

Object-Oriented Programming Concepts

Class

A class is a blueprint or prototype from which objects are created.

In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an *instance* of the *class of objects* known as bicycles. A *class* is the blueprint from which individual objects are created.

Object

An object is a software bundle of related state and behavior.

Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in fields and exposes its behavior through methods.

Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication.

Data Abstraction

Data Abstraction increases the power of programming language by creating user defined data types. Data Abstraction also represents the needed information in the program without presenting the details.

In other ways Data Abstraction is nothing but giving required details by hiding unrelated information.

Data Encapsulation

Data Encapsulation combines data and functions into a single unit called Class. When using Data Encapsulation, data is not accessed directly; it is only accessible through the functions present inside the class. Data Encapsulation enables the important concept of data hiding possible.

Inheritance

In object-oriented programming (OOP), **inheritance** is a way to reuse code of existing objects, or to establish a subtype from an existing object, or both, depending upon programming language support.

In classical inheritance where objects are defined by classes, classes can inherit attributes and behavior from pre-existing classes called base classes, superclasses, parent classes or ancestor classes. The resulting classes are known as derived classes, subclasses or child classes. The relationships of classes through inheritance gives rise to a hierarchy

Polymorphism

Polymorphism allows routines to use variables of different types at different times. An operator or function can be given different meanings or functions. Polymorphism refers to a single function or multi-functioning operator performing in different ways.

Java

Java programming language was originally developed by Sun Microsystems, which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems's Java platform (Java 1.0 [J2SE]).

As of May 2012 the latest release of the Java Standard Edition is 7 (J2SE). With the advancement of Java and its wide spread popularity, multiple configurations were built to suite various types of platforms. Ex: J2EE for Enterprise Applications, J2ME for Mobile Applications.

Java is guaranteed to be Write Once, Run Anywhere

Java is:

- Object Oriented : In java everything is an Object. Java can be easily extended since it is based on the Object model.
- Platform independent: Unlike many other programming languages including C and C++ when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run.
- Simple :Java is designed to be easy to learn. If you understand the basic concept of OOP java would be easy to master.
- Secure : With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
- Architectural- neutral :Java compiler generates an architecture-neutral object file format which makes the compiled code to be executable on many processors, with the presence Java runtime system.

- **Portable** :being architectural neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler and Java is written in ANSI C with a clean portability boundary which is a POSIX subset.
- **Robust** :Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
- **Multi-threaded** : With Java's multi-threaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.
- **Interpreted** :Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light weight process.
- **High Performance**: With the use of Just-In-Time compilers Java enables high performance.
- **Distributed** :Java is designed for the distributed environment of the internet.
- **Dynamic** : Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

History of Java:

James Gosling initiated the Java language project in June 1991 for use in one of his many set-top box projects. The language, initially called Oak after an oak tree that stood outside Gosling's office, also went by the name Green and ended up later renamed as Java, from a list of random words.

Sun released the first public implementation as Java 1.0 in 1995. It promised Write Once, Run Anywhere (WORA), providing no-cost run-times on popular platforms.

On 13 November 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL).

On 8 May 2007 Sun finished the process, making all of Java's core code free and open-source, aside from a small portion of code to which Sun did not hold the copyright.

JDK & JVM

JDK (Java Development Kit)

Java Developer Kit contains tools needed to develop the Java programs, and JRE to run the programs. The tools include compiler (javac.exe), Java application launcher (java.exe), Appletviewer, etc...

Compiler converts java code into byte code. Java application launcher opens a JRE, loads the class, and invokes its main method.

You need JDK, if at all you want to write your own programs, and to compile them. For running java programs, JRE is sufficient.

JRE is targeted for execution of Java files

i.e. JRE = JVM + Java Packages Classes(like util, math, lang, awt,swing etc)+runtime libraries.

JDK is mainly targeted for java development. I.e. You can create a Java file (with the help of Java packages), compile a Java file and run a java file

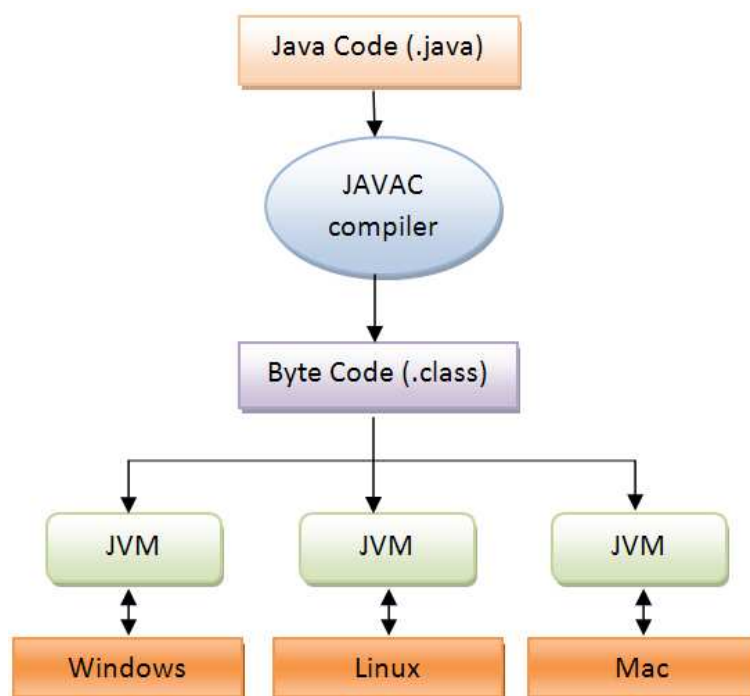
JRE (Java Runtime Environment)

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.

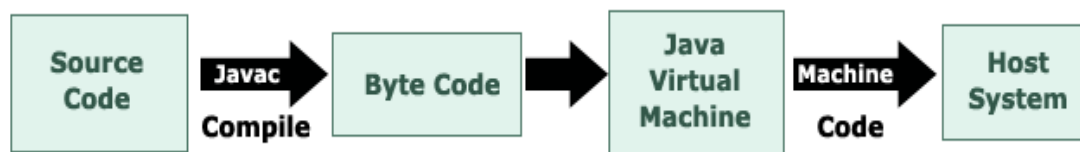
JVM (Java Virtual Machine)

As we all aware when we compile a Java file, output is not an 'exe' but it's a '.class' file. '.class' file consists of Java byte codes which are understandable by JVM. Java Virtual Machine interprets/compiles the byte code into the machine code depending upon the underlying operating system and hardware combination. It is responsible for all the things like garbage collection, array bounds checking, etc... JVM is platform dependent.



The JVM is called "virtual" because it provides a machine interface that does not depend on the underlying operating system and machine hardware architecture. This independence from hardware and operating system is a cornerstone of the write-once run-anywhere value of Java programs.

There are different JVM implementations are there. These may differ in things like performance, reliability, speed, etc. These implementations will differ in those areas where Java specification doesn't mention how to implement the features, like how the garbage collection process works is JVM dependent, Java spec doesn't define any specific way to do this.



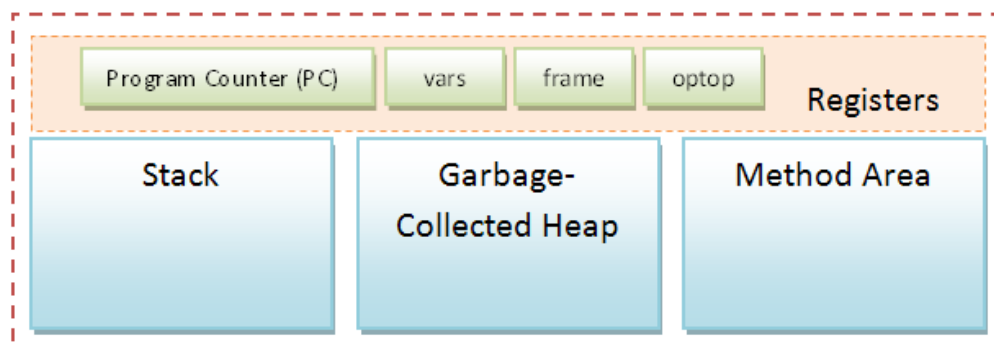
Just in Time Compiler (JIT)

Initially Java has been accused of poor performance because it's both compiles and interpret instruction. Since compilation or Java file to class file is independent of execution of Java program do not confuse. Here compilation word is used for byte code to machine instruction translation. JIT are advanced part of Java Virtual machine which optimize byte code to machine instruction conversion part by compiling similar byte codes at same time and thus reducing overall execution time. JIT is part of Java Virtual Machine and also performs several other optimizations such as in-lining function.

Java Virtual Machine

To execute any code, JVM utilizes different components.

JVM is divided into several components like the stack, the garbage-collected heap, the registers and the method area. Let us see diagram representation of JVM.



The Stack

Stack in Java virtual machine stores various method arguments as well as the local variables of any method. Stack also keep track of each and every method invocation.

This is called **Stack Frame**. There are three registers that help in stack manipulation. They are vars, frame, optop. This registers points to different parts of current Stack. There are three sections in Java stack frame:

Local Variables

The local variables section contains all the local variables being used by the current method invocation. It is pointed to by the **vars** register.

Execution Environment

The execution environment section is used to maintain the operations of the stack itself. It is pointed to by the **frame** register.

Operand Stack

The operand stack is used as a work space by bytecode instructions. It is here that the parameters for bytecode instructions are placed, and results of bytecode instructions are found. The top of the operand stack is pointed to by the **optop** register.

Method Area

This is the area where bytecodes reside. The program counter points to some byte in the method area. It always keep tracks of the current instruction which is being executed (interpreted). After execution of an instruction, the JVM sets the PC to next instruction. Method area is shared among all the threads of a process. Hence if more than one threads are accessing any specific method or any instructions, synchronization is needed. Synchronization in JVM is achieved through Monitors.

Garbage-collected Heap

The Garbage-collected Heap is where the objects in Java programs are stored. Whenever we allocate an object using new operator, the heap comes into picture and memory is allocated from there. Unlike C++, Java does not have free operator to free any previously allocated memory. Java does this automatically using Garbage collection mechanism. Till Java 6.0, **mark and sweep** algorithm is used as a garbage collection logic. Remember that the local object reference resides on Stack but the actual object resides in Heap only. Also, arrays in Java are objects, hence they also resides in Garbage-collected Heap.

Java Environment Setup

Java SE is freely available from the below link

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

You download a version based on your operating system.

Follow the instructions to download java and run the .exe to install Java on your machine. Once you installed Java on your machine, you would need to set environment variables to point to correct installation directories:

Setting up the path for windows 2000/XP:

Assuming you have installed Java in c:\Program Files\java\jdk directory:

Right-click on 'My Computer' and select 'Properties'.

Click on the 'Environment variables' button under the 'Advanced' tab.

Now alter the 'Path' variable so that it also contains the path to the Java executable.

Example, if the path is currently set to 'C:\WINDOWS\SYSTEM32', then change your path to read 'C:\WINDOWS\SYSTEM32;c:\Program Files\java\jdk\bin'.

Setting up the path for Linux, UNIX, Solaris, FreeBSD:

Environment variable PATH should be set to point to where the java binaries have been installed. Refer to your shell documentation if you have trouble doing this.

Example, if you use bash as your shell, then you would add the following line to the end of your '.bashrc: export PATH=/path/to/java:\$PATH'

Popular Java Editors:

To write your java programs you will need a text editor. There are even more sophisticated IDE available in the market. But for now, you can consider one of the following:

Notepad : On Windows machine you can use any simple text editor like Notepad (Recommended for this tutorial), TextPad.

Netbeans :is a Java IDE that is open source and free which can be downloaded from <http://www.netbeans.org/index.html>.

Eclipse : is also a java IDE developed by the eclipse open source community and can be downloaded from <http://www.eclipse.org/>.

First Java Program:

Let us look at a simple code that would print the words Hello World.

```
public class MyFirstJavaProgram{  
  
    /* This is my first java program.  
     * This will print 'Hello World' as the output  
     */  
  
    public static void main(String []args){  
        System.out.println("Hello World"); // prints Hello World  
    }  
}
```

Lets look at how to save the file, compile and run the program. Please follow the steps given below:

1. Open notepad and add the code as above.
2. Save the file as : MyFirstJavaProgram.java.
3. Open a command prompt window and go o the directory where you saved the class. Assume its C:\.
4. Type ' javac MyFirstJavaProgram.java ' and press enter to compile your code. If there are no errors in your code the command prompt will take you to the next line.(Assumption : The path variable is set).
5. Now type ' java MyFirstJavaProgram ' to run your program.

6. You will be able to see 'Hello World' printed on the window.

```
C : > javac MyFirstJavaProgram.java
C : > java MyFirstJavaProgram
Hello World
```

Basic Syntax:

About Java programs, it is very important to keep in mind the following points.

- **Case Sensitivity** - Java is case sensitive which means identifier **Hello** and **hello** would have different meaning in Java.
- **Class Names** - For all class names the first letter should be in Upper Case.

If several words are used to form a name of the class each inner words first letter should be in Upper Case.

Example *class MyFirstJavaClass*

- **Method Names** - All method names should start with a Lower Case letter.

If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.

Example *public void myMethodName()*

- **Program File Name** - Name of the program file should exactly match the class name.

When saving the file you should save it using the class name (Remember java is case sensitive) and append '.java' to the end of the name. (if the file name and the class name do not match your program will not compile).

Example : Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as '*MyFirstJavaProgram.java*'

-
- **public static void main(String args[])** - java program processing starts from the main() method which is a mandatory part of every java program..

Java Language Basics

Class

A class is a blue print from which individual objects are created.

In java everything should be defined as Class

```
class MyClass {  
    // fields,  
    // constructors, and  
    // method declarations  
}
```

This is a class declaration. The *class body* (the area between the braces) contains all the code that provides for the life cycle of the objects created from the class.

A Class will contain below given information.

- Constructors for initializing new objects.
- Declarations for the fields that provide the state of the objects of that class
- Methods to implement the behavior of the class.

Classes contain methods to perform the function assigned to the class. And a class can have more than one constructor.

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

1. Modifiers such as public, private.
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 { }.

```
public class MyClass extends ParentClass implements MyInterface,ParentInterface
```

Constructor

A constructor of the classes automatically initializes the data members of the class when you create an object.

Constructors have the following characteristics:

- A constructor is a method with same of the class name.
- A constructor is automatically invoked every time an instance of a class is created.
- Constructors do not have a return type.

Every class has a constructor. If we do not explicitly write a constructor for a class the java compiler builds a default constructor for that class. This default constructor will call the no-argument constructor of the superclass.

As with methods, the Java platform differentiates constructors on the basis of the number of arguments in the list and their types. You cannot write two constructors that have the same number and type of arguments for the same class, because the platform would not be able to tell them apart. Doing so causes a compile-time error.

Each time a new object is created at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

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

Example of a constructor is given below:

```
class Puppy{
    public puppy() { //Default Constructor
    }

    public puppy(String name){
        // This constructor has one parameter, name.
    }
}
```

Methods

Here is an example of a typical method declaration:

```
public long calculateAnswer(long wingSpan, int numberOfEngines,
                           double length, float grossTons) {
    //do the calculation here
}
```

The only required elements of a method declaration are the method's return type, name, a pair of parentheses, (), and a body between braces, {}.

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.
6. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

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

The signature of the method declared above is:

```
calculateAnswer(long, int, double, float)
```

Naming a Method:

Although a method name can be any legal identifier, code conventions restrict method names. 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. Here are some examples:

```
run
runFast
getBackground
getFinalData
compareTo
setX
isEmpty
```

A sample of a complete java class is given below:

```
public class Dog{

    String breed;
    int age;
    String color;

    public Dog(){ } //Default Constructor

    public Dog(int ag,String col,String bree){
        breed = bree;
        age= ag;
        color = col;
    }

    void barking(){
    }

    void hungry(){
    }

    void sleeping(){
    }

    String getColor(){
        return color;
    }

    String getBreed(){
        return breed;
    }

    int getAge(){
        return age;
    }
}
```

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).

```
int size = 6;
```

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.

A field defining the number of gears for a particular kind of bicycle could be marked as static since conceptually the same number of gears will apply to all instances.

```
static int numGears = 6;
```

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

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". Fields are nothing but variable defined at Instance / class.

Naming Variables

The rules and conventions for naming your variables can be summarized as follows:

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. Subsequent characters may be letters, digits, dollar signs, or underscore characters. Conventions (and common sense) apply to this rule as well. When choosing a name for your variables, use full words instead of cryptic abbreviations. Doing so will make your code easier to read and understand. In many cases it will also make your code self-documenting. Also keep in mind that the name you choose must not be a keyword or reserved word.

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_GEAR = 6`, the convention changes slightly, capitalizing every letter and separating subsequent words with the underscore character.

Object:

As mentioned previously a class provides the blueprints for objects. So basically an object is created from a class. In java the `new` key word is used to create new objects.

There are three steps when creating an object from a class:

Declaration . A variable declaration with a variable name with an object type.

Instantiation . The 'new' key word is used to create the object.

Initialization . The 'new' keyword is followed by a call o a constructor. This call initializes the new object.

Example of creating an object is given below:

```
class Puppy{
    public Puppy(String name){
        // This constructor has one parameter, name.
        System.out.println("Passed Name is : " + name );
    }
    public static void main(String []args){
        // Following statement would create an object myPuppy
        Puppy myPuppy = new Puppy( "tommy" );
    }
}
```

If we compile and run the above program then it would produce following result:

```
Passed Name is :tommy
```

Accessing Instance Variables and Methods:

Instance variables and methods are accessed via created objects. To access an instance variable the fully qualified path should be as follows:

```
/* First create an object */
ObjectReference = new Constructor();

/* Now call a variable as follows */
ObjectReference.variableName;

/* Now you can call a class method as follows */
ObjectReference.MethodName();
```

Example:

This example explains how to access instance variables and methods of a class:

```
class Puppy{

    int puppyAge;

    public Puppy(String name){
        // This constructor has one parameter, name.
        System.out.println("Passed Name is : " + name );
    }

    public setAge( int age ){
        puppyAge = age;
    }

    public getAge( ){
        System.out.println("Puppy's age is : " + puppyAge );
        return puppyAge;
    }

    public static void main(String []args){
        /* Object creation */
        Puppy myPuppy = new Puppy( "tommy" );

        /* Call class method to set puppy's age */
        myPuppy.setAge( 2 );

        /* Call another class method to get puppy's age */
        myPuppy.getAge( );

        /* You can access instance variable as follows as well */
        System.out.println("Variable Value : " + myPuppy.puppyAge );
    }
}
```

If we compile and run the above program then it would produce following result:

```
Passed Name is :tommy
Puppy's age is :2
Variable Value :2
```

Primitive Data Types

The Java programming language is statically-typed, which means that all variables must first be declared before they can be used. This involves stating the variable's type and name, as you've already seen:

```
int gear = 1;
```

Doing so tells your program that a field named "gear" exists, holds numerical data, and has an initial value of "1". A variable's data type determines the values it may contain, plus the operations that may be performed on it. In addition to `int`, the Java programming language supports seven other 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:** The byte data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The byte data type can be useful for saving memory in large arrays, where the memory savings actually matters. They can also be used in place of `int` where their limits help to clarify your code; the fact that a variable's range is limited can serve as a form of documentation.
- **short:** The short data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with byte, the same guidelines apply: you can use `short` to save memory in large arrays, in situations where the memory savings actually matters.
- **int:** The int data type is a 32-bit signed two's complement integer. It has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive). For integral values, this data type is generally the default choice unless there is a reason (like the above) to choose something else. This data type will most likely be large enough for the numbers your program will use, but if you need a wider range of values, use `long` instead.
- **long:** The long data type is a 64-bit signed two's complement integer. It has a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (inclusive). Use this data type when you need a range of values wider than those provided by `int`.
- **float:** The float data type is a single-precision 32-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in section 4.2.3 of the Java Language Specification. As with the recommendations for byte and short, use a float (instead of double) if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency. For that, you will

need to use the `java.math.BigDecimal` class instead. Numbers and Strings covers `BigDecimal` and other useful classes provided by the Java platform.

- **double:** The double data type is a double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in section 4.2.3 of the Java Language Specification. For decimal values, this data type is generally the default choice. As mentioned above, this data type should never be used for precise values, such as currency.
- **boolean:** The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.
- **char:** The char data type is a single 16-bit Unicode character. It has a minimum value of `'\u0000'` (or 0) and a maximum value of `'\uffff'` (or 65,535 inclusive).

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;

for example, `String s = "this is a string";`.

`String` objects are immutable, which means that once created, their values cannot be changed. The `String` class is not technically a primitive data type, but considering the special support given to it by the language, you'll probably tend to think of it as such

Default Values

It's not always necessary to assign a value when a field is declared. Fields that are declared but not initialized will be set to a reasonable default by the compiler. Generally speaking, this default will be zero or null, depending on the data type. Relying on such default values, however, is generally considered bad programming style.

The following chart summarizes the default values for the above data types.

Data Type	Default Value (for fields)
byte	0
short	0

int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Local variables are slightly different; the compiler never assigns a default value to an uninitialized local variable. If you cannot initialize your local variable where it is declared, make sure to assign it a value before you attempt to use it. Accessing an uninitialized local variable will result in a compile-time error.

Literals

You may have noticed that the new keyword isn't used when initializing a variable of a primitive type. Primitive types are special data types built into the language; they are not objects created from a class. A *literal* is the source code representation of a fixed value; literals are represented directly in your code without requiring computation. As shown below, it's possible to assign a literal to a variable of a primitive type:

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
```

Integer Literals

An integer literal is of type long if it ends with the letter L or l; otherwise it is of type int. It is recommended that you use the upper case letter L because the lower case letter l is hard to distinguish from the digit 1.

Values of the integral types byte, short, int, and long can be created from int literals. Values of type long that exceed the range of int can be created from long 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
- Binary: Base 2, whose digits consists of the numbers 0 and 1 (you can create binary literals in Java SE 7 and later)

For general-purpose programming, the decimal system is likely to be the only number system you'll ever use. However, if you need to use another number system, the

following example shows the correct syntax. The prefix 0x indicates hexadecimal and 0b indicates binary:

```
// The number 26, in decimal
```

```
int decVal = 26;
```

```
// The number 26, in hexadecimal
```

```
int hexVal = 0x1a;
```

```
// The number 26, in binary
```

```
int binVal = 0b11010;
```

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).

```
double d1 = 123.4;
// same value as d1, but in scientific notation
double d2 = 1.234e2;
float f1 = 123.4f;
```

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.

For instance, if your code contains numbers with many digits, you can use an underscore character to separate digits in groups of three, similar to how you would use a punctuation mark like a comma, or a space, as a separator.

The following example shows other ways you can use the underscore in numeric literals:

```
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;
```

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

The following examples demonstrate valid and invalid underscore placements (which are highlighted) in numeric literals:

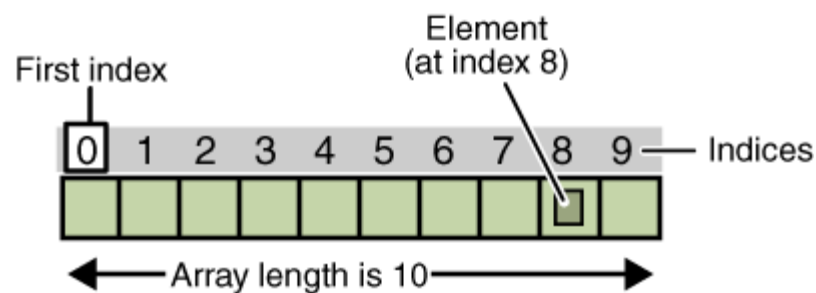
```
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi1 = 3_.1415F;
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi2 = 3._1415F;
// Invalid: cannot put underscores
// prior to an L suffix
long socialSecurityNumber1 = 999_99_9999_L;

// This is an identifier, not
// a numeric literal
int x1 = _52;
// OK (decimal literal)
int x2 = 5_2;
// Invalid: cannot put underscores
// At the end of a literal
int x3 = 52_;
// OK (decimal literal)
int x4 = 5_____2;
```

```
// Invalid: cannot put underscores
// in the 0x radix prefix
int x5 = 0_x52;
// Invalid: cannot put underscores
// at the beginning of a number
int x6 = 0x_52;
// OK (hexadecimal literal)
int x7 = 0x5_2;
// Invalid: cannot put underscores
// at the end of a number
int x8 = 0x52_;
```

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. You've seen an example of arrays already, in the `main` method of the "Hello World!" application. This section discusses arrays in greater detail.



An array of ten elements

Each item in an array is called an *element*, and each element is accessed by its numerical *index*. As shown in the above illustration, numbering begins with 0. The 9th element, for example, would therefore be accessed at index 8.

The following program, [ArrayDemo](#), creates an array of integers, puts some values in it, and prints each value to standard output.

