example, when a program calls a method, the called method must know how to return to its caller, so the return address of the calling method is pushed onto the program-execution stack. If a series of method calls occurs, the successive return addresses are pushed onto the stack in last-in, first-out order so

that each method can return to its caller. Stacks support recursive method calls in the same manner as they do conventional nonrecursive method calls.

The program-execution stack also contains the memory for local variables on each invocation of a method during a program's execution. When the method returns to its caller, the memory for that method's local variables is popped off the stack, and those variables are no longer known to the program. If the local variable is a reference and the object to which it referred has no other variables referring to it, the object can be garbage collected.

Compilers use stacks to evaluate arithmetic expressions and generate machine-language code to process them. The package java.util contains class Stack for implementing and manipulating stacks that can grow and shrink during program execution.

Queues

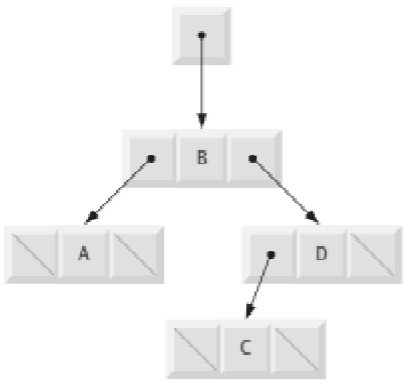
Another commonly used data structure is the queue. A queue is similar to a checkout line in a supermarket, the cashier services the person at the beginning of the line first.Other customers enter the line only at the end and wait for service. Queue nodes are removed only from the head (or front) of the queue and are inserted only at the tail (or end). For this reason, a queue is a first-in, first-out (FIFO) data structure. The insert and remove operations are known as enqueue and dequeue.
Queues have many uses in computer systems. Each CPU in a computer can service only one application at a time. Each application requiring processor time is placed in a queue. The application at the front of the queue is the next to receive service. Each application gradually advances to the front as the applications before it receive service.
Queues are also used to support print spooling. For example, a single printer might be shared by all users of a network. Many users can send print jobs to the printer, even when the printer is already busy. These print jobs are placed in a queue until the printer becomes available. A program called a spooler manages the queue to ensure that, as each print job completes, the next one is sent to the printer.

Information packets also wait in queues in computer networks. Each time a packet arrives at a network node, it must be routed to the next node along the path to the packet's final destination. The routing node routes one packet at a time, so additional packets are enqueued until the router can route them. A file server in a computer network handles file-access requests from many clients throughout the network. Servers have a limited capacity to service requests from clients. When that capacity is exceeded, client requests wait in queues.

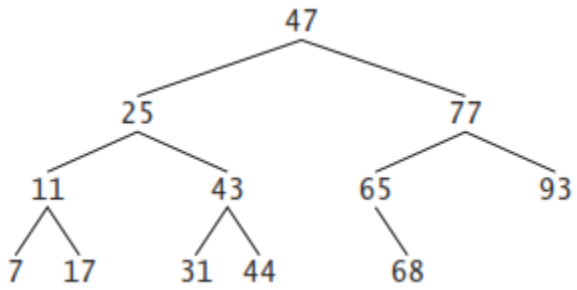
Trees

Lists, stacks and queues are linear data structures (i.e., sequences). A tree is a nonlinear, two dimensional data structure with special properties. Tree nodes contain two or more links.

Binary trees: Trees whose nodes each contain two links (one or both of which may be null). The root node is the first node in a tree. Each link in the root node refers to a child. The left child is the first node in the left subtree (also known as the root node of the left subtree), and the right child is the first node in the right subtree (also known as the root node of the right subtree). The children of a specific node are called siblings. A node with no children is called a leaf node. Computer scientists normally draw trees from the root node down, the opposite of the way most trees grow in nature.



Binary search tree: **A binary search tree (with no duplicate node values) has the characteristic that the values in any left subtree are less than the value in that subtree's parent node, and the values in any right subtree are greater than the value in that subtree's parent node. The shape of the binary search tree that corresponds to a set of data can vary, depending on the order in which the values are inserted into the tree.**



Introduction to Doubly Linked Lists

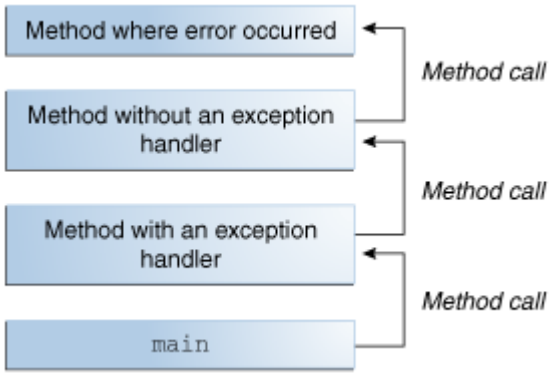
Chapter 7 Exceptions

The term *exception* is shorthand for the phrase "exceptional event."

Definition: **An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.**

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called *throwing an exception*.

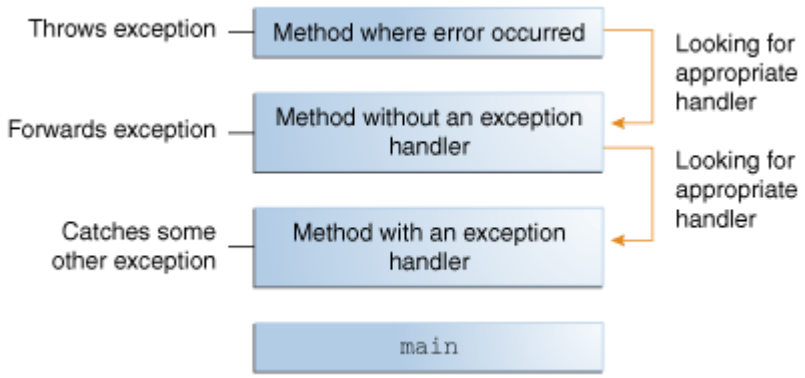
After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the *call stack* (see the next figure).



The call stack.

The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an *exception handler*. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

The exception handler chosen is said to *catch the exception*. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the runtime system (and, consequently, the program) terminates.



Searching the call stack for the exception handler.

Using exceptions to manage errors has some advantages over traditional error-management techniques.

The Catch or Specify Requirement

Valid Java programming language code must honor the Catch or Specify Requirement. This means that code that might throw certain exceptions must be enclosed by either of the following:

- **A try statement that catches the exception. The try must provide a handler for the exception, as described in Catching and Handling Exceptions.**
- **A method that specifies that it can throw the exception. The method must provide a throws clause that lists the exception, as described in Specifying the Exceptions Thrown by a Method.**

The Three Kinds of Exceptions

The first kind of exception is the checked exception. These are exceptional conditions that a well-written application should anticipate and recover from. For example, suppose an application prompts a user for an input file name, then opens the file by passing the name to the constructor for java.io.FileReader. Normally, the user provides the name of an existing, readable file, so the construction of the FileReader object succeeds, and the execution of the application proceeds normally. But sometimes the user supplies the name of a nonexistent file, and the constructor throws java.io.FileNotFoundException. A well-written program will catch this exception and notify the user of the mistake, possibly prompting for a corrected file name.

Checked exceptions are subject to the Catch or Specify Requirement. All exceptions are checked exceptions, except for those indicated by Error, RuntimeException**, and their subclasses.**

The second kind of exception is the error. These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from. For example, suppose that an application successfully opens a file for input, but is unable to read the file because of a hardware or system malfunction. The unsuccessful read will throw java.io.IOError. An application might choose to catch this exception, in order to notify the user of the problem — but it also might make sense for the program to print a stack trace and exit.

Errors are not subject to the Catch or Specify Requirement. Errors are those exceptions indicated by Error and its subclasses.

The third kind of exception is the runtime exception. These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from. These usually indicate programming bugs, such as logic errors or improper use of an API. For example, consider the application described previously that passes a file name to the constructor for FileReader. If a logic error causes a null to be passed to the constructor, the constructor will throw NullPointerException. The application can catch this exception, but it probably makes more sense to eliminate the bug that caused the exception to occur.

Runtime exceptions are not subject to the Catch or Specify Requirement. Runtime exceptions are those indicated by RuntimeException and its subclasses.

Errors and runtime exceptions are collectively known as unchecked exceptions.

Catching and Handling Exceptions

This section describes how to use the three exception handler components — the try, catch, and finally blocks — to write an exception handler. Then, the try-with-resources statement, introduced in Java SE 7, is explained. The try-with-resources statement is particularly suited to situations that use Closeable resources, such as streams.

The first step in constructing an exception handler is to enclose the code that might throw an exception within a try block. In general, a try block looks like the following:

```
private static final int SIZE = 10;

public void writeList() {
    PrintWriter out = null;
    try {
        System.out.println("Entered try statement");
        out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++) {
            out.println("Value at: " + i + " = " + list.get(i));
        }
    }
    catch and finally blocks  . . .
}
```

If an exception occurs within the try block, that exception is handled by an exception handler associated with it. To associate an exception handler with a try block, you must put a catch block after it.

You associate exception handlers with a try block by providing one or more catch blocks directly after the try block. No code can be between the end of the try block and the beginning of the first catch block.

```
try {

} catch (ExceptionType name) {

} catch (ExceptionType name) {

}
```

Each catch block is an exception handler that handles the type of exception indicated by its argument. The argument type, *ExceptionType*, declares the type of exception that the handler can handle and must be the name of a class that inherits from the Throwable class. The handler can refer to the exception with *name*.

The catch block contains code that is executed if and when the exception handler is invoked. The runtime system invokes the exception handler when the handler is the first one in the call stack whose *ExceptionType* matches the type of the exception thrown. The system considers it a match if the thrown object can legally be assigned to the exception handler's argument.

```
try {

} catch (IndexOutOfBoundsException e) {
    System.err.println("IndexOutOfBoundsException: " + e.getMessage());
} catch (IOException e) {
    System.err.println("Caught IOException: " + e.getMessage());
}
```

In Java SE 7 and later, a single catch block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.

In the catch clause, specify the types of exceptions that block can handle, and separate each exception type with a vertical bar (|):

```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

Note: **If a catch block handles more than one exception type, then the catch parameter is implicitly final. In this example, the catch parameter ex is final and therefore you cannot assign any values to it within the catch block.**

The finally block always executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs. But finally is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break. Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.

Note: **If the JVM exits while the try or catch code is being executed, then the finally block may not execute. Likewise, if the thread executing the try or catch code is interrupted or killed, the finally block may not execute even though the application as a whole continues.**

The following finally block for the writeList method cleans up and then closes the PrintWriter.

```
finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close();
    } else {
        System.out.println("PrintWriter not open");
    }
}
```

Important: **The finally block is a key tool for preventing resource leaks. When closing a file or otherwise recovering resources, place the code in a finally block to ensure that resource is always recovered. Consider using the try-with-resources statement in these situations, which automatically releases system resources when no longer needed.**

The try-with-resources statement is a try statement that declares one or more resources. A *resource* is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable, can be used as a resource.

The following example reads the first line from a file. It uses an instance of BufferedReader to read data from the file. BufferedReader is a resource that must be closed after the program is finished with it:

```
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br =
                   new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

Prior to Java SE 7, you can use a finally block to ensure that a resource is closed regardless of whether the try statement completes normally or abruptly. The following example uses a finally block instead of a try-with-resources statement:

```
static String readFirstLineFromFileWithFinallyBlock(String path)
                                                    throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        if (br != null) br.close();
    }
}
```

You may declare one or more resources in a try-with-resources statement. The following example retrieves the names of the files packaged in the zip file zipFileName and creates a text file that contains the names of these files:

```
public static void writeToFileZipFileContents(String zipFileName,
                                               String outputFileName)
                                               throws java.io.IOException {

    java.nio.charset.Charset charset =
         java.nio.charset.StandardCharsets.US_ASCII;
    java.nio.file.Path outputFilePath =
         java.nio.file.Paths.get(outputFileName);

    // Open zip file and create output file with
    // try-with-resources statement

    try (
        java.util.zip.ZipFile zf =
             new java.util.zip.ZipFile(zipFileName);
        java.io.BufferedWriter writer =
            java.nio.file.Files.newBufferedWriter(outputFilePath, charset)
    ) {
        // Enumerate each entry
        for (java.util.Enumeration entries =
                                zf.entries(); entries.hasMoreElements();) {
            // Get the entry name and write it to the output file
            String newLine = System.getProperty("line.separator");
            String zipEntryName =
                 ((java.util.zip.ZipEntry)entries.nextElement()).getName() +
                 newLine;
            writer.write(zipEntryName, 0, zipEntryName.length());
        }
    }
}
```

The following example uses a try-with-resources statement to automatically close a java.sql.Statement object:

```
public static void viewTable(Connection con) throws SQLException {

    String query = "select COF_NAME, SUP_ID, PRICE, SALES, TOTAL from COFFEES";

    try (Statement stmt = con.createStatement()) {
        ResultSet rs = stmt.executeQuery(query);

        while (rs.next()) {
            String coffeeName = rs.getString("COF_NAME");
            int supplierID = rs.getInt("SUP_ID");
            float price = rs.getFloat("PRICE");
            int sales = rs.getInt("SALES");
            int total = rs.getInt("TOTAL");
```

```
            System.out.println(coffeeName + ", " + supplierID + ", " +
                               price + ", " + sales + ", " + total);
        }
    } catch (SQLException e) {
        JDBCTutorialUtilities.printSQLException(e);
    }
}
```

Suppressed Exceptions

An exception can be thrown from the block of code associated with the try-with-resources statement. In the example writeToFileZipFileContents, an exception can be thrown from the try block, and up to two exceptions can be thrown from the try-with-resources statement when it tries to close the ZipFile and BufferedWriter objects. If an exception is thrown from the try block and one or more exceptions are thrown from the try-with-resources statement, then those exceptions thrown from the try-with-resources statement are suppressed, and the exception thrown by the block is the one that is thrown by the writeToFileZipFileContents method. You can retrieve these suppressed exceptions by calling the Throwable.getSuppressed method from the exception thrown by the try block.

Classes That Implement the AutoCloseable or Closeable Interface

See the Javadoc of the AutoCloseable and Closeable interfaces for a list of classes that implement either of these interfaces. The Closeable interface extends the AutoCloseable interface. The close method of the Closeable interface throws exceptions of type IOException while the close method of the AutoCloseable interface throws exceptions of type Exception. Consequently, subclasses of the AutoCloseable interface can override this behavior of the close method to throw specialized exceptions, such as IOException, or no exception at all.

Specifying the Exceptions Thrown by a Method

The previous section showed how to write an exception handler for the writeList method in the ListOfNumbers class. Sometimes, it's appropriate for code to catch exceptions that can occur within it. In other cases, however, it's better to let a method further up the call stack handle the exception. For example, if you were providing the ListOfNumbers class as part of a package of classes, you probably couldn't anticipate the needs of all the users of your package. In this case, it's better to *not* catch the exception and to allow a method further up the call stack to handle it.

If the writeList method doesn't catch the checked exceptions that can occur within it, the writeList method must specify that it can throw these exceptions. Let's modify the original writeList method to specify the exceptions it can throw instead of catching them. To remind you, here's the original version of the writeList method that won't compile.

```
public void writeList() {
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++) {
        out.println("Value at: " + i + " = " + list.get(i));
    }
    out.close();
}
```

To specify that writeList can throw two exceptions, add a throws clause to the method declaration for the writeList method. The throws clause comprises the throws keyword followed by a comma-separated list of all the exceptions thrown by that method. The clause goes after the method name and argument list and before the brace that defines the scope of the method; here's an example.

```
public void writeList() throws IOException, IndexOutOfBoundsException {
```

Remember that IndexOutOfBoundsException is an unchecked exception; including it in the throws clause is not mandatory. You could just write the following.

```
public void writeList() throws IOException {
```

How to Throw Exceptions

Before you can catch an exception, some code somewhere must throw one. Any code can throw an exception: your code, code from a package written by someone else such as the packages that come with the Java platform, or the Java runtime environment. Regardless of what throws the exception, it's always thrown with the throw statement.

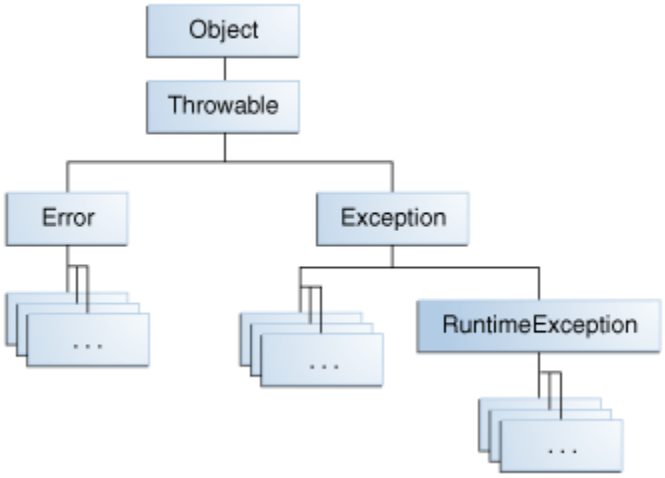
As you have probably noticed, the Java platform provides numerous exception classes. All the classes are descendants of the Throwable class, and all allow programs to differentiate among the various types of exceptions that can occur during the execution of a program.

All methods use the throw **statement to throw an exception. The throw statement requires a single argument: a throwable object. Throwable objects are instances of any subclass of the Throwable class. Here's an example of a throw statement.**

```
throw someThrowableObject;
```

Throwable Class and Its Subclasses

The objects that inherit from the Throwable class include direct descendants (objects that inherit directly from the Throwable class) and indirect descendants (objects that inherit from children or grandchildren of the Throwable class). The figure below illustrates the class hierarchy of the Throwable class and its most significant subclasses. As you can see, Throwable has two direct descendants: Error and Exception.



The Throwable class.

Error Class

When a dynamic linking failure or other hard failure in the Java virtual machine occurs, the virtual machine throws an Error. Simple programs typically do *not* catch or throw Errors.

Exception Class

Most programs throw and catch objects that derive from the Exception class. An Exception indicates that a problem occurred, but it is not a serious system problem. Most programs you write will throw and catch Exceptions as opposed to Errors.

The Java platform defines the many descendants of the Exception class. These descendants indicate various types of exceptions that can occur. For example, IllegalAccessException signals that a particular method could not be found, and NegativeArraySizeException indicates that a program attempted to create an array with a negative size.

One Exception subclass, RuntimeException, is reserved for exceptions that indicate incorrect use of an API. An example of a runtime exception is NullPointerException, which occurs when a method tries to access a member of an object through a null reference.

Chained Exceptions

An application often responds to an exception by throwing another exception. In effect, the first exception causes the second exception. It can be very helpful to know when one exception causes another. Chained Exceptions help the programmer do this.

The following are the methods and constructors in Throwable that support chained exceptions.

```
Throwable getCause()
Throwable initCause(Throwable)
Throwable(String, Throwable)
Throwable(Throwable)
```

The Throwable argument to initCause and the Throwable constructors is the exception that caused the current exception. getCause returns the exception that caused the current exception, and initCause sets the current exception's cause.

The following example shows how to use a chained exception.

```
try {

} catch (IOException e) {
    throw new SampleException("Other IOException", e);
}
```

In this example, when an IOException is caught, a new SampleException exception is created with the original cause attached and the chain of exceptions is thrown up to the next higher level exception handler.

Accessing Stack Trace Information

Definition: **A *stack trace* provides information on the execution history of the current thread and lists the names of the classes and methods that were called at the point when the exception occurred. A stack trace is a useful debugging tool that you'll normally take advantage of when an exception has been thrown.**

The following code shows how to call the getStackTrace method on the exception object.

```
catch (Exception cause) {
    StackTraceElement elements[] = cause.getStackTrace();
    for (int i = 0, n = elements.length; i < n; i++) {
        System.err.println(elements[i].getFileName()
            + ":" + elements[i].getLineNumber()
            + ">> "
            + elements[i].getMethodName() + "()");
    }
}
```

Logging API

The next code snippet logs where an exception occurred from within the catch block. However, rather than manually parsing the stack trace and sending the output to System.err(), it sends the output to a file using the logging facility in the java.util.logging package.

```
try {
    Handler handler = new FileHandler("OutFile.log");
    Logger.getLogger("").addHandler(handler);

} catch (IOException e) {
    Logger logger = Logger.getLogger("package.name");
    StackTraceElement elements[] = e.getStackTrace();
    for (int i = 0, n = elements.length; i < n; i++) {
        logger.log(Level.WARNING, elements[i].getMethodName());
    }
}
```

Creating Exception Classes

When faced with choosing the type of exception to throw, you can either use one written by someone else — the Java platform provides a lot of exception classes you can use — or you can write one of your own. You should write your own exception classes if you answer yes to any of the following questions; otherwise, you can probably use someone else's.

- **Do you need an exception type that isn't represented by those in the Java platform?**
- **Would it help users if they could differentiate your exceptions from those thrown by classes written by other vendors?**
- **Does your code throw more than one related exception?**
- **If you use someone else's exceptions, will users have access to those exceptions? A similar question is, should your package be independent and self-contained?**

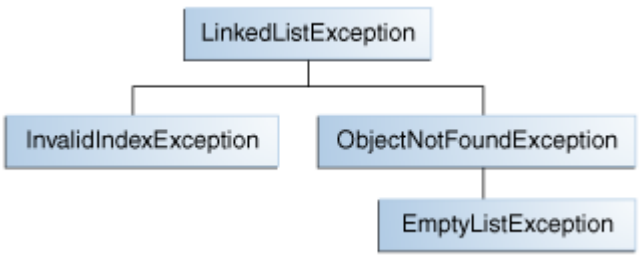
An Example

Suppose you are writing a linked list class. The class supports the following methods, among others:

- objectAt(int n) **— Returns the object in the nth position in the list. Throws an exception if the argument is less than 0 or more than the number of objects currently in the list.**
- firstObject() **— Returns the first object in the list. Throws an exception if the list contains no objects.**
- indexOf(Object o) **— Searches the list for the specified Object and returns its position in the list. Throws an exception if the object passed into the method is not in the list.**

The linked list class can throw multiple exceptions, and it would be convenient to be able to catch all exceptions thrown by the linked list with one exception handler. Also, if you plan to distribute your linked list in a package, all related code should be packaged together. Thus, the linked list should provide its own set of exception classes.

The next figure illustrates one possible class hierarchy for the exceptions thrown by the linked list.



Example exception class hierarchy.

Choosing a Superclass

Any Exception subclass can be used as the parent class of LinkedListException. However, a quick perusal of those subclasses shows that they are inappropriate because they are either too specialized or completely unrelated to LinkedListException. Therefore, the parent class of LinkedListException should be Exception.

Most applets and applications you write will throw objects that are Exceptions. Errors are normally used for serious, hard errors in the system, such as those that prevent the JVM from running.

Note: **For readable code, it's good practice to append the string Exception to the names of all classes that inherit (directly or indirectly) from the Exception class.**

Advantages of Exceptions

Advantage 1: Separating Error-Handling Code from "Regular" Code

Advantage 2: Propagating Errors Up the Call Stack

Advantage 3: Grouping and Differentiating Error Types

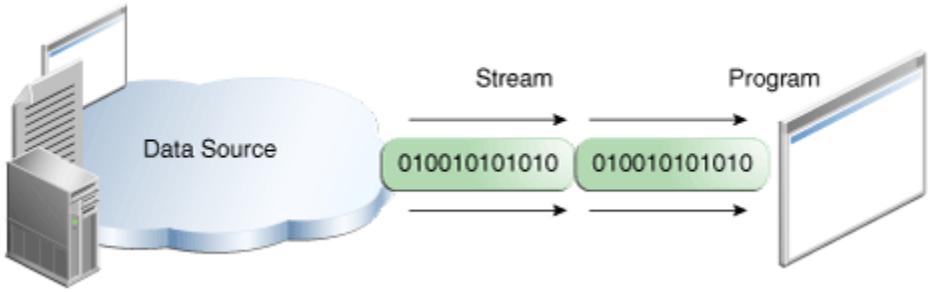
Chapter 8 Files

I/O Streams

An I/O Stream represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.

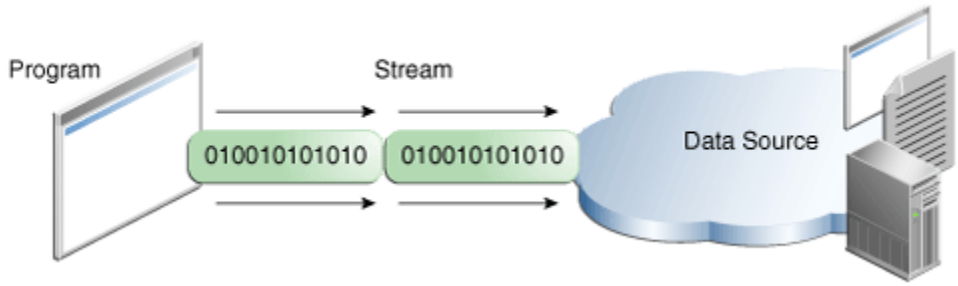
Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Some streams simply pass on data; others manipulate and transform the data in useful ways.

No matter how they work internally, all streams present the same simple model to programs that use them: A stream is a sequence of data. A program uses an input stream to read data from a source, one item at a time:



Reading information into a program.

A program uses an output stream to write data to a destination, one item at time:



Writing information from a program.

In this lesson, we'll see streams that can handle all kinds of data, from primitive values to advanced objects.

The data source and data destination pictured above can be anything that holds, generates, or consumes data. Obviously this includes disk files, but a source or destination can also be another program, a peripheral device, a network socket, or an array.

Byte Streams

Programs use *byte streams* to perform input and output of 8-bit bytes. All byte stream classes are descended from InputStream and OutputStream.

There are many byte stream classes. To demonstrate how byte streams work, we'll focus on the file I/O byte streams, FileInputStream and FileOutputStream. Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed.

Using Byte Streams

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes {
    public static void main(String[] args) throws IOException {

        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("xanadu.txt");
            out = new FileOutputStream("outagain.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Using Character Streams

All character stream classes are descended from Reader and Writer. As with byte streams, there are character stream classes that specialize in file I/O: FileReader and FileWriter. The CopyCharacters example illustrates these classes.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopyCharacters {
    public static void main(String[] args) throws IOException {

        FileReader inputStream = null;
        FileWriter outputStream = null;

        try {
            inputStream = new FileReader("xanadu.txt");
            outputStream = new FileWriter("characteroutput.txt");

            int c;
            while ((c = inputStream.read()) != -1) {
                outputStream.write(c);
            }
        } finally {
```

```
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

CopyCharacters is very similar to CopyBytes. The most important difference is that CopyCharacters uses FileReader and FileWriter for input and output in place of FileInputStream and FileOutputStream. Notice that both CopyBytes and CopyCharacters use an int variable to read to and write from. However, in CopyCharacters, the int variable holds a character value in its last 16 bits; in CopyBytes, the int variable holds a byte value in its last 8 bits.

Character Streams that Use Byte Streams

Character streams are often "wrappers" for byte streams. The character stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes. FileReader, for example, uses FileInputStream, while FileWriter uses FileOutputStream.

There are two general-purpose byte-to-character "bridge" streams: InputStreamReader and OutputStreamWriter. Use them to create character streams when there are no prepackaged character stream classes that meet your needs.

Line-Oriented I/O

Character I/O usually occurs in bigger units than single characters. One common unit is the line: a string of characters with a line terminator at the end. A line terminator can be a carriage-return/line-feed sequence ("\r\n"), a single carriage-return ("\r"), or a single line-feed ("\n"). Supporting all possible line terminators allows programs to read text files created on any of the widely used operating systems.

The CopyLines example invokes BufferedReader.readLine and PrintWriter.println to do input and output one line at a time.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;

public class CopyLines {
    public static void main(String[] args) throws IOException {

        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream = new BufferedReader(new FileReader("xanadu.txt"));
            outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));

            String l;
            while ((l = inputStream.readLine()) != null) {
                outputStream.println(l);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
```

```
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

Invoking readLine returns a line of text with the line. CopyLines outputs each line using println, which appends the line terminator for the current operating system. This might not be the same line terminator that was used in the input file.

Buffered Streams

Most of the examples we've seen so far use unbuffered I/O. This means each read or write request is handled directly by the underlying OS. This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.

To reduce this kind of overhead, the Java platform implements buffered I/O streams. Buffered input streams read data from a memory area known as a buffer; the native input API is called only when the buffer is empty. Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.

A program can convert an unbuffered stream into a buffered stream using the wrapping idiom we've used several times now, where the unbuffered stream object is passed to the constructor for a buffered stream class. Here's how you might modify the constructor invocations in the CopyCharacters example to use buffered I/O:

```
inputStream = new BufferedReader(new FileReader("xanadu.txt"));
outputStream = new BufferedWriter(new FileWriter("characteroutput.txt"));
```

There are four buffered stream classes used to wrap unbuffered streams: BufferedInputStream and BufferedOutputStream create buffered byte streams, while BufferedReader and BufferedWriter create buffered character streams.

Scanning

Objects of type Scanner are useful for breaking down formatted input into tokens and translating individual tokens according to their data type.

Breaking Input into Tokens

By default, a scanner uses white space to separate tokens. (White space characters include blanks, tabs, and line terminators. For the full list, refer to the documentation for Character.isWhitespace.)

```
import java.io.*;
import java.util.Scanner;

public class ScanXan {
    public static void main(String[] args) throws IOException {

        Scanner s = null;

        try {
            s = new Scanner(new BufferedReader(new FileReader("xanadu.txt")));

            while (s.hasNext()) {
                System.out.println(s.next());
            }
        } finally {
            if (s != null) {
```

```
                s.close();
            }
        }
    }
}
```

Translating Individual Tokens

The ScanXan example treats all input tokens as simple String values. Scanner also supports tokens for all of the Java language's primitive types (except for char), as well as BigInteger and BigDecimal. Also, numeric values can use thousands separators. Thus, in a US locale, Scanner correctly reads the string "32,767" as representing an integer value.

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.util.Scanner;
import java.util.Locale;

public class ScanSum {
    public static void main(String[] args) throws IOException {

        Scanner s = null;
        double sum = 0;

        try {
            s = new Scanner(new BufferedReader(new FileReader("usnumbers.txt")));
            s.useLocale(Locale.US);

            while (s.hasNext()) {
                if (s.hasNextDouble()) {
                    sum += s.nextDouble();
                } else {
                    s.next();
                }
            }
        } finally {
            s.close();
        }

        System.out.println(sum);
    }
}
```

Formatting

Stream objects that implement formatting are instances of either PrintWriter, a character stream class, or PrintStream, a byte stream class.

Note: **The only PrintStream objects you are likely to need are System.out and System.err. When you need to create a formatted output stream, instantiate PrintWriter, not PrintStream.**

Like all byte and character stream objects, instances of PrintStream and PrintWriter implement a standard set of write methods for simple byte and character output. In addition, both PrintStream and PrintWriter implement the same set of methods for converting internal data into formatted output. Two levels of formatting are provided:

- **print and println format individual values in a standard way.**
- **format formats almost any number of values based on a format string, with many options for precise formatting.**

The print and println Methods

Invoking print or println outputs a single value after converting the value using the appropriate toString method.

The format Method

The format method formats multiple arguments based on a format string. The format string consists of static text embedded with format specifiers; except for the format specifiers, the format string is output unchanged.

```
public class Root2 {
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);

        System.out.format("The square root of %d is %f.%n", i, r);
    }
}
```

Like the three used in this example, all format specifiers begin with a % and end with a 1- or 2-character conversion that specifies the kind of formatted output being generated. The three conversions used here are:

- **d formats an integer value as a decimal value.**
- **f formats a floating point value as a decimal value.**
- **n outputs a platform-specific line terminator.**

Here are some other conversions:

- **x formats an integer as a hexadecimal value.**
- **s formats any value as a string.**
- **tB formats an integer as a locale-specific month name.**

There are many other conversions.

Note: Except for %% and %n, all format specifiers must match an argument. If they don't, an exception is thrown. In the Java programming language, the \n escape always generates the linefeed character (\u000A). Don't use \n unless you specifically want a linefeed character. To get the correct line separator for the local platform, use %n.

In addition to the conversion, a format specifier can contain several additional elements that further customize the formatted output. Here's an example, Format, that uses every possible kind of element.