```
class ArrayDemo {
    public static void main(String[] args) {
        // declares an array of integers
        int[] anArray;

        // allocates memory for 10 integers
        anArray = new int[10];

        // initialize first element
        anArray[0] = 100;
```

```

// initialize second element
anArray[1] = 200;
// etc.
anArray[2] = 300;
anArray[3] = 400;
anArray[4] = 500;
anArray[5] = 600;
anArray[6] = 700;
anArray[7] = 800;
anArray[8] = 900;
anArray[9] = 1000;

System.out.println("Element at index 0: "
    + anArray[0]);
System.out.println("Element at index 1: "
    + anArray[1]);
System.out.println("Element at index 2: "
    + anArray[2]);
System.out.println("Element at index 3: "
    + anArray[3]);
System.out.println("Element at index 4: "
    + anArray[4]);
System.out.println("Element at index 5: "
    + anArray[5]);
System.out.println("Element at index 6: "
    + anArray[6]);
System.out.println("Element at index 7: "
    + anArray[7]);
System.out.println("Element at index 8: "
    + anArray[8]);
System.out.println("Element at index 9: "
    + anArray[9]);
}
}

```

The output from this program is:

```

Element at index 0: 100
Element at index 1: 200
Element at index 2: 300
Element at index 3: 400
Element at index 4: 500
Element at index 5: 600
Element at index 6: 700
Element at index 7: 800
Element at index 8: 900
Element at index 9: 1000

```

In a real-world programming situation, you'd probably use one of the supported *looping constructs* to iterate through each element of the array, rather than write each line individually as shown above. However, this example clearly illustrates the array syntax. You'll learn about the various looping constructs (for, while, and do-while) in the [Control Flow](#) section.

Declaring a Variable to Refer to an Array

The above program declares `anArray` with the following line of code:

```
// declares an array of integers
int[] anArray;
```

Like declarations for variables of other types, an array declaration has two components: the array's type and the array's name. An array's type is written as `type[]`, where *type* is the data type of the contained elements; the square brackets are special symbols indicating that this variable holds an array. The size of the array is not part of its type (which is why the brackets are empty). An array's name can be anything you want, provided that it follows the rules and conventions as previously discussed in the [naming](#) section. As with variables of other types, the declaration does not actually create an array — it simply tells the compiler that this variable will hold an array of the specified type.

Similarly, you can declare arrays of other types:

```
byte[] anArrayOfBytes;
short[] anArrayOfShorts;
long[] anArrayOfLongs;
float[] anArrayOfFloats;
double[] anArrayOfDoubles;
boolean[] anArrayOfBooleans;
char[] anArrayOfChars;
String[] anArrayOfStrings;
```

You can also place the square brackets after the array's name:

```
// this form is discouraged
float anArrayOfFloats[];
```

However, convention discourages this form; the brackets identify the array type and should appear with the type designation.

Creating, Initializing, and Accessing an Array

One way to create an array is with the `new` operator. The next statement in the `ArrayDemo` program allocates an array with enough memory for ten integer elements and assigns the array to the `anArray` variable.

```
// create an array of integers
anArray = new int[10];
```

If this statement were missing, the compiler would print an error like the following, and compilation would fail:

```
ArrayDemo.java:4: Variable anArray may not have been initialized.
```

The next few lines assign values to each element of the array:

```
anArray[0] = 100; // initialize first element
anArray[1] = 200; // initialize second element
anArray[2] = 300; // etc.
```

Each array element is accessed by its numerical index:

```
System.out.println("Element 1 at index 0: " + anArray[0]);
System.out.println("Element 2 at index 1: " + anArray[1]);
System.out.println("Element 3 at index 2: " + anArray[2]);
```

Alternatively, you can use the shortcut syntax to create and initialize an array:

```
int[] anArray = {
    100, 200, 300,
    400, 500, 600,
    700, 800, 900, 1000
};
```

Here the length of the array is determined by the number of values provided between { and }.

You can also declare an array of arrays (also known as a *multidimensional* array) by using two or more sets of square 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 simply an array whose components are themselves arrays. This is unlike arrays in C or Fortran. A consequence of this is that the rows are allowed to vary in length, as shown in the following [MultiDimArrayDemo](#) program:

```
class MultiDimArrayDemo {
    public static void main(String[] args) {
        String[][] names = {
            {"Mr. ", "Mrs. ", "Ms. "},
            {"Smith", "Jones"}
        };
        // Mr. Smith
        System.out.println(names[0][0] + names[1][0]);
        // Ms. Jones
        System.out.println(names[0][2] + names[1][1]);
    }
}
```

The output from this program is:

```
Mr. Smith
Ms. Jones
```

Finally, you can use the built-in `length` property to determine the size of any array. The code


```
System.out.println(anArray.length);
```

will print the array's size to standard output.

Copying Arrays

The `System` class has an `arraycopy` method that you can use to efficiently copy data from one array into another:

```
public static void arraycopy(Object src, int srcPos,  
                             Object dest, int destPos, int length)
```

The two `Object` arguments specify the array to copy *from* and the array to copy *to*. The three `int` arguments specify the starting position in the source array, the starting position in the destination array, and the number of array elements to copy.

The following program, [ArrayCopyDemo](#), declares an array of `char` elements, spelling the word "decaffeinated". It uses `arraycopy` to copy a subsequence of array components into a second array:

```
class ArrayCopyDemo {  
    public static void main(String[] args) {  
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',  
                             'i', 'n', 'a', 't', 'e', 'd' };  
        char[] copyTo = new char[7];  
  
        System.arraycopy(copyFrom, 2, copyTo, 0, 7);  
        System.out.println(new String(copyTo));  
    }  
}
```

The output from this program is:

```
caffein
```

Operators

Operators are special symbols that perform specific operations on one, two, or three operands, and then return a result.

Simple Assignment Operator

= Simple assignment operator

Arithmetic Operators

+ Additive operator (also used for String concatenation)
- Subtraction operator
* Multiplication operator
/ Division operator
% Remainder operator

Unary Operators

+ Unary plus operator; indicates positive value (numbers are positive without this, however)
- Unary minus operator; negates an expression
++ Increment operator; increments a value by 1
-- Decrement operator; decrements a value by 1
! Logical complement operator; inverts the value of a boolean

Equality and Relational Operators

== Equal to
!= Not equal to
> Greater than
>= Greater than or equal to
< Less than
<= Less than or equal to

Conditional Operators

&& Conditional-AND
|| Conditional-OR
?: Ternary (shorthand for if-then-else statement)

Type Comparison Operator

`instanceof` Compares an object to
 a specified type

Bitwise and Bit Shift Operators

`~` Unary bitwise complement
`<<` Signed left shift
`>>` Signed right shift
`>>>` Unsigned right shift
`&` Bitwise AND
`^` Bitwise exclusive OR
`|` Bitwise inclusive OR

As we explore the operators of the Java programming language, it may be helpful for you to know ahead of time which operators have the highest precedence. The operators in the following table are listed according to precedence order. The closer to the top of the table an operator appears, the higher its precedence. Operators with higher precedence are evaluated before operators with relatively lower precedence. Operators on the same line have equal precedence. When operators of equal precedence appear in the same expression, a rule must govern which is evaluated first. All binary operators except for the assignment operators are evaluated from left to right; assignment operators are evaluated right to left.

Operator Precedence	
Operators	Precedence
postfix	<i>expr++</i> , <i>expr--</i>
unary	<i>++expr</i> , <i>--expr</i> , <i>+expr</i> , <i>-expr</i> , <i>~</i> , <i>!</i>
multiplicative	<i>*</i> / <i>%</i>
additive	<i>+</i> -
shift	<i><<</i> <i>>></i> <i>>>></i>
relational	<i><</i> <i>></i> <i><=</i> <i>>=</i> <i>instanceof</i>
equality	<i>==</i> <i>!=</i>
bitwise AND	<i>&</i>
bitwise exclusive OR	<i>^</i>
bitwise inclusive OR	<i> </i>
logical AND	<i>&&</i>

logical OR	
ternary	? :
assignment	= += -= *= /= %= &= ^= = <<= >>= >>>=

In general-purpose programming, certain operators tend to appear more frequently than others;

for example, the assignment operator "=" is far more common than the unsigned right shift operator ">>>". With that in mind, the following discussion focuses first on the operators that you're most likely to use on a regular basis, and ends focusing on those that are less common. Each discussion is accompanied by sample code that you can compile and run. Studying its output will help reinforce what you've just learned.

The Arithmetic Operators

The Java programming language provides operators that perform addition, subtraction, multiplication, and division. There's a good chance you'll recognize them by their counterparts in basic mathematics. The only symbol that might look new to you is "%", which divides one operand by another and returns the remainder as its result.

+ additive operator (also used for
 String concatenation)
 - subtraction operator
 * multiplication operator
 / division operator
 % remainder operator

The following program, ArithmeticDemo, tests the arithmetic operators.

```
class ArithmeticDemo {

    public static void main (String[] args){

        // result is now 3
        int result = 1 + 2;
        System.out.println(result);

        // result is now 2
        result = result - 1;
        System.out.println(result);

        // result is now 4
        result = result * 2;
        System.out.println(result);
    }
}
```

```

        // result is now 2
        result = result / 2;
        System.out.println(result);

        // result is now 10
        result = result + 8;
        System.out.println(result);

        // result is now 3
        result = result % 7;
        System.out.println(result);
    }
}

```

The Unary Operators

The unary operators require only one operand; they perform various operations such as incrementing/decrementing a value by one, negating an expression, or inverting the value of a boolean.

- + Unary plus operator; indicates positive value (numbers are positive without this, however)
- Unary minus operator; negates an expression
- ++ Increment operator; increments a value by 1
- Decrement operator; decrements a value by 1
- ! Logical complement operator; inverts the value of a boolean

The following program, UnaryDemo, tests the unary operators:

```

class UnaryDemo {

    public static void main(String[] args){
        // result is now 1
        int result = +1;
        System.out.println(result);
        // result is now 0
        result--;
        System.out.println(result);
        // result is now 1
        result++;
        System.out.println(result);
        // result is now -1
        result = -result;
        System.out.println(result);
        boolean success = false;
        // false
        System.out.println(success);
        // true
        System.out.println(!success);
    }
}

```

```
}
```

The increment/decrement operators can be applied before (prefix) or after (postfix) the operand. The code `result++`; and `++result`; will both end in result being incremented by one. The only difference is that the prefix version (`++result`) evaluates to the incremented value, whereas the postfix version (`result++`) evaluates to the original value. If you are just performing a simple increment/decrement, it doesn't really matter which version you choose. But if you use this operator in part of a larger expression, the one that you choose may make a significant difference.

The following program, `PrePostDemo`, illustrates the prefix/postfix unary increment operator:

```
class PrePostDemo {
    public static void main(String[] args){
        int i = 3;
        i++;
        // prints 4
        System.out.println(i);
        ++i;
        // prints 5
        System.out.println(i);
        // prints 6
        System.out.println(++i);
        // prints 6
        System.out.println(i++);
        // prints 7
        System.out.println(i);
    }
}
```

Equality, Relational, and Conditional Operators

The Equality and Relational Operators

The equality and relational operators determine if one operand is greater than, less than, equal to, or not equal to another operand. The majority of these operators will probably look familiar to you as well. Keep in mind that you must use `"=="`, not `"="`, when testing if two primitive values are equal.

<code>==</code>	equal to
<code>!=</code>	not equal to
<code>></code>	greater than
<code>>=</code>	greater than or equal to
<code><</code>	less than
<code><=</code>	less than or equal to

The following program, `ComparisonDemo`, tests the comparison operators:

```
class ComparisonDemo {

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

```

int value1 = 1;
int value2 = 2;
if(value1 == value2)
    System.out.println("value1 == value2");
if(value1 != value2)
    System.out.println("value1 != value2");
if(value1 > value2)
    System.out.println("value1 > value2");
if(value1 < value2)
    System.out.println("value1 < value2");
if(value1 <= value2)
    System.out.println("value1 <= value2");
}
}

```

Output:

```

value1 != value2
value1 < value2
value1 <= value2

```

The Conditional Operators

The `&&` and `||` operators perform *Conditional-AND* and *Conditional-OR* operations on two boolean expressions. These operators exhibit "short-circuiting" behavior, which means that the second operand is evaluated only if needed.

`&&` Conditional-AND
`||` Conditional-OR

The following program, `ConditionalDemo1`, tests these operators:

```

class ConditionalDemo1 {

    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        if((value1 == 1) && (value2 == 2))
            System.out.println("value1 is 1 AND value2 is 2");
        if((value1 == 1) || (value2 == 1))
            System.out.println("value1 is 1 OR value2 is 1");
    }
}

```

Another conditional operator is `?:`, which can be thought of as shorthand for an if-then-else statement (discussed in the Control Flow Statements section of this lesson). This operator is also known as the ternary operator because it uses three operands. In the following example, this operator should be read as: "If someCondition is true, assign the value of value1 to result. Otherwise, assign the value of value2 to result."

The following program, `ConditionalDemo2`, tests the `?:` operator:

```

class ConditionalDemo2 {

```

```

public static void main(String[] args){
    int value1 = 1;
    int value2 = 2;
    int result;
    boolean someCondition = true;
    result = someCondition ? value1 : value2;

    System.out.println(result);
}
}

```

Because someCondition is true, this program prints "1" to the screen. Use the ?: operator instead of an if-then-else statement if it makes your code more readable; for example, when the expressions are compact and without side-effects (such as assignments).

The Type Comparison Operator instanceof

The instanceof operator compares an object to a specified type. You can use it to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface.

The following program, InstanceofDemo, defines a parent class (named Parent), a simple interface (named MyInterface), and a child class (named Child) that inherits from the parent and implements the interface.

```

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

        Parent obj1 = new Parent();
        Parent obj2 = new Child();

        System.out.println("obj1 instanceof Parent: "
            + (obj1 instanceof Parent));
        System.out.println("obj1 instanceof Child: "
            + (obj1 instanceof Child));
        System.out.println("obj1 instanceof MyInterface: "
            + (obj1 instanceof MyInterface));
        System.out.println("obj2 instanceof Parent: "
            + (obj2 instanceof Parent));
        System.out.println("obj2 instanceof Child: "
            + (obj2 instanceof Child));
        System.out.println("obj2 instanceof MyInterface: "
            + (obj2 instanceof MyInterface));
    }
}

class Parent {}
class Child extends Parent implements MyInterface {}
interface MyInterface {}

```

Output:

```
obj1 instanceof Parent: true
```



```
obj1 instanceof Child: false
obj1 instanceof MyInterface: false
obj2 instanceof Parent: true
obj2 instanceof Child: true
obj2 instanceof MyInterface: true
```

When using the instanceof operator, keep in mind that null is not an instance of anything.

Expressions, Statements, and Blocks

Now that you understand variables and operators, it's time to learn about expressions, statements, and blocks. Operators may be used in building expressions, which compute values; expressions are the core components of statements; statements may be grouped into blocks.

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. You've already seen examples of expressions, illustrated in bold below:

```
int cadence = 0;
```

```
anArray[0] = 100;
```

```
System.out.println("Element 1 at index 0: " + anArray[0]);
```

```
int result = 1 + 2; // result is now 3
```

```
if (value1 == value2)
```

```
    System.out.println("value1 == value2");
```

The data type of the value returned by an expression depends on the elements used in the expression. The expression `cadence = 0` returns an int because the assignment operator returns a value of the same data type as its left-hand operand; in this case, `cadence` is an int. As you can see from the other expressions, an expression can return other types of values as well, such as boolean or String.

The Java programming language allows you to construct compound expressions from various smaller expressions as long as the data type required by one part of the expression matches the data type of the other. Here's an example of a compound expression:

```
1 * 2 * 3
```

In this particular example, the order in which the expression is evaluated is unimportant because the result of multiplication is independent of order; the outcome

is always the same, no matter in which order you apply the multiplications. However, this is not true of all expressions. For example, the following expression gives different results, depending on whether you perform the addition or the division operation first:

$x + y / 100$ // ambiguous

You can specify exactly how an expression will be evaluated using balanced parenthesis: (and). For example, to make the previous expression unambiguous, you could write the following:

$(x + y) / 100$ // unambiguous, recommended

If you don't explicitly indicate the order for the operations to be performed, the order is determined by the precedence assigned to the operators in use within the expression. Operators that have a higher precedence get evaluated first. For example, the division operator has a higher precedence than does the addition operator. Therefore, the following two statements are equivalent:

$$\begin{array}{ccccccc} x & & + & & y & & / & & 100 \\ x + (y / 100) & // & \text{unambiguous, recommended} \end{array}$$

When writing compound expressions, be explicit and indicate with parentheses which operators should be evaluated first. This practice makes code easier to read and to maintain.

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*. Here are some examples of expression statements.

```
// assignment statement
aValue = 8933.234;
// increment statement
aValue++;
// method invocation statement
System.out.println("Hello World!");
// object creation statement
Bicycle myBike = new Bicycle();
```

In addition to expression statements, there are two other kinds of statements: declaration statements and control flow statements. A *declaration statement* declares a variable. You've seen many examples of declaration statements already:

```
// declaration statement
double aValue = 8933.234;
```

Finally, *control flow statements* regulate the order in which statements get executed. You'll learn about control flow statements in the next section, Control Flow Statements

Blocks

A block is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed. The following example, BlockDemo, illustrates the use of blocks:

```
class BlockDemo {
    public static void main(String[] args) {
        boolean condition = true;
        if (condition) { // begin block 1
            System.out.println("Condition is true.");
        } // end block one
        else { // begin block 2
            System.out.println("Condition is false.");
        } // end block 2
    }
}
```

```
}  
}
```

Control Flow Statements

The statements inside your source files are generally executed from top to bottom, in the order that they appear. Control flow statements, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to conditionally execute particular blocks of code. This section describes the decision-making statements (if-then, if-then-else, switch), the looping statements (for, while, do-while), and the branching statements (break, continue, return) supported by the Java programming language.

The if-then statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code only if a particular test evaluates to true.

The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false.

Unlike if-then and if-then-else, the switch statement allows for any number of possible execution paths.

The while and do-while statements continually execute a block of statements while a particular condition is true. The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the do block are always executed at least once.

The for statement provides a compact way to iterate over a range of values. It has two forms, one of which was designed for looping through collections and arrays.

The if-then and if-then-else Statements

The if-then Statement

The if-then statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code only if a particular test evaluates to true. For example, the Bicycle class could allow the brakes to decrease the bicycle's speed only if the bicycle is already in motion. One possible implementation of the applyBrakes method could be as follows:

```
void applyBrakes() {  
    // the "if" clause: bicycle must be moving  
    if (isMoving){  
        // the "then" clause: decrease current speed  
        currentSpeed--;  
    }  
}
```

If this test evaluates to false (meaning that the bicycle is not in motion), control jumps to the end of the if-then statement.

In addition, the opening and closing braces are optional, provided that the "then" clause contains only one statement:

```
void applyBrakes() {  
    // same as above, but without braces  
    if (isMoving)  
        currentSpeed--;  
}
```

Deciding when to omit the braces is a matter of personal taste. Omitting them can make the code more brittle. If a second statement is later added to the "then" clause, a common mistake would be forgetting to add the newly required braces. The compiler cannot catch this sort of error; you'll just get the wrong results.

The if-then-else Statement

The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false. You could use an if-then-else statement in the applyBrakes method to take some action if the brakes are applied when the bicycle is not in motion. In this case, the action is to simply print an error message stating that the bicycle has already stopped.

```
void applyBrakes() {  
    if (isMoving) {  
        currentSpeed--;  
    } else {  
        System.err.println("The bicycle has " + "already stopped!");  
    }  
}
```

The following program, IfElseDemo, assigns a grade based on the value of a test score: an A for a score of 90% or above, a B for a score of 80% or above, and so on.

```
class IfElseDemo {  
    public static void main(String[] args) {  
  
        int testscore = 76;  
        char grade;  
  
        if (testscore >= 90) {  
            grade = 'A';  
        } else if (testscore >= 80) {  
            grade = 'B';  
        } else if (testscore >= 70) {  
            grade = 'C';  
        } else if (testscore >= 60) {
```

```

        grade = 'D';
    } else {
        grade = 'F';
    }
    System.out.println("Grade = " + grade);
}
}

```

The output from the program is:

```
Grade = C
```

You may have noticed that the value of testscore can satisfy more than one expression in the compound statement: $76 \geq 70$ and $76 \geq 60$. However, once a condition is satisfied, the appropriate statements are executed (`grade = 'C';`) and the remaining conditions are not evaluated.

The switch Statement

Unlike if-then and if-then-else statements, the switch statement can have a number of possible execution paths. A switch works with the byte, short, char, and int primitive data types. It also works with enumerated types (discussed in Enum Types), the String class, and a few special classes that wrap certain primitive types: Character, Byte, Short, and Integer (discussed in Numbers and Strings).

The following code example, SwitchDemo, declares an int named month whose value represents a month. The code displays the name of the month, based on the value of month, using the switch statement.

```

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

        int month = 8;
        String monthString;
        switch (month) {
            case 1: monthString = "January";
                    break;
            case 2: monthString = "February";
                    break;
            case 3: monthString = "March";
                    break;
            case 4: monthString = "April";
                    break;
            case 5: monthString = "May";
                    break;
            case 6: monthString = "June";
                    break;
            case 7: monthString = "July";
                    break;

```

```

        case 8: monthString = "August";
                break;
        case 9: monthString = "September";
                break;
        case 10: monthString = "October";
                break;
        case 11: monthString = "November";
                break;
        case 12: monthString = "December";
                break;
        default: monthString = "Invalid month";
                break;
    }
    System.out.println(monthString);
}
}

```

In this case, August is printed to standard output.

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.

You could also display the name of the month with if-then-else statements:

```

int month = 8;
if (month == 1) {
    System.out.println("January");
} else if (month == 2) {
    System.out.println("February");
}
... // and so on

```

Deciding whether to use if-then-else statements or a switch statement is based on readability and the expression that the statement is testing. An if-then-else statement can test expressions based on ranges of values or conditions, whereas a switch statement tests expressions based only on a single integer, enumerated value, or String object.

Another point of interest is the break statement. 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. The program SwitchDemoFallThrough shows statements in a switch block that fall through. The program displays the month corresponding to the integer month and the months that follow in the year:

```

public class SwitchDemoFallThrough {

    public static void main(String args[]) {
        java.util.ArrayList<String> futureMonths =
            new java.util.ArrayList<String>();

        int month = 8;

        switch (month) {
            case 1: futureMonths.add("January");
            case 2: futureMonths.add("February");
            case 3: futureMonths.add("March");
            case 4: futureMonths.add("April");
            case 5: futureMonths.add("May");
            case 6: futureMonths.add("June");
            case 7: futureMonths.add("July");
            case 8: futureMonths.add("August");
            case 9: futureMonths.add("September");
            case 10: futureMonths.add("October");
            case 11: futureMonths.add("November");
            case 12: futureMonths.add("December");
                break;
            default: break;
        }

        if (futureMonths.isEmpty()) {
            System.out.println("Invalid month number");
        } else {
            for (String monthName : futureMonths) {
                System.out.println(monthName);
            }
        }
    }
}

```

This is the output from the code:

```

August
September
October
November
December

```

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 following code example, SwitchDemo2, shows how a statement can have multiple case labels. The code example calculates the number of days in a particular month:

```
class SwitchDemo2 {
    public static void main(String[] args) {

        int month = 2;
        int year = 2000;
        int numDays = 0;

        switch (month) {
            case 1: case 3: case 5:
            case 7: case 8: case 10:
            case 12:
                numDays = 31;
                break;
            case 4: case 6:
            case 9: case 11:
                numDays = 30;
                break;
            case 2:
                if (((year % 4 == 0) &&
                    !(year % 100 == 0))
                    || (year % 400 == 0))
                    numDays = 29;
                else
                    numDays = 28;
                break;
            default:
                System.out.println("Invalid month.");
                break;
        }
        System.out.println("Number of Days = "
            + numDays);
    }
}
```

This is the output from the code:

Number of Days = 29

Using Strings in switch Statements

In Java SE 7 and later, you can use a String object in the switch statement's expression. The following code example, StringSwitchDemo, displays the number of the month based on the value of the Stringnamed month:

```
public class StringSwitchDemo {

    public static int getMonthNumber(String month) {

        int monthNumber = 0;

        if (month == null) {
            return monthNumber;
        }

        switch (month.toLowerCase()) {
            case "january":
                monthNumber = 1;
                break;
            case "february":
                monthNumber = 2;
                break;
            case "march":
                monthNumber = 3;
                break;
            case "april":
                monthNumber = 4;
                break;
            case "may":
                monthNumber = 5;
                break;
            case "june":
                monthNumber = 6;
                break;
            case "july":
                monthNumber = 7;
                break;
            case "august":
                monthNumber = 8;
                break;
            case "september":
                monthNumber = 9;
                break;
            case "october":
                monthNumber = 10;
                break;
            case "november":
                monthNumber = 11;
                break;
            case "december":
                monthNumber = 12;
                break;
            default:
                monthNumber = 0;
                break;
        }
    }
}
```

```

    }

    return monthNumber;
}

public static void main(String[] args) {

    String month = "August";

    int returnedMonthNumber =
        StringSwitchDemo.getMonthNumber(month);

    if (returnedMonthNumber == 0) {
        System.out.println("Invalid month");
    } else {
        System.out.println(returnedMonthNumber);
    }
}
}

```

The output from this code is 8.

The String in the switch expression is compared with the expressions associated with each case label as if the String.equals method were being used. In order for the StringSwitchDemo example to accept any month regardless of case, month is converted to lowercase (with the toLowerCase method), and all the strings associated with the case labels are in lowercase.

Note: This example checks if the expression in the switch statement is null. Ensure that the expression in any switch statement is not null to prevent a NullPointerException from being thrown.

The while and do-while Statements

The while statement continually executes a block of statements while a particular condition is true. Its syntax can be expressed as:

```
while (expression) {  
    statement(s)  
}
```

The while statement evaluates *expression*, which must return a boolean value. If the expression evaluates to true, the while statement executes the *statement(s)* in the while block. The while statement continues testing the expression and executing its block until the expression evaluates to false. Using the while statement to print the values from 1 through 10 can be accomplished as in the following [WhileDemo](#) program:

```
class WhileDemo {  
    public static void main(String[] args){  
        int count = 1;  
        while (count < 11) {  
            System.out.println("Count is: "  
                               + count);  
            count++;  
        }  
    }  
}
```

You can implement an infinite loop using the `while` statement as follows:

```
while (true){  
    // your code goes here  
}
```

The Java programming language also provides a `do-while` statement, which can be expressed as follows:

```
do {  
    statement(s)  
} while (expression);
```

The difference between `do-while` and `while` is that `do-while` evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the `do` block are always executed at least once, as shown in the following [DoWhileDemo](#) program:

```
class DoWhileDemo {  
    public static void main(String[] args){  
        int count = 1;  
        do {  
            System.out.println("Count is: "  
                               + count);  
            count++;  
        } while (count < 11);  
    }  
}
```

The for Statement

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)  
}
```

When using this version of the `for` statement, keep in mind that:

- The *initialization* expression initializes the loop; it's executed once, as the loop begins.
- When the *termination* expression evaluates to false, the loop terminates.

- The *increment* expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment *or* decrement a value.

The following program, [ForDemo](#), uses the general form of the `for` statement to print the numbers 1 through 10 to standard output:

```
class ForDemo {
    public static void main(String[] args){
        for(int i=1; i<11; i++){
            System.out.println("Count is: "
                               + i);
        }
    }
}
```

The output of this program is:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

Notice how the code declares a variable within the initialization expression. The scope of this variable extends from its declaration to the end of the block governed by the `for` statement, so it can be used in the termination and increment expressions as well. If the variable that controls a `for` statement is not needed outside of the loop, it's best to declare the variable in the initialization expression. The names `i`, `j`, and `k` are often used to control `for` loops; declaring them within the initialization expression limits their life span and reduces errors.

The three expressions of the `for` loop are optional; an infinite loop can be created as follows:

```
// infinite loop
for ( ; ; ) {

    // your code goes here
}
```

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. To demonstrate, consider the following array, which holds the numbers 1 through 10:

```
int[] numbers = { 1,2,3,4,5,6,7,8,9,10};
```

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);
        }
    }
}
```

In this example, the variable `item` holds the current value from the numbers array. The output from this program is the same as before:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

Above is the recommended form of the `for` statement instead of the general form whenever possible.

Branching Statements

The break Statement

The `break` statement has two forms: labeled and unlabeled. You saw the unlabeled form in the previous discussion of the `switch` statement. You can also use an unlabeled `break` to terminate a `for`, `while`, or `do-while` loop, as shown in the following [BreakDemo](#) program:

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

        int[] arrayOfInts =
            { 32, 87, 3, 589,
              12, 1076, 2000,
              8, 622, 127 };
        int searchfor = 12;

        int i;
        boolean foundIt = false;

        for (i = 0; i < arrayOfInts.length; i++) {
            if (arrayOfInts[i] == searchfor) {
                foundIt = true;
                break;
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor + " at index " + i);
        } else {
            System.out.println(searchfor + " not in the array");
        }
    }
}
```

This program searches for the number 12 in an array. The `break` statement, shown in boldface, terminates the `for` loop when that value is found. Control flow then transfers to the statement after the `for` loop. This program's output is:

```
Found 12 at index 4
```

An unlabeled `break` statement terminates the innermost `switch`, `for`, `while`, or `do-while` statement, but a labeled `break` terminates an outer statement. The following program, [BreakWithLabelDemo](#), is similar to the previous program, but uses nested `for` loops to search for a value in a two-dimensional array. When the value is found, a labeled `break` terminates the outer `for` loop (labeled "search"):

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



```

int[][] arrayOfInts = {
    { 32, 87, 3, 589 },
    { 12, 1076, 2000, 8 },
    { 622, 127, 77, 955 }
};
int searchfor = 12;

int i;
int j = 0;
boolean foundIt = false;

search:
for (i = 0; i < arrayOfInts.length; i++) {
    for (j = 0; j < arrayOfInts[i].length;
        j++) {
        if (arrayOfInts[i][j] == searchfor) {
            foundIt = true;
            break search;
        }
    }
}

if (foundIt) {
    System.out.println("Found " + searchfor +
        " at " + i + ", " + j);
} else {
    System.out.println(searchfor +
        " not in the array");
}
}
}

```

This is the output of the program.

```
Found 12 at 1, 0
```

The `break` statement terminates the labeled statement; it does not transfer the flow of control to the label. Control flow is transferred to the statement immediately following the labeled (terminated) statement.

The continue Statement

The `continue` statement skips the current iteration of a `for`, `while`, or `do-while` loop. The unlabeled form skips to the end of the innermost loop's body and evaluates the boolean expression that controls the loop. The following program, [ContinueDemo](#), steps through a `String`, counting the occurrences of the letter "p". If the current character is not a p, the `continue` statement skips the rest of the loop and proceeds to the next character. If it *is* a "p", the program increments the letter count.

```

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

```

```

String searchMe
    = "peter piper picked a " +
      "peck of pickled peppers";
int max = searchMe.length();
int numPs = 0;

for (int i = 0; i < max; i++) {
    // interested only in p's
    if (searchMe.charAt(i) != 'p')
        continue;

    // process p's
    numPs++;
}
System.out.println("Found " +
    numPs + " p's in the string.");
}
}

```

Here is the output of this program:

Found 9 p's in the string.

To see this effect more clearly, try removing the `continue` statement and recompiling. When you run the program again, the count will be wrong, saying that it found 35 p's instead of 9.

A labeled `continue` statement skips the current iteration of an outer loop marked with the given label. The following example program, `ContinueWithLabelDemo`, uses nested loops to search for a substring within another string. Two nested loops are required: one to iterate over the substring and one to iterate over the string being searched. The following program, [ContinueWithLabelDemo](#), uses the labeled form of `continue` to skip an iteration in the outer loop.

```

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

        String searchMe
            = "Look for a substring in me";
        String substring = "sub";
        boolean foundIt = false;

        int max = searchMe.length() -
            substring.length();

        test:
        for (int i = 0; i <= max; i++) {
            int n = substring.length();
            int j = i;
            int k = 0;
            while (n-- != 0) {
                if (searchMe.charAt(j++)
                    != substring.charAt(k++)) {

```

```

        continue test;
    }
}
foundIt = true;
break test;
}
System.out.println(foundIt ?
    "Found it" : "Didn't find it");
}
}

```

Here is the output from this program.

Found it

The return Statement

The last of the branching statements is the return statement. The return statement exits from the current method, and control flow returns to where the method was invoked. The return statement has two forms: one that returns a value, and one that doesn't. To return a value, simply put the value (or an expression that calculates the value) after the return keyword.

```
return ++count;
```

The data type of the returned value must match the type of the method's declared return value. When a method is declared void, use the form of return that doesn't return a value.

```
return;
```

Modifiers

The Java language has a wide variety of modifiers, including the following:

- Access Modifiers
- Non Access Modifiers

Access modifiers

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:

- Visible to the package the default. No modifiers are needed(default).
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

Non Access Modifiers:

Java provides a number of non-access modifiers to achieve many other functionality.

- The static modifier for creating class methods and variables
- The final modifier for finalizing the implementations of classes, methods, and variables.
- The abstract modifier for creating abstract classes and methods.
- The synchronized and volatile modifiers, which are used for threads.

package

Packages are used in Java in-order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier etc.

A Package can be defined as a grouping of related types(classes, interfaces, enumerations and annotations) providing access protection and name space management.

Some of the existing packages in Java are:

- java.lang - bundles the fundamental classes
- java.io - classes for input , output functions are bundled in this package

Programmers can define their own packages to bundle group of classes/interfaces etc. It is a good practice to group related classes implemented by you so that a programmers can easily determine that the classes, interfaces, enumerations, annotations are related.

Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classed.

Creating a package:

When creating a package, you should choose a name for the package and put a package statement with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

The package statement should 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.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be put into an unnamed package.

import Keyword:

If a class wants to use another class in the same package, the package name does not need to be used. Classes in the same package find each other without any special syntax.

Example:

Here a class named Boss is added to the payroll package that already contains Employee. The Boss can then refer to the Employee class without using the payroll prefix, as demonstrated by the following Boss class.

```
package payroll;

public class Boss
{
    public void payEmployee(Employee e)
    {
        e.mailCheck();
    }
}
```

What happens if Boss is not in the payroll package? The Boss class must then use one of the following techniques for referring to a class in a different package.

- The fully qualified name of the class can be used. For example:

```
payroll.Employee
```

- The package can be imported using the import keyword and the wild card (*). For example:

```
import payroll.*;
```

- The class itself can be imported using the import keyword. For example:

```
import payroll.Employee;
```

Note: A class file can contain any number of import statements. The import statements must appear after the package statement and before the class declaration.

Java Keywords:

The following list shows the reserved words in Java. These reserved words may not be used as constant or variable or any other identifier names.

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

Comments in Java

Java supports single line and multi-line comments very similar to c and c++. All characters available inside any comment are ignored by Java compiler.

```
public class MyFirstJavaProgram{

    /* This is my first java program.
    * This will print 'Hello World' as the output
    * This is an example of multi-line comments.
    */

    public static void main(String []args){
        // This is an example of single line comment
        /* This is also an example of single line comment. */
        System.out.println("Hello World");
    }
}
```

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.

When an escape sequence is encountered in a print statement, the compiler interprets it accordingly. For example, if you want to put quotes within quotes you must use the escape sequence, \", on the interior quotes. To print the sentence

She said "Hello!" to me.

you would write

```
System.out.println("She said \"Hello!\" to me.");
```

Formatting Numeric Print Output

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](#)

Here is a basic example:

```
int i = 461012;  
System.out.format("The value of i is: %d%n", i);
```

The `%d` specifies that the single variable is a decimal integer. The `%n` is a platform-independent newline character. The output is:

The value of i is: 461012

The following table lists some of the converters and flags that are used in the sample program, `TestFormat.java`, that follows the table.

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.

The following program shows some of the formatting that you can do with format. The output is shown within double quotes in the embedded comment:

```
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("%tI:%tM %tp%n", c, c, c); // --> "2:34 am"

    System.out.format("%tD%n", c);    // --> "05/29/06"
}
}

```


Characters and Strings

Most of the time, if you are using a single character value, you will use the primitive `char` type. 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.

Strings are a sequence of characters and are widely used in Java programming. In the Java programming language, strings are objects. The [String](#) class has over 60 methods and 13 constructors.

Most commonly, you create a string with a statement like

```
String s = "Hello world!";
```

rather than using one of the `String` constructors.

The `String` class has many methods to find and retrieve substrings; these can then be easily reassembled into new strings using the `+` concatenation operator.

The `String` class also includes a number of utility methods, among them `split()`, `toLowerCase()`, `toUpperCase()`, and `valueOf()`. The latter method is indispensable in converting user input strings to numbers. The `Number` subclasses also have methods for converting strings to numbers and vice versa.

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 = '\u0391';  
// 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.

Here is an example of boxing,

```
// the primitive char 'a' is boxed
// into the Character object ch
Character ch = 'a';
```

and here is an example of both boxing and unboxing,

```
// method parameter and return
// type = Character object
Character test(Character c) {...}

// primitive 'x' is boxed for method
// test, return is unboxed to char 'c'
char c = test('x');
```

Note: The `Character` class is immutable, so that once it is created, a `Character` object cannot be changed.

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](https://docs.oracle.com/javase/7/docs/api/java/lang/Character.html) 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 <code>String</code> object representing the specified character value — that is, a one-character string.

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.

Creating 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:

```
char[] helloArray = { 'h', 'e', 'l', 'l', 'o', ' ' };  
String helloString = new String(helloArray);  
System.out.println(helloString);
```

The last line of this code snippet displays hello.

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.

String Length

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. After the following two lines of code have been executed, len equals 17:

```
String palindrome = "Dot saw I was Tod";  
int len = palindrome.length();
```

A *palindrome* is a word or sentence that is symmetric—it is spelled the same forward and backward, ignoring case and punctuation. Here is a short and inefficient program to reverse a palindrome string. It invokes the String method charAt(i), which returns the ith character in the string, counting from 0.

```

public class StringDemo {
    public static void main(String[] args) {
        String palindrome = "Dot saw I was Tod";
        int len = palindrome.length();
        char[] tempCharArray = new char[len];
        char[] charArray = new char[len];

        // put original string in an
        // array of chars
        for (int i = 0; i < len; i++) {
            tempCharArray[i] =
                palindrome.charAt(i);
        }

        // reverse array of chars
        for (int j = 0; j < len; j++) {
            charArray[j] =
                tempCharArray[len - 1 - j];
        }

        String reversePalindrome =
            new String(charArray);
        System.out.println(reversePalindrome);
    }
}

```

Running the program produces this output:

```
doT saw I was toD
```

To accomplish the string reversal, the program had to convert the string to an array of characters (first for loop), reverse the array into a second array (second for loop), and then convert back to a string. The [String](#) class includes a method, `getChars()`, to convert a string, or a portion of a string, into an array of characters so we could replace the first for loop in the program above with

```
palindrome.getChars(0, len, tempCharArray, 0);
```

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" + "!"
```

which results in

```
"Hello, world!"
```

The + operator is widely used in print statements. For example:

```
String string1 = "saw I was ";  
System.out.println("Dot " + string1 + "Tod");
```

which prints

```
Dot saw I was Tod
```

Such a concatenation can be a mixture of any objects. For each object that is not a String, its toString() method is called to convert it to a String.

Note: The Java programming language does not permit literal strings to span lines in source files, so you must use the + concatenation operator at the end of each line in a multi-line string. For example:

```
String quote =  
    "Now is the time for all good " +  
    "men to come to the aid of their country.";
```

Breaking strings between lines using the + concatenation operator is, once again, very common in print statements.

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. For example, instead of

```
System.out.printf("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);
```

you can write

```
String fs;  
fs = String.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);
System.out.println(fs);
```

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);
```

The [ToStringDemo](#) example uses the `toString` method to convert a number to a string. The program then uses some string methods to compute the number of digits before and after the decimal point:

```
public class ToStringDemo {

    public static void main(String[] args) {
        double d = 858.48;
        String s = Double.toString(d);

        int dot = s.indexOf('.');

        System.out.println(dot + " digits " +
            "before decimal point.");
        System.out.println( (s.length() - dot - 1) +
            " digits after decimal point.");
    }
}
```

The output of this program is:

```
3 digits before decimal point.
2 digits after decimal point.
```

Manipulating Characters in a String

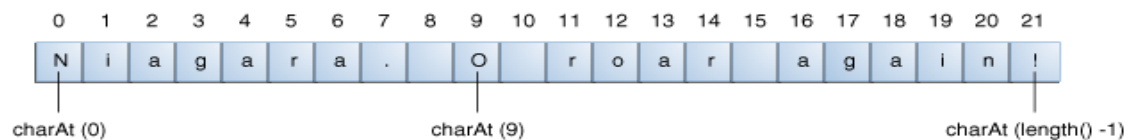
The String class has a number of methods for examining the contents of strings, finding characters or substrings within a string, changing case, and other tasks.

Getting Characters and Substrings by Index

You can get the character at a particular index within a string by invoking the `charAt()` accessor method. The index of the first character is 0, while the index of the last character is `length()-1`. For example, the following code gets the character at index 9 in a string:

```
String anotherPalindrome = "Niagara. O roar again!";  
char aChar = anotherPalindrome.charAt(9);
```

Indices begin at 0, so the character at index 9 is 'O', as illustrated in the following figure:



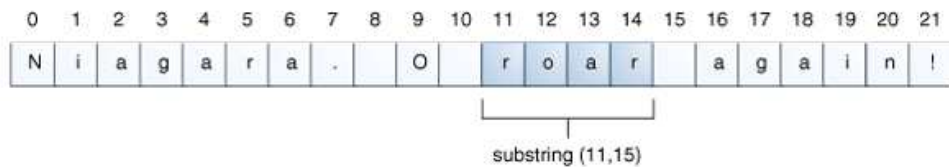
If you want to get more than one consecutive character from a string, you can use the substring method. The substring method has two versions, as shown in the following table:

The substring Methods in the String Class

Method	Description
<code>String substring(int beginIndex, int endIndex)</code>	Returns a new string that is a substring of this string. The first integer argument specifies the index of the first character. The second integer argument is the index of the last character - 1.
<code>String substring(int beginIndex)</code>	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.

The following code gets from the Niagara palindrome the substring that extends from index 11 up to, but not including, index 15, which is the word "roar":

```
String anotherPalindrome = "Niagara. O roar again!";  
String roar = anotherPalindrome.substring(11, 15);
```



Other Methods for Manipulating Strings

Here are several other String methods for manipulating strings:

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.

Searching for Characters and Substrings in a String

Here are some other String methods for finding characters or substrings within a string. The String class provides accessor methods that return the position within the string of a specific character or substring: `indexOf()` and `lastIndexOf()`. The `indexOf()` methods search forward from the beginning of the string, and the `lastIndexOf()` methods search backward from the end of the string. If a character or substring is not found, `indexOf()` and `lastIndexOf()` return -1.

The String class also provides a search method, `contains()`, that returns true if the string contains a particular character sequence. Use this method when you only need to know that the string contains a character sequence, but the precise location isn't important.

The following table describes the various string search methods.

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.

Note: CharSequence is an interface that is implemented by the String class. Therefore, you can use a string as an argument for the contains() method.

Replacing Characters and Substrings into a String

The String class has very few methods for inserting characters or substrings into a string. In general, they are not needed: You can create a new string by concatenation of substrings you have *removed* from a string with the substring that you want to insert.

The String class does have four methods for *replacing* found characters or substrings, however. They are:

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 regular expression regex with the specified replacement.

replacement)	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.

An Example

The following class, [Filename](#), illustrates the use of `lastIndexOf()` and `substring()` to isolate different parts of a file name.

Note: The methods in the following `Filename` class don't do any error checking and assume that their argument contains a full directory path and a filename with an extension. If these methods were production code, they would verify that their arguments were properly constructed.

```
public class Filename {
    private String fullPath;
    private char pathSeparator,
        extensionSeparator;

    public Filename(String str, char sep, char ext) {
        fullPath = str;
        pathSeparator = sep;
        extensionSeparator = ext;
    }

    public String extension() {
        int dot = fullPath.lastIndexOf(extensionSeparator);
        return fullPath.substring(dot + 1);
    }

    // gets filename without extension
    public String filename() {
        int dot = fullPath.lastIndexOf(extensionSeparator);
        int sep = fullPath.lastIndexOf(pathSeparator);
        return fullPath.substring(sep + 1, dot);
    }

    public String path() {
        int sep = fullPath.lastIndexOf(pathSeparator);
        return fullPath.substring(0, sep);
    }
}
```

Here is a program, [FilenameDemo](#), that constructs a `Filename` object and calls all of its methods:

```
public class FilenameDemo {
    public static void main(String[] args) {
        final String FPATH = "/home/user/index.html";
```

```

        Filename myHomePage = new Filename(FPATH, '/', '.');
        System.out.println("Extension = " + myHomePage.extension());
        System.out.println("Filename = " + myHomePage.filename());
        System.out.println("Path = " + myHomePage.path());
    }
}

```

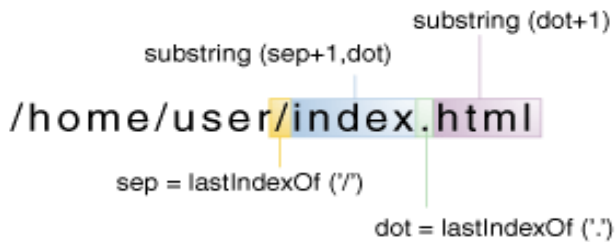
And here's the output from the program:

```

Extension = html
Filename = index
Path = /home/mem

```

As shown in the following figure, our extension method uses `lastIndexOf` to locate the last occurrence of the period (.) in the file name. Then `substring` uses the return value of `lastIndexOf` to extract the file name extension — that is, the substring from the period to the end of the string. This code assumes that the file name has a period in it; if the file name does not have a period, `lastIndexOf` returns -1, and the `substring` method throws a `StringIndexOutOfBoundsException`.



Also, notice that the extension method uses `dot + 1` as the argument to `substring`. If the period character (.) is the last character of the string, `dot + 1` is equal to the length of the string, which is one larger than the largest index into the string (because indices start at 0). This is a legal argument to `substring` because that method accepts an index equal to, but not greater than, the length of the string and interprets it to mean "the end of the string."

Comparing Strings and Portions of Strings

The String class has a number of methods for comparing strings and portions of strings. The following table lists these methods.

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)	<p>Tests whether the specified region of this string matches the specified region of the String argument.</p> <p>Region is of length len and begins at the index toffset for this string and ooffset for the other string.</p> <p>The boolean argument indicates whether case should be ignored; if true, case is ignored when comparing characters.</p>
boolean matches(String regex)	<p>Tests whether this string matches the specified regular expression. Regular expressions are discussed in the lesson titled "Regular Expressions."</p>

The following program, RegionMatchesDemo, uses the regionMatches method to search for a string within another string:

```

public class RegionMatchesDemo {
    public static void main(String[] args) {
        String searchMe = "Green Eggs and Ham";
        String findMe = "Eggs";
        int searchMeLength = searchMe.length();
        int findMeLength = findMe.length();
        boolean foundIt = false;
        for (int i = 0;
            i <= (searchMeLength - findMeLength);
            i++) {
            if (searchMe.regionMatches(i, findMe, 0, findMeLength)) {
                foundIt = true;
                System.out.println(searchMe.substring(i, i + findMeLength));
                break;
            }
        }
        if (!foundIt)
            System.out.println("No match found.");
    }
}

```

The output from this program is Eggs.

The program steps through the string referred to by searchMe one character at a time. For each character, the program calls the regionMatches method to determine whether

the substring beginning with the current character matches the string the program is looking for.

From Now onwards we will see more details about Java Class.

Classes

Subsequent sections of this lesson will back up and explain class declarations step by step.

Declaring Classes

You've seen classes defined in the following way:

```
class MyClass {  
    // field, constructor, and  
    // method declarations  
}
```

This is a class declaration. The *class body* (the area between the braces) contains all the code that provides for the life cycle of the objects created from the class: constructors for initializing new objects, declarations for the fields that provide the state of the class and its objects, and methods to implement the behavior of the class and its objects.

```
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.

You can also add modifiers like *public* or *private* at the very beginning so you can see that the opening line of a class declaration can become quite complicated. The modifiers *public* and *private*, which determine what other classes can access *MyClass*, are discussed later in this lesson.

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.

The `MyClass` class uses the following lines of code to define its fields:

```
class MyClass {  
    public int i;  
    public int j;  
    public int k;  
}
```

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.

The fields of `MyClass` are named `i`, `j`, and `k` and are all of data type integer (`int`). The `public` keyword identifies these fields as public members, accessible by any object that can access the class.

Types

All variables must have a type. You can use primitive types such as `int`, `float`, `boolean`, etc. Or you can use reference types, such as strings, arrays, or objects.

Variable Names

All variables, whether they are fields, local variables, or parameters, follow the same naming rules and conventions that were covered in the variables section.

In this lesson, be aware that the same naming rules and conventions are used for method and class names, except that

- the first letter of a class name should be capitalized, and
- the first (or only) word in a method name should be a verb.

Defining Methods

Here is an example of a typical method declaration:

```
public double calculateAnswer(double wingSpan, int numberOfEngines,
```

```
        double length, double grossTons) {  
    //do the calculation here  
}
```

The only required elements of a method declaration are the method's return type, name, a pair of parentheses, (), and a body between braces, {}.

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.

Modifiers, return types, and parameters will be discussed later in this lesson. Exceptions are discussed in a later lesson.

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

The signature of the method declared above is:

```
calculateAnswer(double, int, double, double)
```

Naming a Method

Although a method name can be any legal identifier, code conventions restrict method names. 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. Here are some examples:

```
run
runFast
getBackground
getFinalData
compareTo
setX
isEmpty
```

Typically, a method has a unique name within its class. However, a method might have the same name as other methods due to *method overloading*.

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 .

Suppose that you have a class that can use to calculate area of various types of shapes (Rectangle, Square and so on) and that contains a method for calculating area for each shape.