```
public class Format {
    public static void main(String[] args) {
        System.out.format("%f, %1$+020.10f %n", Math.PI);
    }
}
```

Here's the output:

```
3.141593, +00000003.1415926536
```

The additional elements are all optional. The following figure shows how the longer specifier breaks down into elements.

Elements of a Format Specifier.

The elements must appear in the order shown. Working from the right, the optional elements are:

- **Precision. For floating point values, this is the mathematical precision of the formatted value. For s and other general conversions, this is the maximum width of the formatted value; the value is right-truncated if necessary.**
- **Width. The minimum width of the formatted value; the value is padded if necessary. By default the value is left-padded with blanks.**
- **Flags specify additional formatting options. In the Format example, the + flag specifies that the number should always be formatted with a sign, and the 0 flag specifies that 0 is the padding character. Other flags include - (pad on the right) and , (format number with locale-specific thousands separators). Note that some flags cannot be used with certain other flags or with certain conversions.**
- **The Argument Index allows you to explicitly match a designated argument. You can also specify < to match the same argument as the previous specifier. Thus the example could have said: System.out.format("%f, %<+020.10f %n", Math.PI);**

I/O from the Command Line

A program is often run from the command line and interacts with the user in the command line environment. The Java platform supports this kind of interaction in two ways: through the Standard Streams and through the Console.

Standard Streams

Standard Streams are a feature of many operating systems. By default, they read input from the keyboard and write output to the display. They also support I/O on files and between programs, but that feature is controlled by the command line interpreter, not the program.

The Java platform supports three Standard Streams: *Standard Input*, accessed through System.in; *Standard Output*, accessed through System.out; and *Standard Error*, accessed through System.err. These objects are defined automatically and do not need to be opened. Standard Output and Standard Error are both for output; having error output separately allows the user to divert regular output to a file and still be able to read error messages.

You might expect the Standard Streams to be character streams, but, for historical reasons, they are byte streams. System.out and System.err are defined as PrintStream objects. Although it is technically a byte stream, PrintStream utilizes an internal character stream object to emulate many of the features of character streams.

By contrast, System.in is a byte stream with no character stream features. To use Standard Input as a character stream, wrap System.in in InputStreamReader.

```
InputStreamReader cin = new InputStreamReader(System.in);
```

The Console

A more advanced alternative to the Standard Streams is the Console. This is a single, predefined object of type Console that has most of the features provided by the Standard Streams, and others besides. The Console is

particularly useful for secure password entry. The Console object also provides input and output streams that are true character streams, through its reader and writer methods.

Before a program can use the Console, it must attempt to retrieve the Console object by invoking System.console(). If the Console object is available, this method returns it. If System.console returns NULL, then Console operations are not permitted, either because the OS doesn't support them or because the program was launched in a noninteractive environment.

The Console object supports secure password entry through its readPassword method. This method helps secure password entry in two ways. First, it suppresses echoing, so the password is not visible on the user's screen. Second, readPassword returns a character array, not a String, so the password can be overwritten, removing it from memory as soon as it is no longer needed.

```
import java.io.Console;
import java.util.Arrays;
import java.io.IOException;

public class Password {

    public static void main (String args[]) throws IOException {

        Console c = System.console();
        if (c == null) {
            System.err.println("No console.");
            System.exit(1);
        }

        String login = c.readLine("Enter your login: ");
        char [] oldPassword = c.readPassword("Enter your old password: ");

        if (verify(login, oldPassword)) {
            boolean noMatch;
            do {
                char [] newPassword1 = c.readPassword("Enter your new password: ");
                char [] newPassword2 = c.readPassword("Enter new password again: ");
                noMatch = ! Arrays.equals(newPassword1, newPassword2);
                if (noMatch) {
                    c.format("Passwords don't match. Try again.%n");
                } else {
                    change(login, newPassword1);
                    c.format("Password for %s changed.%n", login);
                }
                Arrays.fill(newPassword1, ' ');
                Arrays.fill(newPassword2, ' ');
            } while (noMatch);
        }

        Arrays.fill(oldPassword, ' ');
    }

    // Dummy change method.
    static boolean verify(String login, char[] password) {
        // This method always returns
        // true in this example.
        // Modify this method to verify
        // password according to your rules.
        return true;
    }

    // Dummy change method.
```

```
    static void change(String login, char[] password) {
        // Modify this method to change
        // password according to your rules.
    }
}
```

The Password class follows these steps:

1. **Attempt to retrieve the Console object. If the object is not available, abort.**
2. **Invoke Console.readLine to prompt for and read the user's login name.**
3. **Invoke Console.readPassword to prompt for and read the user's existing password.**
4. **Invoke verify to confirm that the user is authorized to change the password. (In this example, verify is a dummy method that always returns true.)**
5. **Repeat the following steps until the user enters the same password twice:**
    1. **Invoke Console.readPassword twice to prompt for and read a new password.**
    2. **If the user entered the same password both times, invoke change to change it. (Again, change is a dummy method.)**
    3. **Overwrite both passwords with blanks.**
6. **Overwrite the old password with blanks.**

Data Streams

Data streams support binary I/O of primitive data type values (boolean, char, byte, short, int, long, float, and double) as well as String values. All data streams implement either the DataInput interface or the DataOutput interface. This section focuses on the most widely-used implementations of these interfaces, DataInputStream and DataOutputStream.

The DataStreams example demonstrates data streams by writing out a set of data records, and then reading them in again. Each record consists of three values related to an item on an invoice, as shown in the following table:

Order in record	Data type	Data description	Output Method	Input Method	Sample Value
1	double	Item price	DataOutputStream.writeDouble	DataInputStream.readDouble	19.99
2	int	Unit count	DataOutputStream.writeInt	DataInputStream.readInt	12
3	String	Item description	DataOutputStream.writeUTF	DataInputStream.readUTF	"Java T-Shirt"

Let's examine crucial code in DataStreams. First, the program defines some constants containing the name of the data file and the data that will be written to it:

```
static final String dataFile = "invoicedata";

static final double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
static final int[] units = { 12, 8, 13, 29, 50 };
static final String[] descs = {
    "Java T-shirt",
    "Java Mug",
    "Duke Juggling Dolls",
    "Java Pin",
    "Java Key Chain"
};
```

Then DataStreams opens an output stream. Since a DataOutputStream can only be created as a wrapper for an existing byte stream object, DataStreams provides a buffered file output byte stream.

```
out = new DataOutputStream(new BufferedOutputStream(
              new FileOutputStream(dataFile)));
```

DataStreams writes out the records and closes the output stream.

```
for (int i = 0; i < prices.length; i ++) {
    out.writeDouble(prices[i]);
    out.writeInt(units[i]);
    out.writeUTF(descs[i]);
}
```

The writeUTF method writes out String values in a modified form of UTF-8. This is a variable-width character encoding that only needs a single byte for common Western characters.

Now DataStreams reads the data back in again. First it must provide an input stream, and variables to hold the input data. Like DataOutputStream, DataInputStream must be constructed as a wrapper for a byte stream.

```
in = new DataInputStream(new
           BufferedInputStream(new FileInputStream(dataFile)));

double price;
int unit;
String desc;
double total = 0.0;
```

Now DataStreams can read each record in the stream, reporting on the data it encounters.

```
try {
    while (true) {
        price = in.readDouble();
        unit = in.readInt();
        desc = in.readUTF();
        System.out.format("You ordered %d" + " units of %s at $%.2f%n",
            unit, desc, price);
        total += unit * price;
    }
} catch (EOFException e) {
}
```

Notice that DataStreams detects an end-of-file condition by catching EOFException, instead of testing for an invalid return value. All implementations of DataInput methods use EOFException instead of return values.

Also notice that each specialized write in DataStreams is exactly matched by the corresponding specialized read. It is up to the programmer to make sure that output types and input types are matched in this way: The input stream consists of simple binary data, with nothing to indicate the type of individual values, or where they begin in the stream.

DataStreams uses one very bad programming technique: it uses floating point numbers to represent monetary values. In general, floating point is bad for precise values. It's particularly bad for decimal fractions, because common values (such as 0.1) do not have a binary representation.

The correct type to use for currency values is java.math.BigDecimal. Unfortunately, BigDecimal is an object type, so it won't work with data streams. However, BigDecimal will work with object streams

Object Streams

Just as data streams support I/O of primitive data types, object streams support I/O of objects. Most, but not all, standard classes support serialization of their objects. Those that do implement the marker interface Serializable.

The object stream classes are ObjectInputStream and ObjectOutputStream. These classes implement ObjectInput and ObjectOutput, which are subinterfaces of DataInput and DataOutput. That means that all the primitive data I/O methods

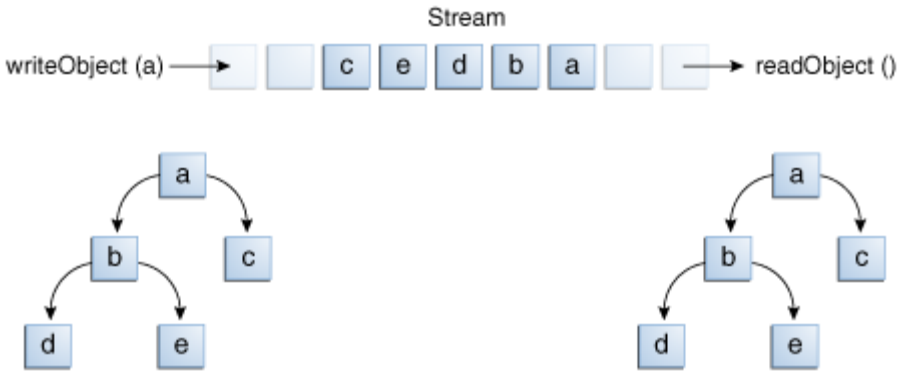
covered in Data Streams are also implemented in object streams. So an object stream can contain a mixture of primitive and object values. The ObjectStreams example illustrates this. ObjectStreams creates the same application as DataStreams, with a couple of changes. First, prices are now BigDecimalobjects, to better represent fractional values. Second, a Calendar object is written to the data file, indicating an invoice date.

If readObject() doesn't return the object type expected, attempting to cast it to the correct type may throw a ClassNotFoundException. In this simple example, that can't happen, so we don't try to catch the exception. Instead, we notify the compiler that we're aware of the issue by adding ClassNotFoundException to the main method's throws clause.

Output and Input of Complex Objects

The writeObject and readObject methods are simple to use, but they contain some very sophisticated object management logic. This isn't important for a class like Calendar, which just encapsulates primitive values. But many objects contain references to other objects. If readObject is to reconstitute an object from a stream, it has to be able to reconstitute all of the objects the original object referred to. These additional objects might have their own references, and so on. In this situation, writeObject traverses the entire web of object references and writes all objects in that web onto the stream. Thus a single invocation of writeObject can cause a large number of objects to be written to the stream.

This is demonstrated in the following figure, where writeObject is invoked to write a single object named a. This object contains references to objects b and c, while b contains references to d and e. Invoking writeobject(a) writes not just a, but all the objects necessary to reconstitute a, so the other four objects in this web are written also. When a is read back by readObject, the other four objects are read back as well, and all the original object references are preserved.



I/O of multiple referred-to objects

You might wonder what happens if two objects on the same stream both contain references to a single object. Will they both refer to a single object when they're read back? The answer is "yes." A stream can only contain one copy of an object, though it can contain any number of references to it. Thus if you explicitly write an object to a stream twice, you're really writing only the reference twice. For example, if the following code writes an object ob twice to a stream:

```
Object ob = new Object();
out.writeObject(ob);
out.writeObject(ob);
```

Each writeObject has to be matched by a readObject, so the code that reads the stream back will look something like this:

```
Object ob1 = in.readObject();
Object ob2 = in.readObject();
```

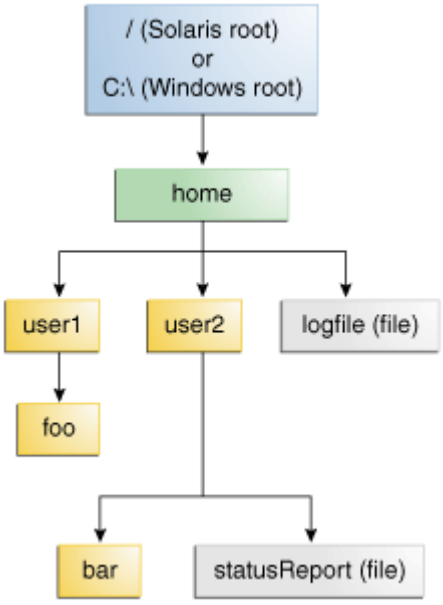
This results in two variables, ob1 and ob2, that are references to a single object.

However, if a single object is written to two different streams, it is effectively duplicated — a single program reading both streams back will see two distinct objects.

What Is a Path?

A file system stores and organizes files on some form of media, generally one or more hard drives, in such a way that they can be easily retrieved. Most file systems in use today store the files in a tree (or *hierarchical*) structure. At the top of the tree is one (or more) root nodes. Under the root node, there are files and directories (*folders* in Microsoft Windows). Each directory can contain files and subdirectories, which in turn can contain files and subdirectories, and so on, potentially to an almost limitless depth.

The following figure shows a sample directory tree containing a single root node. Microsoft Windows supports multiple root nodes. Each root node maps to a volume, such as C:\ or D:\. The Solaris OS supports a single root node, which is denoted by the slash character, /.



Sample Directory Structure

A file is identified by its path through the file system, beginning from the root node. In Microsoft Windows, statusReport is described by the following notation:

```
C:\home\sally\statusReport
```

The character used to separate the directory names (also called the *delimiter*) is specific to the file system: The Solaris OS uses the forward slash (/), and Microsoft Windows uses the backslash slash (\).

Relative or Absolute?

A path is either *relative* or *absolute*. An absolute path always contains the root element and the complete directory list required to locate the file. For example, /home/sally/statusReport is an absolute path. All of the information needed to locate the file is contained in the path string.

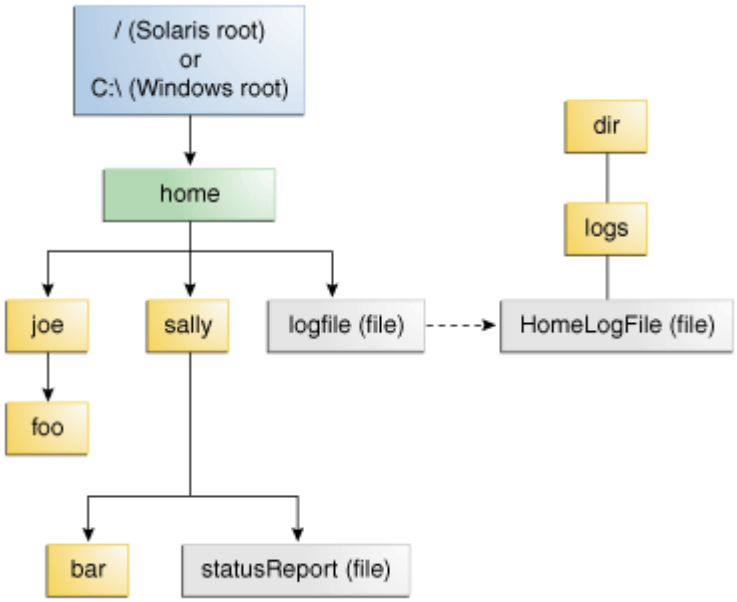
A relative path needs to be combined with another path in order to access a file. For example, joe/foo is a relative path. Without more information, a program cannot reliably locate the joe/foo directory in the file system.

Symbolic Links

File system objects are most typically directories or files. Everyone is familiar with these objects. But some file systems also support the notion of symbolic links. A symbolic link is also referred to as a *symlink* or a *soft link*.

A *symbolic link* is a special file that serves as a reference to another file. For the most part, symbolic links are transparent to applications, and operations on symbolic links are automatically redirected to the target of the link. (The file or directory being pointed to is called the *target* of the link.) Exceptions are when a symbolic link is deleted, or renamed in which case the link itself is deleted, or renamed and not the target of the link.

In the following figure, logFile appears to be a regular file to the user, but it is actually a symbolic link to dir/logs/HomeLogFile. HomeLogFile is the target of the link.



Example of a Symbolic Link.

A symbolic link is usually transparent to the user. Reading or writing to a symbolic link is the same as reading or writing to any other file or directory.

The phrase *resolving a link* means to substitute the actual location in the file system for the symbolic link. In the example, resolving logFile yields dir/logs/HomeLogFile.

The Path Class

The Path class, introduced in the Java SE 7 release, is one of the primary entrypoints of the java.nio.file package. If your application uses file I/O, you will want to learn about the powerful features of this class.

Version Note: **If you have pre-JDK7 code that uses java.io.File, you can still take advantage of the Path class functionality by using the File.toPath method.**

As its name implies, the Path class is a programmatic representation of a path in the file system. A Path object contains the file name and directory list used to construct the path, and is used to examine, locate, and manipulate files.

The file or directory corresponding to the Path might not exist. You can create a Path instance and manipulate it in various ways: you can append to it, extract pieces of it, compare it to another path. At the appropriate time, you can

use the methods in the Files class to check the existence of the file corresponding to the Path, create the file, open it, delete it, change its permissions, and so on.

Creating a Path

A Path instance contains the information used to specify the location of a file or directory. At the time it is defined, a Path is provided with a series of one or more names. A root element or a file name might be included, but neither are required. A Path might consist of just a single directory or file name.

You can easily create a Path object by using one of the following get methods from the Paths (note the plural) helper class:

```
Path p1 = Paths.get("/tmp/foo");
Path p2 = Paths.get(args[0]);
Path p3 = Paths.get(URI.create("file:///Users/joe/FileTest.java"));
```

The Paths.get method is shorthand for the following code:

```
Path p4 = FileSystems.getDefault().getPath("/users/sally");
```

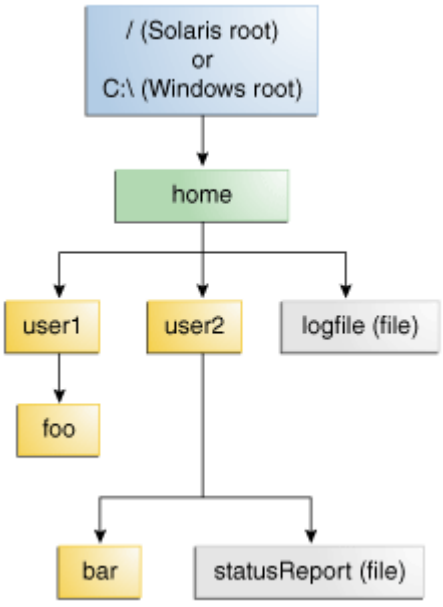
The following example creates /u/joe/logs/foo.log assuming your home directory is /u/joe, or C:\joe\logs\foo.log if you are on Windows.

```
Path p5 = Paths.get(System.getProperty("user.home"),"logs", "foo.log");
```

Retrieving Information about a Path

You can think of the Path as storing these name elements as a sequence. The highest element in the directory structure would be located at index 0. The lowest element in the directory structure would be located at index [n-1], where n is the number of name elements in the Path. Methods are available for retrieving individual elements or a subsequence of the Path using these indexes.

The examples in this lesson use the following directory structure.



Sample Directory Structure

The following code snippet defines a Path instance and then invokes several methods to obtain information about the path:

```
// None of these methods requires that the file corresponding
// to the Path exists.
// Microsoft Windows syntax
Path path = Paths.get("C:\\home\\joe\\foo");

// Solaris syntax
Path path = Paths.get("/home/joe/foo");

System.out.format("toString: %s%n", path.toString());
System.out.format("getFileName: %s%n", path.getFileName());
System.out.format("getName(0): %s%n", path.getName(0));
System.out.format("getNameCount: %d%n", path.getNameCount());
System.out.format("subpath(0,2): %s%n", path.subpath(0,2));
System.out.format("getParent: %s%n", path.getParent());
System.out.format("getRoot: %s%n", path.getRoot());
```

Here is the output for both Windows and the Solaris OS:

Method Invoked	Returns in the Solaris OS	Returns in Microsoft Windows	Comment
`toString`	`/home/joe/foo`	`C:\home\joe\foo`	Returns the string representation of the Path. If the path was created using Filesystems.getDefault().getPath(String) or Paths.get (the latter is a convenience method for getPath), the method performs minor syntactic cleanup. For example, in a UNIX operating system, it will correct the input string //home/joe/foo to /home/joe/foo.
`getFileName`	`foo`	`foo`	Returns the file name or the last element of the sequence of name elements.
`getName(0)`	`home`	`home`	Returns the path element corresponding to the specified index. The 0th element is the path element closest to the root.
`getNameCount`	3	3	Returns the number of elements in the path.
`subpath(0,2)`	`home/joe`	`home\joe`	Returns the subsequence of the Path (not including a root element) as specified by the beginning and ending indexes.
`getParent`	`/home/joe`	`\home\joe`	Returns the path of the parent directory.
`getRoot`	`/`	`C:\`	Returns the root of the path.

The previous example shows the output for an absolute path. In the following example, a relative path is specified:

```
// Solaris syntax
Path path = Paths.get("sally/bar");
or
// Microsoft Windows syntax
Path path = Paths.get("sally\\bar");
```

Here is the output for Windows and the Solaris OS:

Method Invoked	Returns in the Solaris OS	Returns in Microsoft Windows
`toString`	`sally/bar`	`sally\bar`
`getFileName`	`bar`	`bar`
`getName(0)`	`sally`	`sally`
`getNameCount`	2	2
`subpath(0,1)`	`sally`	`sally`

getParent	sally	sally
getRoot	null	null

Removing Redundancies From a Path

Many file systems use "." notation to denote the current directory and ".." to denote the parent directory. You might have a situation where a Path contains redundant directory information. Perhaps a server is configured to save its log files in the "/dir/logs/." directory, and you want to delete the trailing "/." notation from the path.

The following examples both include redundancies:

```
/home/./joe/foo
/home/sally/../joe/foo
```

The normalize method removes any redundant elements, which includes any "." or "*directory*/.." occurrences. Both of the preceding examples normalize to /home/joe/foo.

It is important to note that normalize doesn't check at the file system when it cleans up a path. It is a purely syntactic operation. In the second example, if sally were a symbolic link, removing sally/.. might result in a Path that no longer locates the intended file.

To clean up a path while ensuring that the result locates the correct file, you can use the toRealPath method

Converting a Path

You can use three methods to convert the Path. If you need to convert the path to a string that can be opened from a browser, you can use toUri. For example:

```
Path p1 = Paths.get("/home/logfile");
// Result is file:///home/logfile
System.out.format("%s%n", p1.toUri());
```

The toAbsolutePath method converts a path to an absolute path. If the passed-in path is already absolute, it returns the same Path object. The toAbsolutePath method can be very helpful when processing user-entered file names. For example:

```
public class FileTest {
    public static void main(String[] args) {

        if (args.length < 1) {
            System.out.println("usage: FileTest file");
            System.exit(-1);
        }

        // Converts the input string to a Path object.
        Path inputPath = Paths.get(args[0]);

        // Converts the input Path
        // to an absolute path.
        // Generally, this means prepending
        // the current working
        // directory.  If this example
        // were called like this:
        //     java FileTest foo
        // the getRoot and getParent methods
        // would return null
        // on the original "inputPath"
```

```
        // instance.  Invoking getRoot and
        // getParent on the "fullPath"
        // instance returns expected values.
        Path fullPath = inputPath.toAbsolutePath();
    }
}
```

The toAbsolutePath method converts the user input and returns a Path that returns useful values when queried. The file does not need to exist for this method to work.

The toRealPath method returns the *real* path of an existing file. This method performs several operations in one:

- **If true is passed to this method and the file system supports symbolic links, this method resolves any symbolic links in the path.**
- **If the Path is relative, it returns an absolute path.**
- **If the Path contains any redundant elements, it returns a path with those elements removed.**

This method throws an exception if the file does not exist or cannot be accessed. You can catch the exception when you want to handle any of these cases. For example:

```
try {
    Path fp = path.toRealPath();
} catch (NoSuchFileException x) {
    System.err.format("%s: no such" + " file or directory%n", path);
    // Logic for case when file doesn't exist.
} catch (IOException x) {
    System.err.format("%s%n", x);
    // Logic for other sort of file error.
}
```

Joining Two Paths

You can combine paths by using the resolve method. You pass in a *partial path* , which is a path that does not include a root element, and that partial path is appended to the original path.

For example, consider the following code snippet:

```
// Solaris
Path p1 = Paths.get("/home/joe/foo");
// Result is /home/joe/foo/bar
System.out.format("%s%n", p1.resolve("bar"));

or

// Microsoft Windows
Path p1 = Paths.get("C:\\home\\joe\\foo");
// Result is C:\home\joe\foo\bar
System.out.format("%s%n", p1.resolve("bar"));
```

Passing an absolute path to the resolve method returns the passed-in path:

```
// Result is /home/joe
Paths.get("foo").resolve("/home/joe");
```

Creating a Path Between Two Paths

A common requirement when you are writing file I/O code is the capability to construct a path from one location in the file system to another location. You can meet this using the relativize method. This method constructs a path originating from the original path and ending at the location specified by the passed-in path. The new path is *relative* to the original path.

For example, consider two relative paths defined as joe and sally:

```
Path p1 = Paths.get("joe");
Path p2 = Paths.get("sally");
```

In the absence of any other information, it is assumed that joe and sally are siblings, meaning nodes that reside at the same level in the tree structure. To navigate from joe to sally, you would expect to first navigate one level up to the parent node and then down to sally:

```
// Result is ../sally
Path p1_to_p2 = p1.relativize(p2);
// Result is ../joe
Path p2_to_p1 = p2.relativize(p1);
```

Consider a slightly more complicated example:

```
Path p1 = Paths.get("home");
Path p3 = Paths.get("home/sally/bar");
// Result is sally/bar
Path p1_to_p3 = p1.relativize(p3);
// Result is ../..
Path p3_to_p1 = p3.relativize(p1);
```

In this example, the two paths share the same node, home. To navigate from home to bar, you first navigate one level down to sally and then one more level down to bar. Navigating from bar to home requires moving up two levels.

A relative path cannot be constructed if only one of the paths includes a root element. If both paths include a root element, the capability to construct a relative path is system dependent.

The recursive Copy example uses the relativize and resolve methods.

Comparing Two Paths

The Path class supports equals, enabling you to test two paths for equality. The startsWith and endsWith methods enable you to test whether a path begins or ends with a particular string. These methods are easy to use. For example:

```
Path path = ...;
Path otherPath = ...;
Path beginning = Paths.get("/home");
Path ending = Paths.get("foo");

if (path.equals(otherPath)) {
    // equality logic here
} else if (path.startsWith(beginning)) {
    // path begins with "/home"
} else if (path.endsWith(ending)) {
    // path ends with "foo"
}
```

The Path class implements the Iterable interface. The iterator method returns an object that enables you to iterate over the name elements in the path. The first element returned is that closest to the root in the directory tree. The following code snippet iterates over a path, printing each name element:

```
Path path = ...;
for (Path name: path) {
    System.out.println(name);
}
```

The Path class also implements the Comparable interface. You can compare Path objects by using compareTo which is useful for sorting.

You can also put Path objects into a Collection. See the Collections trail for more information about this powerful feature.

When you want to verify that two Path objects locate the same file, you can use the isSameFile method.

File Operations (Files class)

The Files class is the other primary entrypoint of the java.nio.file package. This class offers a rich set of static methods for reading, writing, and manipulating files and directories. The Files methods work on instances of Path objects.

Releasing System Resources

Many of the resources that are used in this API, such as streams or channels, implement or extend the java.io.Closeable interface. A requirement of a Closeable resource is that the close method must be invoked to release the resource when no longer required. Neglecting to close a resource can have a negative implication on an application's performance. The try-with-resources statement, described in the next section, handles this step for you.

Catching Exceptions

With file I/O, unexpected conditions are a fact of life: a file exists (or doesn't exist) when expected, the program doesn't have access to the file system, the default file system implementation does not support a particular function, and so on. Numerous errors can be encountered.

All methods that access the file system can throw an IOException. It is best practice to catch these exceptions by embedding these methods into a try-with-resources statement, introduced in the Java SE 7 release. The try-with-resources statement has the advantage that the compiler automatically generates the code to close the resource(s) when no longer required. The following code shows how this might look:

```
Charset charset = Charset.forName("US-ASCII");
String s = ...;
try (BufferedWriter writer = Files.newBufferedWriter(file, charset)) {
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

Alternatively, you can embed the file I/O methods in a try block and then catch any exceptions in a catch block. If your code has opened any streams or channels, you should close them in a finally block. The previous example would look something like the following using the try-catch-finally approach:

```
Charset charset = Charset.forName("US-ASCII");
String s = ...;
BufferedWriter writer = null;
try {
    writer = Files.newBufferedWriter(file, charset);
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
} finally {
```

```
    if (writer != null) writer.close();
}
```

In addition to IOException, many specific exceptions extend FileSystemException. This class has some useful methods that return the file involved (getFile), the detailed message string (getMessage), the reason why the file system operation failed (getReason), and the "other" file involved, if any (getOtherFile).

The following code snippet shows how the getFile method might be used:

```
try (...) {
    ...
} catch (NoSuchFileException x) {
    System.err.format("%s does not exist\n", x.getFile());
}
```

For purposes of clarity, the file I/O examples in this lesson may not show exception handling, but your code should always include it.

Varargs

Several Files methods accept an arbitrary number of arguments when flags are specified. For example, in the following method signature, the ellipses notation after the CopyOption argument indicates that the method accepts a variable number of arguments, or *varargs*, as they are typically called:

```
Path Files.move(Path, Path, CopyOption...)
```

When a method accepts a varargs argument, you can pass it a comma-separated list of values or an array (CopyOption[]) of values.

In the move example, the method can be invoked as follows:

```
import static java.nio.file.StandardCopyOption.*;

Path source = ...;
Path target = ...;
Files.move(source,
           target,
           REPLACE_EXISTING,
           ATOMIC_MOVE);
```

Atomic Operations

Several Files methods, such as move, can perform certain operations atomically in some file systems.

An *atomic file operation* is an operation that cannot be interrupted or "partially" performed. Either the entire operation is performed or the operation fails. This is important when you have multiple processes operating on the same area of the file system, and you need to guarantee that each process accesses a complete file.

Method Chaining

Many of the file I/O methods support the concept of *method chaining*.

You first invoke a method that returns an object. You then immediately invoke a method on *that* object, which returns yet another object, and so on. Many of the I/O examples use the following technique:

```
String value = Charset.defaultCharset().decode(buf).toString();
UserPrincipal group =
    file.getFileSystem().getUserPrincipalLookupService().
         lookupPrincipalByName("me");
```

This technique produces compact code and enables you to avoid declaring temporary variables that you don't need.

What Is a Glob?

Two methods in the Files class accept a glob argument, but what is a *glob*?

You can use glob syntax to specify pattern-matching behavior.

A glob pattern is specified as a string and is matched against other strings, such as directory or file names. Glob syntax follows several simple rules:

- **An asterisk, *, matches any number of characters (including none).**
- **Two asterisks, **, works like * but crosses directory boundaries. This syntax is generally used for matching complete paths.**
- **A question mark, ?, matches exactly one character.**
- **Braces specify a collection of subpatterns. For example:**
  - **{sun,moon,stars} matches "sun", "moon", or "stars".**
  - **{temp*,tmp*} matches all strings beginning with "temp" or "tmp".**
- **Square brackets convey a set of single characters or, when the hyphen character (-) is used, a range of characters. For example:**
  - **[aeiou] matches any lowercase vowel.**
  - **[0-9] matches any digit.**
  - **[A-Z] matches any uppercase letter.**
  - **[a-z,A-Z] matches any uppercase or lowercase letter.**

  Within the square brackets, *, ?, and \ match themselves.

- **All other characters match themselves.**
- **To match *, ?, or the other special characters, you can escape them by using the backslash character, \. For example: \\ matches a single backslash, and \? matches the question mark.**

Here are some examples of glob syntax:

- ***.html – Matches all strings that end in *.html***
- **??? – Matches all strings with exactly three letters or digits**
- ***[0-9]* – Matches all strings containing a numeric value**
- ***.{htm,html,pdf} – Matches any string ending with *.htm*, *.html* or *.pdf***
- **a?*.java – Matches any string beginning with a, followed by at least one letter or digit, and ending with *.java***
- **{foo*,*[0-9]*} – Matches any string beginning with *foo* or any string containing a numeric value**

Note: **If you are typing the glob pattern at the keyboard and it contains one of the special characters, you must put the pattern in quotes ("*"), use the backslash (*), or use whatever escape mechanism is supported at the command line.**

Verifying the Existence of a File or Directory

The methods in the Path class are syntactic, meaning that they operate on the Path instance. But eventually you must access the file system to verify that a particular Path exists, or does not exist. You can do so with the exists(Path, LinkOption...) and the notExists(Path, LinkOption...) methods. Note that !Files.exists(path) is not equivalent to Files.notExists(path). When you are testing a file's existence, three results are possible:

- **The file is verified to exist.**
- **The file is verified to not exist.**
- **The file's status is unknown. This result can occur when the program does not have access to the file.**

If both exists and notExists return false, the existence of the file cannot be verified.