It is cumbersome to use a new name for each method

For example, reatangleArea, squareArea, and so on. In the Java programming language, you can use the same name for all the drawing methods but pass a different argument list to each method. Thus, the AreaCalculator class might declare four methods named area, each of which has a different parameter list.

```
public class AreaCalculator {
    ...
    public void area(int i) {
        ...
    }
    public void area(int i,int j) {
        ...
    }
    public void area(double j) {
        ...
    }
    public void draw(int i, double f) {
        ...
    }
}
```

```
}
```

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. In the code sample,

```
area(int s)
```

and

```
area(double i)
```

are distinct and unique methods because they require different argument types.

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.

For example, Bicycle has one constructor:

```
Public class Bicycle {  
    int gear;  
    int cadence;  
    int speed;  
  
    public Bicycle(int startCadence, int startSpeed, int startGear) {  
        gear = startGear;  
        cadence = startCadence;  
        speed = startSpeed;  
    }  
}
```

To create a new Bicycle object called myBike, a constructor is called by the new operator:

```
Bicycle myBike = new Bicycle(30, 0, 8);
```

new Bicycle(30, 0, 8) creates space in memory for the object and initializes its fields.

Although Bicycle only has one constructor, it could have others, including a no-argument constructor:

```
public Bicycle() {
    gear = 1;
    cadence = 10;
    speed = 0;
}
```

`Bicycle yourBike = new Bicycle();` invokes the no-argument constructor to create a new `Bicycle` object called `yourBike`.

Both constructors could have been declared in `Bicycle` because they have different argument lists. As with methods, the Java platform differentiates constructors on the basis of the number of arguments in the list and their types. You cannot write two constructors that have the same number and type of arguments for the same class, because the platform would not be able to tell them apart. Doing so causes a compile-time error.

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. For example, the following is a method that computes the monthly payments for a home loan, based on the amount of the loan, the interest rate, the length of the loan (the number of periods), and the future value of the loan:

```
public double computePayment(
    double loanAmt,
    double rate,
    double futureValue,
    int numPeriods) {
    double interest = rate / 100.0;
    double partial1 = Math.pow((1 + interest),
        - numPeriods);
    double denominator = (1 - partial1) / interest;
    double answer = (-loanAmt / denominator) - ((futureValue * partial1) / denominator);
    return answer;
}
```

This method has four parameters: the loan amount, the interest rate, the future value and the number of periods. The first three are double-precision floating point numbers, and the fourth is an integer. The parameters are used in the method body and at runtime will take on the values of the arguments that are passed in.

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.

Parameter Types

You can use any data type for a parameter of a method or a constructor. This includes primitive data types, such as doubles, floats, and integers, as you saw in the computePayment method, and reference data types, such as objects and arrays.

Here's an example of a method that accepts an array as an argument. In this example, the method creates a new Polygon object and initializes it from an array of Point objects (assume that Point is a class that represents an x, y coordinate):

```
public Polygon polygonFrom(Point[] corners) {  
    // method body goes here  
}
```

Parameter Names

When you declare a parameter to a method or a constructor, you provide a name for that parameter. This name is used within the method body to refer to the passed-in argument.

The name of a parameter must be unique in its scope. It cannot be the same as the name of another parameter for the same method or constructor, and it cannot be the name of a local variable within the method or constructor.

A parameter can have the same name as one of the class's fields. If this is the case, the parameter is said to *shadow* the field. Shadowing fields can make your code difficult to read and is conventionally used only within constructors and methods that set a particular field. For example, consider the following Circle class and its setOrigin method:

```
public class Circle {  
    private int x, y, radius;  
    public void setOrigin(int x, int y) {  
        ...  
    }  
}
```

The Circle class has three fields: x, y, and radius. The setOrigin method has two parameters, each of which has the same name as one of the fields. Each method parameter shadows the field that shares its name. So using the simple names x or y within the body of the method refers to the parameter, *not* to the field.

To access the field, you must use a qualified name. This will be discussed later in this lesson in the section titled "Using the this Keyword."

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. Here is an example:

```
public class PassPrimitiveByValue {  
  
    public static void main(String[] args) {  
  
        int x = 3;  
  
        // invoke passMethod() with  
        // x as argument  
        passMethod(x);  
  
        // print x to see if its  
        // value has changed  
        System.out.println("After invoking passMethod, x = " + x);  
  
    }  
  
    // change parameter in passMethod()  
    public static void passMethod(int p) {  
        p = 10;  
    }  
}
```

When you run this program, the output is:

After invoking passMethod, x = 3

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.

For example, consider a method in an arbitrary class that moves Circle objects:

```
public void moveCircle(Circle circle, int deltaX, int deltaY) {  
    // code to move origin of  
    // circle to x+deltaX, y+deltaY  
    circle.setX(circle.getX() + deltaX);  
    circle.setY(circle.getY() + deltaY);  
  
    // code to assign a new  
    // reference to circle  
    circle = new Circle(0, 0);  
}
```


Let the method be invoked with these arguments:

```
moveCircle(myCircle, 23, 56)
```

Inside the method, `circle` initially refers to `myCircle`. The method changes the `x` and `y` coordinates of the object that `circle` references (i.e., `myCircle`) by 23 and 56, respectively. These changes will persist when the method returns. Then `circle` is assigned a reference to a new `Circle` object with `x = y = 0`. This reassignment has no permanence, however, because the reference was passed in by value and cannot change. Within the method, the object pointed to by `circle` has changed, but, when the method returns, `myCircle` still references the same `Circle` object as before the method was called.

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.

Creating Objects

As you know, a class provides the blueprint for objects; you create an object from a class.

```
Point originOne = new Point(23, 94);  
Rectangle rectOne = new Rectangle(originOne, 100, 200);  
Rectangle rectTwo = new Rectangle(50, 100);
```

The first line creates an object of the [Point](#) class, and the second and third lines each create an object of the [Rectangle](#) class.

Each of these 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.

Declaring a Variable to Refer to an Object

Previously, you learned that to declare a variable, you write:

```
type name;
```


This notifies the compiler that you will use *name* to refer to data whose type is *type*. With a primitive variable, this declaration also reserves the proper amount of memory for the variable.

You can also declare a reference variable on its own line. For example:

```
Point originOne;
```

If you declare `originOne` like this, its value will be undetermined until an object is actually created and assigned to it. Simply declaring a reference variable does not create an object. For that, you need to use the `new` operator, as described in the next section. You must assign an object to `originOne` before you use it in your code. Otherwise, you will get a compiler error.

A variable in this state, which currently references no object, can be illustrated as follows (the variable name, `originOne`, plus a reference pointing to nothing):

`originOne` 

Instantiating a Class

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.

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.

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.

The `new` operator returns a reference to the object it created. This reference is usually assigned to a variable of the appropriate type, like:

```
Point originOne = new Point(23, 94);
```

Initializing an Object

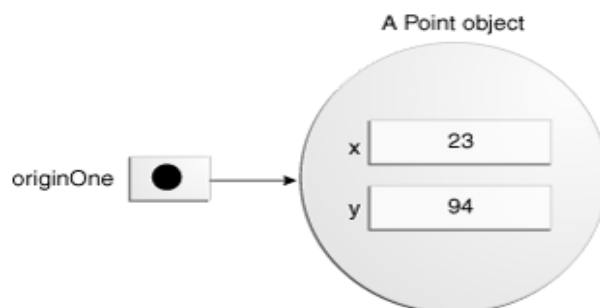
Here's the code for the Point class:

```
public class Point {  
    public int x = 0;  
    public int y = 0;  
    //constructor  
    public Point(int a, int b) {  
        x = a;  
        y = b;  
    }  
}
```

This class contains a single constructor. You can recognize a constructor because its declaration uses the same name as the class and it has no return type. The constructor in the Point class takes two integer arguments, as declared by the code (int a, int b). The following statement provides 23 and 94 as values for those arguments:

```
Point originOne = new Point(23, 94);
```

The result of executing this statement can be illustrated in the next figure:



Using Objects

Once you've created an object, you probably want to use it for something. You may need to use the value of one of its fields, change one of its fields, or call one of its methods to perform an action.

Referencing an Object's Fields

Object fields are accessed by their name. You must use a name that is unambiguous.

You may use a simple name for a field within its own class. For example, we can add a statement within the Rectangle class that prints the width and height:

```
System.out.println("Width and height are: " + width + ", " + height);
```

In this case, width and height are simple names.

Code that is outside the object's class must use an object reference or expression, followed by the dot (.) operator, followed by a simple field name, as in:

```
objectReference.fieldName
```

For example, the code in the `CreateObjectDemo` class is outside the code for the `Rectangle` class. So to refer to the `origin`, `width`, and `height` fields within the `Rectangle` object named `rectOne`, the `CreateObjectDemo` class must use the names `rectOne.origin`, `rectOne.width`, and `rectOne.height`, respectively. The program uses two of these names to display the width and the height of `rectOne`:

```
System.out.println("Width of rectOne: " + rectOne.width);  
System.out.println("Height of rectOne: "+ rectOne.height);
```

Calling an Object's Methods

You also use an object reference to invoke an object's method. You append the method's simple name to the object reference, with an intervening dot operator (.). Also, you provide, within enclosing parentheses, any arguments to the method. If the method does not require any arguments, use empty parentheses.

```
objectReference.methodName(argumentList);
```

or:

```
objectReference.methodName();
```

The `Rectangle` class has two methods: `getArea()` to compute the rectangle's area and `move()` to change the rectangle's origin. Here's the `CreateObjectDemo` code that invokes these two methods:

```
System.out.println("Area of rectOne: " + rectOne.getArea());  
...  
rectTwo.move(40, 72);
```

The first statement invokes `rectOne`'s `getArea()` method and displays the results. The second line moves `rectTwo` because the `move()` method assigns new values to the object's `origin.x` and `origin.y`.

As with instance fields, *objectReference* must be a reference to an object. You can use a variable name, but you also can use any expression that returns an object reference. The `new` operator returns an object reference, so you can use the value returned from `new` to invoke a new object's methods:

```
new Rectangle(100, 50).getArea()
```

The expression `new Rectangle(100, 50)` returns an object reference that refers to a `Rectangle` object. As shown, you can use the dot notation to invoke the new `Rectangle`'s `getArea()` method to compute the area of the new rectangle.

The Garbage Collector

Some object-oriented languages require that you keep track of all the objects you create and that you explicitly destroy them when they are no longer needed. Managing memory explicitly is tedious and error-prone.

The Java platform allows you to create as many objects as you want (limited, of course, by what your system can handle), and you don't have to worry about destroying them. The Java runtime environment deletes objects when it determines that they are no longer being used. This process is called *garbage collection*.

The Java runtime environment has a garbage collector that periodically frees the memory used by objects that are no longer referenced. The garbage collector does its job automatically when it determines that the time is right.

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.

For example, the `Point` class was written like this

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

but it could have been written like this:

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Each argument to the constructor shadows one of the object's fields — inside the constructor **x** is a local copy of the constructor's first argument. To refer to the `Point` field **x**, the constructor must use **this.x**.

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*. Here's another Rectangle class

```
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this(0, 0, 0, 0);
    }
    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    ...
}
```

This class contains a set of constructors. Each constructor initializes some or all of the rectangle's member variables. The constructors provide a default value for any member variable whose initial value is not provided by an argument. For example, the no-argument constructor calls the four-argument constructor with four 0 values and the two-argument constructor calls the four-argument constructor with two 0 values. As before, the compiler determines which constructor to call, based on the number and the type of arguments.

If present, the invocation of another constructor must be the first line in the constructor.

Nested Classes

The Java programming language allows you to define a class within another class. Such a class is called a *nested class* and is illustrated here:

Terminology: Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are simply called *static nested classes*. Non-static nested classes are called *inner classes*.

A nested class is a member of its enclosing class. Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private. Static nested classes do not have access to other members of the enclosing class. As a member of the OuterClass, a nested class can be declared private, public, protected, or *package private*. (Recall that outer classes can only be declared public or *package private*.)

Why Use Nested Classes?

There are several compelling reasons for using nested classes, among them:

- It is a way of logically grouping classes that are only used in one place.
- It increases encapsulation.
- Nested classes can lead to more readable and maintainable code.

Logical grouping of classes—If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.

Increased encapsulation—Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.

More readable, maintainable code—Nesting small classes within top-level classes places the code closer to where it is used.

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.

```
class OuterClass {  
    ...  
    static class StaticNestedClass {  
        ...  
    }  
    class InnerClass {  
        ...  
    }  
}
```

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();
```

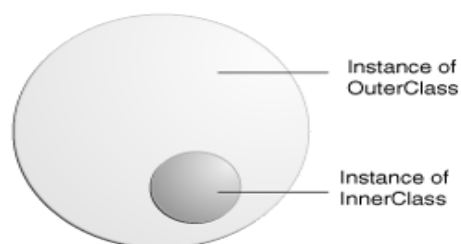
Inner Classes

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. The next figure illustrates this idea.



An Instance of InnerClass Exists Within an Instance of OuterClass

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 outerObject = new OuterClass();
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

Additionally, there are two special kinds of inner classes: local classes and anonymous classes (also called anonymous inner classes). Both of these will be discussed briefly in the next section.

Local and Anonymous Inner Classes

There are two additional types of inner classes. You can declare an inner class within the body of a method. Such a class is known as a local inner class. You can also declare an inner class within the body of a method without naming it. These classes are known as anonymous inner classes.

Local Class

A *local class* is declared locally within a block of Java code, rather than as a member of a class. Typically, a local class is defined within a method, but it can also be defined within a static initializer or instance initializer of a class.

```
class OuterClass {
    public void print(){
        class InnerClass {
            ...
        }
        InnerClass ic = new InnerClass();
    }
}
```

Local classes have the following interesting features:

- Like member classes, local classes are associated with a containing instance, and can access any members, including private members, of the containing class.
- In addition to accessing fields defined by the containing class, local classes can access any local variables, method parameters, or exception parameters that are in the scope of the local method definition and declared final.

Anonymous Classes

An *anonymous class* is a local class without a name. An anonymous class is defined and instantiated in a single succinct expression using the new operator.

While a local class definition is a statement in a block of Java code, an anonymous class definition is an expression, which means that it can be included as part of a larger expression, such as a method call.

When a local class is used only once, consider using anonymous class syntax, which places the definition and use of the class in exactly the same place.

```
new class-name ( [ argument-list ] ) { class-body }
```

or:

```
new interface-name () { class-body }
```

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 do to other class members.

Inheritance

Except for the `Object` class, a class has exactly one direct superclass. A class inherits fields and methods from all its superclasses, whether direct or indirect. A subclass can override methods that it inherits, or it can hide fields or methods that it inherits. (Note that hiding fields is generally bad programming practice.)

The table in [Overriding and Hiding Methods](#) section shows the effect of declaring a method with the same signature as a method in the superclass.

The `Object` class is the top of the class hierarchy. All classes are descendants from this class and inherit methods from it. Useful methods inherited from `Object` include `toString()`, `equals()`, `clone()`, and `getClass()`.

You can prevent a class from being subclassed by using the `final` keyword in the class's declaration. Similarly, you can prevent a method from being overridden by subclasses by declaring it as a `final` method.

An abstract class can only be subclassed; it cannot be instantiated. An abstract class can contain abstract methods—methods that are declared but not implemented. Subclasses then provide the implementations for the abstract methods.

In the preceding lessons, you have seen *inheritance* mentioned several times. In the Java language, classes can be *derived* from other classes, thereby *inheriting* fields and methods from those classes.

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.

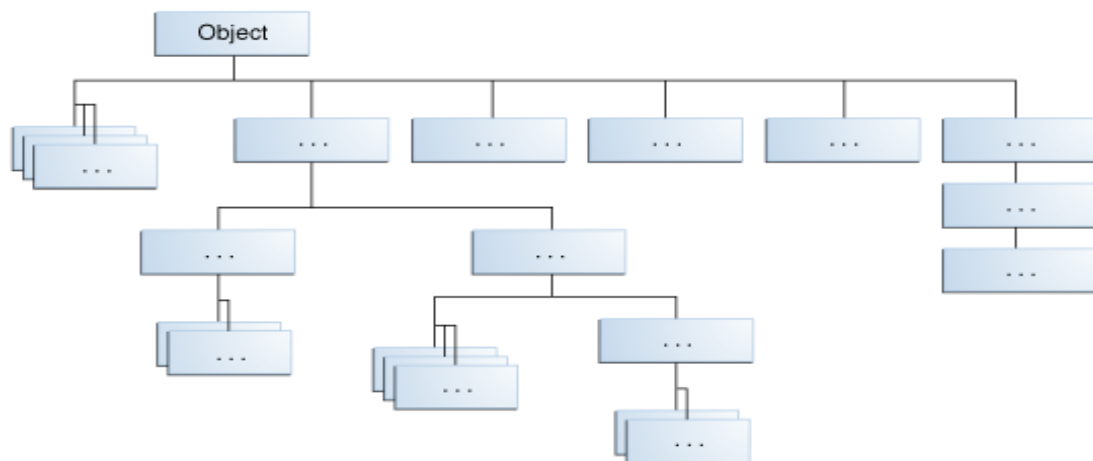
Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, Object. Such a class is said to be *descended* from all the classes in the inheritance chain stretching back to Object.

The idea of inheritance is simple but powerful: When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself.

A subclass inherits all the *members* (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

The Java Platform Class Hierarchy

The [Object](#) class, defined in the java.lang package, defines and implements behavior common to all classes—including the ones that you write. In the Java platform, many classes derive directly from Object, other classes derive from some of those classes, and so on, forming a hierarchy of classes.



All Classes in the Java Platform are Descendants of Object

At the top of the hierarchy, Object is the most general of all classes. Classes near the bottom of the hierarchy provide more specialized behaviour.

An Example of Inheritance

Here is the sample code for a possible implementation of a Bicycle class that was presented in the Classes and Objects lesson:

```
public class Bicycle {

    // the Bicycle class has
    // three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has
    // one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has
    // four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }

}
```

A class declaration for a MountainBike class that is a subclass of Bicycle might look like this:

```
public class MountainBike extends Bicycle {

    // the MountainBike subclass adds
    // one field
    public int seatHeight;

    // the MountainBike subclass has one
    // constructor
    public MountainBike(int startHeight,
                        int startCadence,
                        int startSpeed,
                        int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

}
```

```

    }

    // the MountainBike subclass adds
    // one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}

```

MountainBike inherits all the fields and methods of Bicycle and adds the field `seatHeight` and a method to set it. Except for the constructor, it is as if you had written a new MountainBike class entirely from scratch, with four fields and five methods. However, you didn't have to do all the work. This would be especially valuable if the methods in the Bicycle class were complex and had taken substantial time to debug.

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`.

Private Members in a Superclass

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.

Casting Objects

We have seen that an object is of the data type of the class from which it was instantiated. For example, if we write

```
public MountainBike myBike = new MountainBike();
```

then myBike is of type MountainBike.

MountainBike is descended from Bicycle and Object. Therefore, a MountainBike is a Bicycle and is also an Object, and it can be used wherever Bicycle or Object objects are called for.

The reverse is not necessarily true: a Bicycle *may be* a MountainBike, but it isn't necessarily. Similarly, an Object *may be* a Bicycle or a MountainBike, but it isn't necessarily.

Casting shows the use of an object of one type in place of another type, among the objects permitted by inheritance and implementations. For example, if we write

```
Object obj = new MountainBike();
```

then obj is both an Object and a Mountainbike (until such time as obj is assigned another object that is *not* a Mountainbike). This is called *implicit casting*.

If, on the other hand, we write

```
MountainBike myBike = obj;
```

we would get a compile-time error because obj is not known to the compiler to be a MountainBike. However, we can *tell* the compiler that we promise to assign a MountainBike to obj by *explicit casting*:

```
MountainBike myBike = (MountainBike)obj;
```

This cast inserts a runtime check that obj is assigned a MountainBike so that the compiler can safely assume that obj is a MountainBike. If obj is not a Mountainbike at runtime, an exception will be thrown.

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.

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 it overrides. An overriding method can also return a subtype of the type returned by the overridden method. This 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, it will generate an error. For more information on `@Override`, see [Annotations](#).

Class Methods

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

The distinction between hiding and overriding has important implications. The version of the overridden method that gets invoked is the one in the subclass. The version of the hidden method that gets invoked depends on whether it is invoked from the superclass or the subclass. Let's look at an example that contains two classes. The first is `Animal`, which contains one instance method and one class method:

```
public class Animal {
    public static void testClassMethod() {
        System.out.println("The class" + " method in Animal.");
    }
    public void testInstanceMethod() {
        System.out.println("The instance " + " method in Animal.");
    }
}
```

The second class, a subclass of `Animal`, is called `Cat`:

```
public class Cat extends Animal {
    public static void testClassMethod() {
        System.out.println("The class method" + " in Cat.");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method" + " in Cat.");
    }

    public static void main(String[] args) {
        Cat myCat = new Cat();
    }
}
```

```

        Animal myAnimal = myCat;
        Animal.testClassMethod();
        myAnimal.testInstanceMethod();
    }
}

```

The Cat class overrides the instance method in Animal and hides the class method in Animal. The main method in this class creates an instance of Cat and calls testClassMethod() on the class and testInstanceMethod() on the instance.

The output from this program is as follows:

```

The class method in Animal.
The instance method in Cat.

```

As promised, the version of the hidden method that gets invoked is the one in the superclass, and the version of the overridden method that gets invoked is the one in 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 class 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 methods—they are new methods, unique to the subclass.

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 void printDescription(){
    System.out.println("\nBike is " + "in gear " + this.gear
        + " with a cadence of " + this.cadence +
        " and travelling at a speed of " + this.speed + ". ");
}
```

To demonstrate polymorphic features in the Java language, extend the Bicycle class with a MountainBike and a RoadBike class. For MountainBike, add a field for suspension, which is a String value that indicates if the bike has a front shock absorber, Front. Or, the bike has a front and back shock absorber, Dual.

Here is the updated class:

```
public class MountainBike extends Bicycle {
    private String suspension;

    public MountainBike(
        int startCadence,
        int startSpeed,
        int startGear,
        String suspensionType){
        super(startCadence,
            startSpeed,
            startGear);
        this.setSuspension(suspensionType);
    }

    public String getSuspension(){
        return this.suspension;
    }

    public void setSuspension(String suspensionType) {
        this.suspension = suspensionType;
    }

    public void printDescription() {
        super.printDescription();
        System.out.println("The " + "MountainBike has a" +
            getSuspension() + " suspension.");
    }
}
```

Note the overridden printDescription method. In addition to the information provided before, additional data about the suspension is included to the output.

Next, create the RoadBike class. Because road or racing bikes have skinny tires, add an attribute to track the tire width. Here is the RoadBike class:

```
public class RoadBike extends Bicycle{
    // In millimeters (mm)
    private int tireWidth;

    public RoadBike(int startCadence,
                    int startSpeed,
                    int startGear,
                    int newTireWidth){
        super(startCadence,
              startSpeed,
              startGear);
        this.setTireWidth(newTireWidth);
    }

    public int getTireWidth(){
        return this.tireWidth;
    }

    public void setTireWidth(int newTireWidth){
        this.tireWidth = newTireWidth;
    }

    public void printDescription(){
        super.printDescription();
        System.out.println("The RoadBike"
                           " has " + getTireWidth() +
                           " MM tires.");
    }
}
```

Note that once again, the printDescription method has been overridden. This time, information about the tire width is displayed.

To summarize, there are three classes: Bicycle, MountainBike, and RoadBike. The two subclasses override the printDescription method and print unique information.

Here is a test program that creates three Bicycle variables. Each variable is assigned to one of the three bicycle classes. Each variable is then printed.

```
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 following is the output from the test program:

Bike is in gear 1 with a cadence of 20 and travelling at a speed of 10.

Bike is in gear 5 with a cadence of 20 and travelling at a speed of 10.
The MountainBike has a Dual suspension.

Bike is in gear 8 with a cadence of 40 and travelling at a speed of 20.
The RoadBike has 23 MM tires.

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`, which is covered in the next section.

Generally speaking, java doesn't recommend hiding fields as it makes code difficult to read.

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). Consider this class, `Superclass`:

```
public class Superclass {

    public void printMethod() {
        System.out.println("Printed in Superclass.");
    }
}
```

Here is a subclass, called `Subclass`, that overrides `printMethod()`:

```
public class Subclass extends Superclass {

    // overrides printMethod in Superclass
```

```

    public void printMethod() {
        super.printMethod();
        System.out.println("Printed in Subclass");
    }
    public static void main(String[] args) {
        Subclass s = new Subclass();
        s.printMethod();
    }
}

```

Within Subclass, the simple name `printMethod()` refers to the one declared in Subclass, which overrides the one in Superclass. So, to refer to `printMethod()` inherited from Superclass, Subclass must use a qualified name, using `super` as shown. Compiling and executing Subclass prints the following:

```

Printed in Superclass.
Printed in Subclass

```

Subclass Constructors

The following example illustrates how to use the `super` keyword to invoke a superclass's constructor. Recall from the Bicycle example that MountainBike is a subclass of Bicycle. Here is the MountainBike(subclass) constructor that calls the superclass constructor and then adds initialization code of its own:

```

public MountainBike(int startHeight,
                    int startCadence,
                    int startSpeed,
                    int startGear) {
    super(startCadence, startSpeed, startGear);
    seatHeight = startHeight;
}

```

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)`

Note: There are some subtle aspects to a number of these methods, especially the clone method. You can get information on the correct usage of these methods in the book [Effective Java](#) by Josh Bloch.

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. Exception handling will be covered in a later lesson. 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.

If the object on which clone() was invoked does implement the Cloneable interface, Object's implementation of the clone() method creates an object of the same class as the original object and initializes the new object's member variables to have the same values as the original object's corresponding member variables.

The simplest way to make your class cloneable is to add implements Cloneable to your class's declaration. then your objects can invoke the clone() method.

For some classes, the default behavior of Object's clone() method works just fine. If, however, an object contains a reference to an external object, say ObjExternal, you may need to override clone() to get correct behavior. Otherwise, a change in ObjExternal made by one object will be visible in its clone also. This means that the original object and its clone are not independent—to decouple them, you must override clone() so that it clones the object *and* ObjExternal. Then the original object references ObjExternal and the clone references a clone of ObjExternal, so that the object and its clone are truly independent.

The equals() Method

The equals() method compares two objects for equality and returns true if they are equal. The equals() method provided in the Object class uses the identity operator (==) to determine whether two objects are equal. For primitive data types, this gives the correct result. For objects, however, it does not. The equals() method provided by Object tests whether the object *references* are equal that is, if the objects compared are the exact same object.

To test whether two objects are equal in the sense of *equivalency* (containing the same information), you must override the equals() method. Here is an example of a Book class that overrides equals():

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

Consider this code that tests two instances of the Book class for equality:

```
// Swing Tutorial, 2nd edition
Book firstBook = new Book("0201914670");
Book secondBook = new Book("0201914670");
if (firstBook.equals(secondBook)) {
    System.out.println("objects are equal");
} else {
    System.out.println("objects are not equal");
}
```

This program displays objects are equal even though firstBook and secondBook reference two distinct objects. They are considered equal because the objects compared contain the same ISBN number.

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, such as an instance of `Book`:

```
System.out.println(firstBook.toString());
```

which would, for a properly overridden `toString()` method, print something useful, like this:

ISBN: 0201914670; The Swing Tutorial; A Guide to Constructing GUIs, 2nd Edition

Using `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.

We can declare class fields as final to use them as constants.

Interfaces

An interface defines a protocol of communication between two objects.

An interface declaration contains signatures, but no implementations, for a set of methods, and might also contain constant definitions.

A class that implements an interface must implement all the methods declared in the interface.

An interface name can be used anywhere a type can be used.

Interfaces in Java

In the Java programming language, an *interface* is a reference type, similar to a class, that can contain *only* constants, method signatures, and nested types. There are no method bodies. Interfaces cannot be instantiated they can only be *implemented* by classes or *extended* by other interfaces.

Defining an interface is similar to creating a new class:

```
public interface OperateCar {
    // constant declarations, if any
    // method signatures

    // An enum with values RIGHT, LEFT
    int turn(Direction direction,
             double radius,
             double startSpeed,
```

```

        double endSpeed);
int changeLanes(Direction direction,
                double startSpeed,
                double endSpeed);
int signalTurn(Direction direction,
               boolean signalOn);
int getRadarFront(double distanceToCar,
                  double speedOfCar);
int getRadarRear(double distanceToCar,
                  double speedOfCar);
.....
// more method signatures
}

```

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. For example,

```

public class OperateBMW760i implements OperateCar {

    // the OperateCar method signatures, with implementation --
    // for example:
    int signalTurn(Direction direction, boolean signalOn) {
        // code to turn BMW's LEFT turn indicator lights on
        // code to turn BMW's LEFT turn indicator lights off
        // code to turn BMW's RIGHT turn indicator lights on
        // code to turn BMW's RIGHT turn indicator lights off
    }

    // other members, as needed -- for example, helper classes not
    // visible to clients of the interface
}

```

Interfaces and Multiple Inheritance

Interfaces have another very important role in the Java programming language. Interfaces are not part of the class hierarchy, although they work in combination with classes. The Java programming language does not permit multiple inheritance, but interfaces provide an alternative.

In Java, a class can inherit from only one class but it can implement more than one interface. Therefore, objects can have multiple types: the type of their own class and the types of all the interfaces that they implement. This means that if a variable is declared to be the type of an interface, its value can reference any object that is instantiated from any class that implements the interface

Defining an Interface

An interface declaration consists of modifiers, the keyword `interface`, the interface name, a comma-separated list of parent interfaces (if any), and the interface body. For example:

```
public interface GroupedInterface extends Interface1, Interface2, Interface3 {

    // constant declarations

    // base of natural logarithms
    double E = 2.718282;

    // method signatures
    void doSomething (int i, double x);
    int doSomethingElse(String s);
}
```

The public access specifier indicates that the interface can be used by any class in any package. If you do not specify that the interface is public, your interface will be accessible only to classes defined in the same package as the interface.

An interface can extend other interfaces, just as a class can extend or subclass another class. However, whereas a class can extend only one other class, an interface can extend any number of interfaces. The interface declaration includes a comma-separated list of all the interfaces that it extends.

The Interface Body

The interface body contains method declarations for all the methods included in the interface. A method declaration within an interface is followed by a semicolon, but no braces, because an interface does not provide implementations for the methods declared within it. All methods declared in an interface are implicitly public, so the public modifier can be omitted.

An interface can contain constant declarations in addition to method declarations. All constant values defined in an interface are implicitly public, static, and final. Once again, these modifiers can be omitted.

Implementing an Interface

To declare a class that implements an interface, you include an implements clause in the class declaration. Your class can implement more than one interface, so the implements keyword is followed by a comma-separated list of the interfaces implemented by the class. By convention, the implements clause follows the extends clause, if there is one.

A Sample Interface, Relatable

Consider an interface that defines how to compare the size of objects.

```
public interface Relatable {

    // this (object calling isLargerThan)
    // and other must be instances of
    // the same class returns 1, 0, -1
    // if this is greater // than, equal
    // to, or less than other
    public int isLargerThan(Relatable other);
}
```

```
}
```

If you want to be able to compare the size of similar objects, no matter what they are, the class that instantiates them should implement `Relatable`.

Any class can implement `Relatable` if there is some way to compare the relative "size" of objects instantiated from the class. For strings, it could be number of characters; for books, it could be number of pages; for students, it could be weight; and so forth. For planar geometric objects, area would be a good choice (see the `RectanglePlus` class that follows), while volume would work for three-dimensional geometric objects. All such classes can implement the `isLargerThan()` method.

If you know that a class implements `Relatable`, then you know that you can compare the size of the objects instantiated from that class.

Implementing the Relatable Interface

Here is the `Rectangle` class rewritten to implement `Relatable`.

```
public class RectanglePlus implements Relatable {
    public int width = 0;
    public int height = 0;
    public Point origin;

    // four constructors
    public RectanglePlus() {
        origin = new Point(0, 0);
    }
    public RectanglePlus(Point p) {
        origin = p;
    }
    public RectanglePlus(int w, int h) {
        origin = new Point(0, 0);
        width = w;
        height = h;
    }
    public RectanglePlus(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }

    // a method for moving the rectangle
    public void move(int x, int y) {
        origin.x = x;
        origin.y = y;
    }

    // a method for computing
    // the area of the rectangle
    public int getArea() {
        return width * height;
    }

    // a method required to implement
    // the Relatable interface
```

```

public int isLargerThan(Relatable other) {
    RectanglePlus otherRect
    = (RectanglePlus)other;
    if (this.getArea() < otherRect.getArea())
        return -1;
    else if (this.getArea() > otherRect.getArea())
        return 1;
    else
        return 0;
}
}

```

Because RectanglePlus implements Relatable, the size of any two RectanglePlus objects can be compared.

Note: The isLargerThan method, as defined in the Relatable interface, takes an object of type Relatable. The line of code, shown in bold in the previous example, casts other to a RectanglePlus instance. Type casting tells the compiler what the object really is. Invoking getArea directly on the other instance (other.getArea()) would fail to compile because the compiler does not understand that other is actually an instance of RectanglePlus.

Abstract Methods and Classes

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, the class itself *must* be declared abstract, as in:

```

public abstract class GraphicObject {
    // declare fields
    // declare non-abstract 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, the subclass must also be declared abstract.

Note: All of the methods in an *interface* (see the [Interfaces](#) section) are *implicitly* abstract, so the abstract modifier is not used with interface methods (it could be it's just not necessary).

Abstract Classes versus Interfaces

Unlike interfaces, abstract classes can contain fields that are not static and final, and they can contain implemented methods. Such abstract classes are similar to interfaces, except that they provide a partial implementation, leaving it to subclasses to complete the implementation. If an abstract class contains *only* abstract method declarations, it should be declared as an interface instead.

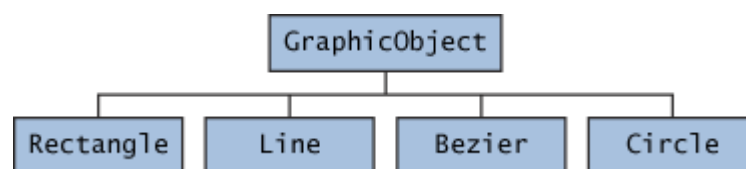
Multiple interfaces can be implemented by classes anywhere in the class hierarchy, whether or not they are related to one another in any way. Think of Comparable or Cloneable, for example.

By comparison, abstract classes are most commonly subclassed to share pieces of implementation. A single abstract class is subclassed by similar classes that have a lot in common (the implemented parts of the abstract class), but also have some differences (the abstract methods).

An Abstract Class Example

In an object-oriented drawing application, you can draw circles, rectangles, lines, Bezier curves, and many other graphic objects. These objects all have certain states (for example: position, orientation, line color, fill color) and behaviors (for example: moveTo, rotate, resize, draw) in common. Some of these states and behaviors are the same for all graphic objects for example: position, fill color, and moveTo. Others require different implementations for example, resize or draw.

All GraphicObjects must know how to draw or resize themselves; they just differ in how they do it. This is a perfect situation for an abstract superclass. You can take advantage of the similarities and declare all the graphic objects to inherit from the same abstract parent object for example, GraphicObject, as shown in the following figure.



Classes Rectangle, Line, Bezier, and Circle inherit from GraphicObject

First, you declare an abstract class, GraphicObject, to provide member variables and methods that are wholly shared by all subclasses, such as the current position and the moveTo method. GraphicObject also declares abstract methods for methods, such as draw or resize, that need to be implemented by all subclasses but must be implemented in different ways. The GraphicObject class can look something like this:

```
abstract class GraphicObject {
    int x, y;
    ...
    void moveTo(int newX, int newY) {
        ...
    }
    abstract void draw();
}
```

```
    abstract void resize();  
}
```

Each non-abstract subclass of `GraphicObject`, such as `Circle` and `Rectangle`, must provide implementations for the `draw` and `resize` methods:

```
class Circle extends GraphicObject {  
    void draw() {  
        ...  
    }  
    void resize() {  
        ...  
    }  
}  
class Rectangle extends GraphicObject {  
    void draw() {  
        ...  
    }  
    void resize() {  
        ...  
    }  
}
```

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 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`.

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.

Exceptions

A program can use exceptions to indicate that an error occurred. To throw an exception, use the throw statement and provide it with an exception object—a descendant of Throwable—to provide information about the specific error that occurred. A method that throws an uncaught, checked exception must include a throws clause in its declaration.

A program can catch exceptions by using a combination of the try, catch, and finally blocks.

- The try block identifies a block of code in which an exception can occur.
- The catch block identifies a block of code, known as an exception handler, that can handle a particular type of exception.
- The finally block identifies a block of code that is guaranteed to execute, and is the right place to close files, recover resources, and otherwise clean up after the code enclosed in the try block.

The try statement should contain at least one catch block or a finally block and may have multiple catch blocks.

The class of the exception object indicates the type of exception thrown. The exception object can contain further information about the error, including an error message. With exception chaining, an exception can point to the exception that caused it, which can in turn point to the exception that caused it, and so on.

What Is an Exception?

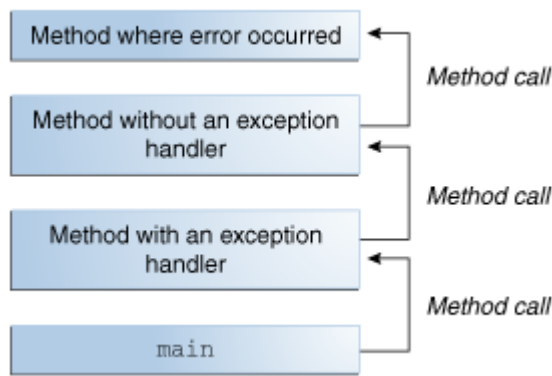
The term *exception* is shorthand for the phrase "exceptional event."

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*.

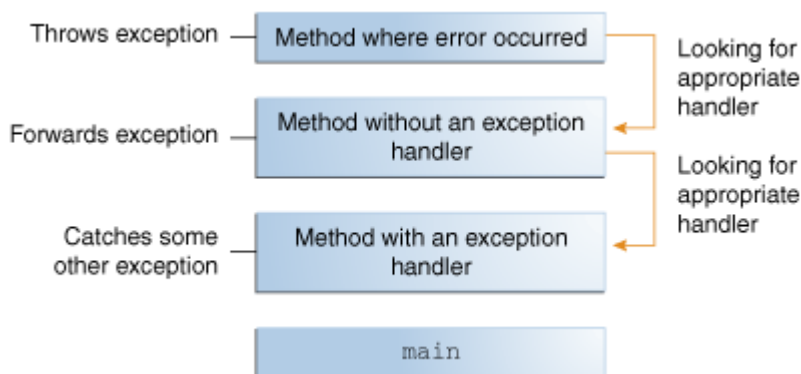
After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the *call stack* (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 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.IOException`. 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 `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 last part of this section walks through an example and analyzes what occurs during various scenarios.

The following example defines and implements a class named `ListOfNumbers`. When constructed, `ListOfNumbers` creates an `ArrayList` that contains 10 `Integer` elements with sequential values 0 through 9. The `ListOfNumbers` class also defines a method

named `writeList`, which writes the list of numbers into a text file called `OutFile.txt`. This example uses output classes defined in `java.io`

```
// Note: This class won't compile by design!
import java.io.*;
import java.util.List;
import java.util.ArrayList;

public class ListOfNumbers {

    private List<Integer> list;
    private static final int SIZE = 10;

    public ListOfNumbers () {
        list = new ArrayList<Integer>(SIZE);
        for (int i = 0; i < SIZE; i++) {
            list.add(new Integer(i));
        }
    }

    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();
    }
}
```

The first line in boldface is a call to a constructor. The constructor initializes an output stream on a file. If the file cannot be opened, the constructor throws an `IOException`.

The second boldface line is a call to the `ArrayList` class's `get` method, which throws an `IndexOutOfBoundsException` if the value of its argument is too small (less than 0) or too large (more than the number of elements currently contained by the `ArrayList`).

If you try to compile the [ListOfNumbers](#) class, the compiler prints an error message about the exception thrown by the `FileWriter` constructor. However, it does not display an error message about the exception thrown by `get`. The reason is that the exception thrown by the constructor, `IOException`, is a checked exception, and the one thrown by the `get` method, `IndexOutOfBoundsException`, is an unchecked exception.

Now that you're familiar with the `ListOfNumbers` class and where the exceptions can be thrown within it, you're ready to write exception handlers to catch and handle those exceptions.

The try Block

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:

```
try {
```

```
    code
}
catch and finally blocks . . .
```

The segment in the example labeled *code* contains one or more legal lines of code that could throw an exception. (The catch and finally blocks are explained in the next two subsections.)

To construct an exception handler for the `writeList` method from the `ListOfNumbers` class, enclose the exception-throwing statements of the `writeList` method within a try block. There is more than one way to do this. You can put each line of code that might throw an exception within its own try block and provide separate exception handlers for each. Or, you can put all the `writeList` code within a single try block and associate multiple handlers with it. The following listing uses one try block for the entire method because the code in question is very short.

```
private List<Integer> list;
private static final int SIZE = 10;

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 statements . . .
```

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; the next section, [The catch Blocks](#), shows you how.

The catch Blocks

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 and 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.

The following are two exception handlers for the writeList method one for two types of checked exceptions that can be thrown within the try statement:

```
try {

} catch (FileNotFoundException e) {
    System.err.println("FileNotFoundException: " + e.getMessage());
    throw new SampleException(e);
} catch (IOException e) {
    System.err.println("Caught IOException: " + e.getMessage());
}
```

Both handlers print an error message. The second handler does nothing else. By catching any IOException that's not caught by the first handler, it allows the program to continue executing.

The first handler, in addition to printing a message, throws a user-defined exception. (Throwing exceptions is covered in detail later in this chapter in the [How to Throw Exceptions](#) section.)

In this example, when the FileNotFoundException is caught it causes a user-defined exception called SampleException to be thrown. You might want to do this if you want your program to handle an exception in this situation in a specific way.

Exception handlers can do more than just print error messages or halt the program. They can do error recovery, prompt the user to make a decision, or propagate the error up to a higher-level handler using chained exceptions, as described in the [Chained Exceptions](#) section.

Catching More Than One Type of Exception with One Exception Handler

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

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 try block of the `writeList` method that you've been working with here opens a `PrintWriter`. The program should close that stream before exiting the `writeList` method. This poses a somewhat complicated problem because `writeList`'s try block can exit in one of three ways.

1. The new `FileWriter` statement fails and throws an `IOException`.
2. The `vector.elementAt(i)` statement fails and throws an `ArrayIndexOutOfBoundsException`.
3. Everything succeeds and the try block exits normally.

The runtime system always executes the statements within the finally block regardless of what happens within the try block. So it's the perfect place to perform cleanup.

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");
    }
}
}

```

In the `writeList` example, you could provide for cleanup without the intervention of a `finally` block. For example, you could put the code to close the `PrintWriter` at the end of the `try` block and again within the exception handler for `ArrayIndexOutOfBoundsException`, as follows.

```

try {

    // Don't do this; it duplicates code.
    out.close();

} catch (FileNotFoundException e) {
    // Don't do this; it duplicates code.
    out.close();

    System.err.println("Caught FileNotFoundException: " + e.getMessage());
    throw new RuntimeException(e);

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

```

However, this duplicates code, thus making the code difficult to read and error-prone should you modify it later. For example, if you add code that can throw a new type of exception to the `try` block, you have to remember to close the `PrintWriter` within the new exception handler.

Putting It All Together

The previous sections described how to construct the `try`, `catch`, and `finally` code blocks for the `writeList` method in the `ListOfNumbers` class. Now, let's walk through the code and investigate what can happen.

When all the components are put together, the `writeList` method looks like the following.

```

public void writeList() {
    PrintWriter out = null;

    try {
        System.out.println("Entering" + " try statement");

        out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++)
            out.println("Value at: " + i + " = " + vector.elementAt(i));

    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("Caught ArrayIndexOutOfBoundsException: "

```

```

        + e.getMessage());

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

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

```

As mentioned previously, this method's try block has three different exit possibilities; here are two of them.

1. Code in the try statement fails and throws an exception. This could be an `IOException` caused by the new `FileWriter` statement or an `ArrayIndexOutOfBoundsException` caused by a wrong index value in the for loop.
2. Everything succeeds and the try statement exits normally.

Let's look at what happens in the `writeList` method during these two exit possibilities.

Scenario 1: An Exception Occurs

The statement that creates a `FileWriter` can fail for a number of reasons. For example, the constructor for the `FileWriter` throws an `IOException` if the program cannot create or write to the file indicated.

When `FileWriter` throws an `IOException`, the runtime system immediately stops executing the try block; method calls being executed are not completed. The runtime system then starts searching at the top of the method call stack for an appropriate exception handler. In this example, when the `IOException` occurs, the `FileWriter` constructor is at the top of the call stack. However, the `FileWriter` constructor doesn't have an appropriate exception handler, so the runtime system checks the next method — the `writeList` method — in the method call stack. The `writeList` method has two exception handlers: one for `IOException` and one for `ArrayIndexOutOfBoundsException`.

The runtime system checks `writeList`'s handlers in the order in which they appear after the try statement. The argument to the first exception handler is `ArrayIndexOutOfBoundsException`. This does not match the type of exception thrown, so the runtime system checks the next exception handler `IOException`. This matches the type of exception that was thrown, so the runtime system ends its search for an appropriate exception handler. Now that the runtime has found an appropriate handler, the code in that catch block is executed.

After the exception handler executes, the runtime system passes control to the finally block. Code in the finally block executes regardless of the exception caught above it. In this scenario, the `FileWriter` was never opened and doesn't need to be closed. After the finally block finishes executing, the program continues with the first statement after the finally block.

Here's the complete output from the `ListOfNumbers` program that appears when an `IOException` is thrown.

```
Entering try statement
Caught IOException: OutFile.txt
PrintWriter not open
```

The boldface code in the following listing shows the statements that get executed during this scenario:

```
public void writeList() {
    PrintWriter out = null;

    try {
        System.out.println("Entering try statement");
        out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++)
            out.println("Value at: " + i + " = " + vector.elementAt(i));

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

    } catch (IOException e) {
        System.err.println("Caught IOException: " + e.getMessage());
    } finally {
        if (out != null) {
            System.out.println("Closing PrintWriter");
            out.close();
        }
        else {
            System.out.println("PrintWriter not open");
        }
    }
}
```

Scenario 2: The try Block Exits Normally

In this scenario, all the statements within the scope of the try block execute successfully and throw no exceptions. Execution falls off the end of the try block, and the runtime system passes control to the finally block. Because everything was successful, the `PrintWriter` is open when control reaches the finally block, which closes the `PrintWriter`. Again, after the finally block finishes executing, the program continues with the first statement after the finally block.

Here is the output from the `ListOfNumbers` program when no exceptions are thrown.

```
Entering try statement
```

Closing PrintWriter

The boldface code in the following sample shows the statements that get executed during this scenario.

```
public void writeList() {  
    PrintWriter out = null;  
    try {  
        System.out.println("Entering try statement");  
        out = new PrintWriter(new FileWriter("OutFile.txt"));  
        for (int i = 0; i < SIZE; i++)  
            out.println("Value at: " + i + " = " + vector.elementAt(i));  
  
    } catch (ArrayIndexOutOfBoundsException e) {  
        System.err.println("Caught ArrayIndexOutOfBoundsException: "  
            + e.getMessage());  
  
    } catch (IOException e) {  
        System.err.println("Caught IOException: " + e.getMessage());  
  
    } finally {  
        if (out != null) {  
            System.out.println("Closing PrintWriter");  
            out.close();  
        }  
        else {  
            System.out.println("PrintWriter not open");  
        }  
    }  
}
```

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.

```
// Note: This method won't compile by design!  
public void writeList() {  
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));  
    for (int i = 0; i < SIZE; i++) {  
        out.println("Value at: " + i + " = " + vector.elementAt(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, ArrayIndexOutOfBoundsException {
```

Remember that `ArrayIndexOutOfBoundsException` 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.

You can also create your own exception classes to represent problems that can occur within the classes you write. In fact, if you are a package developer, you might have to create your own set of exception classes to allow users to differentiate an error that can occur in your package from errors that occur in the Java platform or other packages. You can also create *chained* exceptions.

The throw Statement

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;
```

Let's look at the `throw` statement in context. The following `pop` method is taken from a class that implements a common stack object. The method removes the top element from the stack and returns the object.

```
public Object pop() {  
    Object obj;  
  
    if (size == 0) {  
        throw new EmptyStackException();  
    }  
}
```

```

obj = objectAt(size - 1);
setObjectAt(size - 1, null);
size--;
return obj;
}

```

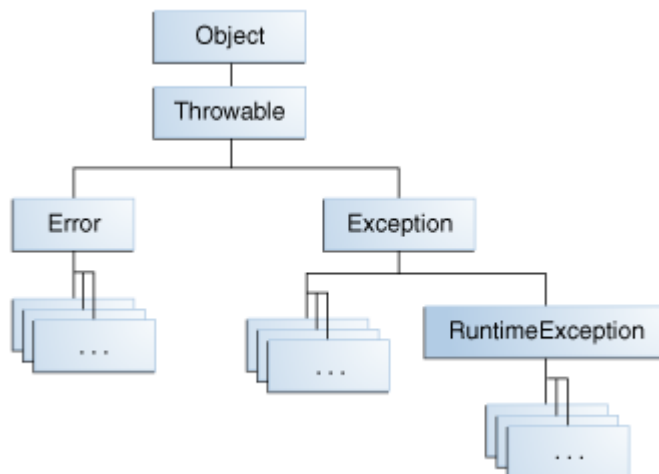
The pop method checks to see whether any elements are on the stack. If the stack is empty (its size is equal to 0), pop instantiates a new EmptyStackException object (a member of java.util) and throws it.