Checking File Accessibility

To verify that the program can access a file as needed, you can use the isReadable(Path), isWritable(Path), and isExecutable(Path) methods.

The following code snippet verifies that a particular file exists and that the program has the ability to execute the file.

```
Path file = ...;
boolean isRegularExecutableFile = Files.isRegularFile(file) &
         Files.isReadable(file) & Files.isExecutable(file);
```

Note: **Once any of these methods completes, there is no guarantee that the file can be accessed. A common security flaw in many applications is to perform a check and then access the file.**

Checking Whether Two Paths Locate the Same File

When you have a file system that uses symbolic links, it is possible to have two different paths that locate the same file. The isSameFile(Path, Path) method compares two paths to determine if they locate the same file on the file system. For example:

```
Path p1 = ...;
Path p2 = ...;

if (Files.isSameFile(p1, p2)) {
    // Logic when the paths locate the same file
}
```

Deleting a File or Directory

You can delete files, directories or links. With symbolic links, the link is deleted and not the target of the link. With directories, the directory must be empty, or the deletion fails.

The Files class provides two deletion methods.

The delete(Path) method deletes the file or throws an exception if the deletion fails. For example, if the file does not exist a NoSuchFileException is thrown. You can catch the exception to determine why the delete failed as follows:

```
try {
    Files.delete(path);
} catch (NoSuchFileException x) {
    System.err.format("%s: no such" + " file or directory%n", path);
} catch (DirectoryNotEmptyException x) {
    System.err.format("%s not empty%n", path);
} catch (IOException x) {
    // File permission problems are caught here.
    System.err.println(x);
}
```

The deleteIfExists(Path) method also deletes the file, but if the file does not exist, no exception is thrown. Failing silently is useful when you have multiple threads deleting files and you don't want to throw an exception just because one thread did so first.

Copying a File or Directory

You can copy a file or directory by using the copy(Path, Path, CopyOption...) method. The copy fails if the target file exists, unless the REPLACE_EXISTING option is specified.

Directories can be copied. However, files inside the directory are not copied, so the new directory is empty even when the original directory contains files.

When copying a symbolic link, the target of the link is copied. If you want to copy the link itself, and not the contents of the link, specify either the NOFOLLOW_LINKS or REPLACE_EXISTING option.

This method takes a varargs argument. The following StandardCopyOption and LinkOption enums are supported:

- **REPLACE_EXISTING – Performs the copy even when the target file already exists. If the target is a symbolic link, the link itself is copied (and not the target of the link). If the target is a non-empty directory, the copy fails with the FileAlreadyExistsException exception.**
- **COPY_ATTRIBUTES – Copies the file attributes associated with the file to the target file. The exact file attributes supported are file system and platform dependent, but last-modified-time is supported across platforms and is copied to the target file.**
- **NOFOLLOW_LINKS – Indicates that symbolic links should not be followed. If the file to be copied is a symbolic link, the link is copied (and not the target of the link).**

The following shows how to use the copy method:

```
import static java.nio.file.StandardCopyOption.*;
...
Files.copy(source, target, REPLACE_EXISTING);
```

In addition to file copy, the Files class also defines methods that may be used to copy between a file and a stream. The copy(InputStream, Path, CopyOptions...) method may be used to copy all bytes from an input stream to a file. The copy(Path, OutputStream) method may be used to copy all bytes from a file to an output stream.

The Copy example uses the copy and Files.walkFileTree methods to support a recursive copy.

Moving a File or Directory

You can move a file or directory by using the move(Path, Path, CopyOption...) method. The move fails if the target file exists, unless the REPLACE_EXISTING option is specified.

Empty directories can be moved. If the directory is not empty, the move is allowed when the directory can be moved without moving the contents of that directory. On UNIX systems, moving a directory within the same partition generally consists of renaming the directory. In that situation, this method works even when the directory contains files.

This method takes a varargs argument – the following StandardCopyOption enums are supported:

- **REPLACE_EXISTING – Performs the move even when the target file already exists. If the target is a symbolic link, the symbolic link is replaced but what it points to is not affected.**
- **ATOMIC_MOVE – Performs the move as an atomic file operation. If the file system does not support an atomic move, an exception is thrown. With an ATOMIC_MOVE you can move a file into a directory and be guaranteed that any process watching the directory accesses a complete file.**

The following shows how to use the move method:

```
import static java.nio.file.StandardCopyOption.*;
...
Files.move(source, target, REPLACE_EXISTING);
```

Though you can implement the move method on a single directory as shown, the method is most often used with the file tree recursion mechanism.

Managing Metadata (File and File Store Attributes)

The definition of *metadata* is "data about other data." With a file system, the data is contained in its files and directories, and the metadata tracks information about each of these objects: Is it a regular file, a directory, or a link? What is its size, creation date, last modified date, file owner, group owner, and access permissions?

A file system's metadata is typically referred to as its *file attributes*. The Files class includes methods that can be used to obtain a single attribute of a file, or to set an attribute.

Methods	Comment
`size(Path)`	Returns the size of the specified file in bytes.
`isDirectory(Path, LinkOption)`	Returns true if the specified Path locates a file that is a directory.
`isRegularFile(Path, LinkOption...)`	Returns true if the specified Path locates a file that is a regular file.
`isSymbolicLink(Path)`	Returns true if the specified Path locates a file that is a symbolic link.
`isHidden(Path)`	Returns true if the specified Path locates a file that is considered hidden by the file system.
`getLastModifiedTime(Path, LinkOption...)` `setLastModifiedTime(Path, FileTime)`	Returns or sets the specified file's last modified time.
`getOwner(Path, LinkOption...)` `setOwner(Path, UserPrincipal)`	Returns or sets the owner of the file.
`getPosixFilePermissions(Path, LinkOption...)` `setPosixFilePermissions(Path, Set<PosixFilePermission>)`	Returns or sets a file's POSIX file permissions.
`getAttribute(Path, String, LinkOption...)` `setAttribute(Path, String, Object, LinkOption...)`	Returns or sets the value of a file attribute.

If a program needs multiple file attributes around the same time, it can be inefficient to use methods that retrieve a single attribute. Repeatedly accessing the file system to retrieve a single attribute can adversely affect performance. For this reason, the Files class provides two readAttributes methods to fetch a file's attributes in one bulk operation.

Method	Comment
`readAttributes(Path, String, LinkOption...)`	Reads a file's attributes as a bulk operation. The String parameter identifies the attributes to be read.
`readAttributes(Path, Class<A>, LinkOption...)`	Reads a file's attributes as a bulk operation. The Class<A> parameter is the type of attributes requested and the method returns an object of that class.

Before showing examples of the readAttributes methods, it should be mentioned that different file systems have different notions about which attributes should be tracked. For this reason, related file attributes are grouped together into views. A *view* maps to a particular file system implementation, such as POSIX or DOS, or to a common functionality, such as file ownership.

The supported views are as follows:

- **BasicFileAttributeView – Provides a view of basic attributes that are required to be supported by all file system implementations.**
- **DosFileAttributeView – Extends the basic attribute view with the standard four bits supported on file systems that support the DOS attributes.**
- **PosixFileAttributeView – Extends the basic attribute view with attributes supported on file systems that support the POSIX family of standards, such as UNIX. These attributes include file owner, group owner, and the nine related access permissions.**
- **FileOwnerAttributeView – Supported by any file system implementation that supports the concept of a file owner.**
- **AclFileAttributeView – Supports reading or updating a file's Access Control Lists (ACL). The NFSv4 ACL model is supported. Any ACL model, such as the Windows ACL model, that has a well-defined mapping to the NFSv4 model might also be supported.**
- **UserDefinedFileAttributeView – Enables support of metadata that is user defined. This view can be mapped to any extension mechanisms that a system supports. In the Solaris OS, for example, you can use this view to store the MIME type of a file.**

A specific file system implementation might support only the basic file attribute view, or it may support several of these file attribute views. A file system implementation might support other attribute views not included in this API.

In most instances, you should not have to deal directly with any of the FileAttributeView interfaces. (If you do need to work directly with the FileAttributeView, you can access it via the getFileAttributeView(Path, Class<V>, LinkOption...) method.)

The readAttributes methods use generics and can be used to read the attributes for any of the file attributes views. The examples in the rest of this page use the readAttributes methods.

Additional topics:

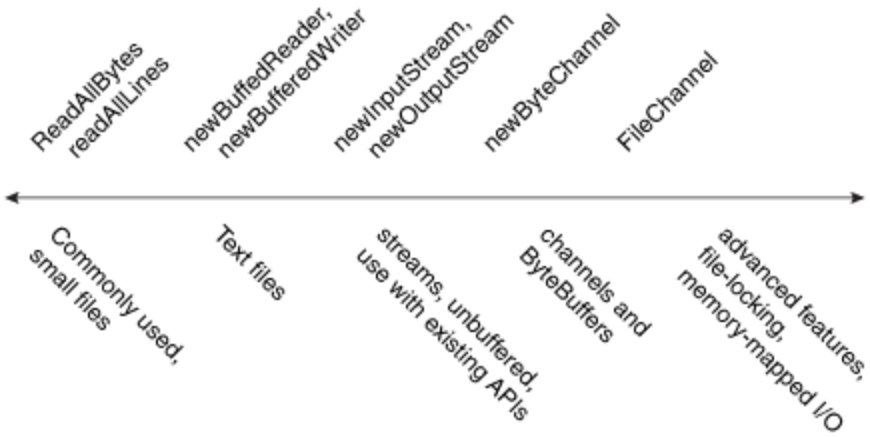
- **Basic File Attributes**
- **Setting Time Stamps**
- **DOS File Attributes**
- **POSIX File Permissions**
- **Setting a File or Group Owner**
- **User-Defined File Attributes**
- **File Store Attributes**

…

…

Reading, Writing, and Creating Files

This page discusses the details of reading, writing, creating, and opening files. There are a wide array of file I/O methods to choose from. To help make sense of the API, the following diagram arranges the file I/O methods by complexity.



File I/O Methods Arranged from Less Complex to More Complex

On the far left of the diagram are the utility methods readAllBytes, readAllLines, and the write methods, designed for simple, common cases. To the right of those are the methods used to iterate over a stream or lines of text, such as newBufferedReader, newBufferedWriter, then newInputStream and newOutputStream. These methods are interoperable with the java.io package. To the right of those are the methods for dealing with ByteChannels, SeekableByteChannels, and ByteBuffers, such as the newByteChannel method. Finally, on the far right are the methods that use FileChannel for advanced applications needing file locking or memory-mapped I/O.

Note: **The methods for creating a new file enable you to specify an optional set of initial attributes for the file. For example, on a file system that supports the POSIX set of standards (such as UNIX), you can specify a file owner, group owner, or file permissions at the time the file is created. The Managing Metadata page explains file attributes, and how to access and set them.**

This page has the following topics:

- **The OpenOptions Parameter**
- **Commonly Used Methods for Small Files**
- **Buffered I/O Methods for Text Files**
- **Methods for Unbuffered Streams and Interoperable with java.io APIs**
- **Methods for Channels and ByteBuffers**
- **Methods for Creating Regular and Temporary Files**

The OpenOptions Parameter

Several of the methods in this section take an optional OpenOptions parameter. This parameter is optional and the API tells you what the default behavior is for the method when none is specified.

The following StandardOpenOptions enums are supported:

- **WRITE – Opens the file for write access.**
- **APPEND – Appends the new data to the end of the file. This option is used with the WRITE or CREATE options.**
- **TRUNCATE_EXISTING – Truncates the file to zero bytes. This option is used with the WRITE option.**
- **CREATE_NEW – Creates a new file and throws an exception if the file already exists.**
- **CREATE – Opens the file if it exists or creates a new file if it does not.**
- **DELETE_ON_CLOSE – Deletes the file when the stream is closed. This option is useful for temporary files.**
- **SPARSE – Hints that a newly created file will be sparse. This advanced option is honored on some file systems, such as NTFS, where large files with data "gaps" can be stored in a more efficient manner where those empty gaps do not consume disk space.**
- **SYNC – Keeps the file (both content and metadata) synchronized with the underlying storage device.**

- **DSYNC – Keeps the file content synchronized with the underlying storage device.**

Commonly Used Methods for Small Files

Reading All Bytes or Lines from a File

If you have a small-ish file and you would like to read its entire contents in one pass, you can use the readAllBytes(Path) or readAllLines(Path, Charset) method. These methods take care of most of the work for you, such as opening and closing the stream, but are not intended for handling large files. The following code shows how to use the readAllBytes method:

```
Path file = ...;
byte[] fileArray;
fileArray = Files.readAllBytes(file);
```

Writing All Bytes or Lines to a File

You can use one of the write methods to write bytes, or lines, to a file.

- `write(Path, byte[], OpenOption...)`
- `write(Path, Iterable< extends CharSequence>, Charset, OpenOption...)`

The following code snippet shows how to use a write method.

```
Path file = ...;
byte[] buf = ...;
Files.write(file, buf);
```

Buffered I/O Methods for Text Files

The java.nio.file package supports channel I/O, which moves data in buffers, bypassing some of the layers that can bottleneck stream I/O.

Reading a File by Using Buffered Stream I/O

The newBufferedReader(Path, Charset) method opens a file for reading, returning a BufferedReader that can be used to read text from a file in an efficient manner.

The following code snippet shows how to use the newBufferedReader method to read from a file. The file is encoded in "US-ASCII."

```
Charset charset = Charset.forName("US-ASCII");
try (BufferedReader reader = Files.newBufferedReader(file, charset)) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

Writing a File by Using Buffered Stream I/O

You can use the newBufferedWriter(Path, Charset, OpenOption...) method to write to a file using a BufferedWriter.

The following code snippet shows how to create a file encoded in "US-ASCII" using this method:

```
Charset charset = Charset.forName("US-ASCII");
String s = ...;
try (BufferedWriter writer = Files.newBufferedWriter(file, charset)) {
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

Methods for Unbuffered Streams and Interoperable with java.io APIs

Reading a File by Using Stream I/O

To open a file for reading, you can use the newInputStream(Path, OpenOption...) method. This method returns an unbuffered input stream for reading bytes from the file.

```
Path file = ...;
try (InputStream in = Files.newInputStream(file);
    BufferedReader reader =
      new BufferedReader(new InputStreamReader(in))) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException x) {
    System.err.println(x);
}
```

Creating and Writing a File by Using Stream I/O

You can create a file, append to a file, or write to a file by using the newOutputStream(Path, OpenOption...) method. This method opens or creates a file for writing bytes and returns an unbuffered output stream.

The method takes an optional OpenOption parameter. If no open options are specified, and the file does not exist, a new file is created. If the file exists, it is truncated. This option is equivalent to invoking the method with the CREATE and TRUNCATE_EXISTING options.

The following example opens a log file. If the file does not exist, it is created. If the file exists, it is opened for appending.

```
import static java.nio.file.StandardOpenOption.*;
import java.nio.file.*;
import java.io.*;

public class LogFileTest {

  public static void main(String[] args) {

    // Convert the string to a
    // byte array.
    String s = "Hello World! ";
    byte data[] = s.getBytes();
    Path p = Paths.get("./logfile.txt");
```

```
    try (OutputStream out = new BufferedOutputStream(
      Files.newOutputStream(p, CREATE, APPEND))) {
      out.write(data, 0, data.length);
    } catch (IOException x) {
      System.err.println(x);
    }
  }
}
```

Methods for Channels and ByteBuffers

Reading and Writing Files by Using Channel I/O

While stream I/O reads a character at a time, channel I/O reads a buffer at a time. The ByteChannel interface provides basic read and write functionality. A SeekableByteChannel is a ByteChannel that has the capability to maintain a position in the channel and to change that position. A SeekableByteChannel also supports truncating the file associated with the channel and querying the file for its size.

The capability to move to different points in the file and then read from or write to that location makes random access of a file possible. See Random Access Files for more information.

There are two methods for reading and writing channel I/O.

- `newByteChannel(Path, OpenOption...)`
- `newByteChannel(Path, Set<? extends OpenOption>, FileAttribute<?>...)`

Note: **The newByteChannel methods return an instance of a SeekableByteChannel. With a default file system, you can cast this seekable byte channel to a FileChannel providing access to more advanced features such mapping a region of the file directly into memory for faster access, locking a region of the file so other processes cannot access it, or reading and writing bytes from an absolute position without affecting the channel's current position.**

Both newByteChannel methods enable you to specify a list of OpenOption options. The same open options used by the newOutputStream methods are supported, in addition to one more option: READ is required because the SeekableByteChannel supports both reading and writing.

Specifying READ opens the channel for reading. Specifying WRITE or APPEND opens the channel for writing. If none of these options is specified, the channel is opened for reading.

The following code snippet reads a file and prints it to standard output:

```
// Defaults to READ
try (SeekableByteChannel sbc = Files.newByteChannel(file)) {
    ByteBuffer buf = ByteBuffer.allocate(10);

    // Read the bytes with the proper encoding for this platform.  If
    // you skip this step, you might see something that looks like
    // Chinese characters when you expect Latin-style characters.
    String encoding = System.getProperty("file.encoding");
    while (sbc.read(buf) > 0) {
        buf.rewind();
        System.out.print(Charset.forName(encoding).decode(buf));
        buf.flip();
    }
} catch (IOException x) {
    System.out.println("caught exception: " + x);
```

The following example, written for UNIX and other POSIX file systems, creates a log file with a specific set of file permissions. This code creates a log file or appends to the log file if it already exists. The log file is created with read/write permissions for owner and read only permissions for group.

```
import static java.nio.file.StandardOpenOption.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.io.*;
import java.util.*;

public class LogFilePermissionsTest {

  public static void main(String[] args) {

    // Create the set of options for appending to the file.
    Set<OpenOption> options = new HashSet<OpenOption>();
    options.add(APPEND);
    options.add(CREATE);

    // Create the custom permissions attribute.
    Set<PosixFilePermission> perms =
      PosixFilePermissions.fromString("rw-r-----");
    FileAttribute<Set<PosixFilePermission>> attr =
      PosixFilePermissions.asFileAttribute(perms);

    // Convert the string to a ByteBuffer.
    String s = "Hello World! ";
    byte data[] = s.getBytes();
    ByteBuffer bb = ByteBuffer.wrap(data);

    Path file = Paths.get("./permissions.log");

    try (SeekableByteChannel sbc =
      Files.newByteChannel(file, options, attr)) {
      sbc.write(bb);
    } catch (IOException x) {
      System.out.println("Exception thrown: " + x);
    }
  }
}
```

Methods for Creating Regular and Temporary Files

Creating Files

You can create an empty file with an initial set of attributes by using the createFile(Path, FileAttribute<?>) method. For example, if, at the time of creation, you want a file to have a particular set of file permissions, use the createFile method to do so. If you do not specify any attributes, the file is created with default attributes. If the file already exists, createFile throws an exception.

In a single atomic operation, the createFile method checks for the existence of the file and creates that file with the specified attributes, which makes the process more secure against malicious code.

The following code snippet creates a file with default attributes:

```
Path file = ...;
```

```
try {
    // Create the empty file with default permissions, etc.
    Files.createFile(file);
} catch (FileAlreadyExistsException x) {
    System.err.format("file named %s" +
        " already exists%n", file);
} catch (IOException x) {
    // Some other sort of failure, such as permissions.
    System.err.format("createFile error: %s%n", x);
}
```

POSIX File Permissions has an example that uses createFile(Path, FileAttribute<?>) to create a file with pre-set permissions.

You can also create a new file by using the newOutputStream methods, as described in Creating and Writing a File using Stream I/O. If you open a new output stream and close it immediately, an empty file is created.

Creating Temporary Files

You can create a temporary file using one of the following createTempFile methods:

- `createTempFile(Path, String, String, FileAttribute<?>)`
- `createTempFile(String, String, FileAttribute<?>)`

The first method allows the code to specify a directory for the temporary file and the second method creates a new file in the default temporary-file directory. Both methods allow you to specify a suffix for the filename and the first method allows you to also specify a prefix. The following code snippet gives an example of the second method:

```
try {
    Path tempFile = Files.createTempFile(null, ".myapp");
    System.out.format("The temporary file" +
        " has been created: %s%n", tempFile)
;
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

The result of running this file would be something like the following:

```
The temporary file has been created: /tmp/509668702974537184.myapp
```

The specific format of the temporary file name is platform specific.

Random Access Files

***Random access files* permit nonsequential, or random, access to a file's contents. To access a file randomly, you open the file, seek a particular location, and read from or write to that file.**

This functionality is possible with the SeekableByteChannel interface. The SeekableByteChannel interface extends channel I/O with the notion of a current position. Methods enable you to set or query the position, and you can then read the data from, or write the data to, that location. The API consists of a few, easy to use, methods:

- **position – Returns the channel's current position**
- **position(long) – Sets the channel's position**
- **read(ByteBuffer) – Reads bytes into the buffer from the channel**

- **write(ByteBuffer) – Writes bytes from the buffer to the channel**
- **truncate(long) – Truncates the file (or other entity) connected to the channel**

Reading and Writing Files With Channel I/O shows that the Path.newByteChannel methods return an instance of a SeekableByteChannel. On the default file system, you can use that channel as is, or you can cast it to a FileChannel giving you access to more advanced features, such as mapping a region of the file directly into memory for faster access, locking a region of the file, or reading and writing bytes from an absolute location without affecting the channel's current position.

The following code snippet opens a file for both reading and writing by using one of the newByteChannel methods. The SeekableByteChannel that is returned is cast to a FileChannel. Then, 12 bytes are read from the beginning of the file, and the string "I was here!" is written at that location. The current position in the file is moved to the end, and the 12 bytes from the beginning are appended. Finally, the string, "I was here!" is appended, and the channel on the file is closed.

```
String s = "I was here!\n";
byte data[] = s.getBytes();
ByteBuffer out = ByteBuffer.wrap(data);

ByteBuffer copy = ByteBuffer.allocate(12);

try (FileChannel fc = (FileChannel.open(file, READ, WRITE))) {
    // Read the first 12
    // bytes of the file.
    int nread;
    do {
        nread = fc.read(copy);
    } while (nread != -1 && copy.hasRemaining());

    // Write "I was here!" at the beginning of the file.
    fc.position(0);
    while (out.hasRemaining())
        fc.write(out);
    out.rewind();

    // Move to the end of the file.  Copy the first 12 bytes to
    // the end of the file.  Then write "I was here!" again.
    long length = fc.size();
    fc.position(length-1);
    copy.flip();
    while (copy.hasRemaining())
        fc.write(copy);
    while (out.hasRemaining())
        fc.write(out);
} catch (IOException x) {
    System.out.println("I/O Exception: " + x);
}
```

Summary

Reading and writing text files

In JDK 7, the most important classes for text files are:

- **`Paths` and `Path` - file locations/names, but not their content.**
- **`Files` - operations on file content.**
- **`StandardCharsets` and `Charset` (an older class), for encodings of text files.**

- **the `File.toPath` method, which lets older code interact nicely with the newer java.nio API.**

In addition, the following classes are also commonly used with text files, for both JDK 7 and earlier versions:

- **`Scanner` - allows reading files in a compact way**
- **`BufferedReader` - readLine**
- **`BufferedWriter` - write + newLine**

When reading and writing text files:

- **it's often a good idea to use buffering (default size is 8K)**
- **there's always a need to pay attention to exceptions (in particular, IOException and FileNotFoundException)**

Character Encoding

In order to correctly read and write text files, you need to understand that those read/write operations always use an implicit character encoding to translate raw bytes - the 1s and 0s - into text. When a text file is saved, the tool that saves it must always use a character encoding (UTF-8 is recommended). There's a problem, however. The character encoding is not, in general, explicit: it's not saved as part of the file itself. Thus, a program that consumes a text file should know beforehand what its encoding is. If it doesn't, then the best it can do is make an assumption. Problems with encoding usually show up as weird characters in a tool that has read the file.

The `FileReader` and `FileWriter` classes are a bit tricky, since they implicitly use the system's default character encoding. If this default is not appropriate, the recommended alternatives are, for example:

```
FileInputStream fis = new FileInputStream("test.txt");
InputStreamReader in = new InputStreamReader(fis, "UTF-8");

FileOutputStream fos = new FileOutputStream("test.txt");
OutputStreamWriter out = new OutputStreamWriter(fos, "UTF-8");

Scanner scanner = new Scanner(file, "UTF-8");
```

Reading and writing binary files

In JDK 7, the most important classes for binary files are:

- Paths **and** Path **- file locations/names, but not their content.**
- Files **- operations on file content.**
- **the** File.toPath **method, which lets older code interact nicely with the newer java.nio API.**

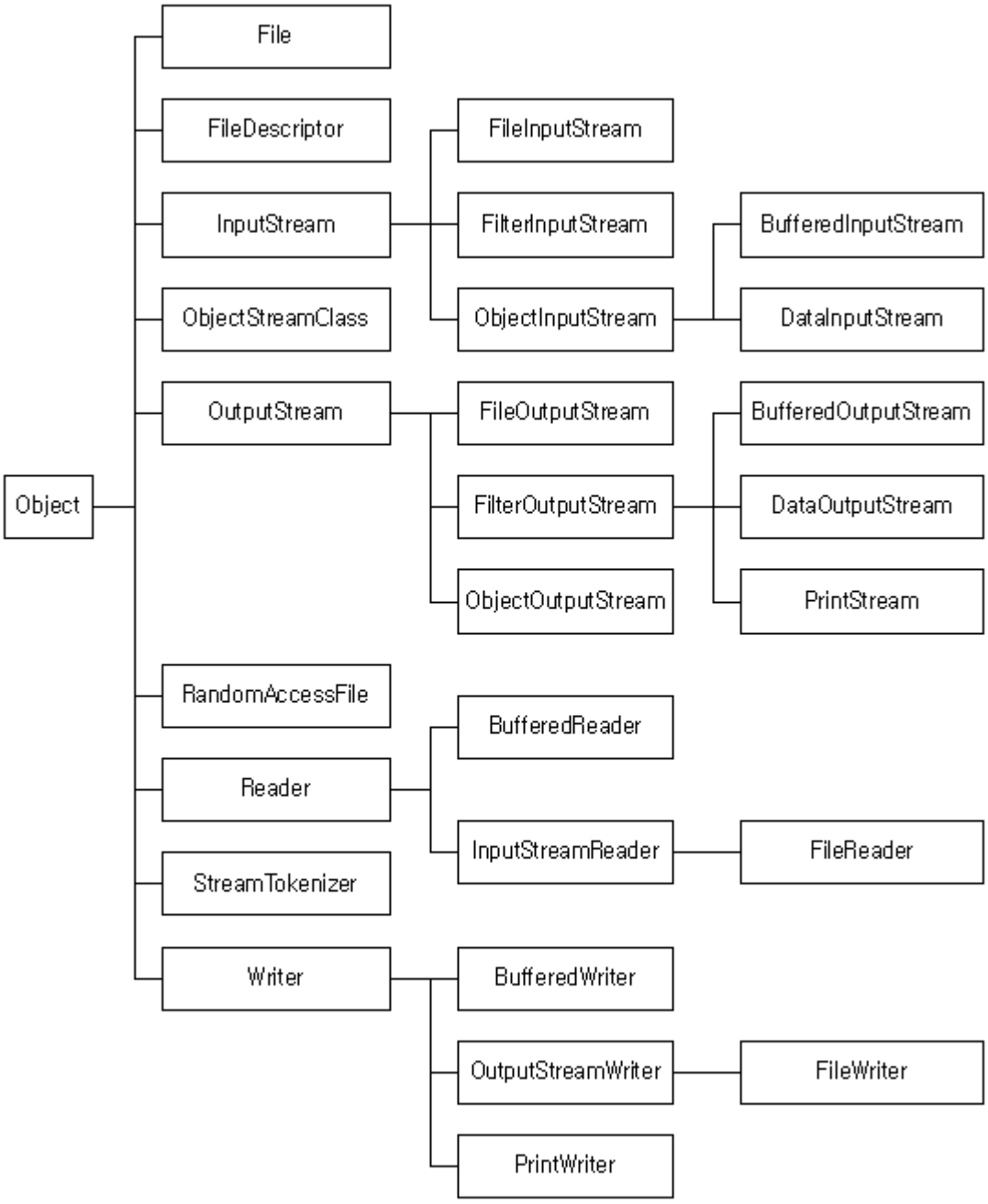
In addition, the following classes are also commonly used with binary files, for both JDK 7 and earlier versions:

Input	Output
FileInputStream	FileOutputStream
BufferedInputStream	BufferedOutputStream
ByteArrayInputStream	ByteArrayOutputStream
DataInput	DataOutput

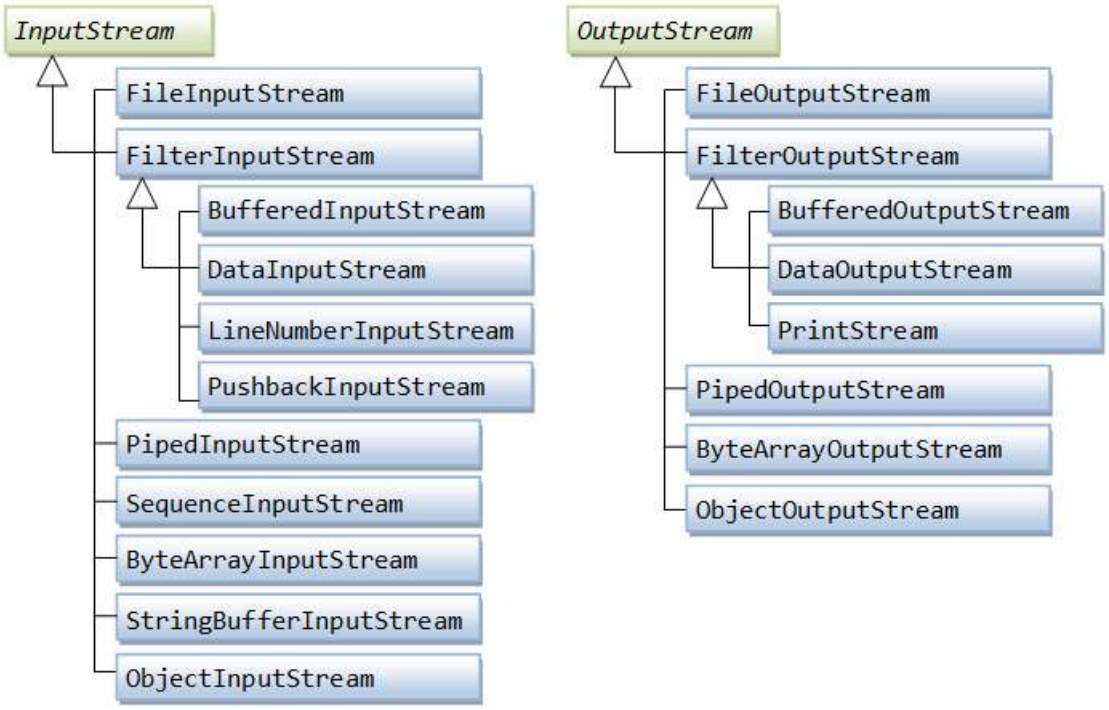
When reading and writing binary files:

- **it's almost always a good idea to use buffering (default buffer size is 8K)**
- **it's often possible to use references to abstract base classes, instead of references to specific concrete classes**
- **there's always a need to pay attention to exceptions (in particular,** IOException **and** FileNotFoundException**)**

IO Class Hierarchy:



OutputStream and InputStream hierarchies, raw byte streams:


InputStream
FileInputStream
FilterInputStream
BufferedInputStream
DataInputStream
LineNumberInputStream
PushbackInputStream
PipedInputStream
SequenceInputStream
ByteArrayInputStream
StringBufferInputStream
ObjectInputStream
OutputStream
FileOutputStream
FilterOutputStream
BufferedOutputStream
DataOutputStream
PrintStream
PipedOutputStream
ByteArrayOutputStream
ObjectOutputStream

Appendix 1 Inner classes

Static Nested Classes

As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference.

Note: A static nested class interacts with the instance members of its outer class (and other classes) just like any other top-level class. In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.

Static nested classes are accessed using the enclosing class name:

```
OuterClass.StaticNestedClass
```

For example, to create an object for the static nested class, use this syntax:

```
OuterClass.StaticNestedClass nestedObject =
     new OuterClass.StaticNestedClass();
```

Inner Classes

As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields. Also, because an inner class is associated with an instance, it cannot define any static members itself.

Objects that are instances of an inner class exist *within* an instance of the outer class. Consider the following classes:

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

An instance of `InnerClass` can exist only within an instance of `OuterClass` and has direct access to the methods and fields of its enclosing instance.

To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

You can use inner classes to implement helper. To handle user interface events, you must know how to use inner classes, because the event-handling mechanism makes extensive use of them.

Modifiers

You can use the same modifiers for inner classes that you use for other members of the outer class. For example, you can use the access specifiers `private`, `public`, and `protected` to restrict access to inner classes, just as you use them to restrict access do to other class members.

There are two special kinds of inner classes: local classes and anonymous classes.

Local Classes

You can define a local class inside any block (see Expressions, Statements, and Blocks for more information). For example, you can define a local class in a method body, a `for` loop, or an `if` clause.

A local class has access to the members of its enclosing class.

In addition, a local class has access to local variables. However, a local class can only access local variables that are declared final. When a local class accesses a local variable or parameter of the enclosing block, it *captures* that variable or parameter. However, starting in Java SE 8, a local class can access local variables and parameters of the enclosing block that are final or *effectively final*. A variable or parameter whose value is never changed after it is initialized is effectively final.

Starting in Java SE 8, if you declare the local class in a method, it can access the method's parameters.

Local classes are similar to inner classes because they cannot define or declare any static members. Local classes in static methods can only refer to static members of the enclosing class. For example, if you do not define the member variable `regularExpression` as static, then the Java compiler generates an error similar to "non-static variable `regularExpression` cannot be referenced from a static context."

Local classes are non-static because they have access to instance members of the enclosing block. Consequently, they cannot contain most kinds of static declarations.

You cannot declare an interface inside a block; interfaces are inherently static.

You cannot declare static initializers or member interfaces in a local class.

A local class can have static members provided that they are constant variables. (A *constant variable* is a variable of primitive type or type `String` that is declared final and initialized with a compile-time constant expression. A compile-time constant expression is typically a string or an arithmetic expression that can be evaluated at compile time.

Anonymous Classes

While local classes are class declarations, anonymous classes are expressions, which means that you define the class in another expression.

The syntax of an anonymous class expression is like the invocation of a constructor, except that there is a class definition contained in a block of code.

Consider the instantiation of the `frenchGreeting` object:

```
public class HelloWorldAnonymousClasses {

    interface HelloWorld {
        public void greet();
        public void greetSomeone(String someone);
    }

    public void sayHello() {

        class EnglishGreeting implements HelloWorld {
```

```
        String name = "world";
        public void greet() {
            greetSomeone("world");
        }
        public void greetSomeone(String someone) {
            name = someone;
            System.out.println("Hello " + name);
        }
    }

    HelloWorld englishGreeting = new EnglishGreeting();

HelloWorld frenchGreeting = new HelloWorld() {
    String name = "tout le monde";
    public void greet() {
        greetSomeone("tout le monde");
    }
    public void greetSomeone(String someone) {
        name = someone;
        System.out.println("Salut " + name);
    }
};
```

The anonymous class expression consists of the following:

- **The `new` operator**
- **The name of an interface to implement or a class to extend. In this example, the anonymous class is implementing the interface `HelloWorld`.**
- **Parentheses that contain the arguments to a constructor, just like a normal class instance creation expression. Note: When you implement an interface, there is no constructor, so you use an empty pair of parentheses, as in this example.**
- **A body, which is a class declaration body. More specifically, in the body, method declarations are allowed but statements are not.**

Because an anonymous class definition is an expression, it must be part of a statement. In this example, the anonymous class expression is part of the statement that instantiates the `frenchGreeting` object. (This explains why there is a semicolon after the closing brace.)

Like local classes, anonymous classes can capture variables; they have the same access to local variables of the enclosing scope:

- **An anonymous class has access to the members of its enclosing class.**
- **An anonymous class cannot access local variables in its enclosing scope that are not declared as `final` or effectively final.**
- **Like a nested class, a declaration of a type (such as a variable) in an anonymous class shadows any other declarations in the enclosing scope that have the same name.**

Anonymous classes also have the same restrictions as local classes with respect to their members:

- **You cannot declare static initializers or member interfaces in an anonymous class.**
- **An anonymous class can have static members provided that they are constant variables.**

Note that you can declare the following in anonymous classes:

- **Fields**
- **Extra methods (even if they do not implement any methods of the supertype)**
- **Instance initializers**
- **Local classes**

However, you cannot declare constructors in an anonymous class.

Examples of Anonymous Classes

```
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World!");
        Button btn = new Button();
        btn.setText("Say 'Hello World'");
        btn.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });

        StackPane root = new StackPane();
        root.getChildren().add(btn);
        primaryStage.setScene(new Scene(root, 300, 250));
        primaryStage.show();
    }

}
```

In this example, the method invocation `btn.setOnAction` specifies what happens when you select the Say 'Hello World' button. This method requires an object of type `EventHandler<ActionEvent>`. The `EventHandler<ActionEvent>` interface contains only one method, handle. Instead of implementing this method with a new class, the example uses an anonymous class expression. Notice that this expression is the argument passed to the `btn.setOnAction` method.

Because the `EventHandler<ActionEvent>` interface contains only one method, you can use a lambda expression instead of an anonymous class expression.

Lambda Expressions

Consider the `CheckPerson` interface:

```
  interface CheckPerson {
      boolean test(Person p);  }

  public static void printPersons(
      List<Person> roster, CheckPerson tester) {
      for (Person p : roster) {
          if (tester.test(p)) {
```

```
                p.printPerson();
            }
        }
    }

    printPersons(
        roster,
        new CheckPerson() {
            public boolean test(Person p) {
                return p.getGender() == Person.Sex.MALE
                    && p.getAge() >= 18
                    && p.getAge() <= 25;
            }
        }
    );
```

The `CheckPerson` interface is a *functional interface*. A functional interface is any interface that contains only one abstract method. (A functional interface may contain one or more default methods or static methods.) Because a functional interface contains only one abstract method, you can omit the name of that method when you implement it. To do this, instead of using an anonymous class expression, you use a *lambda expression*, which is highlighted in the following method invocation:

```
    printPersons(
        roster,
        (Person p) -> p.getGender() == Person.Sex.MALE
            && p.getAge() >= 18
            && p.getAge() <= 25
    );
```

Use Standard Functional Interfaces with Lambda Expressions:

```
    interface Predicate<Person> {
        boolean test(Person t);

}

    public static void printPersonsWithPredicate(
        List<Person> roster, Predicate<Person> tester) {
        for (Person p : roster) {
            if (tester.test(p)) {
                p.printPerson();
            }
        }

}
```

method invocation:

```
    printPersonsWithPredicate(
        roster,
        p -> p.getGender() == Person.Sex.MALE
            && p.getAge() >= 18
            && p.getAge() <= 25

);
```

Remember, to use a lambda expression, you need to implement a functional interface. In this case, you need a functional interface that contains an abstract method that can take one argument of type `Person` and returns void. The `Consumer<T>` interface contains the method `void accept(T t)`, which has these characteristics.

```
  public static void processPersons(
      List<Person> roster,
      Predicate<Person> tester,
      Consumer<Person> block) {
          for (Person p : roster) {
              if (tester.test(p)) {
                  block.accept(p);
              }
          }

}

  processPersons(
       roster,
       p -> p.getGender() == Person.Sex.MALE
           && p.getAge() >= 18
           && p.getAge() <= 25,
       p -> p.printPerson()

);

----------------------
```

Suppose that you want to validate the members' profiles or retrieve their contact information? In this case, you need a functional interface that contains an abstract method that returns a value. The `Function<T,R>` interface contains the method `R apply(T t)`. The following method retrieves the data specified by the parameter `mapper`, and then performs an action on it specified by the parameter `block`:

```
  public static void processPersonsWithFunction(
      List<Person> roster,
      Predicate<Person> tester,
      Function<Person, String> mapper,
      Consumer<String> block) {
      for (Person p : roster) {
          if (tester.test(p)) {
              String data = mapper.apply(p);
              block.accept(data);
          }
      }

}

  processPersonsWithFunction(
      roster,
      p -> p.getGender() == Person.Sex.MALE
          && p.getAge() >= 18
          && p.getAge() <= 25,
      p -> p.getEmailAddress(),
      email -> System.out.println(email)

);

----------------------
```

Use Generics More Extensively

```
  public static <X, Y> void processElements(
      Iterable<X> source,
      Predicate<X> tester,
      Function <X, Y> mapper,
      Consumer<Y> block) {
      for (X p : source) {
          if (tester.test(p)) {
              Y data = mapper.apply(p);
              block.accept(data);
          }
      }

}

  processElements(
      roster,
      p -> p.getGender() == Person.Sex.MALE
          && p.getAge() >= 18
          && p.getAge() <= 25,
      p -> p.getEmailAddress(),
      email -> System.out.println(email)

);
```

This method invocation performs the following actions:

1. **Obtains a source of objects from the collection source. In this example, it obtains a source of Person objects from the collection roster. Notice that the collection roster, which is a collection of type List, is also an object of type Iterable.**
2. **Filters objects that match the Predicate object tester. In this example, the Predicate object is a lambda expression that specifies which members would be eligible for Selective Service.**
3. **Maps each filtered object to a value as specified by the Function object mapper. In this example, the Function object is a lambda expression that returns the e-mail address of a member.**
4. **Performs an action on each mapped object as specified by the Consumer object block. In this example, the Consumer object is a lambda expression that prints a string, which is the e-mail address returned by the Function object.**

Use Aggregate Operations That Accept Lambda Expressions as Parameters

```
  roster
      .stream()
      .filter(
          p -> p.getGender() == Person.Sex.MALE
              && p.getAge() >= 18
              && p.getAge() <= 25)
      .map(p -> p.getEmailAddress())

    .forEach(email -> System.out.println(email));
```

`processElements` Action	Aggregate Operation
Obtain a source of objects	`Stream<E> stream()`
Filter objects that match a `Predicate` object	`Stream<T> filter(Predicate<? super T> predicate)`
Map objects to another value as specified by a `Function` object	`<R> Stream<R> map(Function<? super T,? extends R> mapper)`

Perform an action as specified by a `Consumer` object	`void forEach(Consumer<? super T> action)`

The operations `filter`, `map`, and `forEach` are *aggregate operations*. Aggregate operations process elements from a stream, not directly from a collection (which is the reason why the first method invoked in this example is `stream`). A *stream* is a sequence of elements. Unlike a collection, it is not a data structure that stores elements. Instead, a stream carries values from a source, such as collection, through a pipeline. A *pipeline* is a sequence of stream operations, which in this example is `filter- map-forEach`. In addition, aggregate operations typically accept lambda expressions as parameters, enabling you to customize how they behave.

`-------------------------`

Lambda Expressions in GUI Applications

```
btn.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }

      });

btn.setOnAction(
          event -> System.out.println("Hello World!")

      );
```

Syntax of Lambda Expressions

A lambda expression consists of the following:

- **A comma-separated list of formal parameters enclosed in parentheses. The `CheckPerson.test` method contains one parameter, `p`, which represents an instance of the `Person` class.**

  Note: You can omit the data type of the parameters in a lambda expression. In addition, you can omit the parentheses if there is only one parameter. For example, the following lambda expression is also valid:

  ```
  p -> p.getGender() == Person.Sex.MALE
      && p.getAge() >= 18
      && p.getAge() <= 25
  ```

- **The arrow token, `->`**
- **A body, which consists of a single expression or a statement block. This example uses the following expression:**
  - `p.getGender() == Person.Sex.MALE`
  - `&& p.getAge() >= 18`
    `&& p.getAge() <= 25`

  If you specify a single expression, then the Java runtime evaluates the expression and then returns its value. Alternatively, you can use a return statement:

  ```
  p -> {
      return p.getGender() == Person.Sex.MALE
          && p.getAge() >= 18
          && p.getAge() <= 25;
  ```

```
}
```

A return statement is not an expression; in a lambda expression, you must enclose statements in braces ({}). However, you do not have to enclose a void method invocation in braces. For example, the following is a valid lambda expression:

```
email -> System.out.println(email)
```

Note that a lambda expression looks a lot like a method declaration; you can consider lambda expressions as anonymous methods—methods without a name.

The following example, `Calculator`, is an example of lambda expressions that take more than one formal `parameter`:

```
public class Calculator {

    interface IntegerMath {
        int operation(int a, int b);
    }

    public int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }

    public static void main(String... args) {

        Calculator myApp = new Calculator();
        IntegerMath addition = (a, b) -> a + b;
        IntegerMath subtraction = (a, b) -> a - b;
        System.out.println("40 + 2 = " +
            myApp.operateBinary(40, 2, addition));
        System.out.println("20 - 10 = " +
            myApp.operateBinary(20, 10, subtraction));
    }
}
```

The method `operateBinary` performs a mathematical operation on two integer operands. The operation itself is specified by an instance of `IntegerMath`. The example defines two operations with lambda expressions, `addition` and `subtraction`. The example prints the following:

```
40 + 2 = 42
20 - 10 = 10
```

Accessing Local Variables of the Enclosing Scope

Like local and anonymous classes, lambda expressions can capture variables; they have the same access to local variables of the enclosing scope. However, unlike local and anonymous classes, lambda expressions do not have any shadowing issues. Lambda expressions are lexically scoped. This means that they do not inherit any names from a supertype or introduce a new level of scoping. Declarations in a lambda expression are interpreted just as they are in the enclosing environment.

```
import java.util.function.Consumer;

public class LambdaScopeTest {

    public int x = 0;
```

```
    class FirstLevel {

        public int x = 1;

        void methodInFirstLevel(int x) {

            // The following statement causes the compiler to generate
            // the error "local variables referenced from a lambda expression
            // must be final or effectively final" in statement A:
            //
            // x = 99;

            Consumer<Integer> myConsumer = (y) ->
            {
                System.out.println("x = " + x); // Statement A
                System.out.println("y = " + y);
                System.out.println("this.x = " + this.x);
                System.out.println("LambdaScopeTest.this.x = " +
                    LambdaScopeTest.this.x);
            };

            myConsumer.accept(x);

        }
    }

    public static void main(String... args) {
        LambdaScopeTest st = new LambdaScopeTest();
        LambdaScopeTest.FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
```

This example generates the following output:

```
x = 23
y = 23
this.x = 1
LambdaScopeTest.this.x = 0
```

If you substitute the parameter `x` in place of `y` in the declaration of the lambda expression `myConsumer`, then the compiler generates an error:

```
Consumer<Integer> myConsumer = (x) -> {
    // ...
}
```

The compiler generates the error "variable x is already defined in method methodInFirstLevel(int)" because the lambda expression does not introduce a new level of scoping. Consequently, you can directly access fields, methods, and local variables of the enclosing scope. For example, the lambda expression directly accesses the parameter `x` of the method `methodInFirstLevel`. To access variables in the enclosing class, use the keyword `this`. In this example, `this.x` refers to the member variable `FirstLevel.x`.

However, like local and anonymous classes, a lambda expression can only access local variables and parameters of the enclosing block that are final or effectively final. For example, suppose that you add the following assignment statement immediately after the `methodInFirstLevel` definition statement:

```
void methodInFirstLevel(int x) {
    x = 99;
```

```
        // ...
    }
```

Because of this assignment statement, the variable `FirstLevel.x` is not effectively final anymore. As a result, the Java compiler generates an error message similar to "local variables referenced from a lambda expression must be final or effectively final" where the lambda expression `myConsumer` tries to access the `FirstLevel.x` variable:

```
System.out.println("x = " + x);
```

Method References

There are four kinds of method references:

Kind	Example
Reference to a static method	`ContainingClass::staticMethodName`
Reference to an instance method of a particular object	`containingObject::instanceMethodName`
Reference to an instance method of an arbitrary object of a particular type	`ContainingType::methodName`
Reference to a constructor	`ClassName::new`

When to Use Nested Classes, Local Classes, Anonymous Classes, and Lambda Expressions

As mentioned in the section Nested Classes, nested classes enable you to logically group classes that are only used in one place, increase the use of encapsulation, and create more readable and maintainable code. Local classes, anonymous classes, and lambda expressions also impart these advantages; however, they are intended to be used for more specific situations:

- **Local class: Use it if you need to create more than one instance of a class, access its constructor, or introduce a new, named type (because, for example, you need to invoke additional methods later).**
- **Anonymous class: Use it if you need to declare fields or additional methods.**
- **Lambda expression:**
    - **Use it if you are encapsulating a single unit of behavior that you want to pass to other code. For example, you would use a lambda expression if you want a certain action performed on each element of a collection, when a process is completed, or when a process encounters an error.**
    - **Use it if you need a simple instance of a functional interface and none of the preceding criteria apply (for example, you do not need a constructor, a named type, fields, or additional methods).**
- **Nested class: Use it if your requirements are similar to those of a local class, you want to make the type more widely available, and you don't require access to local variables or method parameters.**
    - **Use a non-static nested class (or inner class) if you require access to an enclosing instance's non-public fields and methods. Use a static nested class if you don't require this access.**

Appendix 2 Numbers and Strings

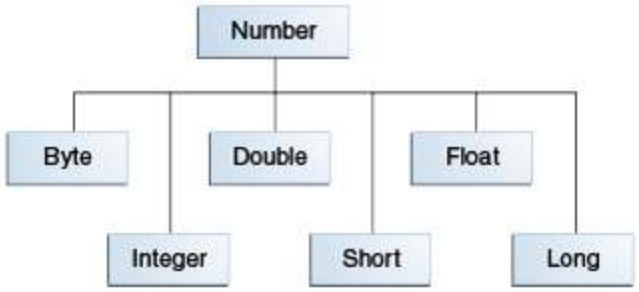
The Numbers Classes

When working with numbers, most of the time you use the primitive types in your code. For example:

```
int i = 500;
float gpa = 3.65f;
byte mask = 0xff;
```

There are, however, reasons to use objects in place of primitives, and the Java platform provides wrapper classes for each of the primitive data types. These classes "wrap" the primitive in an object. Often, the wrapping is done by the compiler—if you use a primitive where an object is expected, the compiler boxes the primitive in its wrapper class for you. Similarly, if you use a number object when a primitive is expected, the compiler unboxes the object for you.

All of the numeric wrapper classes are subclasses of the abstract class `Number`:



Note: There are four other subclasses of `Number` that are not discussed here. `BigDecimal` and `BigInteger` are used for high-precision calculations. `AtomicInteger` and `AtomicLong` are used for multi-threaded applications.

The `Number` classes include constants and useful class methods. The `MIN_VALUE` and `MAX_VALUE` constants contain the smallest and largest values that can be contained by an object of that type. The `byteValue`, `shortValue`, and similar methods convert one numeric type to another. The `valueOf` method converts a string to a number, and the `toString` method converts a number to a string.

There are three reasons that you might use a `Number` object rather than a primitive:

1. **As an argument of a method that expects an object (often used when manipulating collections of numbers).**
2. **To use constants defined by the class, such as `MIN_VALUE` and `MAX_VALUE`, that provide the upper and lower bounds of the data type.**
3. **To use class methods for converting values to and from other primitive types, for converting to and from strings, and for converting between number systems (decimal, octal, hexadecimal, binary).**

The following table lists the instance methods that all the subclasses of the `Number` class implement.

Methods Implemented by all Subclasses of Number	
Method	**Description**

`byte byteValue()`
`short shortValue()`
`int intValue()`
`long longValue()`
	Converts the value of this `Number` object to the primitive data type returned.

`float floatValue()` `double doubleValue()`	

`int compareTo(Byte anotherByte)`
`int compareTo(Double anotherDouble)`
`int compareTo(Float anotherFloat)`
`int compareTo(Integer anotherInteger)`
`int compareTo(Long anotherLong)`
`int compareTo(Short anotherShort)`
	Compares this `Number` object to the argument.

`boolean equals(Object obj)`	Determines whether this number object is equal to the argument. The methods return `true` if the argument is not `null` and is an object of the same type and with the same numeric value. There are some extra requirements for `Double` and `Float` objects that are described in the Java API documentation.

Each `Number` class contains other methods that are useful for converting numbers to and from strings and for converting between number systems. The following table lists these methods in the `Integer` class. Methods for the other `Number` subclasses are similar:

Conversion Methods, `Integer` Class	
Method	**Description**
`static Integer decode(String s)`	Decodes a string into an integer. Can accept string representations of decimal, octal, or hexadecimal numbers as input.
`static int parseInt(String s)`	Returns an integer (decimal only).
`static int parseInt(String s, int radix)`	Returns an integer, given a string representation of decimal, binary, octal, or hexadecimal (`radix` equals 10, 2, 8, or 16 respectively) numbers as input.
`String toString()`	Returns a `String` object representing the value of this `Integer`.
`static String toString(int i)`	Returns a `String` object representing the specified integer.
`static Integer valueOf(int i)`	Returns an `Integer` object holding the value of the specified primitive.
`static Integer valueOf(String s)`	Returns an `Integer` object holding the value of the specified string representation.
`static Integer valueOf(String s, int radix)`	Returns an `Integer` object holding the integer value of the specified string representation, parsed with the value of radix. For example, if s = "333" and radix = 8, the method returns the base-ten integer equivalent of the octal number 333.

Scanner class

The Scanner class is a class in java.util, which allows the user to read values of various types. There are far more methods in class Scanner than you will need in this course. We only cover a small useful subset, ones that allow us to read in numeric values from either the keyboard or file without having to convert them from strings and determine if there are more values to be read.

Class Constructors

There are two constructors that are particularly useful: one takes an InputStream object as a parameter and the other takes a FileReader object as a parameter.

```
Scanner in = new Scanner(System.in);  // System.in is an InputStream

Scanner inFile = new Scanner(new FileReader("myFile"));
```

If the file "myFile" is not found, a FileNotFoundException is thrown. This is a checked exception, so it must be caught or forwarded by putting the phrase "throws FileNotFoundException" on the header of the method in which the instantiation occurs and the header of any method that calls the method in which the instantiation occurs.

Numeric and String Methods

Method	Returns
`int nextInt()`	Returns the next token as an int. If the next token is not an integer,InputMismatchException is thrown.
`long nextLong()`	Returns the next token as a long. If the next token is not an integer,InputMismatchException is thrown.
`float nextFloat()`	Returns the next token as a float. If the next token is not a float or is out of range, InputMismatchException is thrown.
`double nextDouble()`	Returns the next token as a long. If the next token is not a float or is out of range, InputMismatchException is thrown.
`String next()`	Finds and returns the next complete token from this scanner and returns it as a string; a token is usually ended by whitespace such as a blank or line break. If not token exists,NoSuchElementException is thrown.
`String nextLine()`	Returns the rest of the current line, excluding any line separator at the end.
`void close()`	Closes the scanner.
`...`	...

The Scanner looks for tokens in the input. A token is a series of characters that ends with what Java calls whitespace. A whitespace character can be a blank, a tab character, a carriage return, or the end of the file. Thus, if we read a line that has a series of numbers separated by blanks, the scanner will take each number as a separate token. Although we have only shown four numeric methods, each numeric data type has a corresponding method that reads values of that type.

The numeric values may all be on one line with blanks between each value or may be on separate lines. Whitespace characters (blanks or carriage returns) act as separators. The next method returns the next input value as a string, regardless of what is keyed.

Boolean Methods

We said that the Scanner methods that read numeric data throw a InputMismatchException exception if the next value isnπt what the method expects. We can avoid that problem using Boolean methods. Here are four useful Boolean methods that allow us to check to be sure that the next value is what we expect.

Method	Returns
`boolean hasNextLine()`	Returns true if the scanner has another line in its input; false otherwise.
`boolean hasNextInt()`	Returns true if the next token in the scanner can be interpreted as an int value.
`boolean hasNextFloat()`	Returns true if the next toke in the scanner can be interpreted as a float value.
`boolean hasNext()`	Returns true if the scanner has another token in its input; false otherwise.
`boolean hasBoolean()`	Returns true if the next toke in the scanner can be interpreted as a boolean value.
`...`	...

Note:

```
int x;
Scanner input = new Scanner( System.in );
// to terminate the input: type end-of-file indicator
// On UNIX/Linux/Mac OS X systems: <Ctrl>+d
// On Windows systems: <Ctrl>+z
```

```
while (input.hasNext())
{
    x = input.nextInt();
    ....
}
```

Note: read char using scanner

```
Scanner in = new Scanner(System.in);
char c = in.next().charAt(0);
```

or:

```
Scanner in = new Scanner(System.in).useDelimiter("\\s*");
    while (!in.hasNext("z")) {
        char ch = in.next().charAt(0);
        System.out.print("[" + ch + "] ");
    }
```

input: `123 a b c x   y   z`
output: `[1] [2] [3] [a] [b] [c] [x] [y]`

The `Scanner` uses \s* as delimiter, which is the regex for "zero or more whitespace characters". This skips spaces etc in the input, so you only get non-whitespace characters, one at a time.

The printf and format Methods

The `java.io` package includes a `PrintStream` class that has two formatting methods that you can use to replace `print` and `println`. These methods, `format` and `printf`, are equivalent to one another. The familiar `System.out` that you have been using happens to be a `PrintStream` object, so you can invoke `PrintStream` methods on `System.out`. Thus, you can use `format` or `printf` anywhere in your code where you have previously been using `print` or `println`. For example,

```
System.out.format(.....);
```

The syntax for these two `java.io.PrintStream` methods is the same:

```
public PrintStream format(String format, Object... args)
```

where `format` is a string that specifies the formatting to be used and `args` is a list of the variables to be printed using that formatting. A simple example would be

```
System.out.format("The value of " + "the float variable is " +
     "%f, while the value of the " + "integer variable is %d, " +
     "and the string is %s", floatVar, intVar, stringVar);
```

The first parameter, `format`, is a format string specifying how the objects in the second parameter, `args`, are to be formatted. The format string contains plain text as well as *format specifiers*, which are special characters that format the arguments of `Object... args`. (The notation `Object... args` is called *varargs*, which means that the number of arguments may vary.)

Format specifiers begin with a percent sign (%) and end with a *converter*. The converter is a character indicating the type of argument to be formatted. In between the percent sign (%) and the converter you can have optional flags and specifiers. There are many converters, flags, and specifiers, which are documented in `java.util.Formatter`

Converters and Flags Used in `TestFormat.java`

Converter	Flag	Explanation
d		A decimal integer.
f		A float.
n		A new line character appropriate to the platform running the application. You should always use %n, rather than \n.
tB		A date & time conversion—locale-specific full name of month.
td, te		A date & time conversion—2-digit day of month. td has leading zeroes as needed, te does not.
ty, tY		A date & time conversion—ty = 2-digit year, tY = 4-digit year.
tl		A date & time conversion—hour in 12-hour clock.
tM		A date & time conversion—minutes in 2 digits, with leading zeroes as necessary.
tp		A date & time conversion—locale-specific am/pm (lower case).
tm		A date & time conversion—months in 2 digits, with leading zeroes as necessary.
tD		A date & time conversion—date as %tm%td%ty
	08	Eight characters in width, with leading zeroes as necessary.
	+	Includes sign, whether positive or negative.
	,	Includes locale-specific grouping characters.
	-	Left-justified.
	.3	Three places after decimal point.
	10.3	Ten characters in width, right justified, with three places after decimal point.
and:		
b or B		A boolean.
s or S		A string.
e or E		a decimal number in computerized scientific notation
g or G		The result is formatted using computerized scientific notation or decimal format, depending on the precision and the value after rounding.

```
import java.util.Calendar;
import java.util.Locale;

public class TestFormat {

    public static void main(String[] args) {
      long n = 461012;
      System.out.format("%d%n", n);      //  -->  "461012"
      System.out.format("%08d%n", n);    //  -->  "00461012"
      System.out.format("%+8d%n", n);    //  -->  " +461012"
      System.out.format("%,8d%n", n);    // -->  " 461,012"
      System.out.format("%+,8d%n%n", n); //  -->  "+461,012"

      double pi = Math.PI;

      System.out.format("%f%n", pi);       // -->  "3.141593"
      System.out.format("%.3f%n", pi);     // -->  "3.142"
      System.out.format("%10.3f%n", pi);   // -->  "     3.142"
      System.out.format("%-10.3f%n", pi);  // -->  "3.142"
      System.out.format(Locale.FRANCE,
                        "%-10.4f%n%n", pi); // -->  "3,1416"
```

```
        Calendar c = Calendar.getInstance();
        System.out.format("%tB %te, %tY%n", c, c, c); // -->  "May 29, 2006"

        System.out.format("%tl:%tM %tp%n", c, c, c);  // -->  "2:34 am"

        System.out.format("%tD%n", c);    // -->  "05/29/06"
    }
}
```

The DecimalFormat Class

You can use the `java.text.DecimalFormat` class to control the display of leading and trailing zeros, prefixes and suffixes, grouping (thousands) separators, and the decimal separator. `DecimalFormat` offers a great deal of flexibility in the formatting of numbers, but it can make your code more complex.

The example that follows creates a `DecimalFormat` object, `myFormatter`, by passing a pattern string to the `DecimalFormat` constructor. The `format()` method, which `DecimalFormat` inherits from `NumberFormat`, is then invoked by `myFormatter`—it accepts a `double` value as an argument and returns the formatted number in a string:

Here is a sample program that illustrates the use of `DecimalFormat`:

```
import java.text.*;

public class DecimalFormatDemo {

   static public void customFormat(String pattern, double value ) {
      DecimalFormat myFormatter = new DecimalFormat(pattern);
      String output = myFormatter.format(value);
      System.out.println(value + "  " + pattern + "  " + output);
   }

   static public void main(String[] args) {

      customFormat("###,###.###", 123456.789);
      customFormat("###.##", 123456.789);
      customFormat("000000.000", 123.78);
      customFormat("$###,###.###", 12345.67);
   }
}
```

The output is:

```
123456.789  ###,###.###  123,456.789
123456.789  ###.##  123456.79
123.78  000000.000  000123.780
12345.67  $###,###.###  $12,345.67
```

The following table explains each line of output.

`DecimalFormat.java` Output

Value	Pattern	Output	Explanation
123456.789	###,###.###	123,456.789	The pound sign (#) denotes a digit, the comma is a placeholder for the grouping separator, and the period is a placeholder for the decimal separator.

123456.789	###.##	123456.79	The `value` has three digits to the right of the decimal point, but the `pattern` has only two. The `format` method handles this by rounding up.
123.78	000000.000	000123.780	The `pattern` specifies leading and trailing zeros, because the 0 character is used instead of the pound sign (#).
12345.67	$###,###.###	$12,345.67	The first character in the `pattern` is the dollar sign ($). Note that it immediately precedes the leftmost digit in the formatted `output`.

Math class

The methods in the `Math` class are all static, so you call them directly from the class, like this:

```
Math.cos(angle);
```

Note: Using the `static import` language feature, you don't have to write `Math` in front of every math function:
```
import static java.lang.Math.*;
```

This allows you to invoke the `Math` class methods by their simple names. For example:

```
cos(angle);
```

The `Math` class includes two constants:

- `Math.E`, which is the base of natural logarithms, and
- `Math.PI`, which is the ratio of the circumference of a circle to its diameter.

The `Math` class also includes more than 40 static methods. The following table lists a number of the basic methods.

Basic Math Methods	
Method	Description

`double abs(double d)`
`float abs(float f)`
`int abs(int i)`
`long abs(long lng)`
	Returns the absolute value of the argument.

`double ceil(double d)`	Returns the smallest integer that is greater than or equal to the argument. Returned as a double.
`double floor(double d)`	Returns the largest integer that is less than or equal to the argument. Returned as a double.
`double rint(double d)`	Returns the integer that is closest in value to the argument. Returned as a double.
`long round(double d)` `int round(float f)`	Returns the closest long or int, as indicated by the method's return type, to the argument.

`double min(double arg1, double arg2)`
`float min(float arg1, float arg2)`
`int min(int arg1, int arg2)`
`long min(long arg1, long arg2)`
	Returns the smaller of the two arguments.

`double max(double arg1, double arg2)` `float max(float arg1, float arg2)`	Returns the larger of the two arguments.

Basic Math Methods	
Method	**Description**
`int max(int arg1, int arg2)` `long max(long arg1, long arg2)`	
Exponential and Logarithmic Methods	
Method	**Description**
`double exp(double d)`	Returns the base of the natural logarithms, e, to the power of the argument.
`double log(double d)`	Returns the natural logarithm of the argument.
`double pow(double base, double exponent)`	Returns the value of the first argument raised to the power of the second argument.
`double sqrt(double d)`	Returns the square root of the argument.
Trigonometric Methods	
Method	**Description**
`double sin(double d)`	Returns the sine of the specified double value.
`double cos(double d)`	Returns the cosine of the specified double value.
`double tan(double d)`	Returns the tangent of the specified double value.
`double asin(double d)`	Returns the arcsine of the specified double value.
`double acos(double d)`	Returns the arccosine of the specified double value.
`double atan(double d)`	Returns the arctangent of the specified double value.
`double atan2(double y, double x)`	Converts rectangular coordinates `(x, y)` to polar coordinate `(r, theta)` and returns `theta`.
`double toDegrees(double d)` `double toRadians(double d)`	Converts the argument to degrees or radians.

Random Numbers

The `random()` method returns a pseudo-randomly selected number between 0.0 and 1.0. The range includes 0.0 but not 1.0. In other words: `0.0 <= Math.random() < 1.0`. To get a number in a different range, you can perform arithmetic on the value returned by the random method. For example, to generate an integer between 0 and 9, you would write:

```
int number = (int)(Math.random() * 10);
```

By multiplying the value by 10, the range of possible values becomes `0.0 <= number < 10.0`.

Using `Math.random` works well when you need to generate a single random number. If you need to generate a series of random numbers, you should create an instance of `java.util.Random` and invoke methods on that object to generate numbers.

The java.util.Random class

The java.util.Random class instance is used to generate a stream of pseudorandom numbers. Following are the important points about Random:

The class uses a 48-bit seed, which is modified using a linear congruential formula.

The algorithms implemented by class Random use a protected utility method that on each invocation can supply up to 32 pseudorandomly generated bits.

Following is the declaration for java.util.Random class:

```
public class Random
   extends Object
   implements Serializable
```

Class constructors:

`Random()`	This creates a new random number generator.
`Random(long seed)`	This creates a new random number generator using a single long seed.

Class methods:

`protected int next(int bits)`	This method generates the next pseudorandom number.
`boolean nextBoolean()`	This method returns the next pseudorandom, uniformly distributed boolean value from this random number generator's sequence.
`void nextBytes(byte[] bytes)`	This method generates random bytes and places them into a user-supplied byte array.
`double nextDouble()`	This method returns the next pseudorandom, uniformly distributed double value between 0.0 and 1.0 from this random number generator's sequence.
`float nextFloat()`	This method returns the next pseudorandom, uniformly distributed float value between 0.0 and 1.0 from this random number generator's sequence.
`double nextGaussian()`	This method returns the next pseudorandom, Gaussian ("normally") distributed double value with mean 0.0 and standard deviation 1.0 from this random number generator's sequence.
`int nextInt()`	This method returns the next pseudorandom, uniformly distributed int value from this random number generator's sequence.
`int nextInt(int n)`	This method returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.
`long nextLong()`	This method returns the next pseudorandom, uniformly distributed long value from this random number generator's sequence.
`void setSeed(long seed)`	This method sets the seed of this random number generator using a single long seed.

Characters

Most of the time, if you are using a single character value, you will use the primitive `char` type. For example:

```
char ch = 'a';
// Unicode for uppercase Greek omega character
char uniChar = '\u03A9';
// an array of chars
char[] charArray = { 'a', 'b', 'c', 'd', 'e' };
```

There are times, however, when you need to use a char as an object—for example, as a method argument where an object is expected. The Java programming language provides a *wrapper* class that "wraps" the `char` in a `Character` object for this purpose. An object of type `Character` contains a single field, whose type is `char`. This Character class also offers a number of useful class (i.e., static) methods for manipulating characters.

You can create a `Character` object with the `Character` constructor:

```
Character ch = new Character('a');
```

The Java compiler will also create a `Character` object for you under some circumstances. For example, if you pass a primitive `char` into a method that expects an object, the compiler automatically converts the `char` to a `Character` for you. This feature is called *autoboxing*—or *unboxing*, if the conversion goes the other way.

The following table lists some of the most useful methods in the `Character` class, but is not exhaustive. For a complete listing of all methods in this class (there are more than 50), refer to the java.lang.Character API specification.

Useful Methods in the `Character` Class

Method	Description
`boolean isLetter(char ch)` `boolean isDigit(char ch)`	Determines whether the specified char value is a letter or a digit, respectively.
`boolean isWhitespace(char ch)`	Determines whether the specified char value is white space.
`boolean isUpperCase(char ch)` `boolean isLowerCase(char ch)`	Determines whether the specified char value is uppercase or lowercase, respectively.
`char toUpperCase(char ch)` `char toLowerCase(char ch)`	Returns the uppercase or lowercase form of the specified char value.
`toString(char ch)`	Returns a `String` object representing the specified character value — that is, a one-character string.

Escape Sequences

A character preceded by a backslash (\) is an *escape sequence* and has special meaning to the compiler. The following table shows the Java escape sequences:

Escape Sequences

Escape Sequence	Description
`\t`	Insert a tab in the text at this point.
`\b`	Insert a backspace in the text at this point.
`\n`	Insert a newline in the text at this point.
`\r`	Insert a carriage return in the text at this point.
`\f`	Insert a formfeed in the text at this point.
`\'`	Insert a single quote character in the text at this point.
`\"`	Insert a double quote character in the text at this point.
`\\`	Insert a backslash character in the text at this point.

Strings

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects.

The Java platform provides the `String` class to create and manipulate strings.

The most direct way to create a string is to write:

```
String greeting = "Hello world!";
```

In this case, "Hello world!" is a *string literal*—a series of characters in your code that is enclosed in double quotes. Whenever it encounters a string literal in your code, the compiler creates a `String` object with its value—in this case, `Hello world!`.

As with any other object, you can create `String` objects by using the `new` keyword and a constructor. The `String` class has thirteen constructors that allow you to provide the initial value of the string using different sources, such as an array of characters:

Note: The `String` class is immutable, so that once it is created a `String` object cannot be changed. The `String` class has a number of methods, some of which will be discussed below, that appear to modify strings. Since strings are immutable, what these methods really do is create and return a new string that contains the result of the operation.

Methods used to obtain information about an object are known as *accessor methods*. One accessor method that you can use with strings is the `length()` method, which returns the number of characters contained in the string object.

Concatenating Strings

The `String` class includes a method for concatenating two strings:

```
string1.concat(string2);
```

This returns a new string that is string1 with string2 added to it at the end.

You can also use the `concat()` method with string literals, as in:

```
"My name is ".concat("Rumplestiltskin");
```

Strings are more commonly concatenated with the + operator, as in

```
"Hello," + " world" + "!"
```

Creating Format Strings

You have seen the use of the `printf()` and `format()` methods to print output with formatted numbers. The `String` class has an equivalent class method, `format()`, that returns a `String` object rather than a `PrintStream` object.

Using `String's` static `format()` method allows you to create a formatted string that you can reuse, as opposed to a one-time print statement.

Converting Strings to Numbers

The `Number` subclasses that wrap primitive numeric types (`Byte`, `Integer`, `Double`, `Float`, `Long`, and `Short`) each provide a class method named `valueOf` that converts a string to an object of that type.

```
float a = (Float.valueOf(args[0])).floatValue();
float b = (Float.valueOf(args[1])).floatValue();
```

Note: Each of the `Number` subclasses that wrap primitive numeric types also provides a `parseXXXX()` method (for example, `parseFloat()`) that can be used to convert strings to primitive numbers. Since a primitive type is returned instead of an object, the `parseFloat()` method is more direct than the `valueOf()` method. For example, in the `ValueOfDemo` program, we could use:

```
float a = Float.parseFloat(args[0]);
float b = Float.parseFloat(args[1]);
```

Converting Numbers to Strings

Sometimes you need to convert a number to a string because you need to operate on the value in its string form. There are several easy ways to convert a number to a string:

```
int i;
// Concatenate "i" with an empty string; conversion is handled for you.
String s1 = "" + i;
```

or

```
// The valueOf class method.
String s2 = String.valueOf(i);
```

Each of the `Number` subclasses includes a class method, `toString()`, that will convert its primitive type to a string. For example:

```
int i;
double d;
String s3 = Integer.toString(i);
String s4 = Double.toString(d);
```

Manipulating Characters in a String

You can get the character at a particular index within a string by invoking the `charAt()` accessor method.

The `substring` Methods in the `String` Class	
Method	**Description**
`String substring(int beginIndex, int endIndex)`	Returns a new string that is a substring of this string. The substring begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`.
`String substring(int beginIndex)`	Returns a new string that is a substring of this string. The integer argument specifies the index of the first character. Here, the returned substring extends to the end of the original string.

Other Methods in the `String` Class for Manipulating Strings	
Method	**Description**
`String[] split(String regex)` `String[] split(String regex, int limit)`	Searches for a match as specified by the string argument (which contains a regular expression) and splits this string into an array of strings accordingly. The optional integer argument specifies the maximum size of the returned array. Regular expressions are covered in the lesson titled "Regular Expressions."
`CharSequence subSequence(int beginIndex, int endIndex)`	Returns a new character sequence constructed from `beginIndex` index up until `endIndex` - 1.
`String trim()`	Returns a copy of this string with leading and trailing white space removed.
`String toLowerCase()` `String toUpperCase()`	Returns a copy of this string converted to lowercase or uppercase. If no conversions are necessary, these methods return the original string.

The Search Methods in the `String` Class

Method	Description
`int indexOf(int ch)` `int lastIndexOf(int ch)`	Returns the index of the first (last) occurrence of the specified character.
`int indexOf(int ch, int fromIndex)` `int lastIndexOf(int ch, int fromIndex)`	Returns the index of the first (last) occurrence of the specified character, searching forward (backward) from the specified index.
`int indexOf(String str)` `int lastIndexOf(String str)`	Returns the index of the first (last) occurrence of the specified substring.
`int indexOf(String str, int fromIndex)` `int lastIndexOf(String str, int fromIndex)`	Returns the index of the first (last) occurrence of the specified substring, searching forward (backward) from the specified index.
`boolean contains(CharSequence s)`	Returns true if the string contains the specified character sequence.

Methods in the `String` Class for Manipulating Strings

Method	Description
`String replace(char oldChar, char newChar)`	Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
`String replace(CharSequence target, CharSequence replacement)`	Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.
`String replaceAll(String regex, String replacement)`	Replaces each substring of this string that matches the given regular expression with the given replacement.
`String replaceFirst(String regex, String replacement)`	Replaces the first substring of this string that matches the given regular expression with the given replacement.

Methods for Comparing Strings

Method	Description
`boolean endsWith(String suffix)` `boolean startsWith(String prefix)`	Returns `true` if this string ends with or begins with the substring specified as an argument to the method.
`boolean startsWith(String prefix, int offset)`	Considers the string beginning at the index `offset`, and returns `true` if it begins with the substring specified as an argument.
`int compareTo(String anotherString)`	Compares two strings lexicographically. Returns an integer indicating whether this string is greater than (result is > 0), equal to (result is = 0), or less than (result is < 0) the argument.
`int compareToIgnoreCase(String str)`	Compares two strings lexicographically, ignoring differences in case. Returns an integer indicating whether this string is greater than (result is > 0), equal to (result is = 0), or less than (result is < 0) the argument.
`boolean equals(Object anObject)`	Returns `true` if and only if the argument is a `String` object that represents the same sequence of characters as this object.
`boolean equalsIgnoreCase(String anotherString)`	Returns `true` if and only if the argument is a `String` object that represents the same sequence of characters as this object, ignoring differences in case.
`boolean regionMatches(int toffset, String other, int ooffset, int len)`	Tests whether the specified region of this string matches the specified region of the String argument.

	Region is of length `len` and begins at the index `toffset` for this string and `ooffset` for the other string.
`boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)`	Tests whether the specified region of this string matches the specified region of the String argument. Region is of length `len` and begins at the index `toffset` for this string and `ooffset` for the other string. The boolean argument indicates whether case should be ignored; if true, case is ignored when comparing characters.
`boolean matches(String regex)`	Tests whether this string matches the specified regular expression.

The StringBuilder Class

`StringBuilder` objects are like `String` objects, except that they can be modified. Internally, these objects are treated like variable-length arrays that contain a sequence of characters. At any point, the length and content of the sequence can be changed through method invocations.

Strings should always be used unless string builders offer an advantage in terms of simpler code (see the sample program at the end of this section) or better performance. For example, if you need to concatenate a large number of strings, appending to a `StringBuilder` object is more efficient.

The `StringBuilder` class, like the `String` class, has a `length()` method that returns the length of the character sequence in the builder.

Unlike strings, every string builder also has a *capacity*, the number of character spaces that have been allocated. The capacity, which is returned by the `capacity()` method, is always greater than or equal to the length (usually greater than) and will automatically expand as necessary to accommodate additions to the string builder.

`StringBuilder` Constructors	
Constructor	**Description**
`StringBuilder()`	Creates an empty string builder with a capacity of 16 (16 empty elements).
`StringBuilder(CharSequence cs)`	Constructs a string builder containing the same characters as the specified `CharSequence`, plus an extra 16 empty elements trailing the `CharSequence`.
`StringBuilder(int initCapacity)`	Creates an empty string builder with the specified initial capacity.
`StringBuilder(String s)`	Creates a string builder whose value is initialized by the specified string, plus an extra 16 empty elements trailing the string.
Length and Capacity Methods	
Method	**Description**
`void setLength(int newLength)`	Sets the length of the character sequence. If `newLength` is less than `length()`, the last characters in the character sequence are truncated. If `newLength` is greater than `length()`, null characters are added at the end of the character sequence.

`void ensureCapacity(int minCapacity)`	Ensures that the capacity is at least equal to the specified minimum.

Various `StringBuilder` Methods

Method	Description

`StringBuilder append(boolean b)`
`StringBuilder append(char c)`
`StringBuilder append(char[] str)`
`StringBuilder append(char[] str, int offset, int len)`
`StringBuilder append(double d)`
`StringBuilder append(float f)`
`StringBuilder append(int i)`
`StringBuilder append(long lng)`
`StringBuilder append(Object obj)`
`StringBuilder append(String s)`
	Appends the argument to this string builder. The data is converted to a string before the append operation takes place.

`StringBuilder delete(int start, int end)` `StringBuilder deleteCharAt(int index)`	The first method deletes the subsequence from start to end-1 (inclusive) in the `StringBuilder`'s char sequence. The second method deletes the character located at `index`.

`StringBuilder insert(int offset, boolean b)`
`StringBuilder insert(int offset, char c)`
`StringBuilder insert(int offset, char[] str)`
`StringBuilder insert(int index, char[] str, int offset, int len)`
`StringBuilder insert(int offset, double d)`
`StringBuilder insert(int offset, float f)`
`StringBuilder insert(int offset, int i)`
`StringBuilder insert(int offset, long lng)`
`StringBuilder insert(int offset, Object obj)`
`StringBuilder insert(int offset, String s)`
	Inserts the second argument into the string builder. The first integer argument indicates the index before which the data is to be inserted. The data is converted to a string before the insert operation takes place.

`StringBuilder replace(int start, int end, String s)` `void setCharAt(int index, char c)`	Replaces the specified character(s) in this string builder.
`StringBuilder reverse()`	Reverses the sequence of characters in this string builder.
`String toString()`	Returns a string that contains the character sequence in the builder.

Note: You can use any `String` method on a `StringBuilder` object by first converting the string builder to a string with the `toString()` method of the `StringBuilder` class. Then convert the string back into a string builder using the `StringBuilder(String str)` constructor.

Note: There is also a `StringBuffer` class that is *exactly* the same as the `StringBuilder` class, except that it is thread-safe by virtue of having its methods synchronized. Threads will be discussed in the lesson on concurrency.

Autoboxing and Unboxing

***Autoboxing* is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an `int` to an `Integer`, a `double` to a `Double`, and so on. If the conversion goes the other way, this is called *unboxing*.**

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2)
    li.add(i);
```

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2)
    li.add(Integer.valueOf(i));
```

Converting a primitive value (an `int`, for example) into an object of the corresponding wrapper class (`Integer`) is called autoboxing. The Java compiler applies autoboxing when a primitive value is:

- **Passed as a parameter to a method that expects an object of the corresponding wrapper class.**
- **Assigned to a variable of the corresponding wrapper class.**

Consider the following method:

```
public static int sumEven(List<Integer> li) {
    int sum = 0;
    for (Integer i: li)
        if (i % 2 == 0)
            sum += i;
        return sum;
}
```

Because the remainder (`%`) and unary plus (`+=`) operators do not apply to `Integer` objects, you may wonder why the Java compiler compiles the method without issuing any errors. The compiler does not generate an error because it invokes the `intValue` method to convert an `Integer` to an `int` at runtime:

```
public static int sumEven(List<Integer> li) {
    int sum = 0;
    for (Integer i : li)
        if (i.intValue() % 2 == 0)
            sum += i.intValue();
        return sum;
}
```

Converting an object of a wrapper type (`Integer`) to its corresponding primitive (`int`) value is called unboxing. The Java compiler applies unboxing when an object of a wrapper class is:

- **Passed as a parameter to a method that expects a value of the corresponding primitive type.**
- **Assigned to a variable of the corresponding primitive type.**

The `Unboxing` example shows how this works:

```
import java.util.ArrayList;
import java.util.List;

public class Unboxing {

    public static void main(String[] args) {
        Integer i = new Integer(-8);

        // 1. Unboxing through method invocation
        int absVal = absoluteValue(i);
        System.out.println("absolute value of " + i + " = " + absVal);

        List<Double> ld = new ArrayList<>();
        ld.add(3.1416);    // Π is autoboxed through method invocation.

        // 2. Unboxing through assignment
        double pi = ld.get(0);
```

```
        System.out.println("pi = " + pi);
    }

    public static int absoluteValue(int i) {
        return (i < 0) ? -i : i;
    }
}
```

The program prints the following:

```
absolute value of -8 = 8
pi = 3.1416
```

Autoboxing and unboxing lets developers write cleaner code, making it easier to read. The following table lists the primitive types and their corresponding wrapper classes, which are used by the Java compiler for autoboxing and unboxing:

Primitive type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

Appendix 3 Generics

In a nutshell, generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods. Much like the more familiar *formal parameters* used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

Code that uses generics has many benefits over non-generic code:

- **Stronger type checks at compile time.**
  A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

- **Elimination of casts.**
  The following code snippet without generics requires casting:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

  When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

- **Enabling programmers to implement generic algorithms.**
  By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

Generic Types

A *generic type* is a generic class or interface that is parameterized over types.

A *generic class* is defined with the following format:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the *type parameters* (also called *type variables*) `T1`, `T2`, ..., and `Tn`.

```
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

As you can see, all occurrences of `Object` are replaced by `T`. A type variable can be any non-primitive type you specify: any class type, any interface type, any array type, or even another type variable.

This same technique can be applied to create generic interfaces.

By convention, type parameter names are single, uppercase letters. The most commonly used type parameter names are:

- **E - Element (used extensively by the Java Collections Framework)**
- **K - Key**
- **N - Number**
- **T - Type**
- **V - Value**
- **S,U,V etc. - 2nd, 3rd, 4th types**

To reference the generic `Box` class from within your code, you must perform a *generic type invocation*, which replaces `T` with some concrete value, such as `Integer`:

```
Box<Integer> integerBox;
Box<Integer> integerBox = new Box<Integer>();
```

Type Parameter and Type Argument Terminology: Many developers use the terms "type parameter" and "type argument" interchangeably, but these terms are not the same. When coding, one provides type arguments in order to create a parameterized type. Therefore, the `T` in `Foo<T>` is a type parameter and the `String` in `Foo<String> f` is a type argument. This lesson observes this definition when using these terms.

In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, <>, is informally called *the diamond*. For example, you can create an instance of `Box<Integer>` with the following statement:

```
Box<Integer> integerBox = new Box<>();
```

Parameterized Types

You can also substitute a type parameter (i.e., `K` or `V`) with a parameterized type (i.e., `List<String>`). For example, using the `OrderedPair<K, V>` example:

```
OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new Box<Integer>(...));
```

Raw Types

A *raw type* is the name of a generic class or interface without any type arguments. For example, given the generic `Box` class:

```
public class Box<T> {
    public void set(T t) { /* ... */ }
    // ...
}
```

To create a parameterized type of `Box<T>`, you supply an actual type argument for the formal type parameter `T`:

```
Box<Integer> intBox = new Box<>();
```

If the actual type argument is omitted, you create a raw type of `Box<T>`:

```
Box rawBox = new Box();
```

Therefore, `Box` is the raw type of the generic type `Box<T>`. However, a non-generic class or interface type is *not* a raw type.

Raw types show up in legacy code because lots of API classes (such as the `Collections` classes) were not generic prior to JDK 5.0. When using raw types, you essentially get pre-generics behavior — a `Box` gives you `Objects`. For backward compatibility, assigning a parameterized type to its raw type is allowed:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;               // OK
```

But if you assign a raw type to a parameterized type, you get a warning:

```
Box rawBox = new Box();           // rawBox is a raw type of Box<T>
Box<Integer> intBox = rawBox;     // warning: unchecked conversion
```

You also get a warning if you use a raw type to invoke generic methods defined in the corresponding generic type:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;
rawBox.set(8);  // warning: unchecked invocation to set(T)
```

The warning shows that raw types bypass generic type checks, deferring the catch of unsafe code to runtime. Therefore, you should avoid using raw types.

The term "unchecked" means that the compiler does not have enough type information to perform all type checks necessary to ensure type safety. The "unchecked" warning is disabled, by default, though the compiler gives a hint. To see all "unchecked" warnings, recompile with `-Xlint:unchecked`.

To completely disable unchecked warnings, use the `-Xlint:-unchecked` flag. The `@SuppressWarnings("unchecked")` annotation suppresses unchecked warnings. If you are unfamiliar with the `@SuppressWarnings` syntax.

Generic Methods

Generic methods **are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Static and non-static generic methods are allowed, as well as generic class constructors.**

The syntax for a generic method includes a type parameter, inside angle brackets, and appears before the method's return type. For static generic methods, the type parameter section must appear before the method's return type.

The `Util` class includes a generic method, `compare`, which compares two `Pair` objects:

```
public class Util {
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}

public class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }
```

```
    public void setKey(K key) { this.key = key; }
    public void setValue(V value) { this.value = value; }
    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

The complete syntax for invoking this method would be:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

The type has been explicitly provided, as shown in bold. Generally, this can be left out and the compiler will infer the type that is needed:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.compare(p1, p2);
```

This feature, known as *type inference*, allows you to invoke a generic method as an ordinary method, without specifying a type between angle brackets.

Bounded Type Parameters

There may be times when you want to restrict the types that can be used as type arguments in a parameterized type. For example, a method that operates on numbers might only want to accept instances of `Number` or its subclasses. This is what *bounded type parameters* are for.

To declare a bounded type parameter, list the type parameter's name, followed by the `extends` keyword, followed by its *upper bound*, which in this example is `Number`. Note that, in this context, `extends` is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

```
public class Box<T> {

    private T t;

    public void set(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.set(new Integer(10));
        integerBox.inspect("some text"); // error: this is still String!
    }
}
```

By modifying our generic method to include this bounded type parameter, compilation will now fail, since our invocation of `inspect` still includes a `String`:

```
Box.java:21: <U>inspect(U) in Box<java.lang.Integer> cannot
  be applied to (java.lang.String)
                        integerBox.inspect("10");
                                   ^
1 error
```

In addition to limiting the types you can use to instantiate a generic type, bounded type parameters allow you to invoke methods defined in the bounds:

```
public class NaturalNumber<T extends Integer> {

    private T n;

    public NaturalNumber(T n)  { this.n = n; }

    public boolean isEven() {
        return n.intValue() % 2 == 0;
    }

    // ...
}
```

The `isEven` method invokes the `intValue` method defined in the `Integer` class through `n`.

Multiple Bounds

A type variable with multiple bounds is a subtype of all the types listed in the bound. If one of the bounds is a class, it must be specified first. For example:

```
Class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }
```

Generic Methods and Bounded Type Parameters

Bounded type parameters are key to the implementation of generic algorithms. Consider the following method that counts the number of elements in an array `T[]` that are greater than a specified element `elem`.

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem)  // compiler error
            ++count;
    return count;
}
```

The implementation of the method is straightforward, but it does not compile because the greater than operator (>) applies only to primitive types such as `short`, `int`, `double`, `long`, `float`, `byte`, and `char`. You cannot use the > operator to compare objects. To fix the problem, use a type parameter bounded by the `Comparable<T>` interface:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

The resulting code will be:

```
public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

Generics, Inheritance, and Subtypes

As you already know, it is possible to assign an object of one type to an object of another type provided that the types are compatible. For example, you can assign an `Integer` to an `Object`, since `Object` is one of `Integer`'s supertypes:

```
Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger;   // OK
```

In object-oriented terminology, this is called an "is a" relationship. Since an `Integer` *is a* kind of `Object`, the assignment is allowed. But `Integer` is also a kind of `Number`, so the following code is valid as well:

```
public void someMethod(Number n) { /* ... */ }

someMethod(new Integer(10));   // OK
someMethod(new Double(10.1));   // OK
```

The same is also true with generics. You can perform a generic type invocation, passing `Number` as its type argument, and any subsequent invocation of `add` will be allowed if the argument is compatible with `Number`:

```
Box<Number> box = new Box<Number>();
box.add(new Integer(10));   // OK
box.add(new Double(10.1));  // OK
```

Now consider the following method:

```
public void boxTest(Box<Number> n) { /* ... */ }
```

What type of argument does it accept? By looking at its signature, you can see that it accepts a single argument whose type is `Box<Number>`. But what does that mean? Are you allowed to pass in `Box<Integer>` or `Box<Double>`, as you might expect? The answer is "no", because `Box<Integer>` and `Box<Double>` are not subtypes of `Box<Number>`.

Now consider the following method:

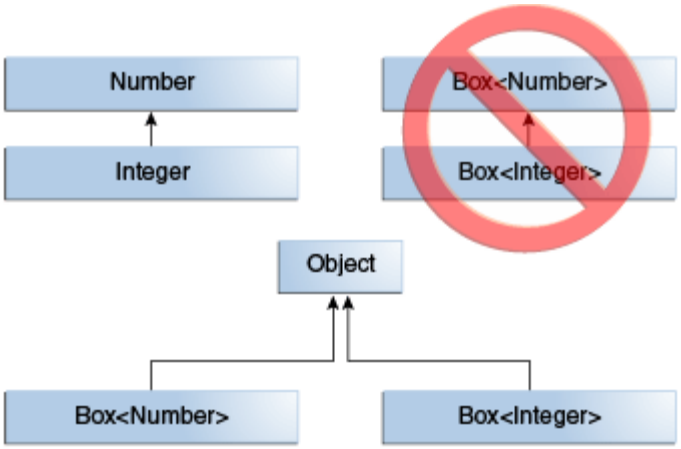
```
public void boxTest(Box<Number> n) { /* ... */ }
```

What type of argument does it accept? By looking at its signature, you can see that it accepts a single argument whose type is `Box<Number>`. But what does that mean? Are you allowed to pass in `Box<Integer>` or `Box<Double>`, as you might expect? The answer is "no", because `Box<Integer>` and `Box<Double>` are not subtypes of `Box<Number>`.

Now consider the following method:

```
public void boxTest(Box<Number> n) { /* ... */ }
```

What type of argument does it accept? By looking at its signature, you can see that it accepts a single argument whose type is `Box<Number>`. But what does that mean? Are you allowed to pass in `Box<Integer>` or `Box<Double>`, as you might expect? The answer is "no", because `Box<Integer>` and `Box<Double>` are not subtypes of `Box<Number>`.

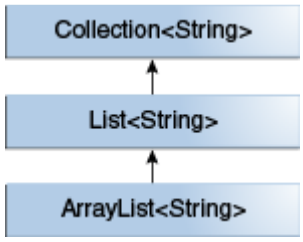


Note: Given two concrete types `A` and `B` (for example, `Number` and `Integer`), `MyClass<A>` has no relationship to `MyClass<B>`, regardless of whether or not `A` and `B` are related. The common parent of `MyClass<A>` and `MyClass<B>` is `Object`.

Generic Classes and Subtyping

You can subtype a generic class or interface by extending or implementing it. The relationship between the type parameters of one class or interface and the type parameters of another are determined by the `extends` and `implements` clauses.

Using the `Collections` classes as an example, `ArrayList<E>` implements `List<E>`, and `List<E> extends Collection<E>`. So `ArrayList<String>` is a subtype of `List<String>`, which is a subtype of `Collection<String>`. So long as you do not vary the type argument, the subtyping relationship is preserved between the types.

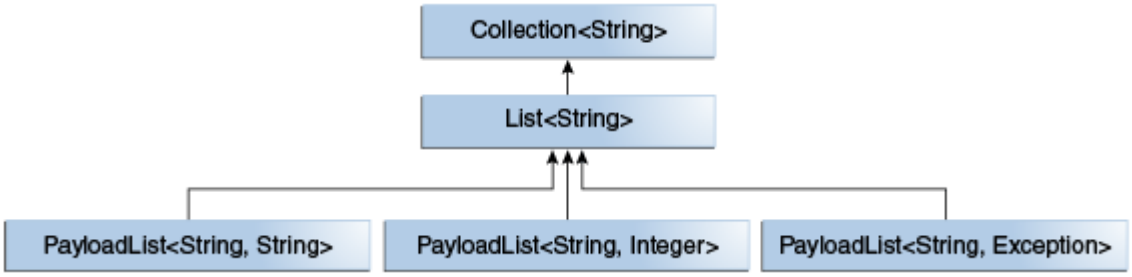


Now imagine we want to define our own list interface, `PayloadList`, that associates an optional value of generic type `P` with each element. Its declaration might look like:

```
interface PayloadList<E,P> extends List<E> {
  void setPayload(int index, P val);
  ...
}
```

The following parameterizations of `PayloadList` are subtypes of `List<String>`:

- **`PayloadList<String,String>`**
- **`PayloadList<String,Integer>`**
- **`PayloadList<String,Exception>`**



A sample `PayloadList` hierarchy

Type Inference

Type inference **is a Java compiler's ability to look at each method invocation and corresponding declaration to determine the type argument (or arguments) that make the invocation applicable. The inference algorithm determines the types of the arguments and, if available, the type that the result is being assigned, or returned. Finally, the inference algorithm tries to find the *most specific* type that works with all of the arguments.**

To illustrate this last point, in the following example, inference determines that the second argument being passed to the `pick` method is of type `Serializable`:

```
static <T> T pick(T a1, T a2) { return a2; }
Serializable s = pick("d", new ArrayList<String>());
```

Generally, a Java compiler can infer the type parameters of a generic method call. Consequently, in most cases, you do not have to specify them.

You can replace the type arguments required to invoke the constructor of a generic class with an empty set of type parameters (<>) as long as the compiler can infer the type arguments from the context. This pair of angle brackets is informally called the diamond.

For example, consider the following variable declaration:

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

You can substitute the parameterized type of the constructor with an empty set of type parameters (<>):

```
Map<String, List<String>> myMap = new HashMap<>();
```

Note that to take advantage of type inference during generic class instantiation, you must use the diamond. In the following example, the compiler generates an unchecked conversion warning because the `HashMap()` constructor refers to the `HashMap` raw type, not the `Map<String, List<String>>` type:

```
Map<String, List<String>> myMap = new HashMap(); // unchecked conversion warning
```

Type Inference and Generic Constructors of Generic and Non-Generic Classes

Note that constructors can be generic (in other words, declare their own formal type parameters) in both generic and non-generic classes. Consider the following example:

```
class MyClass<X> {
  <T> MyClass(T t) {
    // ...
  }
}
```

Consider the following instantiation of the class `MyClass`:

```
new MyClass<Integer>("")
```

or:

```
MyClass<Integer> myObject = new MyClass<>("");
```

Note: It is important to note that the inference algorithm uses only invocation arguments, target types, and possibly an obvious expected return type to infer types. The inference algorithm does not use results from later in the program.

Target Types

The Java compiler takes advantage of target typing to infer the type parameters of a generic method invocation. The *target type* of an expression is the data type that the Java compiler expects depending on where the expression appears. Consider the method `Collections.emptyList`, which is declared as follows:

```
static <T> List<T> emptyList();
```

Consider the following assignment statement:

```
List<String> listOne = Collections.emptyList();
```

This statement is expecting an instance of `List<String>`; this data type is the target type. Because the method `emptyList` returns a value of type `List<T>`, the compiler infers that the type argument `T` must be the value `String`. This works in both Java SE 7 and 8. Alternatively, you could use a type witness and specify the value of `T` as follows:

```
List<String> listOne = Collections.<String>emptyList();
```

However, this is not necessary in this context. It was necessary in other contexts, though. Consider the following method:

```
void processStringList(List<String> stringList) {
    // process stringList
}
```

Suppose you want to invoke the method `processStringList` with an empty list. In Java SE 7, the following statement does not compile:

```
processStringList(Collections.emptyList());
```

The Java SE 7 compiler generates an error message similar to the following:

```
List<Object> cannot be converted to List<String>
```

The compiler requires a value for the type argument `T` so it starts with the value `Object`. Consequently, the invocation of `Collections.emptyList` returns a value of type `List<Object>`, which is incompatible with the method `processStringList`. Thus, in Java SE 7, you must specify the value of the value of the type argument as follows:

```
processStringList(Collections.<String>emptyList());
```

This is no longer necessary in Java SE 8. The notion of what is a target type has been expanded to include method arguments, such as the argument to the method `processStringList`. In this case, `processStringList` requires an argument of type `List<String>`. The method `Collections.emptyList` returns a value of `List<T>`, so using

the target type of `List<String>`, the compiler infers that the type argument `T` has a value of `String`. Thus, in Java SE 8, the following statement compiles:

```
processStringList(Collections.emptyList());
```

Wildcards In generic code

In generic code, the question mark (`?`), called the *wildcard*, represents an unknown type. The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable; sometimes as a return type (though it is better programming practice to be more specific). The wildcard is never used as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

Upper Bounded Wildcards

You can use an upper bounded wildcard to relax the restrictions on a variable. For example, say you want to write a method that works on `List<Integer>`, `List<Double>`, *and* `List<Number>`; you can achieve this by using an upper bounded wildcard.