The [Creating Exception Classes](#) section in this chapter explains how to create your own exception classes. For now, all you need to remember is that you can throw only objects that inherit from the java.lang.Throwable class.

Note that the declaration of the pop method does not contain a throws clause. EmptyStackException is not a checked exception, so pop is not required to state that it might occur.

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](#).



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, `IllegalArgumentException` 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. You can create your own exceptions as follows .

```
public class MyRunTimeException extends RuntimeException {  
  
    public MyRunTimeException(String msg) {  
  
        super(msg);  
  
    }  
  
}
```

We can consider the above exception as example of *un checked* Exception. The below exception we can consider as *Checked* Exception.

```
public class MyException extends Exception {  
  
    public MyException(String message) {  
  
        super(message);  
  
    }  
  
}
```

Below is the code to see the usage of throw key word.

```
public class MyTest {  
  
    public static void main(String a[]){  
  
        try {
```

```
        testOne();

    } catch (MyException e) {

        e.printStackTrace

    }

    testTwo();

}

private static void testOne() throws MyException {

    throw new MyException("Exception");

}

private static void testTwo(){

    throw new MyRunTimeException("Runtime Exception");

}

}
```


Processes and Threads

In concurrent programming, there are two basic units of execution: *processes* and *threads*. In the Java programming language, concurrent programming is mostly concerned with threads. However, processes are also important.

A computer system normally has many active processes and threads. This is true even in systems that only have a single execution core, and thus only have one thread actually executing at any given moment. Processing time for a single core is shared among processes and threads through an OS feature called time slicing.

It's becoming more and more common for computer systems to have multiple processors or processors with multiple execution cores. This greatly enhances a system's capacity for concurrent execution of processes and threads but concurrency is possible even on simple systems, without multiple processors or execution cores.

Processes

A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.

Processes are often seen as synonymous with programs or applications. However, what the user sees as a single application may in fact be a set of cooperating processes. To facilitate communication between processes, most operating systems support *Inter Process Communication* (IPC) resources, such as pipes and sockets. IPC is used not just for communication between processes on the same system, but processes on different systems.

Most implementations of the Java virtual machine run as a single process. A Java application can create additional processes using a [ProcessBuilder](#) object.

Threads

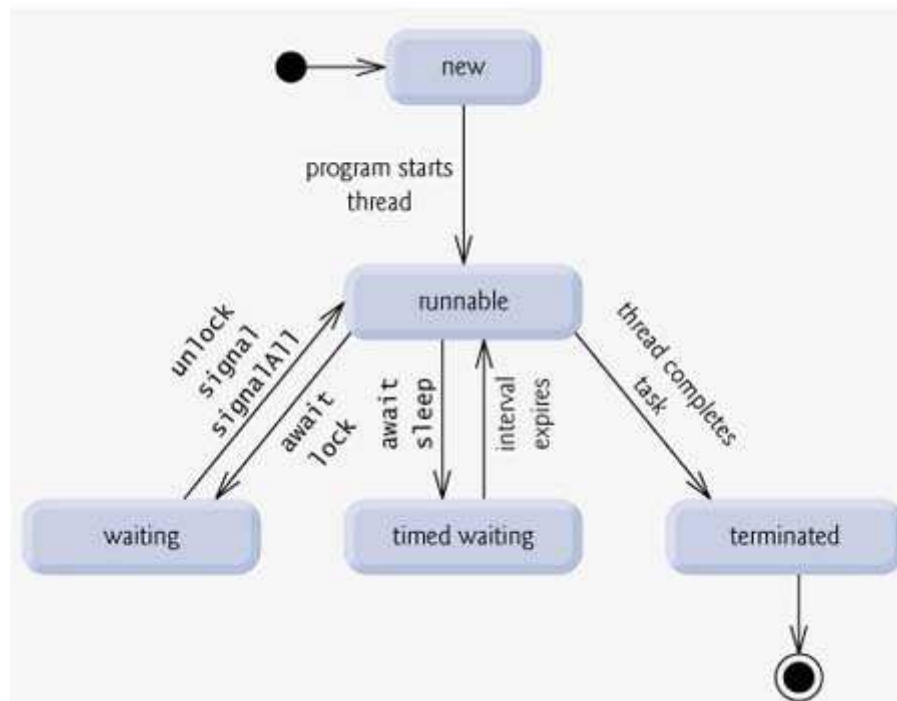
Threads are sometimes called *lightweight processes*. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Threads exist within a process every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

Multithreaded execution is an essential feature of the Java platform. Every application has at least one thread or several, if you count "system" threads that do things like memory management and signal handling. But from the application programmer's point of view, you start with just one thread, called the *main thread*. This thread has the ability to create additional threads, as we'll demonstrate in the next section.

Life Cycle of a Thread:

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread.



Above mentioned stages are explained here:

- **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting:** Sometimes a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Priorities:

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java priorities are in the range between `MIN_PRIORITY` (a constant of 1) and `MAX_PRIORITY` (a constant of 10). By default, every thread is given priority `NORM_PRIORITY` (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

Thread Creation

There are two ways to create thread in java;

- Implement the Runnable interface (java.lang.Runnable)
- By Extending the Thread class (java.lang.Thread)

Implementing the Runnable Interface

The Runnable Interface Signature

```
public interface Runnable {  
  
    void run();  
  
}
```

One way to create a thread in java is to implement the Runnable Interface and then instantiate an object of the class. We need to override the run() method into our class which is the only method that needs to be implemented. The run() method contains the logic of the thread.

The procedure for creating threads based on the Runnable interface is as follows:

1. A class implements the Runnable interface, providing the run() method that will be executed by the thread. An object of this class is a Runnable object.
2. An object of Thread class is created by passing a Runnable object as argument to the Thread constructor. The Thread object now has a Runnable object that implements the run() method.
3. The start() method is invoked on the Thread object created in the previous step. The start() method returns immediately after a thread has been spawned.
4. The thread ends when the run() method ends, either by normal completion or by throwing an uncaught exception.

Below is a program that illustrates instantiation and running of threads using the runnable interface instead of extending the Thread class. To start the thread you need to invoke the **start()** method on your object.

```

class RunnableThread implements Runnable {

    Thread runner;
    public RunnableThread() {
    }
    public RunnableThread(String threadName) {
        runner = new Thread(this, threadName); // (1) Create a new thread.
        System.out.println(runner.getName());
        runner.start(); // (2) Start the thread.
    }
    public void run() {
        //Display info about this particular thread
        System.out.println(Thread.currentThread());
    }
}

public class RunnableExample {

    public static void main(String[] args) {
        Thread thread1 = new Thread(new RunnableThread(), "thread1");
        Thread thread2 = new Thread(new RunnableThread(), "thread2");
        RunnableThread thread3 = new RunnableThread("thread3");
        //Start the threads
        thread1.start();
        thread2.start();
        try {
            //delay for one second
            Thread.currentThread().sleep(1000);
        } catch (InterruptedException e) {
        }
        //Display info about the main thread
        System.out.println(Thread.currentThread());
    }
}

```

Output

```

thread3
Thread[thread1,5,main]
Thread[thread2,5,main]
Thread[thread3,5,main]
Thread[main,5,main]private

```

This approach of creating a thread by implementing the Runnable Interface must be used whenever the class being used to instantiate the thread object is required to extend some other class.

Extending Thread Class

The procedure for creating threads based on extending the Thread is as follows:

1. A class extending the Thread class overrides the run() method from the Thread class to define the code executed by the thread.
2. This subclass may call a Thread constructor explicitly in its constructors to initialize the thread, using the super() call.
3. The start() method inherited from the Thread class is invoked on the object of the class to make the thread eligible for running.

Below is a program that illustrates instantiation and running of threads by extending the Thread class instead of implementing the Runnable interface. To start the thread you need to invoke the **start()** method on your object.

```
class XThread extends Thread {  
  
    XThread() {  
    }  
    XThread(String threadName) {  
        super(threadName); // Initialize thread.  
        System.out.println(this);  
        start();  
    }  
    public void run() {  
        //Display info about this particular thread  
        System.out.println(Thread.currentThread().getName());  
    }  
}  
  
public class ThreadExample {  
  
    public static void main(String[] args) {  
        Thread thread1 = new Thread(new XThread(), "thread1");  
        Thread thread2 = new Thread(new XThread(), "thread2");  
        //          The below 2 threads are assigned default names  
        Thread thread3 = new XThread();  
        Thread thread4 = new XThread();  
        Thread thread5 = new XThread("thread5");  
        //Start the threads  
        thread1.start();  
        thread2.start();  
        thread3.start();  
        thread4.start();  
        try {  
            //The sleep() method is invoked on the main thread to cause a one second delay.  
            Thread.currentThread().sleep(1000);  
        }  
    }  
}
```

```

        } catch (InterruptedException e) {
        }
        //Display info about the main thread
        System.out.println(Thread.currentThread());
    }
}

```

Output

```

Thread[thread5,5,main]
thread1
thread5
thread2
Thread-3
Thread-2
Thread[main,5,main]

```

When creating threads, there are two reasons why implementing the Runnable interface may be preferable to extending the Thread class:

- Extending the Thread class means that the subclass cannot extend any other class, whereas a class implementing the Runnable interface has this option.
- A class might only be interested in being runnable, and therefore, inheriting the full overhead of the Thread class would be excessive.

An example of an anonymous class below shows how to create a thread and start it:

```

( new Thread() {

    public void run() {

        for(;;) System.out.println("Stop the world!");

    }

}

).start();

```

Thread Methods:

Following is the list of important methods available in the Thread class.

SN	Methods with Description
1	public void start() Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	public void run() If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
3	public final void setName(String name) Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	public final void setPriority(int priority) Sets the priority of this Thread object. The possible values are between 1 and 10.
5	public final void setDaemon(boolean on) A parameter of true denotes this Thread as a daemon thread.
6	public final void join(long millisec) The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
7	public void interrupt() Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	public final boolean isAlive() Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread

SN	Methods with Description
1	public static void yield() Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled
2	public static void sleep(long millisec) Causes the currently running thread to block for at least the specified number of milliseconds
3	public static boolean holdsLock(Object x) Returns true if the current thread holds the lock on the given Object.
4	public static Thread currentThread() Returns a reference to the currently running thread, which is the thread that invokes this method.
5	public static void dumpStack() Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

Example:

The following ThreadClassDemo program demonstrates some of these methods of the Thread class:

```

// File Name : DisplayMessage.java
// Create a thread to implement Runnable
public class DisplayMessage implements Runnable
{
    private String message;
    public DisplayMessage(String message)
    {
        this.message = message;
    }
    public void run()
    {
        while(true)
        {
            System.out.println(message);
        }
    }
}

// File Name : GuessANumber.java
// Create a thread to extend Thread
public class GuessANumber extends Thread
{
    private int number;
    public GuessANumber(int number)
    {
        this.number = number;
    }
    public void run()
    {
        int counter = 0;
        int guess = 0;
        do
        {
            guess = (int) (Math.random() * 100 + 1);
            System.out.println(this.getName()
                               + " guesses " + guess);
            counter++;
        } while(guess != number);
        System.out.println("*** Correct! " + this.getName()
                           + " in " + counter + " guesses.**");
    }
}

// File Name : ThreadClassDemo.java
public class ThreadClassDemo
{
    public static void main(String [] args)
    {
        Runnable hello = new DisplayMessage("Hello");
        Thread thread1 = new Thread(hello);
        thread1.setDaemon(true);
        thread1.setName("hello");
        System.out.println("Starting hello thread...");
        thread1.start();

        Runnable bye = new DisplayMessage("Goodbye");
        Thread thread2 = new Thread(hello);
        thread2.setPriority(Thread.MIN_PRIORITY);
        thread2.setDaemon(true);
        System.out.println("Starting goodbye thread...");
        thread2.start();

        System.out.println("Starting thread3...");
        Thread thread3 = new GuessANumber(27);
        thread3.start();
        try
        {

```

```

        thread3.join();
    } catch (InterruptedException e)
    {
        System.out.println("Thread interrupted.");
    }
    System.out.println("Starting thread4...");
    Thread thread4 = new GuessANumber(75);

    thread4.start();
    System.out.println("main() is ending...");
}
}

```

This would produce following result. You can try this example again and again and you would get different result every time.

```

Starting hello thread...
Starting goodbye thread...
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Thread-2 guesses 27
Hello
** Correct! Thread-2 in 102 guesses.**
Hello
Starting thread4...
Hello
Hello
.....remaining result produced.

```

Major Thread Concepts:

While doing Multithreading programming, you would need to have following concepts very handy:

- [Thread Synchronization](#)
- [Interthread Communication](#)
- [Thread Deadlock](#)
- [Thread Control: Suspend, Stop and Resume](#)

Thread Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.

The process by which this synchronization is achieved is called thread synchronization.

The synchronized keyword in Java creates a block of code referred to as a critical section. Every Java object with a critical section of code gets a lock associated with the object. To enter a critical section, a thread needs to obtain the corresponding object's lock.

This is the general form of the synchronized statement:

```
synchronized(object) {  
    // statements to be synchronized  
}
```

Here, object is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor.

Here is an example, using a synchronized block within the run() method:

```
// File Name : Callme.java  
// This program uses a synchronized block.  
class Callme {  
    void call(String msg) {  
        System.out.print("[ " + msg);  
        try {  
            Thread.sleep(1000);  
        } catch (InterruptedException e) {  
            System.out.println("Interrupted");  
        }  
        System.out.println("]");  
    }  
}  
  
// File Name : Caller.java  
class Caller implements Runnable {  
    String msg;  
    Callme target;  
    Thread t;  
    public Caller(Callme targ, String s) {  
        target = targ;  
        msg = s;  
        t = new Thread(this);  
        t.start();  
    }  
  
    // synchronize calls to call()  
    public void run() {  
        synchronized(target) { // synchronized block  
            target.call(msg);  
        }  
    }  
}  
  
// File Name : Synch.java  
class Synch {  
    public static void main(String args[]) {  
        Callme target = new Callme();  
        Caller ob1 = new Caller(target, "Hello");  
        Caller ob2 = new Caller(target, "Synchronized");  
        Caller ob3 = new Caller(target, "World");  
  
        // wait for threads to end  
        try {  
            ob1.t.join();  
            ob2.t.join();  
            ob3.t.join();  
        } catch (InterruptedException e) {  
            System.out.println("Interrupted");  
        }  
    }  
}
```

This would produce following result:

```
[Hello]
[World]
[Synchronized]
```

Interthread Communication

Consider the classic queuing problem, where one thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data.

In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on. Clearly, this situation is undesirable.

To avoid polling, Java includes an elegant interprocess communication mechanism via the following methods:

- **wait()**: This method tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify().
- **notify()**: This method wakes up the first thread that called wait() on the same object.
- **notifyAll()**: This method wakes up all the threads that called wait() on the same object. The highest priority thread will run first.

These methods are implemented as final methods in Object, so all classes have them. All three methods can be called only from within a synchronized context.

These methods are declared within Object. Various forms of wait() exist that allow you to specify a period of time to wait.

Example:

The following sample program consists of four classes: Q, the queue that you're trying to synchronize; Producer, the threaded object that is producing queue entries; Consumer, the threaded object that is consuming queue entries; and PC, the tiny class that creates the single Q, Producer, and Consumer.

The proper way to write this program in Java is to use wait() and notify() to signal in both directions, as shown here:

```
class Q {
    int n;
    boolean valueSet = false;
    synchronized int get() {
        if(!valueSet)
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
        System.out.println("Got: " + n);
        valueSet = false;
    }
}
```

```

        notify();
        return n;
    }

    synchronized void put(int n) {
        if(valueSet)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
        this.n = n;
        valueSet = true;
        System.out.println("Put: " + n);
        notify();
    }
}

class Producer implements Runnable {
    Q q;
    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }

    public void run() {
        int i = 0;
        while(true) {
            q.put(i++);
        }
    }
}

class Consumer implements Runnable {
    Q q;
    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }
    public void run() {
        while(true) {
            q.get();
        }
    }
}

class PCFixed {
    public static void main(String args[]) {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);
        System.out.println("Press Control-C to stop.");
    }
}

```

Inside `get()`, `wait()` is called. This causes its execution to suspend until the Producer notifies you that some data is ready.

When this happens, execution inside `get()` resumes. After the data has been obtained, `get()` calls `notify()`. This tells Producer that it is okay to put more data in the queue.

Inside `put()`, `wait()` suspends execution until the Consumer has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and `notify()` is called. This tells the Consumer that it should now remove it.

Here is some output from this program, which shows the clean synchronous behavior:

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

Thread Control

While the `suspend()`, `resume()`, and `stop()` methods defined by `Thread` class seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs and obsolete in newer versions of Java.

The following example illustrates how the `wait()` and `notify()` methods that are inherited from `Object` can be used to control the execution of a thread.

This example is similar to the program in the previous section. However, the deprecated method calls have been removed. Let us consider the operation of this program.

The `NewThread` class contains a boolean instance variable named `suspendFlag`, which is used to control the execution of the thread. It is initialized to false by the constructor.

The `run()` method contains a synchronized statement block that checks `suspendFlag`. If that variable is true, the `wait()` method is invoked to suspend the execution of the thread. The `mysuspend()` method sets `suspendFlag` to true. The `myresume()` method sets `suspendFlag` to false and invokes `notify()` to wake up the thread. Finally, the `main()` method has been modified to invoke the `mysuspend()` and `myresume()` methods.

Example:

```
// Suspending and resuming a thread for Java 2
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;
    boolean suspendFlag;
    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        suspendFlag = false;
        t.start(); // Start the thread
    }
    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 15; i > 0; i--) {
```

```

        System.out.println(name + ": " + i);
        Thread.sleep(200);
        synchronized(this) {
            while(suspendFlag) {
                wait();
            }
        }
    } catch (InterruptedException e) {
        System.out.println(name + " interrupted.");
    }
    System.out.println(name + " exiting.");
}
void mysuspend() {
    suspendFlag = true;
}
synchronized void myresume() {
    suspendFlag = false;
    notify();
}
}

class SuspendResume {
    public static void main(String args[]) {
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");
        try {
            Thread.sleep(1000);
            ob1.mysuspend();
            System.out.println("Suspending thread One");
            Thread.sleep(1000);
            ob1.myresume();
            System.out.println("Resuming thread One");
            ob2.mysuspend();
            System.out.println("Suspending thread Two");
            Thread.sleep(1000);
            ob2.myresume();
            System.out.println("Resuming thread Two");
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
        // wait for threads to finish
        try {
            System.out.println("Waiting for threads to finish.");
            ob1.t.join();
            ob2.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
        System.out.println("Main thread exiting.");
    }
}

```

Here is the output produced by the above program:

```

New thread: Thread[One,5,main]
One: 15
New thread: Thread[Two,5,main]
Two: 15
One: 14
Two: 14
One: 13
Two: 13
One: 12
Two: 12

```

```

One: 11
Two: 11
Suspending thread One
Two: 10
Two: 9
Two: 8
Two: 7
Two: 6
Resuming thread One
Suspending thread Two
One: 10
One: 9
One: 8
One: 7
One: 6
Resuming thread Two
Waiting for threads to finish.
Two: 5
One: 5
Two: 4
One: 4
Two: 3
One: 3
Two: 2
One: 2
Two: 1
One: 1
Two exiting.
One exiting.
Main thread exiting.

```

Thread Deadlock

A special type of error that you need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects.

For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.

Example:

To understand deadlock fully, it is useful to see it in action. The next example creates two classes, A and B, with methods `foo()` and `bar()`, respectively, which pause briefly before trying to call a method in the other class.

The main class, named `Deadlock`, creates an A and a B instance, and then starts a second thread to set up the deadlock condition. The `foo()` and `bar()` methods use `sleep()` as a way to force the deadlock condition to occur.

```

class A {
    synchronized void foo(B b) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered A.foo");
        try {

```

```

        Thread.sleep(1000);
    } catch(Exception e) {
        System.out.println("A Interrupted");
    }
    System.out.println(name + " trying to call B.last()");
    b.last();
}
synchronized void last() {
    System.out.println("Inside A.last");
}
}
class B {
    synchronized void bar(A a) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered B.bar");
        try {
            Thread.sleep(1000);
        } catch(Exception e) {
            System.out.println("B Interrupted");
        }
        System.out.println(name + " trying to call A.last()");
        a.last();
    }
    synchronized void last() {
        System.out.println("Inside A.last");
    }
}
}
class Deadlock implements Runnable {
    A a = new A();
    B b = new B();
    Deadlock() {
        Thread.currentThread().setName("MainThread");
        Thread t = new Thread(this, "RacingThread");
        t.start();
        a.foo(b); // get lock on a in this thread.
        System.out.println("Back in main thread");
    }
    public void run() {
        b.bar(a); // get lock on b in other thread.
        System.out.println("Back in other thread");
    }
    public static void main(String args[]) {
        new Deadlock();
    }
}

```

Here is some output from this program:

```

MainThread entered A.foo
RacingThread entered B.bar
MainThread trying to call B.last()
RacingThread trying to call A.last()

```

Because the program has deadlocked, you need to press CTRL-C to end the program. You can see a full thread and monitor cache dump by pressing CTRL-BREAK on a PC .

You will see that RacingThread owns the monitor on b, while it is waiting for the monitor on a. At the same time, MainThread owns a and is waiting to get b. This program will never complete.

As this example illustrates, if your multithreaded program locks up occasionally, deadlock is one of the first conditions that you should check for.

Ordering Locks:

A common threading trick to avoid the deadlock is to order the locks. By ordering the locks, it gives threads a specific order to obtain multiple locks.

Deadlock Example:

Following is the depiction of a dead lock:

```
// File Name ThreadSafeBankAccount.java
public class ThreadSafeBankAccount
{
    private double balance;
    private int number;
    public ThreadSafeBankAccount(int num, double initialBalance)
    {
        balance = initialBalance;
        number = num;
    }
    public int getNumber()
    {
        return number;
    }
    public double getBalance()
    {
        return balance;
    }
    public void deposit(double amount)
    {
        synchronized(this)
        {
            double prevBalance = balance;
            try
            {
                Thread.sleep(4000);
            } catch (InterruptedException e)
            {}
            balance = prevBalance + amount;
        }
    }
    public void withdraw(double amount)
    {
        synchronized(this)
        {
            double prevBalance = balance;
            try
            {
                Thread.sleep(4000);
            } catch (InterruptedException e)
            {}
            balance = prevBalance - amount;
        }
    }
}

// File Name LazyTeller.java
public class LazyTeller extends Thread
{
    private ThreadSafeBankAccount source, dest;
    public LazyTeller(ThreadSafeBankAccount a,
```



```

        ThreadSafeBankAccount b)
    {
        source = a;
        dest = b;
    }
    public void run()
    {
        transfer(250.00);
    }
    public void transfer(double amount)
    {
        System.out.println("Transferring from "
            + source.getNumber() + " to " + dest.getNumber());
        synchronized(source)
        {
            Thread.yield();
            synchronized(dest)
            {
                System.out.println("Withdrawing from "
                    + source.getNumber());
                source.withdraw(amount);
                System.out.println("Depositing into "
                    + dest.getNumber());
                dest.deposit(amount);
            }
        }
    }
}
}
public class DeadlockDemo
{
    public static void main(String [] args)
    {
        System.out.println("Creating two bank accounts...");
        ThreadSafeBankAccount checking =
            new ThreadSafeBankAccount(101, 1000.00);
        ThreadSafeBankAccount savings =
            new ThreadSafeBankAccount(102, 5000.00);

        System.out.println("Creating two teller threads...");
        Thread teller1 = new LazyTeller(checking, savings);
        Thread teller2 = new LazyTeller(savings, checking);
        System.out.println("Starting both threads...");
        teller1.start();
        teller2.start();
    }
}

```

This would produce following result:

```

Creating two bank accounts...
Creating two teller threads...
Starting both threads...
Transferring from 101 to 102
Transferring from 102 to 101

```

The problem with the LazyTeller class is that it does not consider the possibility of a race condition, a common occurrence in multithreaded programming.

After the two threads are started, teller1 grabs the checking lock and teller2 grabs the savings lock. When teller1 tries to obtain the savings lock, it is not available. Therefore, teller1 blocks until the savings lock becomes available. When the teller1 thread blocks, teller1 still has the checking lock and does not let it go.

Similarly, teller2 is waiting for the checking lock, so teller2 blocks but does not let go of the savings lock. This leads to one result: deadlock!

Deadlock Solution Example:

Here transfer() method, in a class named OrderedTeller, in stead of arbitrarily synchronizing on locks, this transfer() method obtains locks in a specified order based on the number of the bank account.

```
// File Name ThreadSafeBankAccount.java
public class ThreadSafeBankAccount
{
    private double balance;
    private int number;
    public ThreadSafeBankAccount(int num, double initialBalance)
    {
        balance = initialBalance;
        number = num;
    }
    public int getNumber()
    {
        return number;
    }
    public double getBalance()
    {
        return balance;
    }
    public void deposit(double amount)
    {
        synchronized(this)
        {
            double prevBalance = balance;
            try
            {
                Thread.sleep(4000);
            } catch (InterruptedException e)
            {}
            balance = prevBalance + amount;
        }
    }
    public void withdraw(double amount)
    {
        synchronized(this)
        {
            double prevBalance = balance;
            try
            {
                Thread.sleep(4000);
            } catch (InterruptedException e)
            {}
            balance = prevBalance - amount;
        }
    }
}

// File Name OrderedTeller.java
public class OrderedTeller extends Thread
{
    private ThreadSafeBankAccount source, dest;
    public OrderedTeller(ThreadSafeBankAccount a,
                        ThreadSafeBankAccount b)
    {
        source = a;
        dest = b;
    }
}
```

```

public void run()
{
    transfer(250.00);
}
public void transfer(double amount)
{
    System.out.println("Transferring from " + source.getNumber()
        + " to " + dest.getNumber());
    ThreadSafeBankAccount first, second;
    if(source.getNumber() < dest.getNumber())
    {
        first = source;
        second = dest;
    }
    else
    {
        first = dest;
        second = source;
    }
    synchronized(first)
    {
        Thread.yield();
        synchronized(second)
        {
            System.out.println("Withdrawing from "
                + source.getNumber());
            source.withdraw(amount);
            System.out.println("Depositing into "
                + dest.getNumber());
            dest.deposit(amount);
        }
    }
}
}

// File Name DeadlockDemo.java
public class DeadlockDemo
{
    public static void main(String [] args)
    {
        System.out.println("Creating two bank accounts...");
        ThreadSafeBankAccount checking =
            new ThreadSafeBankAccount(101, 1000.00);
        ThreadSafeBankAccount savings =
            new ThreadSafeBankAccount(102, 5000.00);

        System.out.println("Creating two teller threads...");
        Thread teller1 = new OrderedTeller(checking, savings);
        Thread teller2 = new OrderedTeller(savings, checking);
        System.out.println("Starting both threads...");
        teller1.start();
        teller2.start();
    }
}

```

This would remove deadlock problem and would produce following result:

```

Creating two bank accounts...
Creating two teller threads...
Starting both threads...
Transferring from 101 to 102
Transferring from 102 to 101
Withdrawing from 101
Depositing into 102
Withdrawing from 102

```

Daemon threads

We mentioned that a Java program exits when all of its threads have completed, but this is not exactly correct. What about the hidden system threads, such as the garbage collection thread and others created by the JVM? We have no way of stopping these. If those threads are running, how does any Java program ever exit? These system threads are called daemon threads.

A Java program actually exits when all its non-daemon threads have completed. Any thread can become a daemon thread. You can indicate a thread is a daemon thread by calling the `Thread.setDaemon()` method. You might want to use daemon threads for background threads that you create in your programs, such as timer threads or other deferred event threads, which are only useful while there are other non-daemon threads running.

Volatile

If a variable is declared as volatile then is guaranteed that any thread which reads the field will see the most recently written value. The keyword volatile will not perform any mutual exclusive lock on the variable.

As of Java 5 write access to a volatile variable will also update non-volatile variables which were modified by the same thread. This can also be used to update values within a reference variable, e.g. for a volatile variable person. In this case you must use a temporary variable person and use the setter to initialize the variable and then assign the temporary variable to the final variable. This will then make the address changes of this variable and the values visible to other threads.

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. The following Box class will be modified to demonstrate the concept.

A Simple Box Class

Begin by examining a non-generic Box class that operates on objects of any type. It needs only to provide two methods: `set`, which adds an object to the box, and `get`, which retrieves it:

```
public class Box {
    private Object object;

    public void set(Object object) {
this.object = object; }
    public Object get() { return object; }
}
```

Since its methods accept or return an `Object`, you are free to pass in whatever you want, provided that it is not one of the primitive types. There is no way to verify, at compile time, how the class is used. One part of the code may place an `Integer` in the box and expect to get `Integers` out of it, while another part of the code may mistakenly pass in a `String`, resulting in a runtime error.

A Generic Version of the Box Class

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.

To update the Box class to use generics, you create a *generic type declaration* by changing the code "public class Box" to "public class Box<T>". This introduces the type variable, T, that can be used anywhere inside the class.

With this change, the Box class becomes:

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
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.

Type Parameter Naming Conventions

By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable naming conventions that you already know about, and with good reason: Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.

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

You'll see these names used throughout the Java SE API and the rest of this lesson.

Invoking and Instantiating a Generic Type

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;
```

You can think of a generic type invocation as being similar to an ordinary method invocation, but instead of passing an argument to a method, you are passing a *type argument* — `Integer` in this case — to the `Box` class itself.

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.

Like any other variable declaration, this code does not actually create a new `Box` object. It simply declares that `integerBox` will hold a reference to a "Box of Integer", which is how `Box<Integer>` is read.

An invocation of a generic type is generally known as a *parameterized type*.

To instantiate this class, use the `new` keyword, as usual, but place `<Integer>` between the class name and the parenthesis:

```
Box<Integer> integerBox = new Box<Integer>();
```

The Diamond

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<>();
```

Multiple Type Parameters

As mentioned previously, a generic class can have multiple type parameters. For example, the generic `OrderedPair` class, which implements the generic `Pair` interface:

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() { return key; }
    public V getValue() { return value; }
}
```

The following statements create two instantiations of the `OrderedPair` class:

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String> p2 = new OrderedPair<String, String>("hello", "world");
```

The code, `new OrderedPair<String, Integer>`, instantiates `K` as a `String` and `V` as an `Integer`. Therefore, the parameter types of `OrderedPair`'s constructor are `String` and `Integer`, respectively. Due to [autoboxing](#), is it valid to pass a `String` and an `int` to the class.

As mentioned in [The Diamond](#), because a Java compiler can infer the `K` and `V` types from the declaration `OrderedPair<String, Integer>`, these statements can be shortened using diamond notation:

```
OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);
OrderedPair<String, String> p2 = new OrderedPair<>("hello", "world");
```

To create a generic interface, follow the same conventions as for creating a generic class.

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<V, K>` example:

```
OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new Box<Integer>(...));
```


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 {
    // Generic static method
    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;

    // Generic constructor
    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    // Generic methods
    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.

Introduction to Collections

A collection sometimes called a container is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. Typically, they represent data items that form a natural group a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers).

If you've used the Java programming language or just about any other programming language you're already familiar with collections. Collection implementations in earlier (pre-1.2) versions of the Java platform included Vector, Hashtable, and array. However, those earlier versions did not contain a collections framework.

What Is a Collections Framework?

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

- **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
- **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be *polymorphic*: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

Benefits of the Java Collections Framework

The Java Collections Framework provides the following benefits:

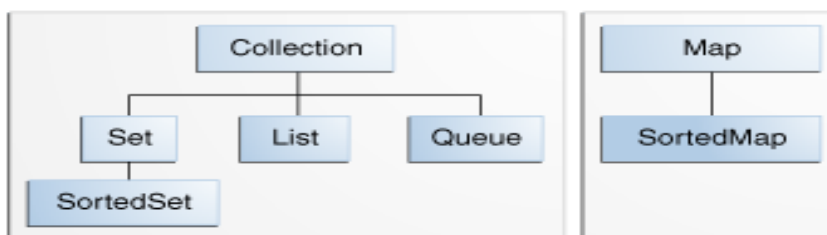
- **Reduces programming effort:** By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work. By facilitating interoperability among unrelated APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.
- **Increases program speed and quality:** This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you're freed from the drudgery of writing your own

data structures, you'll have more time to devote to improving programs' quality and performance.

- **Allows interoperability among unrelated APIs:** The collection interfaces are the vernacular by which APIs pass collections back and forth. If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.
- **Reduces effort to learn and to use new APIs:** Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard collection interfaces, the problem went away.
- **Reduces effort to design new APIs:** This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
- **Fosters software reuse:** New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

Collection - Interfaces

The core collection interfaces encapsulate different types of collections, which are shown in the figure below. These interfaces allow collections to be manipulated independently of the details of their representation. Core collection interfaces are the foundation of the Java Collections Framework. As you can see in the following figure, the core collection interfaces form a hierarchy.



The core collection interfaces.

A Set is a special kind of Collection, a SortedSet is a special kind of Set, and so forth. Note also that the hierarchy consists of two distinct trees a Map is not a true Collection.

Note that all the core collection interfaces are generic. For example, this is the declaration of the Collection interface.

```
public interface Collection<E>...
```

The <E> syntax tells you that the interface is generic. When you declare a Collection instance you can *and should* specify the type of object contained in the

collection. Specifying the type allows the compiler to verify (at compile-time) that the type of object you put into the collection is correct, thus reducing errors at runtime

When you understand how to use these interfaces, you will know most of what there is to know about the Java Collections Framework

To keep the number of core collection interfaces manageable, the Java platform doesn't provide separate interfaces for each variant of each collection type. (Such variants might include immutable, fixed-size, and append-only.) Instead, the modification operations in each interface are designated optional a given implementation may elect not to support all operations.

If an unsupported operation is invoked, a collection throws an `UnsupportedOperationException`. Implementations are responsible for documenting which of the optional operations they support. All of the Java platform's general-purpose implementations support all of the optional operations.

The following list describes the core collection interfaces:

- **Collection** — the root of the collection hierarchy. A collection represents a group of objects known as its *elements*. The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as Set and List.
- **Set** — a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine.
- **List** — an ordered collection (sometimes called a *sequence*). Lists can contain duplicate elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position). If you've used Vector, you're familiar with the general flavor of List.
- **Queue** — a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.

Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator or the elements' natural ordering. Whatever the ordering used, the head of the queue is the element that would be removed by a call to `remove` or `poll`. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties.

- Map — an object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value. If you've used Hashtable, you're already familiar with the basics of Map.

The last two core collection interfaces are merely sorted versions of Set and Map:

- SortedSet — a Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls.
- SortedMap — a Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

The Collection Interface

A [Collection](#) represents a group of objects known as its elements. The Collection interface is used to pass around collections of objects where maximum generality is desired. For example, by convention all general-purpose collection implementations have a constructor that takes a Collection argument. This constructor, known as a *conversion constructor*, initializes the new collection to contain all of the elements in the specified collection, whatever the given collection's subinterface or implementation type. In other words, it allows you to *convert* the collection's type.

Suppose, for example, that you have a `Collection<String> c`, which may be a List, a Set, or another kind of Collection. This idiom creates a new `ArrayList` (an implementation of the List interface), initially containing all the elements in `c`.

```
List<String> list = new ArrayList<String>(c);
```

The following shows the Collection interface.

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    // optional
    boolean add(E element);
    // optional
    boolean remove(Object element);
    Iterator<E> iterator();

    // Bulk operations
```

```

    boolean containsAll(Collection<?> c);
    // optional
    boolean addAll(Collection<? extends E> c);
    // optional
    boolean removeAll(Collection<?> c);
    // optional
    boolean retainAll(Collection<?> c);
    // optional
    void clear();

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}

```

The interface does about what you'd expect given that a `Collection` represents a group of objects. The interface has methods to tell you how many elements are in the collection (`size`, `isEmpty`), to check whether a given object is in the collection (`contains`), to add and remove an element from the collection (`add`, `remove`), and to provide an iterator over the collection (`iterator`).

The `add` method is defined generally enough so that it makes sense for collections that allow duplicates as well as those that don't. It guarantees that the `Collection` will contain the specified element after the call completes, and returns `true` if the `Collection` changes as a result of the call. Similarly, the `remove` method is designed to remove a single instance of the specified element from the `Collection`, assuming that it contains the element to start with, and to return `true` if the `Collection` was modified as a result.

Traversing Collections

There are two ways to traverse collections: (1) with the `for-each` construct and (2) by using `Iterator`.

for-each Construct

The `for-each` construct allows you to concisely traverse a collection or array using a `for` loop — see [The for Statement](#). The following code uses the `for-each` construct to print out each element of a collection on a separate line.

```

for (Object o : collection)
    System.out.println(o);

```

Iterators

An [Iterator](#) is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired. You get an `Iterator` for a collection by calling its `iterator` method. The following is the `Iterator` interface.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

The `hasNext` method returns `true` if the iteration has more elements, and the `next` method returns the next element in the iteration. The `remove` method removes the last element that was returned by `next` from the underlying `Collection`. The `remove` method may be called only once per call to `next` and throws an exception if this rule is violated.

Note that `Iterator.remove` is the *only* safe way to modify a collection during iteration; the behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress.

Use `Iterator` instead of the `for-each` construct when you need to:

- Remove the current element. The `for-each` construct hides the iterator, so you cannot call `remove`. Therefore, the `for-each` construct is not usable for filtering.
- Iterate over multiple collections in parallel.

The following method shows you how to use an `Iterator` to filter an arbitrary `Collection` — that is, traverse the collection removing specific elements.

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext(); )
        if (!cond(it.next()))
            it.remove();
}
```

This simple piece of code is polymorphic, which means that it works for *any* `Collection` regardless of implementation. This example demonstrates how easy it is to write a polymorphic algorithm using the Java Collections Framework.

Collection Interface Bulk Operations

Bulk operations perform an operation on an entire `Collection`. You could implement these shorthand operations using the basic operations, though in most cases such implementations would be less efficient. The following are the bulk operations:

- `containsAll` — returns `true` if the target `Collection` contains all of the elements in the specified `Collection`.

- `addAll` — adds all of the elements in the specified `Collection` to the target `Collection`.
- `removeAll` — removes from the target `Collection` all of its elements that are also contained in the specified `Collection`.
- `retainAll` — removes from the target `Collection` all its elements that are *not* also contained in the specified `Collection`. That is, it retains only those elements in the target `Collection` that are also contained in the specified `Collection`.
- `clear` — removes all elements from the `Collection`.

The `addAll`, `removeAll`, and `retainAll` methods all return `true` if the target `Collection` was modified in the process of executing the operation.

As a simple example of the power of bulk operations, consider the following idiom to remove *all* instances of a specified element, `e`, from a `Collection`, `c`.

```
c.removeAll(Collections.singleton(e));
```

More specifically, suppose you want to remove all of the `null` elements from a `Collection`.

```
c.removeAll(Collections.singleton(null));
```

This idiom uses `Collections.singleton`, which is a static factory method that returns an immutable `Set` containing only the specified element.

Collection Interface Array Operations

The `toArray` methods are provided as a bridge between collections and older APIs that expect arrays on input. The array operations allow the contents of a `Collection` to be translated into an array. The simple form with no arguments creates a new array of `Object`. The more complex form allows the caller to provide an array or to choose the runtime type of the output array.

For example, suppose that `c` is a `Collection`. The following snippet dumps the contents of `c` into a newly allocated array of `Object` whose length is identical to the number of elements in `c`.

```
Object[] a = c.toArray();
```

Suppose that `c` is known to contain only strings (perhaps because `c` is of type `Collection<String>`). The following snippet dumps the contents of `c` into a newly allocated array of `String` whose length is identical to the number of elements in `c`.

```
String[] a = c.toArray(new String[0]);
```


The Set Interface

A [Set](#) is a [Collection](#) that cannot contain duplicate elements. It models the mathematical set abstraction. The `Set` interface contains *only* methods inherited from `Collection` and adds the restriction that duplicate elements are prohibited. `Set` also adds a stronger contract on the behavior of the `equals` and `hashCode` operations, allowing `Set` instances to be compared meaningfully even if their implementation types differ. Two `Set` instances are equal if they contain the same elements.

The following is the `Set` interface.

```
public interface Set<E> extends Collection<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    // optional
    boolean add(E element);
    // optional
    boolean remove(Object element);
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    // optional
    boolean addAll(Collection<? extends E> c);
    // optional
    boolean removeAll(Collection<?> c);
    // optional
    boolean retainAll(Collection<?> c);
    // optional
    void clear();

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

The Java platform contains three general-purpose `Set` implementations: `HashSet`, `TreeSet`, and `LinkedHashSet`. [HashSet](#), which stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration. [TreeSet](#), which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than `HashSet`. [LinkedHashSet](#), which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order). `LinkedHashSet` spares its clients from the unspecified, generally chaotic ordering provided by `HashSet` at a cost that is only slightly higher.

Here's a simple but useful Set idiom. Suppose you have a Collection, c, and you want to create another Collection containing the same elements but with all duplicates eliminated. The following one-liner does the trick.

```
Collection<Type> noDups = new HashSet<Type>(c);
```

It works by creating a Set (which, by definition, cannot contain a duplicate), initially containing all the elements in c. It uses the standard conversion constructor described in the [The Collection Interface](#) section.

Here is a minor variant of this idiom that preserves the order of the original collection while removing duplicate element.

```
Collection<Type> noDups = new LinkedHashSet<Type>(c);
```

The following is a generic method that encapsulates the preceding idiom, returning a Set of the same generic type as the one passed.

```
public static <E> Set<E> removeDups(Collection<E> c) {  
    return new LinkedHashSet<E>(c);  
}
```

Set Interface Basic Operations

The size operation returns the number of elements in the Set (its *cardinality*). The isEmpty method does exactly what you think it would. The add method adds the specified element to the Set if it's not already present and returns a boolean indicating whether the element was added. Similarly, the remove method removes the specified element from the Set if it's present and returns a boolean indicating whether the element was present. The iterator method returns an Iterator over the Set.

The following [program](#) takes the words in its argument list and prints out any duplicate words, the number of distinct words, and a list of the words with duplicates eliminated.

```
import java.util.*;  
  
public class FindDups {  
    public static void main(String[] args) {  
        Set<String> s = new HashSet<String>();  
        for (String a : args)  
            if (!s.add(a))  
                System.out.println("Duplicate detected: " + a);  
  
        System.out.println(s.size() + " distinct words: " + s);  
    }  
}
```

Now run the program.

```
java FindDups i came i saw i left
```

The following output is produced.

```
Duplicate detected: i
Duplicate detected: i
4 distinct words: [i, left, saw, came]
```

Note that the code always refers to the `Collection` by its interface type (`Set`) rather than by its implementation type (`HashSet`). This is a *strongly* recommended programming practice because it gives you the flexibility to change implementations merely by changing the constructor. If either of the variables used to store a collection or the parameters used to pass it around are declared to be of the `Collection`'s implementation type rather than its interface type, *all* such variables and parameters must be changed in order to change its implementation type.

Furthermore, there's no guarantee that the resulting program will work. If the program uses any nonstandard operations present in the original implementation type but not in the new one, the program will fail. Referring to collections only by their interface prevents you from using any nonstandard operations.

The implementation type of the `Set` in the preceding example is `HashSet`, which makes no guarantees as to the order of the elements in the `Set`. If you want the program to print the word list in alphabetical order, merely change the `Set`'s implementation type from `HashSet` to `TreeSet`. Making this trivial one-line change causes the command line in the previous example to generate the following output.

```
java FindDups i came i saw i left
Duplicate detected: i
Duplicate detected: i
4 distinct words: [came, i, left, saw]
```

Set Interface Bulk Operations

Bulk operations are particularly well suited to `Set`s; when applied, they perform standard set-algebraic operations. Suppose `s1` and `s2` are sets. Here's what bulk operations do:

- `s1.containsAll(s2)` — returns `true` if `s2` is a **subset** of `s1`. (`s2` is a subset of `s1` if set `s1` contains all of the elements in `s2`.)
- `s1.addAll(s2)` — transforms `s1` into the **union** of `s1` and `s2`. (The union of two sets is the set containing all of the elements contained in either set.)

- `s1.retainAll(s2)` — transforms `s1` into the intersection of `s1` and `s2`. (The intersection of two sets is the set containing only the elements common to both sets.)
- `s1.removeAll(s2)` — transforms `s1` into the (asymmetric) set difference of `s1` and `s2`. (For example, the set difference of `s1` minus `s2` is the set containing all of the elements found in `s1` but not in `s2`.)

To calculate the union, intersection, or set difference of two sets *nondestructively* (without modifying either set), the caller must copy one set before calling the appropriate bulk operation. The following are the resulting idioms.

```
Set<Type> union = new HashSet<Type>(s1);
union.addAll(s2);
```

```
Set<Type> intersection = new HashSet<Type>(s1);
intersection.retainAll(s2);
```

```
Set<Type> difference = new HashSet<Type>(s1);
difference.removeAll(s2);
```

The implementation type of the result `Set` in the preceding idioms is `HashSet`, which is, as already mentioned, the best all-around `Set` implementation in the Java platform. However, any general-purpose `Set` implementation could be substituted.

Let's revisit the `FindDups` program. Suppose you want to know which words in the argument list occur only once and which occur more than once, but you do not want any duplicates printed out repeatedly. This effect can be achieved by generating two sets — one containing every word in the argument list and the other containing only the duplicates. The words that occur only once are the set difference of these two sets, which we know how to compute. Here's how [the resulting program](#) looks.

```
import java.util.*;

public class FindDups2 {
    public static void main(String[] args) {
        Set<String> uniques = new HashSet<String>();
        Set<String> dups = new HashSet<String>();

        for (String a : args)
            if (!uniques.add(a))
                dups.add(a);

        // Destructive set-difference
        uniques.removeAll(dups);

        System.out.println("Unique words: " + uniques);
        System.out.println("Duplicate words: " + dups);
    }
}
```

```
}
```

When run with the same argument list used earlier (i came i saw i left), the program yields the following output.

```
Unique words: [left, saw, came]
Duplicate words: [i]
```

A less common set-algebraic operation is the *symmetric set difference* — the set of elements contained in either of two specified sets but not in both. The following code calculates the symmetric set difference of two sets nondestructively.

```
Set<Type> symmetricDiff = new HashSet<Type>(s1);
symmetricDiff.addAll(s2);
Set<Type> tmp = new HashSet<Type>(s1);
tmp.retainAll(s2);
symmetricDiff.removeAll(tmp);
```

Set Interface Array Operations

The array operations don't do anything special for Sets beyond what they do for any other Collection.

The List Interface

A [List](#) is an ordered [Collection](#) (sometimes called a *sequence*). Lists may contain duplicate elements. In addition to the operations inherited from Collection, the List interface includes operations for the following:

- Positional access — manipulates elements based on their numerical position in the list
- Search — searches for a specified object in the list and returns its numerical position
- Iteration — extends Iterator semantics to take advantage of the list's sequential nature
- Range-view — performs arbitrary *range operations* on the list.

The Java platform contains two general-purpose List implementations. [ArrayList](#), which is usually the better-performing implementation, and [LinkedList](#) which offers better performance under certain circumstances. Also, [Vector](#) has been retrofitted to implement List.

Collection Operations

The `remove` operation always removes *the first* occurrence of the specified element from the list. The `add` and `addAll` operations always append the new

element(s) to the *end* of the list. Thus, the following idiom concatenates one list to another.

List Algorithms

Most polymorphic algorithms in the `Collections` class apply specifically to `List`. Having all these algorithms at your disposal makes it very easy to manipulate lists. Here's a summary of these algorithms, which are described in more detail in the [Algorithms](#) section.

- `sort` — sorts a `List` using a merge sort algorithm, which provides a fast, stable sort. (A *stable sort* is one that does not reorder equal elements.)
- `shuffle` — randomly permutes the elements in a `List`.
- `reverse` — reverses the order of the elements in a `List`.
- `rotate` — rotates all the elements in a `List` by a specified distance.
- `swap` — swaps the elements at specified positions in a `List`.
- `replaceAll` — replaces all occurrences of one specified value with another.
- `fill` — overwrites every element in a `List` with the specified value.
- `copy` — copies the source `List` into the destination `List`.
- `binarySearch` — searches for an element in an ordered `List` using the binary search algorithm.
- `indexOfSubList` — returns the index of the first sublist of one `List` that is equal to another.
- `lastIndexOfSubList` — returns the index of the last sublist of one `List` that is equal to another.

The Queue Interface

A [Queue](#) is a collection for holding elements prior to processing. Besides basic `Collection` operations, queues provide additional insertion, removal, and inspection operations. The `Queue` interface follows.

```
public interface Queue<E> extends Collection<E> {  
    E element();  
    boolean offer(E e);  
    E peek();  
    E poll();  
    E remove();  
}
```

Each `Queue` method exists in two forms: (1) one throws an exception if the operation fails, and (2) the other returns a special value if the operation fails (either `null` or `false`, depending on the operation). The regular structure of the interface is illustrated in the following table.

Queue Interface Structure

Type of Operation	Throws exception	Returns special value
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

Queues typically, but not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to their values — see the [Object Ordering](#) section for details). Whatever ordering is used, the head of the queue is the element that would be removed by a call to `remove` or `poll`. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every `Queue` implementation must specify its ordering properties.

It is possible for a `Queue` implementation to restrict the number of elements that it holds; such queues are known as *bounded*. Some `Queue` implementations in `java.util.concurrent` are bounded, but the implementations in `java.util` are not.

The `add` method, which `Queue` inherits from `Collection`, inserts an element unless it would violate the queue's capacity restrictions, in which case it throws `IllegalStateException`. The `offer` method, which is intended solely for use on bounded queues, differs from `add` only in that it indicates failure to insert an element by returning `false`.

The `remove` and `poll` methods both remove and return the head of the queue. Exactly which element gets removed is a function of the queue's ordering policy. The `remove` and `poll` methods differ in their behavior only when the queue is empty. Under these circumstances, `remove` throws `NoSuchElementException`, while `poll` returns `null`.

The `element` and `peek` methods return, but do not remove, the head of the queue. They differ from one another in precisely the same fashion as `remove` and `poll`: If the queue is empty, `element` throws `NoSuchElementException`, while `peek` returns `null`.

`Queue` implementations generally do not allow insertion of `null` elements. The `LinkedList` implementation, which was retrofitted to implement `Queue`, is an exception. For historical reasons, it permits `null` elements, but you should refrain from taking advantage of this, because `null` is used as a special return value by the `poll` and `peek` methods.

Queue implementations generally do not define element-based versions of the `equals` and `hashCode` methods but instead inherit the identity-based versions from `Object`.

The Map Interface

A [Map](#) is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value. It models the mathematical *function* abstraction. The `Map` interface follows.