To declare an upper-bounded wildcard, use the wildcard character ('`?`'), followed by the `extends` keyword, followed by its *upper bound*. Note that, in this context, `extends` is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

To write the method that works on lists of `Number` and the subtypes of `Number`, such as `Integer`, `Double`, and `Float`, you would specify `List<? extends Number>`. The term `List<Number>` is more restrictive than `List<? extends Number>` because the former matches a list of type `Number` only, whereas the latter matches a list of type `Number` or any of its subclasses.

Unbounded Wildcards

The unbounded wildcard type is specified using the wildcard character (`?`), for example, `List<?>`. This is called a *list of unknown type*. There are two scenarios where an unbounded wildcard is a useful approach:

- If you are writing a method that can be implemented using functionality provided in the `Object` class.
- When the code is using methods in the generic class that don't depend on the type parameter. For example, `List.size` or `List.clear`. In fact, `Class<?>` is so often used because most of the methods in `Class<T>` do not depend on `T`.

Consider the following method, `printList`:

```
public static void printList(List<Object> list) {
    for (Object elem : list)
        System.out.println(elem + " ");
    System.out.println();
}
```

The goal of `printList` is to print a list of any type, but it fails to achieve that goal — it prints only a list of `Object` instances; it cannot print `List<Integer>`, `List<String>`, `List<Double>`, and so on, because they are not subtypes of `List<Object>`. To write a generic `printList` method, use `List<?>`:

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}
```

Because for any concrete type `A`, `List<A>` is a subtype of `List<?>`, you can use `printList` to print a list of any type:

```
List<Integer> li = Arrays.asList(1, 2, 3);
List<String>  ls = Arrays.asList("one", "two", "three");
printList(li);
printList(ls);
```

Note: The `Arrays.asList` method is used in examples throughout this lesson. This static factory method converts the specified array and returns a fixed-size list.

It's important to note that `List<Object>` and `List<?>` are not the same. You can insert an `Object`, or any subtype of `Object`, into a `List<Object>`. But you can only insert `null` into a `List<?>`.

Lower Bounded Wildcards

The Upper Bounded Wildcards section shows that an upper bounded wildcard restricts the unknown type to be a specific type or a subtype of that type and is represented using the `extends` keyword. In a similar way, a *lower bounded* wildcard restricts the unknown type to be a specific type or a *super type* of that type.

A lower bounded wildcard is expressed using the wildcard character ('`?`'), following by the `super` keyword, followed by its *lower bound*: `<? super A>`.

Note: You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

Say you want to write a method that puts `Integer` objects into a list. To maximize flexibility, you would like the method to work on `List<Integer>`, `List<Number>`, and `List<Object>` — anything that can hold `Integer` values.

To write the method that works on lists of `Integer` and the supertypes of `Integer`, such as `Integer`, `Number`, and `Object`, you would specify `List<? super Integer>`. The term `List<Integer>` is more restrictive than `List<? super Integer>` because the former matches a list of type `Integer` only, whereas the latter matches a list of any type that is a supertype of `Integer`.

The following code adds the numbers 1 through 10 to the end of a list:

```
public static void addNumbers(List<? super Integer> list) {
    for (int i = 1; i <= 10; i++) {
        list.add(i);
    }
}
```

Wildcards and Subtyping

Given the following two regular (non-generic) classes:

```
class A { /* ... */ }
class B extends A { /* ... */ }
```

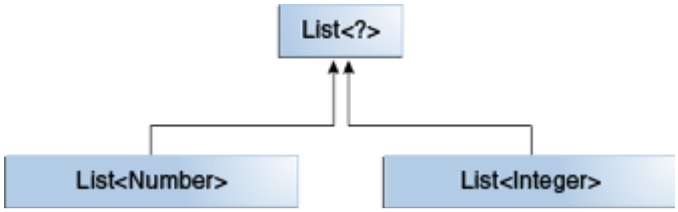
It would be reasonable to write the following code:

```
B b = new B();
A a = b;
```

This example shows that inheritance of regular classes follows this rule of subtyping: class `B` is a subtype of class `A` if `B` extends `A`. This rule does not apply to generic types:

```
List<B> lb = new ArrayList<>();
List<A> la = lb;   // compile-time error
```

Given that `Integer` is a subtype of `Number`, what is the relationship between `List<Integer>` and `List<Number>`?



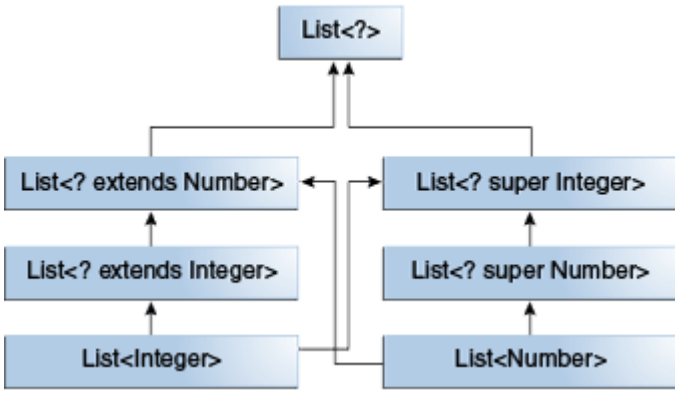
The common parent is `List<?>`.

Although `Integer` is a subtype of `Number`, `List<Integer>` is not a subtype of `List<Number>` and, in fact, these two types are not related. The common parent of `List<Number>` and `List<Integer>` is `List<?>`.

In order to create a relationship between these classes so that the code can access `Number`'s methods through `List<Integer>`'s elements, use an upper bounded wildcard:

```
List<? extends Integer> intList = new ArrayList<>();
List<? extends Number>  numList = intList;  // OK. List<? extends Integer> is a subtype
of List<? extends Number>
```

Because `Integer` is a subtype of `Number`, and `numList` is a list of `Number` objects, a relationship now exists between `intList` (a list of `Integer` objects) and `numList`. The following diagram shows the relationships between several `List` classes declared with both upper and lower bounded wildcards.



Wildcard Capture and Helper Methods

In some cases, the compiler infers the type of a wildcard. For example, a list may be defined as `List<?>` but, when evaluating an expression, the compiler infers a particular type from the code. This scenario is known as *wildcard capture*.

For the most part, you don't need to worry about wildcard capture, except when you see an error message that contains the phrase "capture of".

The `WildcardError` example produces a capture error when compiled:

```
import java.util.List;

public class WildcardError {
```

```
    void foo(List<?> i) {
        i.set(0, i.get(0));
    }
}
```

In this example, the compiler processes the `i` input parameter as being of type `Object`. When the `foo` method invokes List.set(int, E), the compiler is not able to confirm the type of object that is being inserted into the list, and an error is produced. When this type of error occurs it typically means that the compiler believes that you are assigning the wrong type to a variable. Generics were added to the Java language for this reason — to enforce type safety at compile time.

The `WildcardError` example generates the following error when compiled by Oracle's JDK 7 `javac` implementation:

```
WildcardError.java:6: error: method set in interface List<E> cannot be applied to given
types;
    i.set(0, i.get(0));
     ^
  required: int,CAP#1
  found: int,Object
  reason: actual argument Object cannot be converted to CAP#1 by method invocation
conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
1 error
```

In this example, the code is attempting to perform a safe operation, so how can you work around the compiler error? You can fix it by writing a *private helper method* which captures the wildcard. In this case, you can work around the problem by creating the private helper method, `fooHelper`, as shown in `WildcardFixed`:

```
public class WildcardFixed {

    void foo(List<?> i) {
        fooHelper(i);
    }


    // Helper method created so that the wildcard can be captured
    // through type inference.
    private <T> void fooHelper(List<T> l) {
        l.set(0, l.get(0));
    }

}
```

Thanks to the helper method, the compiler uses inference to determine that `T` is `CAP#1`, the capture variable, in the invocation. The example now compiles successfully.

By convention, helper methods are generally named *`originalMethodName`*`Helper`.

Guidelines for Wildcard Use

One of the more confusing aspects when learning to program with generics is determining when to use an upper bounded wildcard and when to use a lower bounded wildcard. This page provides some guidelines to follow when designing your code.

For purposes of this discussion, it is helpful to think of variables as providing one of two functions:

An "In" Variable

An "in" variable serves up data to the code. Imagine a copy method with two arguments: `copy(src, dest)`. The `src` argument provides the data to be copied, so it is the "in" parameter.

An "Out" Variable

An "out" variable holds data for use elsewhere. In the copy example, `copy(src, dest)`, the `dest` argument accepts data, so it is the "out" parameter.

Of course, some variables are used both for "in" and "out" purposes — this scenario is also addressed in the guidelines.

You can use the "in" and "out" principle when deciding whether to use a wildcard and what type of wildcard is appropriate. The following list provides the guidelines to follow:

Wildcard Guidelines:

- **An "in" variable is defined with an upper bounded wildcard, using the `extends` keyword.**
- **An "out" variable is defined with a lower bounded wildcard, using the `super` keyword.**
- **In the case where the "in" variable can be accessed using methods defined in the `Object` class, use an unbounded wildcard.**
- **In the case where the code needs to access the variable as both an "in" and an "out" variable, do not use a wildcard.**

These guidelines do not apply to a method's return type. Using a wildcard as a return type should be avoided because it forces programmers using the code to deal with wildcards.

A list defined by `List<? extends ...>` can be informally thought of as read-only, but that is not a strict guarantee. Suppose you have the following two classes:

```
class NaturalNumber {

    private int i;

    public NaturalNumber(int i) { this.i = i; }
    // ...
}

class EvenNumber extends NaturalNumber {

    public EvenNumber(int i) { super(i); }
    // ...
}
```

Consider the following code:

```
List<EvenNumber> le = new ArrayList<>();
List<? extends NaturalNumber> ln = le;
ln.add(new NaturalNumber(35));  // compile-time error
```

Because `List<EvenNumber>` is a subtype of `List<? extends NaturalNumber>`, you can assign `le` to `ln`. But you cannot use `ln` to add a natural number to a list of even numbers. The following operations on the list are possible:

- **You can add `null`.**
- **You can invoke `clear`.**
- **You can get the iterator and invoke `remove`.**
- **You can capture the wildcard and write elements that you've read from the list.**

You can see that the list defined by `List<? extends NaturalNumber>` is not read-only in the strictest sense of the word, but you might think of it that way because you cannot store a new element or change an existing element in the list.

Type Erasure

Generics were introduced to the Java language to provide tighter type checks at compile time and to support generic programming. To implement generics, the Java compiler applies type erasure to:

- **Replace all type parameters in generic types with their bounds or `Object` if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.**
- **Insert type casts if necessary to preserve type safety.**
- **Generate bridge methods to preserve polymorphism in extended generic types.**

Type erasure ensures that no new classes are created for parameterized types; consequently, generics incur no runtime overhead.

Erasure of Generic Types

During the type erasure process, the Java compiler erases all type parameters and replaces each with its first bound if the type parameter is bounded, or `Object` if the type parameter is unbounded.

Consider the following generic class that represents a node in a singly linked list:

```
public class Node<T> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) }
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // ...
}
```

Because the type parameter `T` is unbounded, the Java compiler replaces it with `Object`:

```
public class Node {

    private Object data;
    private Node next;

    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }

    public Object getData() { return data; }
    // ...
}
```

In the following example, the generic `Node` class uses a bounded type parameter:

```
public class Node<T extends Comparable<T>> {
```

```
    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // ...
}
```

The Java compiler replaces the bounded type parameter `T` with the first bound class, `Comparable`:

```
public class Node {

    private Comparable data;
    private Node next;

    public Node(Comparable data, Node next) {
        this.data = data;
        this.next = next;
    }

    public Comparable getData() { return data; }
    // ...
}
```

Erasure of Generic Methods

The Java compiler also erases type parameters in generic method arguments. Consider the following generic method:

```
// Counts the number of occurrences of elem in anArray.
//
public static <T> int count(T[] anArray, T elem) {
    int cnt = 0;
    for (T e : anArray)
        if (e.equals(elem))
            ++cnt;
        return cnt;
}
```

Because `T` is unbounded, the Java compiler replaces it with `Object`:

```
public static int count(Object[] anArray, Object elem) {
    int cnt = 0;
    for (Object e : anArray)
        if (e.equals(elem))
            ++cnt;
        return cnt;
}
```

Suppose the following classes are defined:

```
class Shape { /* ... */ }
class Circle extends Shape { /* ... */ }
class Rectangle extends Shape { /* ... */ }
```

You can write a generic method to draw different shapes:

```
public static <T extends Shape> void draw(T shape) { /* ... */ }
```

The Java compiler replaces `T` with `Shape`:

```
public static void draw(Shape shape) { /* ... */ }
```

Effects of Type Erasure and Bridge Methods

Sometimes type erasure causes a situation that you may not have anticipated. The following example shows how this can occur. The example (described in Bridge Methods) shows how a compiler sometimes creates a synthetic method, called a bridge method, as part of the type erasure process.

When compiling a class or interface that extends a parameterized class or implements a parameterized interface, the compiler may need to create a synthetic method, called a *bridge method*, as part of the type erasure process. You normally don't need to worry about bridge methods, but you might be puzzled if one appears in a stack trace.

After type erasure, the `Node` and `MyNode` classes become:

```
public class Node {

    public Object data;

    public Node(Object data) { this.data = data; }

    public void setData(Object data) {
        System.out.println("Node.setData");
        this.data = data;
    }
}

public class MyNode extends Node {

    public MyNode(Integer data) { super(data); }

    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }
}
```

After type erasure, the method signatures do not match. The `Node` method becomes `setData(Object)` and the `MyNode` method becomes `setData(Integer)`. Therefore, the `MyNode` `setData` method does not override the `Node` `setData` method.

To solve this problem and preserve the polymorphism of generic types after type erasure, a Java compiler generates a bridge method to ensure that subtyping works as expected. For the `MyNode` class, the compiler generates the following bridge method for `setData`:

```
class MyNode extends Node {

    // Bridge method generated by the compiler
    //
    public void setData(Object data) {
        setData((Integer) data);
    }
```

```
    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }

    // ...
}
```

As you can see, the bridge method, which has the same method signature as the `Node` class's `setData` method after type erasure, delegates to the original `setData` method.

Non-Reifiable Types

The section Type Erasure discusses the process where the compiler removes information related to type parameters and type arguments. Type erasure has consequences related to variable arguments (also known as *varargs*) methods whose varargs formal parameter has a non-reifiable type.

Non-Reifiable Types

A *reifiable* type is a type whose type information is fully available at runtime. This includes primitives, non-generic types, raw types, and invocations of unbound wildcards.

***Non-reifiable types* are types where information has been removed at compile-time by type erasure — invocations of generic types that are not defined as unbounded wildcards. A non-reifiable type does not have all of its information available at runtime. Examples of non-reifiable types are `List<String>` and `List<Number>`; the JVM cannot tell the difference between these types at runtime. There are certain situations where non-reifiable types cannot be used: in an `instanceof` expression, for example, or as an element in an array.**

Heap Pollution

***Heap pollution* occurs when a variable of a parameterized type refers to an object that is not of that parameterized type. This situation occurs if the program performed some operation that gives rise to an unchecked warning at compile-time. An *unchecked warning* is generated if, either at compile-time (within the limits of the compile-time type checking rules) or at runtime, the correctness of an operation involving a parameterized type (for example, a cast or method call) cannot be verified. For example, heap pollution occurs when mixing raw types and parameterized types, or when performing unchecked casts.**

In normal situations, when all code is compiled at the same time, the compiler issues an unchecked warning to draw your attention to potential heap pollution. If you compile sections of your code separately, it is difficult to detect the potential risk of heap pollution. If you ensure that your code compiles without warnings, then no heap pollution can occur.

Potential Vulnerabilities of Varargs Methods with Non-Reifiable Formal Parameters

Generic methods that include vararg input parameters can cause heap pollution.

Consider the following `ArrayBuilder` class:

```
public class ArrayBuilder {

  public static <T> void addToList (List<T> listArg, T... elements) {
    for (T x : elements) {
      listArg.add(x);
    }
  }
```

```
  public static void faultyMethod(List<String>... l) {
    Object[] objectArray = l;     // Valid
    objectArray[0] = Arrays.asList(42);
    String s = l[0].get(0);       // ClassCastException thrown here
  }

}
```

The following example, `HeapPollutionExample` uses the `ArrayBuiler` class:

```
public class HeapPollutionExample {

  public static void main(String[] args) {

    List<String> stringListA = new ArrayList<String>();
    List<String> stringListB = new ArrayList<String>();

    ArrayBuilder.addToList(stringListA, "Seven", "Eight", "Nine");
    ArrayBuilder.addToList(stringListB, "Ten", "Eleven", "Twelve");
    List<List<String>> listOfStringLists =
      new ArrayList<List<String>>();
    ArrayBuilder.addToList(listOfStringLists,
      stringListA, stringListB);

    ArrayBuilder.faultyMethod(Arrays.asList("Hello!"), Arrays.asList("World!"));
  }
}
```

When compiled, the following warning is produced by the definition of the `ArrayBuilder.addToList` method:

```
warning: [varargs] Possible heap pollution from parameterized vararg type T
```

When the compiler encounters a varargs method, it translates the varargs formal parameter into an array. However, the Java programming language does not permit the creation of arrays of parameterized types. In the method `ArrayBuilder.addToList`, the compiler translates the varargs formal parameter `T... elements` to the formal parameter `T[] elements`, an array. However, because of type erasure, the compiler converts the varargs formal parameter to `Object[] elements`. Consequently, there is a possibility of heap pollution.

The following statement assigns the varargs formal parameter `l` to the `Object` array `objectArgs`:

```
Object[] objectArray = l;
```

This statement can potentially introduce heap pollution. A value that does match the parameterized type of the varargs formal parameter `l` can be assigned to the variable `objectArray`, and thus can be assigned to `l`. However, the compiler does not generate an unchecked warning at this statement. The compiler has already generated a warning when it translated the varargs formal parameter `List<String>... l` to the formal parameter `List[] l`. This statement is valid; the variable `l` has the type `List[]`, which is a subtype of `Object[]`.

Consequently, the compiler does not issue a warning or error if you assign a `List` object of any type to any array component of the `objectArray` array as shown by this statement:

```
objectArray[0] = Arrays.asList(42);
```

This statement assigns to the first array component of the `objectArray` array with a `List` object that contains one object of type `Integer`.

Suppose you invoke `ArrayBuilder.faultyMethod` with the following statement:

```
ArrayBuilder.faultyMethod(Arrays.asList("Hello!"), Arrays.asList("World!"));
```

At runtime, the JVM throws a `ClassCastException` at the following statement:

```
// ClassCastException thrown here
String s = l[0].get(0);
```

The object stored in the first array component of the variable `l` has the type `List<Integer>`, but this statement is expecting an object of type `List<String>`.

Prevent Warnings from Varargs Methods with Non-Reifiable Formal Parameters

If you declare a varargs method that has parameters of a parameterized type, and you ensure that the body of the method does not throw a `ClassCastException` or other similar exception due to improper handling of the varargs formal parameter, you can prevent the warning that the compiler generates for these kinds of varargs methods by adding the following annotation to static and non-constructor method declarations:

```
@SafeVarargs
```

The `@SafeVarargs` annotation is a documented part of the method's contract; this annotation asserts that the implementation of the method will not improperly handle the varargs formal parameter.

It is also possible, though less desirable, to suppress such warnings by adding the following to the method declaration:

```
@SuppressWarnings({"unchecked", "varargs"})
```

However, this approach does not suppress warnings generated from the method's call site.

Restrictions on Generics

To use Java generics effectively, you must consider the following restrictions:

- **Cannot Instantiate Generic Types with Primitive Types**
- **Cannot Create Instances of Type Parameters**
- **Cannot Declare Static Fields Whose Types are Type Parameters**
- **Cannot Use Casts or `instanceof` With Parameterized Types**
- **Cannot Create Arrays of Parameterized Types**
- **Cannot Create, Catch, or Throw Objects of Parameterized Types**
- **Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type**

Cannot Instantiate Generic Types with Primitive Types

Consider the following parameterized type:

```
class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
```

```
    }

    // ...
}
```

When creating a `Pair` object, you cannot substitute a primitive type for the type parameter `K` or `V`:

```
Pair<int, char> p = new Pair<>(8, 'a');  // compile-time error
```

You can substitute only non-primitive types for the type parameters `K` and `V`:

```
Pair<Integer, Character> p = new Pair<>(8, 'a');
```

Note that the Java compiler autoboxes `8` to `Integer.valueOf(8)` and `'a'` to `Character('a')`:

```
Pair<Integer, Character> p = new Pair<>(Integer.valueOf(8), new Character('a'));
```

Cannot Create Instances of Type Parameters

You cannot create an instance of a type parameter. For example, the following code causes a compile-time error:

```
public static <E> void append(List<E> list) {
    E elem = new E();  // compile-time error
    list.add(elem);
}
```

As a workaround, you can create an object of a type parameter through reflection:

```
public static <E> void append(List<E> list, Class<E> cls) throws Exception {
    E elem = cls.newInstance();   // OK
    list.add(elem);
}
```

You can invoke the `append` method as follows:

```
List<String> ls = new ArrayList<>();
append(ls, String.class);
```

Cannot Declare Static Fields Whose Types are Type Parameters

A class's static field is a class-level variable shared by all non-static objects of the class. Hence, static fields of type parameters are not allowed. Consider the following class:

```
public class MobileDevice<T> {
    private static T os;

    // ...
}
```

If static fields of type parameters were allowed, then the following code would be confused:

```
MobileDevice<Smartphone> phone = new MobileDevice<>();
MobileDevice<Pager> pager = new MobileDevice<>();
MobileDevice<TabletPC> pc = new MobileDevice<>();
```

Because the static field `os` is shared by `phone`, `pager`, and `pc`, what is the actual type of `os`? It cannot be `Smartphone`, `Pager`, and `TabletPC` at the same time. You cannot, therefore, create static fields of type parameters.

Cannot Use Casts or instanceof with Parameterized Types

Because the Java compiler erases all type parameters in generic code, you cannot verify which parameterized type for a generic type is being used at runtime:

```
public static <E> void rtti(List<E> list) {
    if (list instanceof ArrayList<Integer>) {  // compile-time error
        // ...
    }
}
```

The set of parameterized types passed to the `rtti` method is:

```
S = { ArrayList<Integer>, ArrayList<String> LinkedList<Character>, ... }
```

The runtime does not keep track of type parameters, so it cannot tell the difference between an `ArrayList<Integer>` and an `ArrayList<String>`. The most you can do is to use an unbounded wildcard to verify that the list is an `ArrayList`:

```
public static void rtti(List<?> list) {
    if (list instanceof ArrayList<?>) {  // OK; instanceof requires a reifiable type
        // ...
    }
}
```

Typically, you cannot cast to a parameterized type unless it is parameterized by unbounded wildcards. For example:

```
List<Integer> li = new ArrayList<>();
List<Number>  ln = (List<Number>) li;  // compile-time error
```

However, in some cases the compiler knows that a type parameter is always valid and allows the cast. For example:

```
List<String> l1 = ...;
ArrayList<String> l2 = (ArrayList<String>)l1;  // OK
```

Cannot Create Arrays of Parameterized Types

You cannot create arrays of parameterized types. For example, the following code does not compile:

```
List<Integer>[] arrayOfLists = new List<Integer>[2];  // compile-time error
```

The following code illustrates what happens when different types are inserted into an array:

```
Object[] strings = new String[2];
strings[0] = "hi";   // OK
strings[1] = 100;    // An ArrayStoreException is thrown.
```

If you try the same thing with a generic list, there would be a problem:

```
Object[] stringLists = new List<String>[];  // compiler error, but pretend it's allowed
stringLists[0] = new ArrayList<String>();   // OK
stringLists[1] = new ArrayList<Integer>();  // An ArrayStoreException should be thrown,
```

```
                                        // but the runtime can't detect it.
```

If arrays of parameterized lists were allowed, the previous code would fail to throw the desired `ArrayStoreException`.

Cannot Create, Catch, or Throw Objects of Parameterized Types

A generic class cannot extend the `Throwable` class directly or indirectly. For example, the following classes will not compile:

```
// Extends Throwable indirectly
class MathException<T> extends Exception { /* ... */ }    // compile-time error

// Extends Throwable directly
class QueueFullException<T> extends Throwable { /* ... */ // compile-time error
```

A method cannot catch an instance of a type parameter:

```
public static <T extends Exception, J> void execute(List<J> jobs) {
    try {
        for (J job : jobs)
            // ...
    } catch (T e) {   // compile-time error
        // ...
    }
}
```

You can, however, use a type parameter in a `throws` clause:

```
class Parser<T extends Exception> {
    public void parse(File file) throws T {     // OK
        // ...
    }
}
```

Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

A class cannot have two overloaded methods that will have the same signature after type erasure.

```
public class Example {
    public void print(Set<String> strSet) { }
    public void print(Set<Integer> intSet) { }
}
```

The overloads would all share the same classfile representation and will generate a compile-time error.

Appendix 4 Packages

To make types easier to find and use, to avoid naming conflicts, and to control access, programmers bundle groups of related types into packages.

Definition: A *package* is a grouping of related types providing access protection and name space management. Note that *types* refers to classes, interfaces, enumerations, and annotation types. Enumerations and annotation types are special kinds of classes and interfaces, respectively, so *types* are often referred to in this lesson simply as *classes and interfaces*.

The types that are part of the Java platform are members of various packages that bundle classes by function: fundamental classes are in `java.lang`, classes for reading and writing (input and output) are in `java.io`, and so on. You can put your types in packages too.

You should bundle these classes and the interface in a package for several reasons, including the following:

- **You and other programmers can easily determine that these types are related.**
- **You and other programmers know where to find types that can provide graphics-related functions.**
- **The names of your types won't conflict with the type names in other packages because the package creates a new namespace.**
- **You can allow types within the package to have unrestricted access to one another yet still restrict access for types outside the package.**

Creating a Package

To create a package, you choose a name for the package (naming conventions are discussed in the next section) and put a `package` statement with that name at the top of *every source file* that contains the types (classes, interfaces, enumerations, and annotation types) that you want to include in the package.

The package statement (for example, `package graphics;`) must be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

Note: If you put multiple types in a single source file, only one can be `public`, and it must have the same name as the source file. For example, you can define `public class Circle` in the file `Circle.java`, define `public interface Draggable` in the file `Draggable.java`, define `public enum Day` in the file `Day.java`, and so forth.

You can include non-public types in the same file as a public type (this is strongly discouraged, unless the non-public types are small and closely related to the public type), but only the public type will be accessible from outside of the package. All the top-level, non-public types will be *package private*.

If you do not use a `package` statement, your type ends up in an unnamed package. Generally speaking, an unnamed package is only for small or temporary applications or when you are just beginning the development process. Otherwise, classes and interfaces belong in named packages.

Naming a Package

With programmers worldwide writing classes and interfaces using the Java programming language, it is likely that many programmers will use the same name for different types. In fact, the previous example does just that: It defines a `Rectangle` class when there is already a `Rectangle` class in the `java.awt` package. Still, the compiler allows both classes to have the same name if they are in different packages. The fully qualified name of each `Rectangle` class includes the package name. That is, the fully qualified name of the `Rectangle` class in the `graphics` package is

graphics.Rectangle, and the fully qualified name of the Rectangle class in the java.awt package is java.awt.Rectangle.

This works well unless two independent programmers use the same name for their packages. What prevents this problem? Convention.

Naming Conventions

Package names are written in all lower case to avoid conflict with the names of classes or interfaces.

Companies use their reversed Internet domain name to begin their package names—for example, com.example.mypackage for a package named mypackage created by a programmer at example.com.

Name collisions that occur within a single company need to be handled by convention within that company, perhaps by including the region or the project name after the company name (for example, com.example.region.mypackage).

Packages in the Java language itself begin with java. or javax.

In some cases, the internet domain name may not be a valid package name. This can occur if the domain name contains a hyphen or other special character, if the package name begins with a digit or other character that is illegal to use as the beginning of a Java name, or if the package name contains a reserved Java keyword, such as "int". In this event, the suggested convention is to add an underscore. For example:

Legalizing Package Names	
Domain Name	**Package Name Prefix**
hyphenated-name.example.org	org.example.hyphenated_name
example.int	int_.example
123name.example.com	com.example._123name

Using Package Members

The types that comprise a package are known as the *package members*.

To use a public package member from outside its package, you must do one of the following:

- **Refer to the member by its fully qualified name**
- **Import the package member**
- **Import the member's entire package**

Each is appropriate for different situations, as explained in the sections that follow.

Referring to a Package Member by Its Qualified Name

So far, most of the examples in this tutorial have referred to types by their simple names, such as Rectangle and StackOfInts. You can use a package member's simple name if the code you are writing is in the same package as that member or if that member has been imported.

However, if you are trying to use a member from a different package and that package has not been imported, you must use the member's fully qualified name, which includes the package name. Here is the fully qualified name for the Rectangle class declared in the graphics package in the previous example.

```
graphics.Rectangle
```

You could use this qualified name to create an instance of `graphics.Rectangle`:

```
graphics.Rectangle myRect = new graphics.Rectangle();
```

Qualified names are all right for infrequent use. When a name is used repetitively, however, typing the name repeatedly becomes tedious and the code becomes difficult to read. As an alternative, you can *import* the member or its package and then use its simple name.

Importing a Package Member

To import a specific member into the current file, put an `import` statement at the beginning of the file before any type definitions but after the `package` statement, if there is one. Here's how you would import the `Rectangle` class from the `graphics` package created in the previous section.

```
import graphics.Rectangle;
```

Now you can refer to the `Rectangle` class by its simple name.

```
Rectangle myRectangle = new Rectangle();
```

This approach works well if you use just a few members from the `graphics` package. But if you use many types from a package, you should import the entire package.

Importing an Entire Package

To import all the types contained in a particular package, use the `import` statement with the asterisk `(*)` wildcard character.

```
import graphics.*;
```

Now you can refer to any class or interface in the `graphics` package by its simple name.

```
Circle myCircle = new Circle();
Rectangle myRectangle = new Rectangle();
```

The asterisk in the `import` statement can be used only to specify all the classes within a package, as shown here. It cannot be used to match a subset of the classes in a package. For example, the following does not match all the classes in the `graphics` package that begin with `A`.

```
// does not work
import graphics.A*;
```

Instead, it generates a compiler error. With the `import` statement, you generally import only a single package member or an entire package.

Note: Another, less common form of `import` allows you to import the public nested classes of an enclosing class. For example, if the `graphics.Rectangle` class contained useful nested classes, such as `Rectangle.DoubleWide` and `Rectangle.Square`, you could import `Rectangle` and its nested classes by using the following *two* statements.

```
import graphics.Rectangle;
import graphics.Rectangle.*;
```

Be aware that the second import statement will *not* import `Rectangle`.

For convenience, the Java compiler automatically imports two entire packages for each source file: (1) the `java.lang` package and (2) the current package (the package for the current file).

Apparent Hierarchies of Packages

At first, packages appear to be hierarchical, but they are not. For example, the Java API includes a `java.awt` package, a `java.awt.color` package, a `java.awt.font` package, and many others that begin with `java.awt`. However, the `java.awt.color` package, the `java.awt.font` package, and other `java.awt.xxxx` packages are *not included* in the `java.awt` package. The prefix `java.awt` (the Java Abstract Window Toolkit) is used for a number of related packages to make the relationship evident, but not to show inclusion.

Importing `java.awt.*` imports all of the types in the `java.awt` package, but it *does not import* `java.awt.color`, `java.awt.font`, or any other `java.awt.xxxx` packages. If you plan to use the classes and other types in `java.awt.color` as well as those in `java.awt`, you must import both packages with all their files:

```
import java.awt.*;
import java.awt.color.*;
```

Name Ambiguities

If a member in one package shares its name with a member in another package and both packages are imported, you must refer to each member by its qualified name. For example, the `graphics` package defined a class named `Rectangle`. The `java.awt` package also contains a `Rectangle` class. If both `graphics` and `java.awt` have been imported, the following is ambiguous.

```
Rectangle rect;
```

In such a situation, you have to use the member's fully qualified name to indicate exactly which `Rectangle` class you want. For example,

```
graphics.Rectangle rect;
```

The Static Import Statement

There are situations where you need frequent access to static final fields (constants) and static methods from one or two classes. Prefixing the name of these classes over and over can result in cluttered code. The *static import* statement gives you a way to import the constants and static methods that you want to use so that you do not need to prefix the name of their class.

The `java.lang.Math` class defines the `PI` constant and many static methods, including methods for calculating sines, cosines, tangents, square roots, maxima, minima, exponents, and many more. For example,

```
public static final double PI
    = 3.141592653589793;
public static double cos(double a)
{
    ...
}
```

Ordinarily, to use these objects from another class, you prefix the class name, as follows.

```
double r = Math.cos(Math.PI * theta);
```

You can use the static import statement to import the static members of java.lang.Math so that you don't need to prefix the class name, `Math`. The static members of `Math` can be imported either individually:

```
import static java.lang.Math.PI;
```

or as a group:

```
import static java.lang.Math.*;
```

Once they have been imported, the static members can be used without qualification. For example, the previous code snippet would become:

```
double r = cos(PI * theta);
```

Obviously, you can write your own classes that contain constants and static methods that you use frequently, and then use the static import statement. For example,

```
import static mypackage.MyConstants.*;
```

Note: Use static import very sparingly. Overusing static import can result in code that is difficult to read and maintain, because readers of the code won't know which class defines a particular static object. Used properly, static import makes code more readable by removing class name repetition.

Managing Source and Class Files

Many implementations of the Java platform rely on hierarchical file systems to manage source and class files, although *The Java Language Specification* does not require this. The strategy is as follows.

Put the source code for a class, interface, enumeration, or annotation type in a text file whose name is the simple name of the type and whose extension is `.java`. For example:

```
//in the Rectangle.java file
package graphics;
public class Rectangle {
    ...
}
```

Then, put the source file in a directory whose name reflects the name of the package to which the type belongs:

```
.....\graphics\Rectangle.java
```

The qualified name of the package member and the path name to the file are parallel, assuming the Microsoft Windows file name separator backslash (for UNIX, use the forward slash).

- **class name – `graphics.Rectangle`**
- **pathname to file – `graphics\Rectangle.java`**

As you should recall, by convention a company uses its reversed Internet domain name for its package names. The Example company, whose Internet domain name is `example.com`, would precede all its package names with `com.example`. Each component of the package name corresponds to a subdirectory. So, if the Example company had a `com.example.graphics` package that contained a `Rectangle.java` source file, it would be contained in a series of subdirectories like this:

```
....\com\example\graphics\Rectangle.java
```

When you compile a source file, the compiler creates a different output file for each type defined in it. The base name of the output file is the name of the type, and its extension is `.class`. For example, if the source file is like this

```
//in the Rectangle.java file
package com.example.graphics;
public class Rectangle {
```

```
        . . .
}

class Helper{
        . . .
}
```

then the compiled files will be located at:

```
<path to the parent directory of the output files>\com\example\graphics\Rectangle.class
<path to the parent directory of the output files>\com\example\graphics\Helper.class
```

Like the `.java` source files, the compiled `.class` files should be in a series of directories that reflect the package name. However, the path to the `.class` files does not have to be the same as the path to the `.java` source files. You can arrange your source and class directories separately, as:

```
<path_one>\sources\com\example\graphics\Rectangle.java

<path_two>\classes\com\example\graphics\Rectangle.class
```

By doing this, you can give the `classes` directory to other programmers without revealing your sources. You also need to manage source and class files in this manner so that the compiler and the Java Virtual Machine (JVM) can find all the types your program uses.

The full path to the `classes` directory, `<path_two>\classes`, is called the *class path*, and is set with the `CLASSPATH` system variable. Both the compiler and the JVM construct the path to your `.class` files by adding the package name to the class path. For example, if

```
<path_two>\classes
```

is your class path, and the package name is

```
com.example.graphics,
```

then the compiler and JVM look for `.class files` in

```
<path_two>\classes\com\example\graphics.
```

A class path may include several paths, separated by a semicolon (Windows) or colon (UNIX). By default, the compiler and the JVM search the current directory and the JAR file containing the Java platform classes so that these directories are automatically in your class path.

Setting the CLASSPATH System Variable

To display the current `CLASSPATH` variable, use these commands in Windows and UNIX (Bourne shell):

```
In Windows:   C:\> set CLASSPATH
In UNIX:      % echo $CLASSPATH
```

To delete the current contents of the `CLASSPATH` variable, use these commands:

```
In Windows:   C:\> set CLASSPATH=
In UNIX:      % unset CLASSPATH; export CLASSPATH
```

To set the `CLASSPATH` variable, use these commands (for example):

```
In Windows:   C:\> set CLASSPATH=C:\users\george\java\classes
```
In UNIX: % CLASSPATH=/home/george/java/classes; export CLASSPATH

Using Packages For Introduction to Java Programming

Packages can be used to organize classes. Each class in the Java API belongs to a package that contains a group of related classes. These packages are defined once, but can be imported into many programs. As applications become more complex, packages help you manage the complexity of application components. Packages also facilitate software reuse by enabling programs to *import* classes from other packages, rather than *copying* the classes into each program that uses them. Another benefit of packages is that they provide a convention for unique class names, which helps prevent class-name conflicts.

To do so, you need to add the following line as the first noncomment and nonblank statement in the program:

package packagename;

The following code gives a program that places class Welcome in package chapter0.

Welcome.java

```
/** Use package for the class */
package chapter0;

public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

This code is identical to code in the text except that the Welcome class in that code is placed in package chapter0. A package corresponds to a directory. You need to create a directory named chapter0 and place Welcome.java in the directory. If you use an IDE such as NetBeans and Eclipse, the directory is automatically created. Suppose all source code in chapter*i* are placed in the directory chapter*i* in this text.

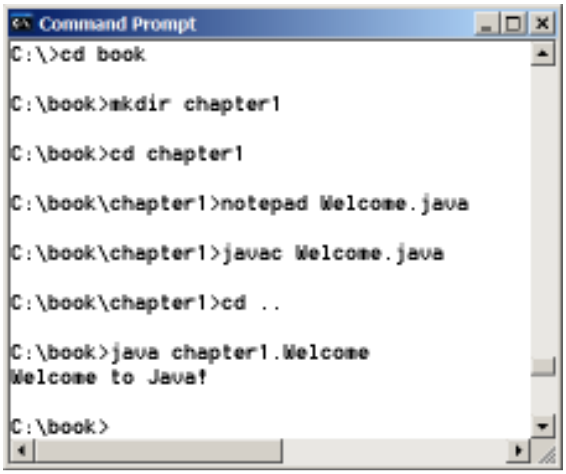
The root directory where the .class files (including the packages) are stored is known as the *classpath* directory.

NOTE
To compile and run programs from the command window rather using an IDE, you need to know at least two DOS commands: mkdir and cd.

- **mkdir dirName -- Creates a new directory named dirName.**
- **cd dirName -- Changes to the specified directory.**
- **cd .. -- Changes to the parent directory.**

For example, cd c:\book changes to the directory c:\book.

To compile Welcome.java from the command window, change the directory to chapter0, and type javac Welcome.java. To run the class, change to the classpath directory, and type java chapter1.Welcome.



NOTE
If a class is defined without the package statement, the class is said to be placed in the *default package*. The Welcome class in the text is placed in the default package.

Steps for Declaring a Reusable Class

Before a class can be imported into multiple applications, it must be placed in a package to make it reusable. The steps for creating a reusable class are:

1. **Declare a public class. If the class is not public, it can be used only by other classes in the same package.**
2. **Choose a unique package name and add a package declaration to the source-code file for the reusable class declaration. In each Java source-code file there can be only one package declaration, and it must precede all other declarations and statements. Comments are not statements, so comments can be placed before a package statement in a file. *Note:* If no package statement is provided, the class is placed in the so-called default package and is accessible only to other classes in the default package that are located in the same directory.]**
3. **Compile the class so that it's placed in the appropriate package directory.**
4. **Import the reusable class into a program and use the class.**

Steps 1 and 2: Creating a* public *Class and Adding the* package *Statement

```
package a.b.c.d;

public class Class1
{
    . . .
}
```

Placing a package declaration at the beginning of a Java source file indicates that the class declared in the file is part of the specified package (`a.b.c.d`). Only package declarations, import declarations and comments can appear outside the braces of a class declaration. A Java source-code file must have the following order:

1. **a package declaration (if any),**
2. **import declarations (if any), then**
3. **class declarations.**

Only one of the class declarations in a particular file can be public. Other classes in the file are placed in the package and can be used only by the other classes in the package.

Nonpublic classes are in a package to support the reusable classes in the package.
To provide unique package names, start each one with your Internet domain name in reverse order.

Step 3: Compiling the Packaged Class
***Step 3* is to compile the class so that it's stored in the appropriate package. When a Java file containing a package declaration is compiled, the resulting class file is placed in the directory specified by the declaration. The package declaration indicates that `Class1` should be placed in the directory `a\b\c\d`.**

The names in the package declaration specify the exact location of the package's classes. When compiling a class in a package, the javac command-line option -d causes the javac compiler to create appropriate directories based on the class's package declaration. The option also specifies where the directories should be stored. For example, in a command window, we used the compilation command
javac -d . Class1.java
to specify that the first directory in our package name should be placed in the current directory. The dot (.) after -d in the preceding command represents the current directory on the Windows. After execution of the compilation command, the current directory contains a directory called `a`, `a` contains a directory called `b`, `b` contains a directory called `c` and `c` contains a directory called `d`. In the `d` directory, you can find the file *Class1.class*. *Note:* If you do not use the -d option, then you must copy or move the class file to the appropriate package directory after compiling it.
The package name is part of the fully qualified class name, so the name of class `Class1` is actually `a.b.c.d.Class1`. You can use this fully qualified name in your programs, or you can import the class and use its simple name (the class name by itself in the program. If another package also contains a `Class1` class, the fully qualified class names can be used to distinguish between the classes in the program and prevent a name conflict (also called a name collision).

Step 4: Importing the Reusable Class
Once it's compiled and stored in its package, the class can be imported into programs (*Step 4*).

```
import a.b.c.d.Class1;

public class Class2
{
    Class1 c1 = new Class1();  //invokes class1 constructor
    . . .
}
```

The first line specifies that class `Class1` should be imported for use in class `Class2`. This class is in the default package because its .java file does not contain a package declaration. Since the two classes are in different packages, the import statement is required so that class `Class2` can use class `Class1`.

Note:

- **single-type-import declaration: that is, the import declaration specifies one class to import (`import a.b.c.d.Class1;`).**
- **type-import-ondemand declaration When your program uses multiple classes from the same package, you can import those classes with a single import declaration. For example, the import declaration import java.util.*; uses an asterisk (*) at its end to inform the compiler that all public classes from the java.util package are available for use in the program.**

Specifying the Classpath During Compilation
When compiling `Class2`, javac must locate the .class file for `Class1` to ensure that class `Class2` uses class `Class1` correctly. The compiler uses a special object called a class loader to locate the classes it needs. The class loader begins by searching the standard Java classes that are bundled with the JDK. Then it searches for optional packages. Java provides an extension mechanism that enables new (optional) packages to be added to Java for development and execution purposes. If the class is not found in the standard Java classes or in the extension

classes, the class loader searches the classpath, which contains a list of locations in which classes are stored. The classpath consists of a list of directories or archive files, each separated by a directory separator, a semicolon (;) on Windows or a colon (:) on UNIX/Linux/Mac OS X. Archive files are individual files that contain directories of other files, typically in a compressed format. For example, the standard classes used by your programs are contained in the archive file rt.jar, which is installed with the JDK. Archive files normally end with the .jar or .zip file-name extensions. The directories and archive files specified in the classpath contain the classes you wish to make available to the Java compiler and the JVM.
By default, the classpath consists only of the current directory (.). However, the classpath can be modified by:

1. **providing the -classpath option to the javac compiler or**
2. **setting the CLASSPATH environment variable (a special variable that you define and the operating system maintains so that applications can search for classes in the specified locations).**

Note:
Specifying an explicit classpath eliminates the current directory from the classpath. This prevents classes in the current directory (including packages in the current directory) from loading properly. If classes must be loaded from the current directory, include a dot (.) in the classpath to specify the current directory.
In general, it's a better practice to use the -classpath option of the compiler, rather than the CLASSPATH environment variable, to specify the classpath for a program. This enables each application to have its own classpath. Specifying the classpath with the CLASSPATH environment variable can cause subtle and difficult-to-locate errors in programs that use different versions of the same package.

In the example, we didn't specify an explicit classpath. Thus, to locate the classes in the `a.b.c.d` package from this example, the class loader looks in the current directory for the first name in the package (`a`) then navigates the directory structure.
Directory `a` contains the subdirectory `b`, `b` contains the subdirectory `c`, and `c` contains subdirectory `d`. In the `d` directory is the file `Class1`.class, which is loaded by the class loader to ensure that the class is used properly in our program.

Specifying the Classpath When Executing an Application
When you execute an application, the JVM must be able to locate the .class files of the classes used in that application. Like the compiler, the java command uses a class loader that searches the standard classes and extension classes first, then searches the classpath (the current directory by default). The classpath can be specified explicitly by using either of the techniques discussed for the compiler. As with the compiler, it's better to specify an individual program's classpath via command-line JVM options. You can specify the classpath in the java command via the -classpath or -cp command-line options, followed by a list of directories or archive files separated by semicolons (;) on Microsoft Windows or by colons (:) on UNIX/Linux/Mac OS X. Again, if classes must be loaded from the current directory, be sure to include a dot (.) in the classpath to specify the current directory.

Package Access
If no access modifier (public, protected or private) is specified for a method or variable when it's declared in a class, the method or variable is considered to have package access. In a program that consists of one class declaration, this has no specific effect. However, if a program uses multiple classes from the same package (i.e., a group of related classes), these classes can access each other's package-access members directly through references to objects of the appropriate classes, or in the case of static members through the class name. Package access is rarely used.

Appendix 5 NetBeans IDE

Creating Your First Application (NetBeans IDE)

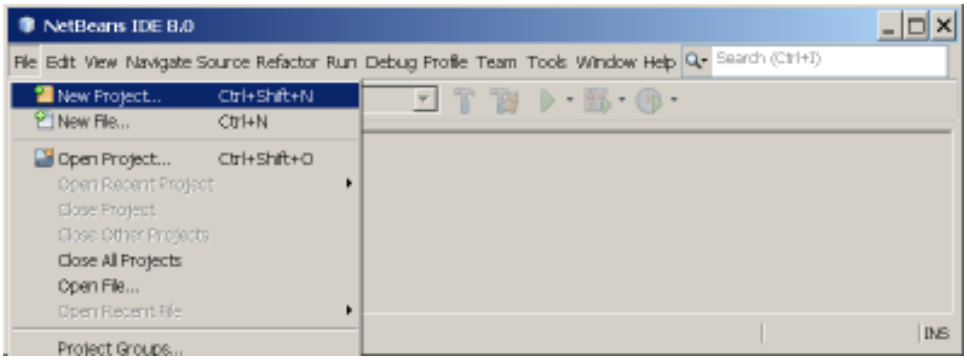
Your first application, HelloWorldApp, will simply display the greeting "Hello World!" To create this program, you will:

- **Create an IDE project**
  When you create an IDE project, you create an environment in which to build and run your applications. Using IDE projects eliminates configuration issues normally associated with developing on the command line. You can build or run your application by choosing a single menu item within the IDE.
- **Add code to the generated source file**
  A source file contains code, written in the Java programming language, that you and other programmers can understand. As part of creating an IDE project, a skeleton source file will be automatically generated. You will then modify the source file to add the "Hello World!" message.
- **Compile the source file into a .class file**
  The IDE invokes the Java programming language *compiler* (javac), which takes your source file and translates its text into instructions that the Java virtual machine can understand. The instructions contained within this file are known as *bytecodes*.
- **Run the program**
  The IDE invokes the Java application launcher tool (java), which uses the Java virtual machine to run your application.

Create an IDE Project

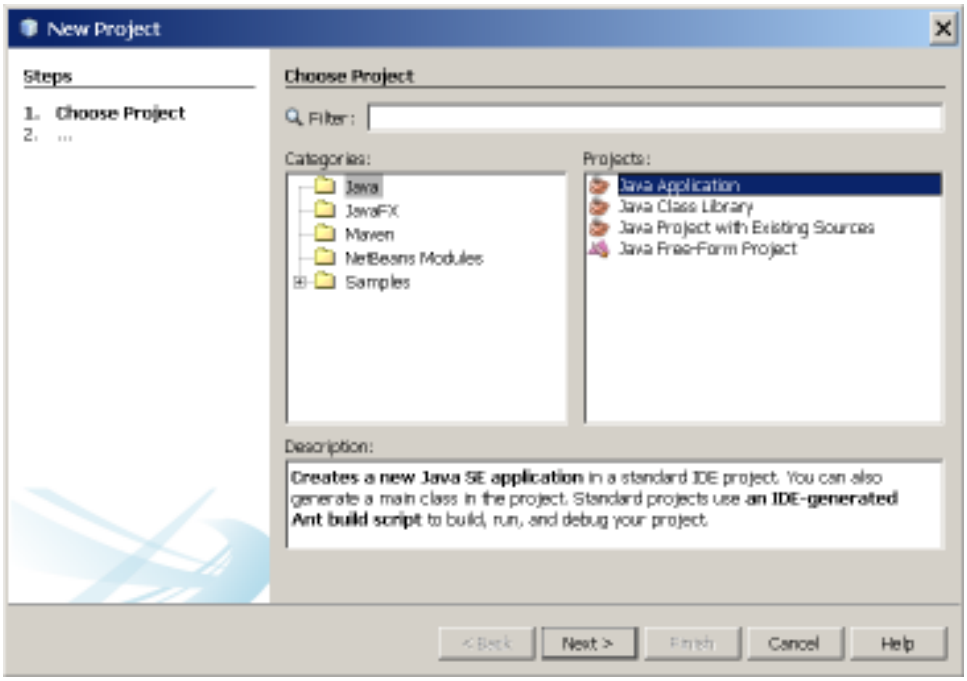
To create an IDE project:

1. **Launch the NetBeans IDE.**
   - **On Microsoft Windows systems, you can use the NetBeans IDE item in the Start menu.**
   - **On Solaris OS and Linux systems, you execute the IDE launcher script by navigating to the IDE's bin directory and typing ./netbeans.**
   - **On Mac OS X systems, click the NetBeans IDE application icon.**
2. **In the NetBeans IDE, choose File | New Project....**



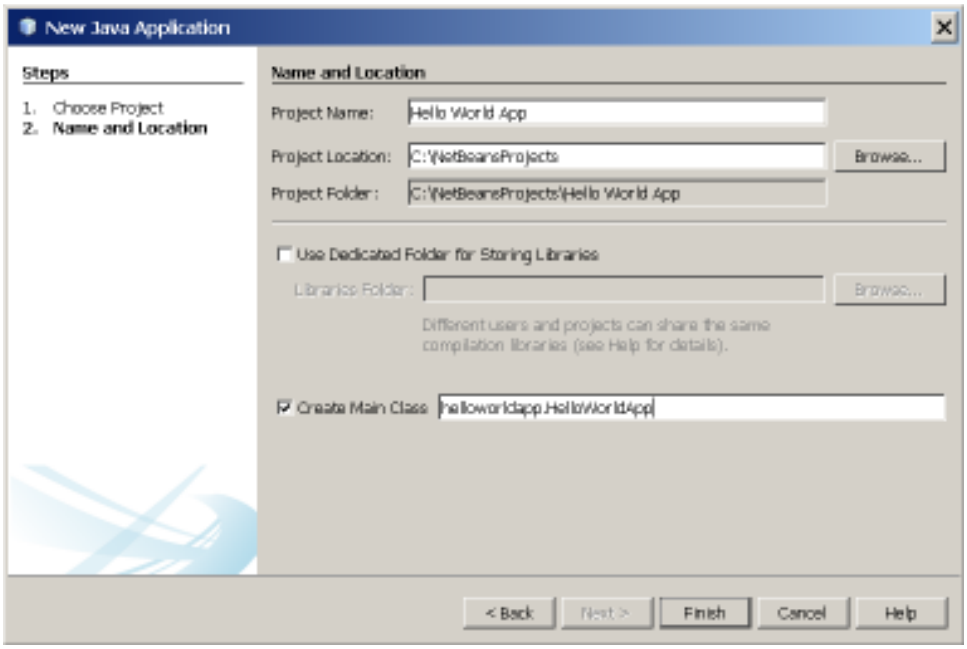
NetBeans IDE with the File | New Project menu item selected.

3. **In the New Project wizard, expand the Java category and select Java Application as shown in the following figure:**



NetBeans IDE, New Project wizard, Choose Project page.

4. **In the Name and Location page of the wizard, do the following (as shown in the figure below):**
   - **In the Project Name field, type Hello World App.**
   - **In the Create Main Class field, type helloworldapp.HelloWorldApp.**

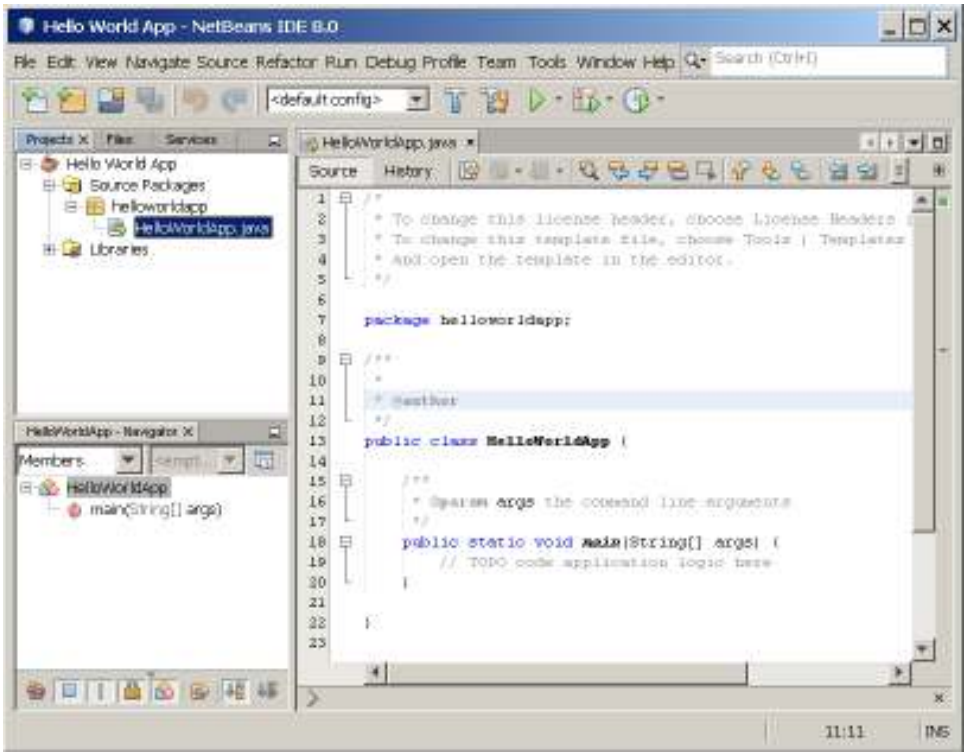


NetBeans IDE, New Project wizard, Name and Location page.

5. **Click Finish.**

The project is created and opened in the IDE. You should see the following components:

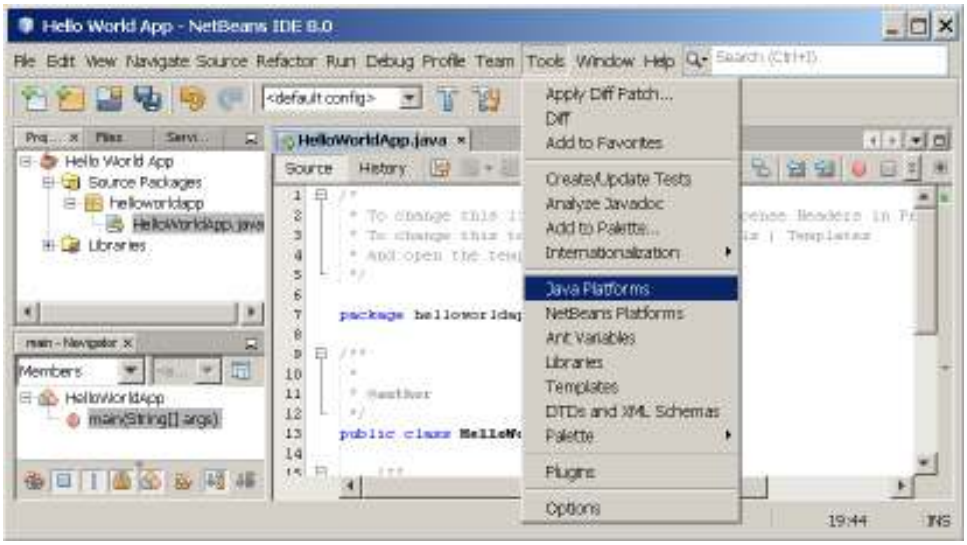
- **The Projects window, which contains a tree view of the components of the project, including source files, libraries that your code depends on, and so on.**
- **The Source Editor window with a file called HelloWorldApp.java open.**
- **The Navigator window, which you can use to quickly navigate between elements within the selected class.**



NetBeans IDE with the HelloWorldApp project open.

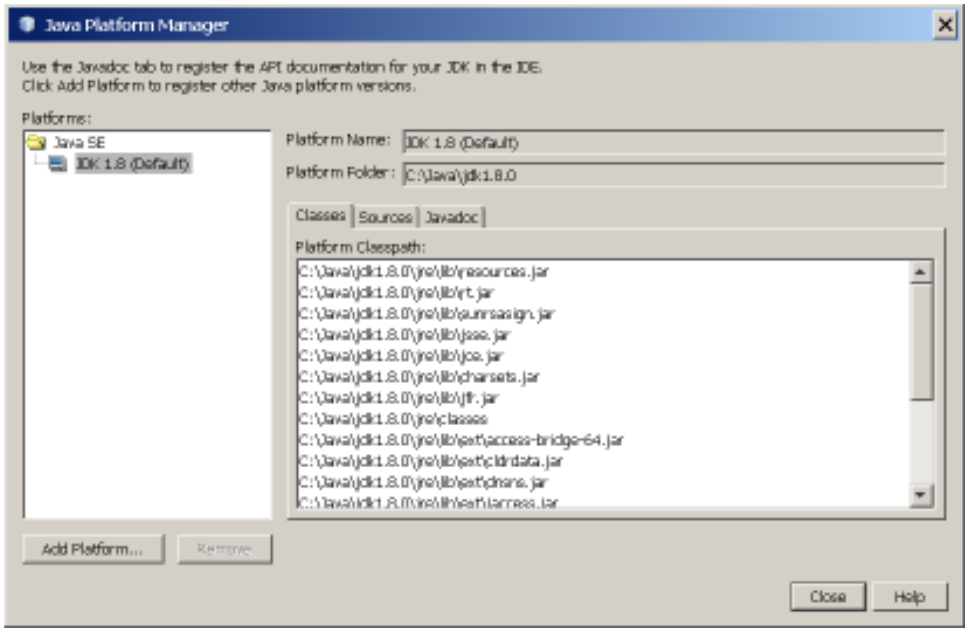
Add JDK 8 to the Platform List (if necessary)

It may be necessary to add JDK 8 to the IDE's list of available platforms. To do this, choose Tools | Java Platforms as shown in the following figure:



Selecting the Java Platform Manager from the Tools Menu

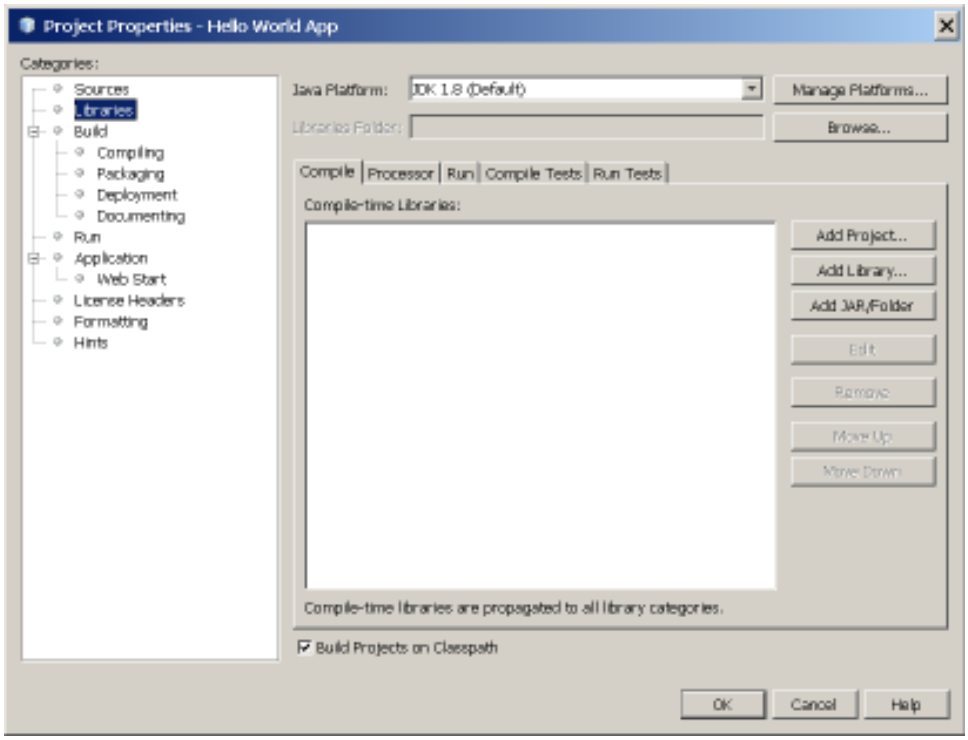
If you don't see JDK 8 (which might appear as 1.8 or 1.8.0) in the list of installed platforms, click Add Platform, navigate to your JDK 8 install directory, and click Finish. You should now see this newly added platform:



The Java Platform Manager

To set this JDK as the default for all projects, you can run the IDE with the --jdkhome switch on the command line, or by entering the path to the JDK in the netbeans_j2sdkhome property of your INSTALLATION_DIRECTORY/etc/netbeans.conf file.

To specify this JDK for the current project only, select Hello World App in the Projects pane, choose File | Project Properties (Hello World App), click Libraries, then select JDK 1.8 in the Java Platform pulldown menu. You should see a screen similar to the following:



The IDE is now configured for JDK 8.

Add Code to the Generated Source File

When you created this project, you left the Create Main Class checkbox selected in the New Project wizard. The IDE has therefore created a skeleton class for you. You can add the "Hello World!" message to the skeleton code by replacing the line:

// TODO code application logic here

with the line:

System.out.println("Hello World!"); // Display the string.

Optionally, you can replace these four lines of generated code:

```
/**
 *
 * @author
 */
```

with these lines:

```
/**
 * The HelloWorldApp class implements an application that
 * simply prints "Hello World!" to standard output.
 */
```

These four lines are a code comment and do not affect how the program runs. Later sections of this tutorial explain the use and format of code comments.

Note: Type all code, commands, and file names exactly as shown. Both the compiler (javac) and launcher (java) are *case-sensitive*, so you must capitalize consistently.
HelloWorldApp is *not* the same as helloworldapp.

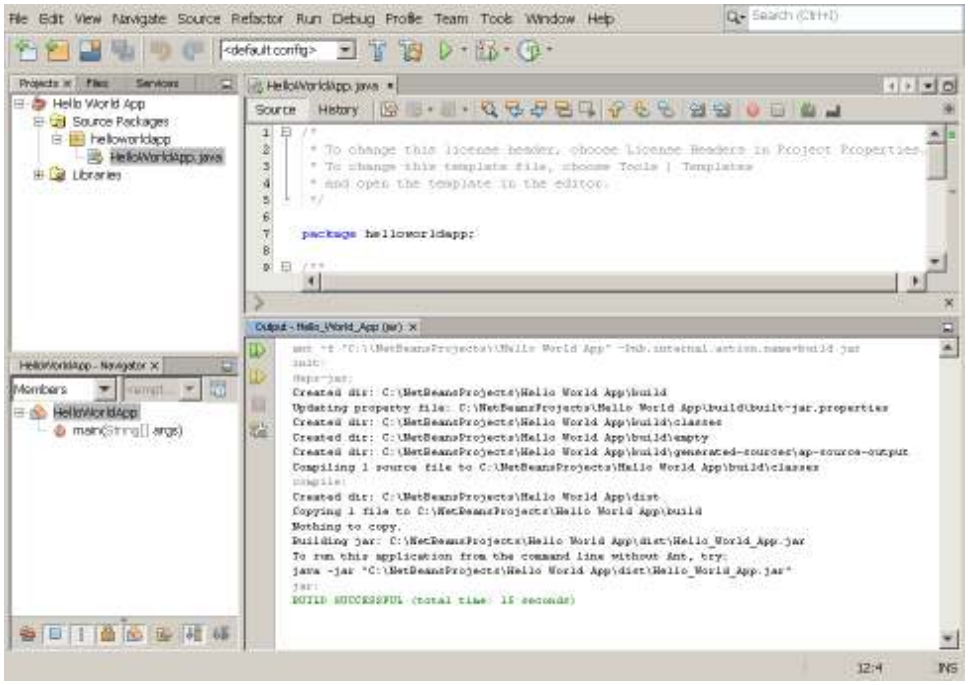
Save your changes by choosing File | Save.

The file should look something like the following:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package helloworldapp;

/**
 * The HelloWorldApp class implements an application that
 * simply prints "Hello World!" to standard output.
 */
public class HelloWorldApp {


    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Display the string.
    }

}
```

Compile the Source File into a .class File

To compile your source file, choose Run | Build Project (Hello World App) from the IDE's main menu. The Output window opens and displays output similar to what you see in the following figure:

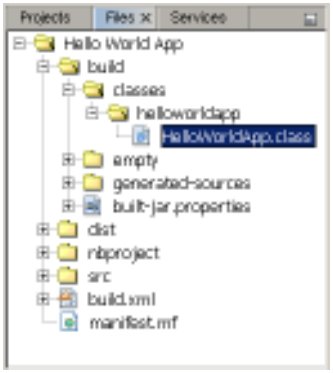


Output window showing results of building the HelloWorld project.

If the build output concludes with the statement BUILD SUCCESSFUL, congratulations! You have successfully compiled your program!

If the build output concludes with the statement BUILD FAILED, you probably have a syntax error in your code. Errors are reported in the Output window as hyperlinked text. You double-click such a hyperlink to navigate to the source of an error. You can then fix the error and once again choose Run | Build Project.

When you build the project, the bytecode file HelloWorldApp.class is generated. You can see where the new file is generated by opening the Files window and expanding the Hello World App/build/classes/helloworldapp node as shown in the following figure.



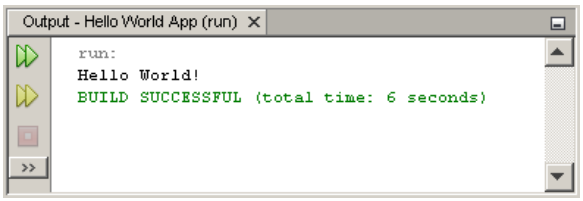
Files window, showing the generated .class file.

Now that you have built the project, you can run your program.

Run the Program

From the IDE's menu bar, choose Run | Run Main Project.

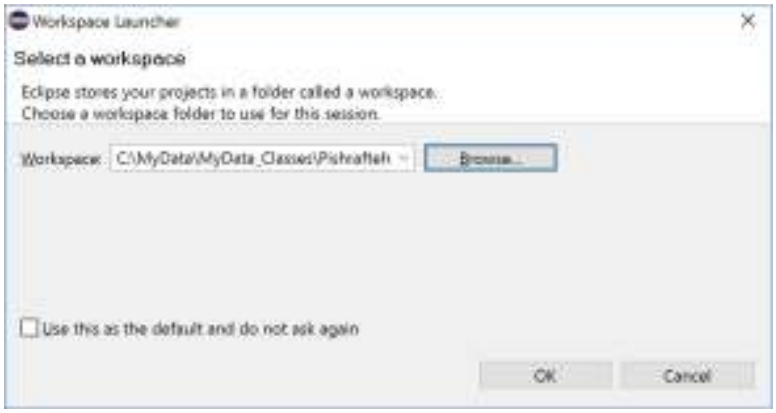
The next figure shows what you should now see.



The program prints "Hello World!" to the Output window (along with other output from the build script).

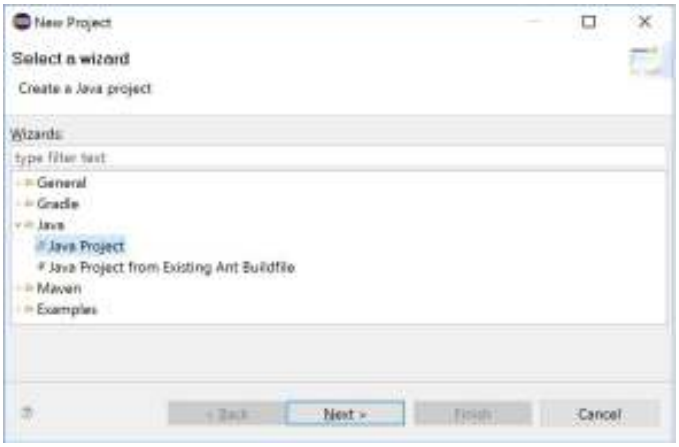
Appendix 6 Eclipse IDE

Select a workspace (Choose a workspace folder)

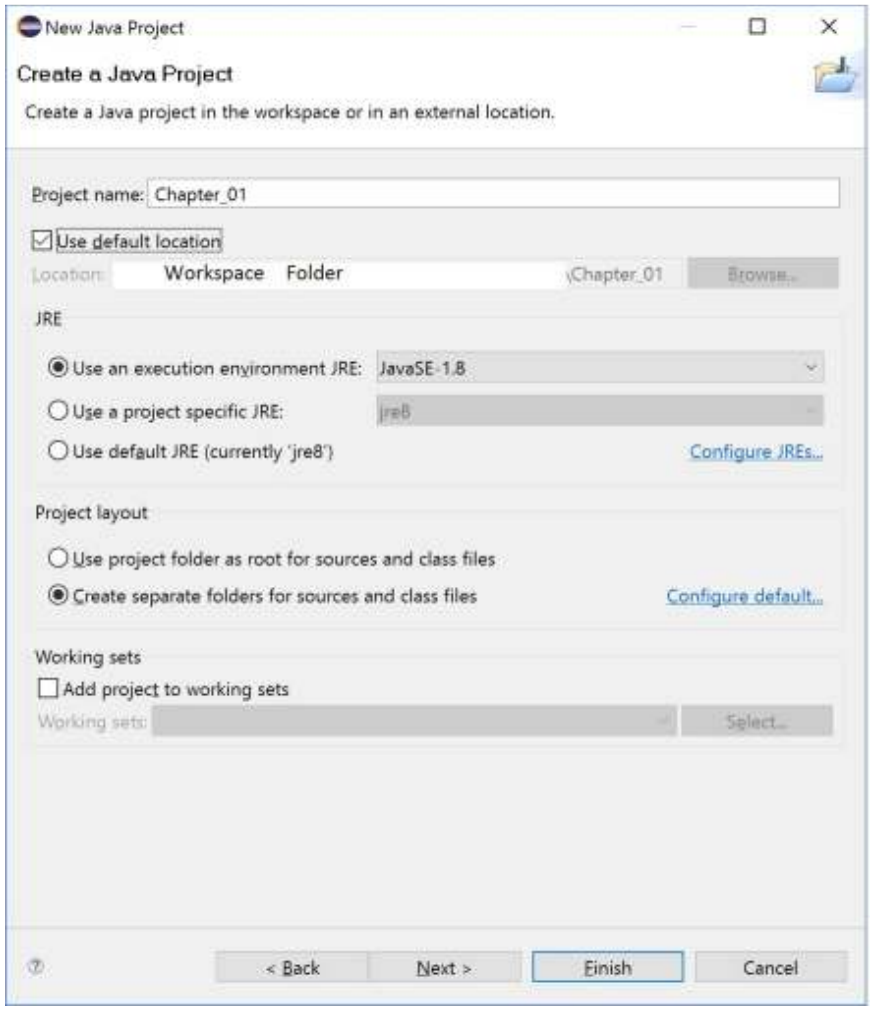


Create a project

Select a wizard (Java project)

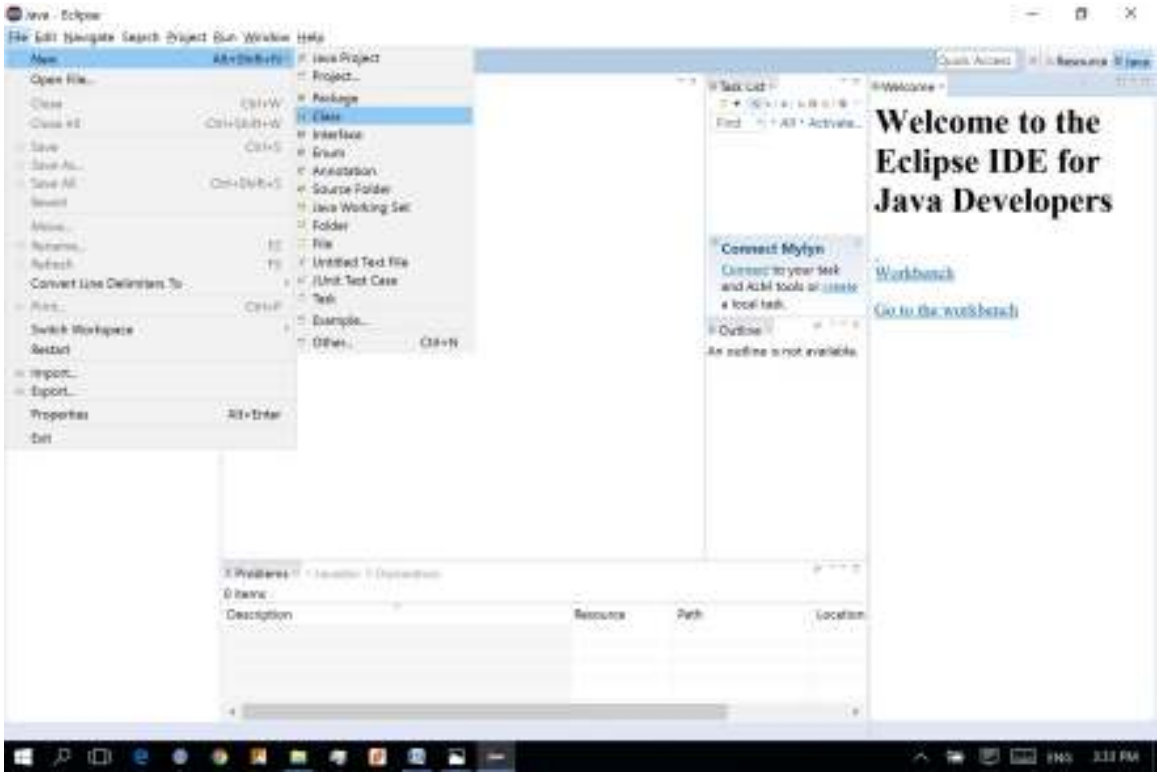


Create a Java project (Use default location: workspace folder)

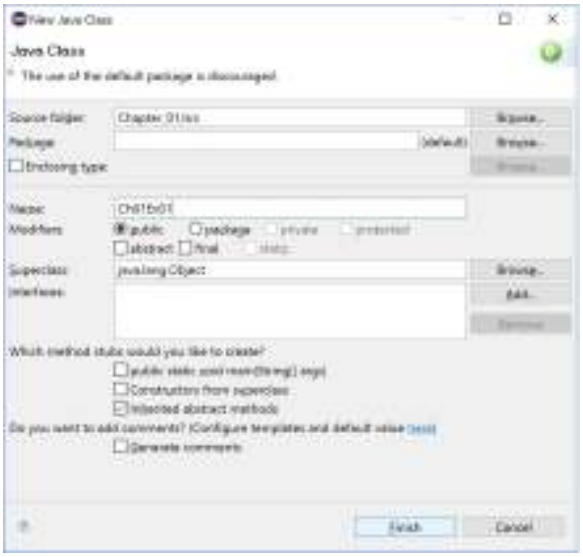


Open Associated Perspective (: Yes)

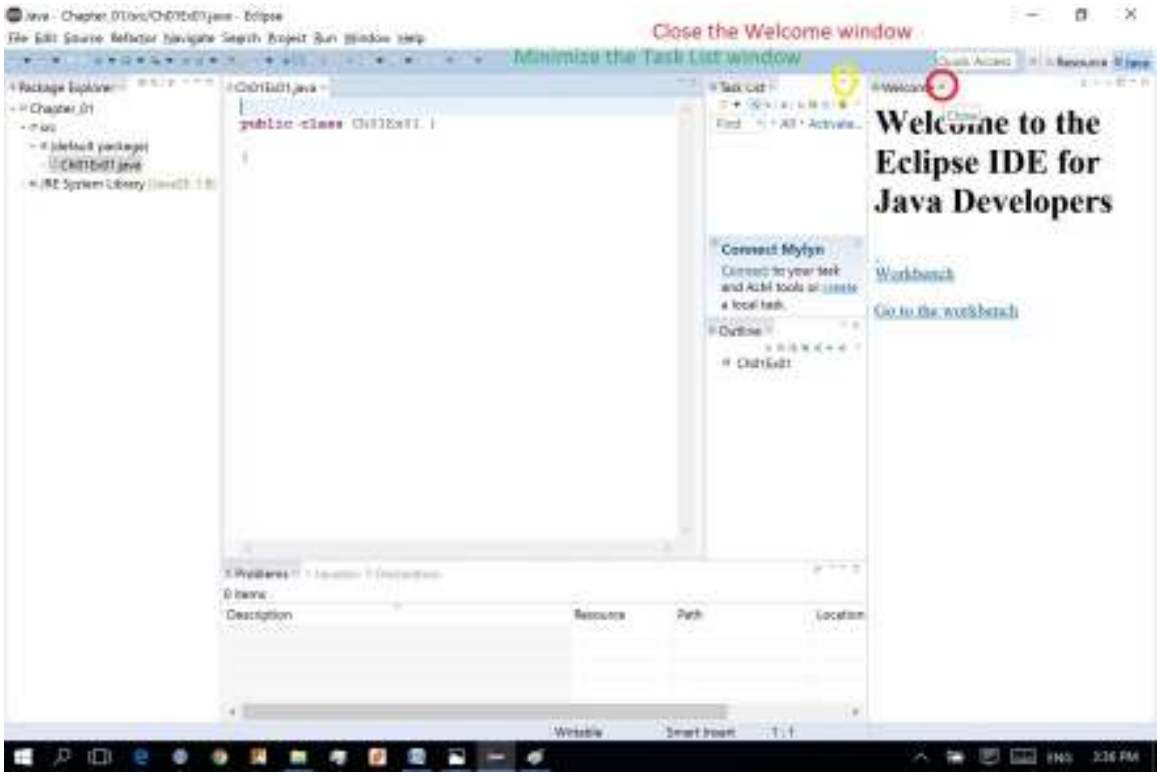
Create new class



Select name for new class (in default package)



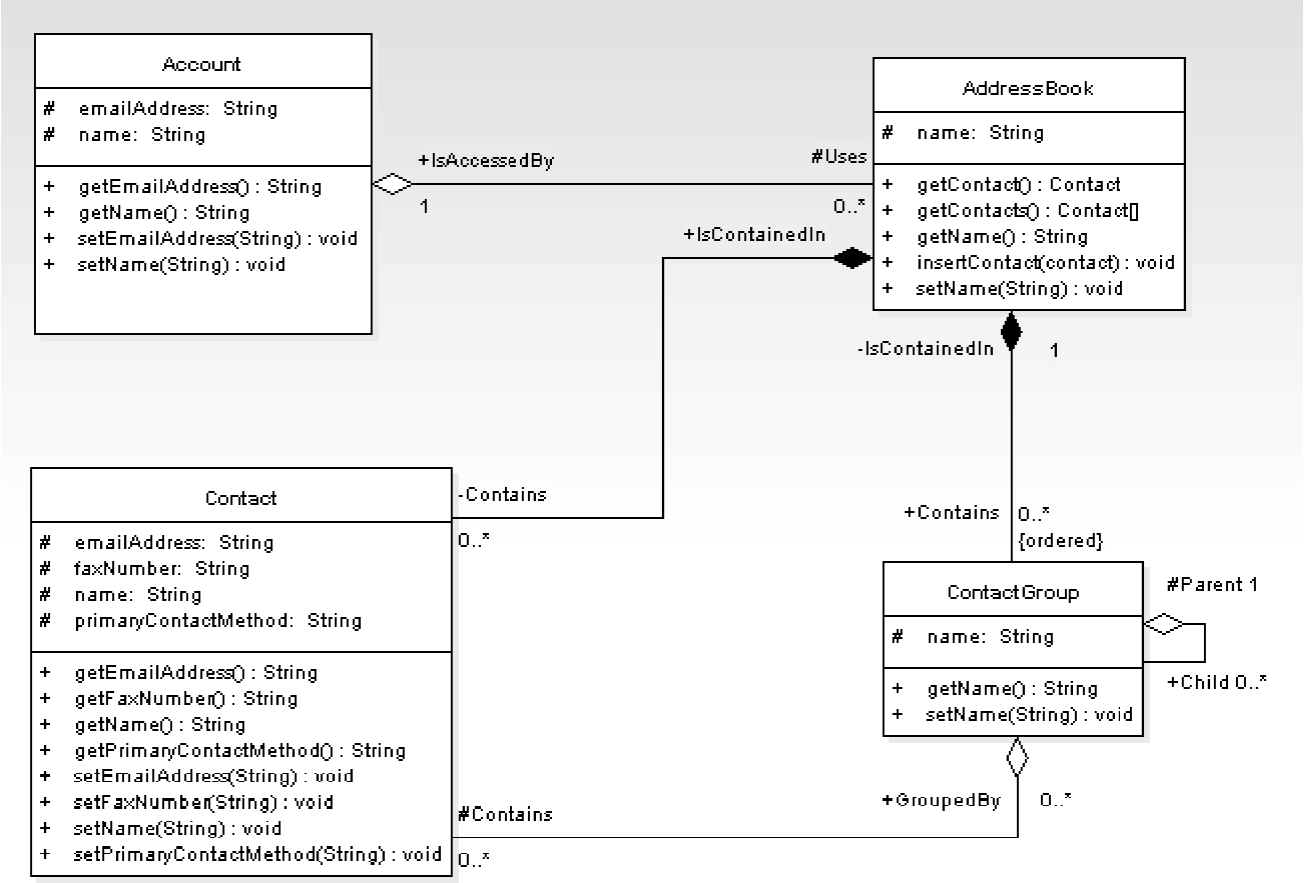
Start coding



Appendix 7 UML 2.0 Class Diagram

The class diagram shows the building blocks of any object-orientated system. Class diagrams depict a static view of the model, or part of the model, describing what attributes and behavior it has rather than detailing the methods for achieving operations. Class diagrams are most useful in illustrating relationships between classes and interfaces. Generalizations, aggregations, and associations are all valuable in reflecting inheritance, composition or usage, and connections respectively.

The diagram below illustrates aggregation relationships between classes. The lighter aggregation indicates that the class "Account" uses AddressBook, but does not necessarily contain an instance of it. The strong, composite aggregations by the other connectors indicate ownership or containment of the source classes by the target classes, for example Contact and ContactGroup values will be contained in AddressBook.



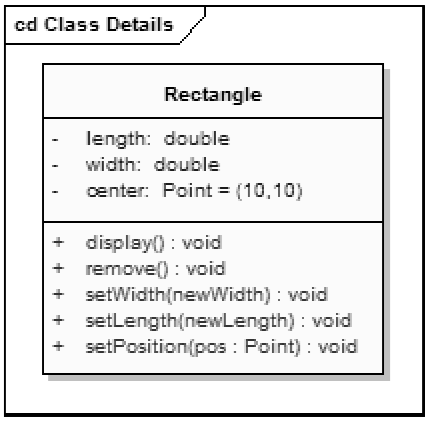
Classes

A class is an element that defines the attributes and behaviors that an object is able to generate. The behavior is described by the possible messages the class is able to understand, along with operations that are appropriate for each message. Classes may also have definitions of constraints, tagged values and stereotypes.

Class Notation

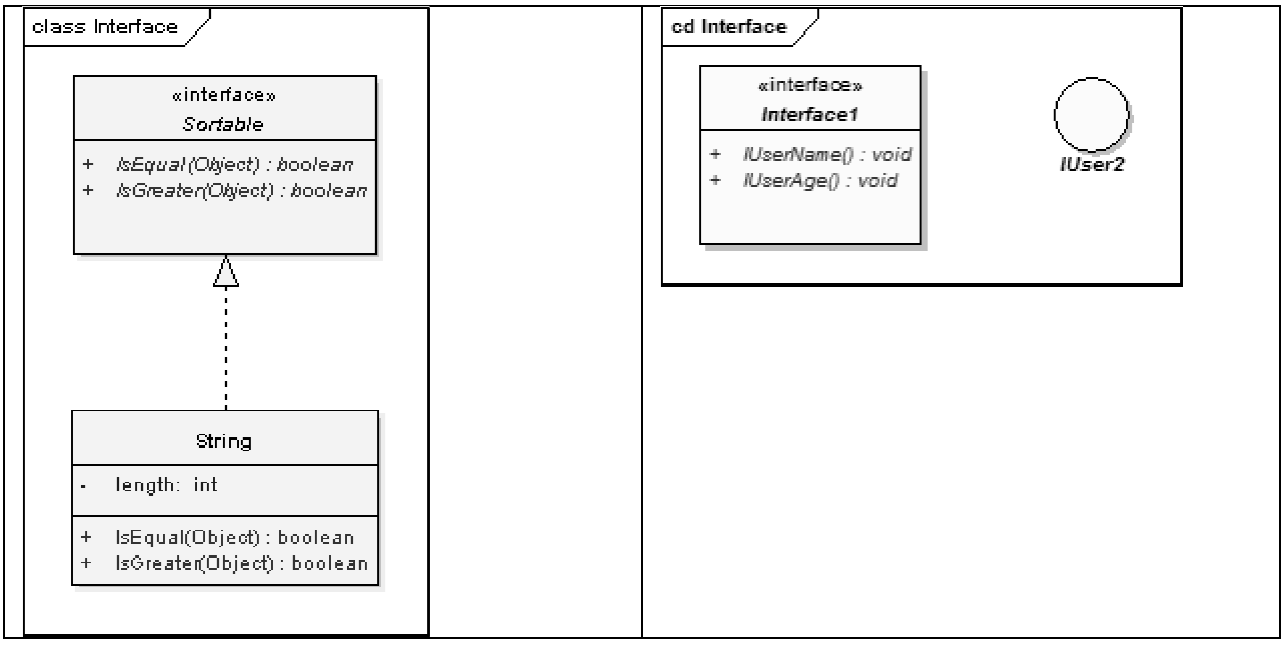
Classes are represented by rectangles which show the name of the class and optionally the name of the operations and attributes. Compartments are used to divide the class name, attributes and operations.

In the diagram below the class contains the class name in the topmost compartment, the next compartment details the attributes, with the "center" attribute showing initial values. The final compartment shows the operations setWidth, setLength and setPosition and their parameters. The notation that precedes the attribute, or operation name, indicates the visibility of the element: if the + symbol is used, the attribute, or operation, has a public level of visibility; if a - symbol is used, the attribute, or operation, is private. In addition the # symbol allows an operation, or attribute, to be defined as protected, while the ~ symbol indicates package visibility.



Interfaces

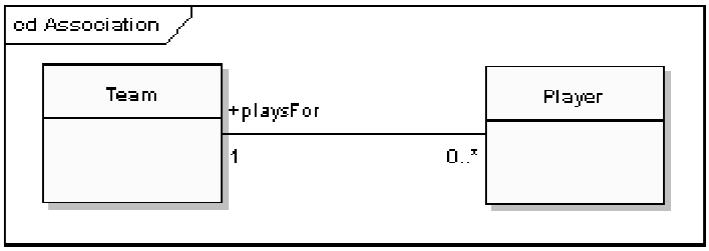
An interface is a specification of behavior that implementers agree to meet; it is a contract. By realizing an interface, classes are guaranteed to support a required behavior, which allows the system to treat non-related elements in the same way – that is, through the common interface.



Interfaces may be drawn in a similar style to a class, with operations specified, as shown above. They may also be drawn as a circle with no explicit operations detailed. When drawn as a circle, realization links to the circle form of notation are drawn without target arrows.

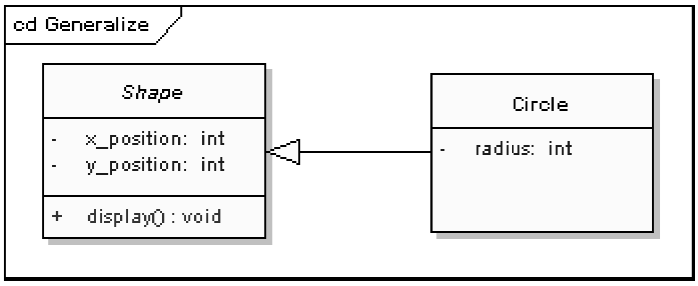
Associations

An association implies two model elements have a relationship - usually implemented as an instance variable in one class. This connector may include named roles at each end, cardinality, direction and constraints. Association is the general relationship type between elements. For more than two elements, a diamond representation toolbox element can be used as well. When code is generated for class diagrams, named association ends become instance variables in the target class. So, for the example below, "playsFor" will become an instance variable in the "Player" class.

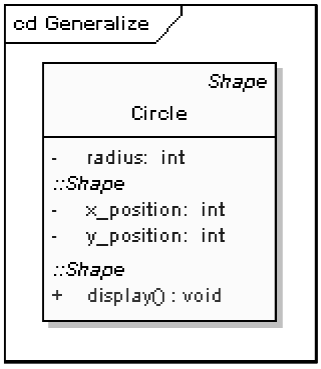


Generalizations

A generalization is used to indicate inheritance. Drawn from the specific classifier to a general classifier, the generalize implication is that the source inherits the target's characteristics. The following diagram shows a parent class generalizing a child class. Implicitly, an instantiated object of the Circle class will have attributes x_position, y_position and radius and a method display(). Note that the class "Shape" is abstract, shown by the name being italicized.



The following diagram shows an equivalent view of the same information.



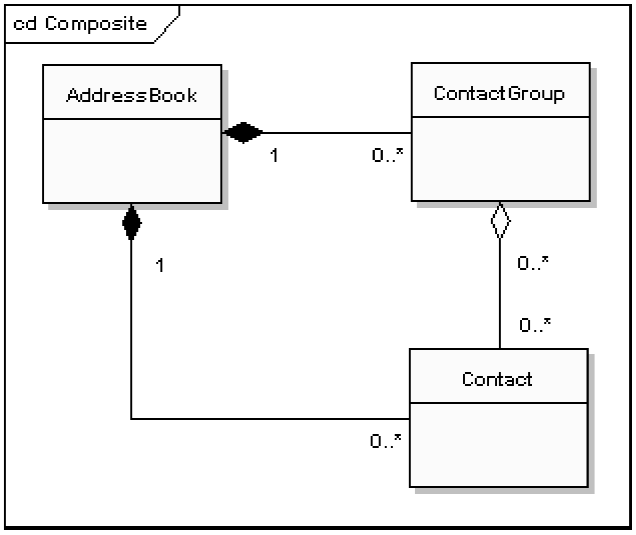
Aggregations

Aggregations are used to depict elements which are made up of smaller components. Aggregation relationships are shown by a white diamond-shaped arrowhead pointing towards the target or parent class.

A stronger form of aggregation - a composite aggregation - is shown by a black diamond-shaped arrowhead and is used where components can be included in a maximum of one composition at a time. If the parent of a composite

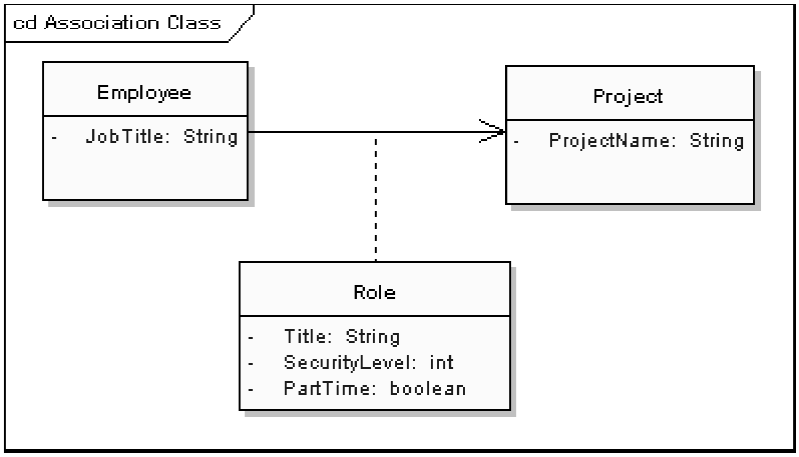
aggregation is deleted, usually all of its parts are deleted with it; however a part can be individually removed from a composition without having to delete the entire composition. Compositions are transitive, asymmetric relationships and can be recursive.

The following diagram illustrates the difference between weak and strong aggregations. An address book is made up of a multiplicity of contacts and contact groups. A contact group is a virtual grouping of contacts; a contact may be included in more than one contact group. If you delete an address book, all the contacts and contact groups will be deleted too; if you delete a contact group, no contacts will be deleted.



Association Classes

An association class is a construct that allows an association connection to have operations and attributes. The following example shows that there is more to allocating an employee to a project than making a simple association link between the two classes: the role the employee takes up on the project is a complex entity in its own right and contains detail that does not belong in the employee or project class. For example, an employee may be working on several projects at the same time and have different job titles and security levels on each.



Notes:

Dependencies

A dependency is used to model a wide range of dependent relationships between model elements. It would normally be used early in the design process where it is known that there is some kind of link between two

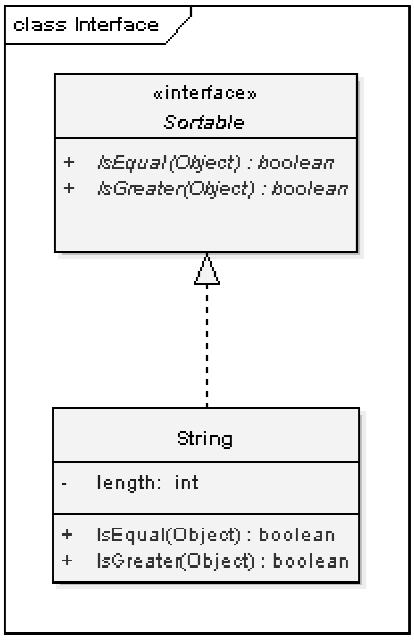
elements, but it is too early to know exactly what the relationship is. Later in the design process, dependencies will be stereotyped (stereotypes available include «instantiate», «trace», «import», and others), or replaced with a more specific type of connector.

Traces

The trace relationship is a specialization of a dependency, linking model elements or sets of elements that represent the same idea across models. Traces are often used to track requirements and model changes. As changes can occur in both directions, the order of this dependency is usually ignored. The relationship's properties can specify the trace mapping, but the trace is usually bi-directional, informal and rarely computable.

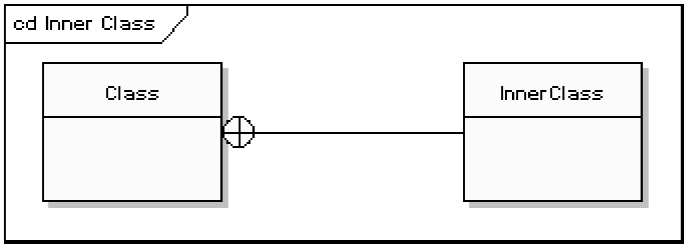
Realizations

The source object implements or realizes the destination. Realizations are used to express traceability and completeness in the model - a business process or requirement is realized by one or more use cases, which are in turn realized by some classes, which in turn are realized by a component, etc. Mapping requirements, classes, etc. across the design of your system, up through the levels of modeling abstraction, ensures the big picture of your system remembers and reflects all the little pictures and details that constrain and define it. A realization is shown as a dashed line with a solid arrowhead.



Nestings

A nesting is connector that shows the source element is nested within the target element. The following diagram shows the definition of an inner class, although in EA it is more usual to show them by their position in the project view hierarchy.



References

- **Paul Deitel and Harvey Deitel, Java How to Program (9th Edition).**
- **B. Eckel, Thinking in Java (4th Edition).**
- **The Java™ Tutorials - Oracle Documentation (https://docs.oracle.com/javase/tutorial/)**