```
public interface Map<K,V> {

    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();

    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

The Java platform contains three general-purpose `Map` implementations: [HashMap](#), [TreeMap](#), and [LinkedHashMap](#). Their behavior and performance are precisely analogous to `HashSet`, `TreeSet`, and `LinkedHashSet`, as described in [The Set Interface](#) section. Also, `Hashtable` was retrofitted to implement `Map`.

Comparison to Hashtable

If you've used `Hashtable`, you're already familiar with the general basics of `Map`. (Of course, `Map` is an interface, while `Hashtable` is a concrete implementation.) The following are the major differences:

- Map provides Collection views instead of direct support for iteration via Enumeration objects. Collection views greatly enhance the expressiveness of the interface, as discussed later in this section.
- Map allows you to iterate over keys, values, or key-value pairs; Hashtable does not provide the third option.
- Map provides a safe way to remove entries in the midst of iteration; Hashtable did not.

Finally, Map fixes a minor deficiency in the Hashtable interface. Hashtable has a method called `contains`, which returns `true` if the Hashtable contains a given *value*. Given its name, you'd expect this method to return `true` if the Hashtable contained a given *key*, because the key is the primary access mechanism for a Hashtable. The Map interface eliminates this source of confusion by renaming the method `containsValue`. Also, this improves the interface's consistency — `containsValue` parallels `containsKey`.

Map Interface Basic Operations

The basic operations of Map (`put`, `get`, `containsKey`, `containsValue`, `size`, and `isEmpty`) behave exactly like their counterparts in Hashtable. The [following program](#) generates a frequency table of the words found in its argument list. The frequency table maps each word to the number of times it occurs in the argument list.

```
import java.util.*;

public class Freq {
    public static void main(String[] args) {
        Map<String, Integer> m = new HashMap<String, Integer>();

        // Initialize frequency table from command line
        for (String a : args) {
            Integer freq = m.get(a);
            m.put(a, (freq == null) ? 1 : freq + 1);
        }

        System.out.println(m.size() + " distinct words:");
        System.out.println(m);
    }
}
```

The only tricky thing about this program is the second argument of the `put` statement. That argument is a conditional expression that has the effect of setting the frequency to one if the word has never been seen before or one more than its current value if the word has already been seen. Try running this program with the command:

```
java Freq if it is to be it is up to me to delegate
```

The program yields the following output.

8 distinct words:

```
{to=3, delegate=1, be=1, it=2, up=1, if=1, me=1, is=2}
```

Suppose you'd prefer to see the frequency table in alphabetical order. All you have to do is change the implementation type of the Map from `HashMap` to `TreeMap`. Making this four-character change causes the program to generate the following output from the same command line.

8 distinct words:

```
{be=1, delegate=1, if=1, is=2, it=2, me=1, to=3, up=1}
```

Similarly, you could make the program print the frequency table in the order the words first appear on the command line simply by changing the implementation type of the map to `LinkedHashMap`. Doing so results in the following output.

8 distinct words:

```
{if=1, it=2, is=2, to=3, be=1, up=1, me=1, delegate=1}
```

This flexibility provides a potent illustration of the power of an interface-based framework.

Like the [Set](#) and [List](#) interfaces, `Map` strengthens the requirements on the `equals` and `hashCode` methods so that two `Map` objects can be compared for logical equality without regard to their implementation types. Two `Map` instances are equal if they represent the same key-value mappings.

By convention, all general-purpose `Map` implementations provide constructors that take a `Map` object and initialize the new `Map` to contain all the key-value mappings in the specified `Map`. This standard `Map` conversion constructor is entirely analogous to the standard `Collection` constructor: It allows the caller to create a `Map` of a desired implementation type that initially contains all of the mappings in another `Map`, regardless of the other `Map`'s implementation type. For example, suppose you have a `Map`, named `m`. The following one-liner creates a new `HashMap` initially containing all of the same key-value mappings as `m`.

```
Map<K, V> copy = new HashMap<K, V>(m);
```

Map Interface Bulk Operations

The `clear` operation does exactly what you would think it could do: It removes all the mappings from the `Map`. The `putAll` operation is the `Map` analogue of the `Collection` interface's `addAll` operation. In addition to its obvious use of dumping one `Map` into another, it has a second, more subtle use. Suppose a `Map` is used to represent a collection of attribute-value pairs; the `putAll` operation, in combination with the `Map` conversion constructor,

provides a neat way to implement attribute map creation with default values. The following is a static factory method that demonstrates this technique.

```
static <K, V> Map<K, V> newAttributeMap(Map<K, V>defaults, Map<K, V> overrides) {  
    Map<K, V> result = new HashMap<K, V>(defaults);  
    result.putAll(overrides);  
    return result;  
}
```

Collection Views

The Collection view methods allow a Map to be viewed as a Collection in these three ways:

- `keySet` — the Set of keys contained in the Map.
- `values` — The Collection of values contained in the Map. This Collection is not a Set, because multiple keys can map to the same value.
- `entrySet` — the Set of key-value pairs contained in the Map. The Map interface provides a small nested interface called `Map.Entry`, the type of the elements in this Set.

The Collection views provide the *only* means to iterate over a Map. This example illustrates the standard idiom for iterating over the keys in a Map with a for-each construct:

```
for (KeyType key : m.keySet())  
    System.out.println(key);
```

and with an iterator:

```
// Filter a map based on some  
// property of its keys.  
for (Iterator<Type> it = m.keySet().iterator(); it.hasNext(); )  
    if (it.next().isBogus())  
        it.remove();
```

The idiom for iterating over values is analogous. Following is the idiom for iterating over key-value pairs.

```
for (Map.Entry<KeyType, ValType> e : m.entrySet())  
    System.out.println(e.getKey() + ": " + e.getValue());
```

At first, many people worry that these idioms may be slow because the Map has to create a new Collection instance each time a Collection view operation is called. Rest easy: There's no reason that a Map cannot always return the same object each time it is asked for a given Collection view. This is precisely what all the Map implementations in `java.util` do.

With all three Collection views, calling an Iterator's `remove` operation removes the associated entry from the backing Map, assuming that the backing Map supports element removal to begin with. This is illustrated by the preceding filtering idiom.

With the `entrySet` view, it is also possible to change the value associated with a key by calling a `Map.Entry`'s `setValue` method during iteration (again, assuming the Map supports value modification to begin with). Note that these are the *only* safe ways to modify a Map during iteration; the behavior is unspecified if the underlying Map is modified in any other way while the iteration is in progress.

The Collection views support element removal in all its many forms — `remove`, `removeAll`, `retainAll`, and `clear` operations, as well as the `Iterator.remove` operation. (Yet again, this assumes that the backing Map supports element removal.)

The Collection views *do not* support element addition under any circumstances. It would make no sense for the `keySet` and `values` views, and it's unnecessary for the `entrySet` view, because the backing Map's `put` and `putAll` methods provide the same functionality.

Object Ordering

A List may be sorted as follows.

```
Collections.sort(l);
```

If the List consists of String elements, it will be sorted into alphabetical order. If it consists of Date elements, it will be sorted into chronological order. How does this happen? String and Date both implement the [Comparable](#) interface. Comparable implementations provide a *natural ordering* for a class, which allows objects of that class to be sorted automatically. The following table summarizes some of the more important Java platform classes that implement Comparable.

Classes Implementing Comparable	
Class	Natural Ordering
Byte	Signed numerical
Character	Unsigned numerical
Long	Signed numerical
Integer	Signed numerical
Short	Signed numerical
Double	Signed numerical

Float	Signed numerical
BigInteger	Signed numerical
BigDecimal	Signed numerical
Boolean	Boolean.FALSE < Boolean.TRUE
File	System-dependent lexicographic on path name
String	Lexicographic
Date	Chronological
CollationKey	Locale-specific lexicographic

If you try to sort a list, the elements of which do not implement `Comparable`, `Collections.sort(list)` will throw a [ClassCastException](#). Similarly, `Collections.sort(list, comparator)` will throw a `ClassCastException` if you try to sort a list whose elements cannot be compared to one another using the comparator. Elements that can be compared to one another are called *mutually comparable*. Although elements of different types may be mutually comparable, none of the classes listed here permit interclass comparison.

This is all you really need to know about the `Comparable` interface if you just want to sort lists of comparable elements or to create sorted collections of them. The next section will be of interest to you if you want to implement your own `Comparable` type.

Writing Your Own Comparable Types

The `Comparable` interface consists of the following method.

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

The `compareTo` method compares the receiving object with the specified object and returns a negative integer, 0, or a positive integer depending on whether the receiving object is less than, equal to, or greater than the specified object. If the specified object cannot be compared to the receiving object, the method throws a `ClassCastException`.

The [following class representing a person's name](#) implements `Comparable`.

```
import java.util.*;

public class Name implements Comparable<Name> {
    private final String firstName, lastName;

    public Name(String firstName, String lastName) {
```

```

        if (firstName == null || lastName == null)
            throw new NullPointerException();
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String firstName() { return firstName; }
    public String lastName() { return lastName; }

    public boolean equals(Object o) {
        if (!(o instanceof Name))
            return false;
        Name n = (Name) o;
        return n.firstName.equals(firstName) && n.lastName.equals(lastName);
    }

    public int hashCode() {
        return 31*firstName.hashCode() + lastName.hashCode();
    }

    public String toString() {
        return firstName + " " + lastName;
    }

    public int compareTo(Name n) {
        int lastCmp = lastName.compareTo(n.lastName);
        return (lastCmp != 0 ? lastCmp : firstName.compareTo(n.firstName));
    }
}

```

To keep the preceding example short, the class is somewhat limited: It doesn't support middle names, it demands both a first and a last name, and it is not internationalized in any way. Nonetheless, it illustrates the following important points:

- Name objects are *immutable*. All other things being equal, immutable types are the way to go, especially for objects that will be used as elements in Sets or as keys in Maps. These collections will break if you modify their elements or keys while they're in the collection.
- The constructor checks its arguments for null. This ensures that all Name objects are well formed so that none of the other methods will ever throw a NullPointerException.
- The hashCode method is redefined. This is essential for any class that redefines the equals method. (Equal objects must have equal hash codes.)
- The equals method returns false if the specified object is null or of an inappropriate type. The compareTo method throws a runtime exception under these circumstances. Both of these behaviors are required by the general contracts of the respective methods.
- The toString method has been redefined so it prints the Name in human-readable form. This is always a good idea, especially for objects that are going to get put into collections. The various collection

`types.toString` methods depend on the `toString` methods of their elements, keys, and values.

Since this section is about element ordering, let's talk a bit more about `Name`'s `compareTo` method. It implements the standard name-ordering algorithm, where last names take precedence over first names. This is exactly what you want in a natural ordering. It would be very confusing indeed if the natural ordering were unnatural!

Take a look at how `compareTo` is implemented, because it's quite typical. First, you compare the most significant part of the object (in this case, the last name). Often, you can just use the natural ordering of the part's type. In this case, the part is a `String` and the natural (lexicographic) ordering is exactly what's called for. If the comparison results in anything other than zero, which represents equality, you're done: You just return the result. If the most significant parts are equal, you go on to compare the next most-significant parts. In this case, there are only two parts — first name and last name. If there were more parts, you'd proceed in the obvious fashion, comparing parts until you found two that weren't equal or you were comparing the least-significant parts, at which point you'd return the result of the comparison.

Just to show that it all works, here's [a program that builds a list of names and sorts them](#).

```
import java.util.*;

public class NameSort {
    public static void main(String[] args) {
        Name nameArray[] = {
            new Name("John", "Smith"),
            new Name("Karl", "Ng"),
            new Name("Jeff", "Smith"),
            new Name("Tom", "Rich")
        };

        List<Name> names = Arrays.asList(nameArray);
        Collections.sort(names);
        System.out.println(names);
    }
}
```

If you run this program, here's what it prints.

```
[Karl Ng, Tom Rich, Jeff Smith, John Smith]
```

There are four restrictions on the behavior of the `compareTo` method, which we won't go into now because they're fairly technical and boring and are better left in the API documentation. It's really important that all classes that implement `Comparable` obey these restrictions, so read the documentation for `Comparable` if you're writing a class that implements it. Attempting to sort a

list of objects that violate the restrictions has undefined behavior. Technically speaking, these restrictions ensure that the natural ordering is a *total order* on the objects of a class that implements it; this is necessary to ensure that sorting is well defined.

Comparators

What if you want to sort some objects in an order other than their natural ordering? Or what if you want to sort some objects that don't implement `Comparable`? To do either of these things, you'll need to provide a [Comparator](#) — an object that encapsulates an ordering. Like the `Comparable` interface, the `Comparator` interface consists of a single method.

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

The `compare` method compares its two arguments, returning a negative integer, 0, or a positive integer depending on whether the first argument is less than, equal to, or greater than the second. If either of the arguments has an inappropriate type for the `Comparator`, the `compare` method throws a `ClassCastException`.

Much of what was said about `Comparable` applies to `Comparator` as well. Writing a `compare` method is nearly identical to writing a `compareTo` method, except that the former gets both objects passed in as arguments. The `compare` method has to obey the same four technical restrictions as `Comparable`'s `compareTo` method for the same reason — a `Comparator` must induce a total order on the objects it compares.

Suppose you have a class called `Employee`, as follows.

```
public class Employee implements Comparable<Employee> {  
    public Name name() { ... }  
    public int number() { ... }  
    public Date hireDate() { ... }  
    ...  
}
```

Let's assume that the natural ordering of `Employee` instances is Name ordering (as defined in the previous example) on employee name. Unfortunately, the boss has asked for a list of employees in order of seniority. This means we have to do some work, but not much. The following program will produce the required list.

```
import java.util.*;  
public class EmpSort {  
    static final Comparator<Employee> SENIORITY_ORDER =  
        new Comparator<Employee>() {
```



```

        public int compare(Employee e1, Employee e2) {
            return e2.hireDate().compareTo(e1.hireDate());
        }
    };

    // Employee database
    static final Collection<Employee> employees = ... ;

    public static void main(String[] args) {
        List<Employee> e = new ArrayList<Employee>(employees);
        Collections.sort(e, SENIORITY_ORDER);
        System.out.println(e);
    }
}

```

The `Comparator` in the program is reasonably straightforward. It relies on the natural ordering of `Date` applied to the values returned by the `hireDate` accessor method. Note that the `Comparator` passes the hire date of its second argument to its first rather than vice versa. The reason is that the employee who was hired most recently is the least senior; sorting in the order of hire date would put the list in reverse seniority order. Another technique people sometimes use to achieve this effect is to maintain the argument order but to negate the result of the comparison.

```

// Don't do this!!
return -r1.hireDate().compareTo(r2.hireDate());

```

You should always use the former technique in favor of the latter because the latter is not guaranteed to work. The reason for this is that the `compareTo` method can return any negative `int` if its argument is less than the object on which it is invoked. There is one negative `int` that remains negative when negated, strange as it may seem.

```
-Integer.MIN_VALUE == Integer.MIN_VALUE
```

The `Comparator` in the preceding program works fine for sorting a `List`, but it does have one deficiency: It cannot be used to order a sorted collection, such as `TreeSet`, because it generates an ordering that is *not compatible with equals*. This means that this `Comparator` equates objects that the `equals` method does not. In particular, any two employees who were hired on the same date will compare as equal. When you're sorting a `List`, this doesn't matter; but when you're using the `Comparator` to order a sorted collection, it's fatal. If you use this `Comparator` to insert multiple employees hired on the same date into a `TreeSet`, only the first one will be added to the set; the second will be seen as a duplicate element and will be ignored.

To fix this problem, simply tweak the `Comparator` so that it produces an ordering that is *compatible with equals*. In other words, tweak it so that the only elements seen as equal when using `compare` are those that are also seen as

equal when compared using `equals`. The way to do this is to perform a two-part comparison (as for `Name`), where the first part is the one we're interested in — in this case, the hire date — and the second part is an attribute that uniquely identifies the object. Here the employee number is the obvious attribute. This is the `Comparator` that results.

```
static final Comparator<Employee> SENIORITY_ORDER =
    new Comparator<Employee>() {
        public int compare(Employee e1, Employee e2) {
            int dateCmp = e2.hireDate().compareTo(e1.hireDate());
            if (dateCmp != 0)
                return dateCmp;

            return (e1.number() < e2.number() ? -1 :
                (e1.number() == e2.number() ? 0 : 1));
        }
    };
```

One last note: You might be tempted to replace the final `return` statement in the `Comparator` with the simpler:

```
return e1.number() - e2.number();
```

The SortedSet Interface

A [SortedSet](#) is a [Set](#) that maintains its elements in ascending order, sorted according to the elements' natural ordering or according to a `Comparator` provided at `SortedSet` creation time. In addition to the normal `Set` operations, the `SortedSet` interface provides operations for the following:

- Range view — allows arbitrary range operations on the sorted set
- Endpoints — returns the first or last element in the sorted set
- Comparator access — returns the `Comparator`, if any, used to sort the set

The code for the `SortedSet` interface follows.

```
public interface SortedSet<E> extends Set<E> {
    // Range-view
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);

    // Endpoints
    E first();
    E last();

    // Comparator access
    Comparator<? super E> comparator();
}
```

Set Operations

The operations that `SortedSet` inherits from `Set` behave identically on sorted sets and normal sets with two exceptions:

- The `Iterator` returned by the `iterator` operation traverses the sorted set in order.
- The array returned by `toArray` contains the sorted set's elements in order.

Although the interface doesn't guarantee it, the `toString` method of the Java platform's `SortedSet` implementations returns a string containing all the elements of the sorted set, in order.

Standard Constructors

By convention, all general-purpose `Collection` implementations provide a standard conversion constructor that takes a `Collection`; `SortedSet` implementations are no exception. In `TreeSet`, this constructor creates an instance that sorts its elements according to their natural ordering. This was probably a mistake. It would have been better to check dynamically to see whether the specified collection was a `SortedSet` instance and, if so, to sort the new `TreeSet` according to the same criterion (comparator or natural ordering). Because `TreeSet` took the approach that it did, it also provides a constructor that takes a `SortedSet` and returns a new `TreeSet` containing the same elements sorted according to the same criterion. Note that it is the compile-time type of the argument, not its runtime type, that determines which of these two constructors is invoked (and whether the sorting criterion is preserved).

`SortedSet` implementations also provide, by convention, a constructor that takes a `Comparator` and returns an empty set sorted according to the specified `Comparator`. If `null` is passed to this constructor, it returns a set that sorts its elements according to their natural ordering.

The SortedMap Interface

A [SortedMap](#) is a [Map](#) that maintains its entries in ascending order, sorted according to the keys' natural ordering, or according to a `Comparator` provided at the time of the `SortedMap` creation. Natural ordering and `Comparators` are discussed in the [Object Ordering](#) section. The `SortedMap` interface provides operations for normal `Map` operations and for the following:

- Range view — performs arbitrary range operations on the sorted map
- Endpoints — returns the first or the last key in the sorted map
- Comparator access — returns the `Comparator`, if any, used to sort the map

The following interface is the Map analog of [SortedSet](#).

```
public interface SortedMap<K, V> extends Map<K, V>{
    Comparator<? super K> comparator();
    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
}
```

Map Operations

The operations SortedMap inherits from Map behave identically on sorted maps and normal maps with two exceptions:

- The Iterator returned by the iterator operation on any of the sorted map's Collection views traverse the collections in order.
- The arrays returned by the Collection views' toArray operations contain the keys, values, or entries in order.

Although it isn't guaranteed by the interface, the toString method of the Collection views in all the Java platform's SortedMap implementations returns a string containing all the elements of the view, in order.

Standard Constructors

By convention, all general-purpose Map implementations provide a standard conversion constructor that takes a Map; SortedMap implementations are no exception. In TreeMap, this constructor creates an instance that orders its entries according to their keys' natural ordering. This was probably a mistake. It would have been better to check dynamically to see whether the specified Map instance was a SortedMap and, if so, to sort the new map according to the same criterion (comparator or natural ordering). Because TreeMap took the approach it did, it also provides a constructor that takes a SortedMap and returns a new TreeMap containing the same mappings as the given SortedMap, sorted according to the same criterion. Note that it is the compile-time type of the argument, not its runtime type, that determines whether the SortedMap constructor is invoked in preference to the ordinary map constructor.

SortedMap implementations also provide, by convention, a constructor that takes a Comparator and returns an empty map sorted according to the specified Comparator. If null is passed to this constructor, it returns a Map that sorts its mappings according to their keys' natural ordering.

Examples of collections

HashSet

```
import java.util.*;

public class FindDups {
    public static void main(String args[]){
        Set s = new HashSet();
        for (int i = 0; i < args.length; i++){
            if (!s.add(args[i]))
                System.out.println("Duplicate detected: " +
                    args[i]);
        }
        System.out.println(s.size() + " distinct words detected: " +s);
    }
}
```

ArrayList

```
import java.util.*;

public class Shuffle {
    public static void main(String args[] ) {
        List l = new ArrayList();
        for (int i = 0; i < args.length; i++)
            l.add(args[i]);
        Collections.shuffle(l, new Random());
        System.out.println(l);
    }
}
```

LinkedList

```
import java.util.*;

public class MyStack {
    private LinkedList list = new LinkedList();
    public void push(Object o){
        list.addFirst(o);
    }
    public Object top(){
        return list.getFirst();
    }
    public Object pop(){
        return list.removeFirst();
    }
    public static void main(String args[]) {
        Car myCar;
```

```

        MyStack s = new MyStack();
        s.push (new Car());
        myCar = (Car)s.pop();
    }

}

```

HashMap

```

import java.util.*;
public class Freq {
    private static final Integer ONE = new Integer(1);
    public static void main(String args[]) {
        Map m = new HashMap();
        // Initialize frequency table from command line
        for (int i=0; i < args.length; i++) {
            Integer freq = (Integer) m.get(args[i]);
            m.put(args[i], (freq==null ? ONE :
                new Integer(freq.intValue() + 1)));
        }
        System.out.println(m.size()+" distinct words detected:");
        System.out.println(m);
    }
}

```

Varargs

In past releases, a method that took an arbitrary number of values required you to create an array and put the values into the array prior to invoking the method. For example, here is how one used the [MessageFormat](#) class to format a message:

```

Object[] arguments = {
    new Integer(7),
    new Date(),
    "a disturbance in the Force"
};

String result = MessageFormat.format(
    "At {1,time} on {1,date}, there was {2} on planet "
    + "{0,number,integer}.", arguments);

```

It is still true that multiple arguments must be passed in an array, but the varargs feature automates and hides the process. Furthermore, it is upward compatible with preexisting APIs. So, for example, the `MessageFormat.format` method now has this declaration:

```

public static String format(String pattern, Object... arguments);

```

The three periods after the final parameter's type indicate that the final argument may be passed as an array *or* as a sequence of arguments. Varargs

can be used *only* in the final argument position. Given the new varargs declaration for `MessageFormat.format`, the above invocation may be replaced by the following shorter and sweeter invocation:

```
String result = MessageFormat.format(
    "At {1,time} on {1,date}, there was {2} on planet "
    + "{0,number,integer}.",
    7, new Date(), "a disturbance in the Force");
```

Basic I/O

This lesson covers the Java platform classes used for basic I/O. It first focuses on *I/O Streams*, a powerful concept that greatly simplifies I/O operations. The lesson also looks at serialization, which lets a program write whole objects out to streams and read them back again. Then the lesson looks at file I/O and file system operations, including random access files.

Most of the classes covered in the `I/O Streams` section are in the `java.io` package. Most of the classes covered in the `File I/O` section are in the `java.nio.file` package.

I/O Streams

- Byte Streams handle I/O of raw binary data.
- Character Streams handle I/O of character data, automatically handling translation to and from the local character set.
- Buffered Streams optimize input and output by reducing the number of calls to the native API.
- Scanning and Formatting allows a program to read and write formatted text.
- I/O from the Command Line describes the Standard Streams and the Console object.
- Data Streams handle binary I/O of primitive data type and String values.
- Object Streams handle binary I/O of objects.

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

We'll explore `FileInputStream` and `FileOutputStream` by examining an example program named `CopyBytes`, which uses byte streams to copy `xanadu.txt`, one byte at a time.

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

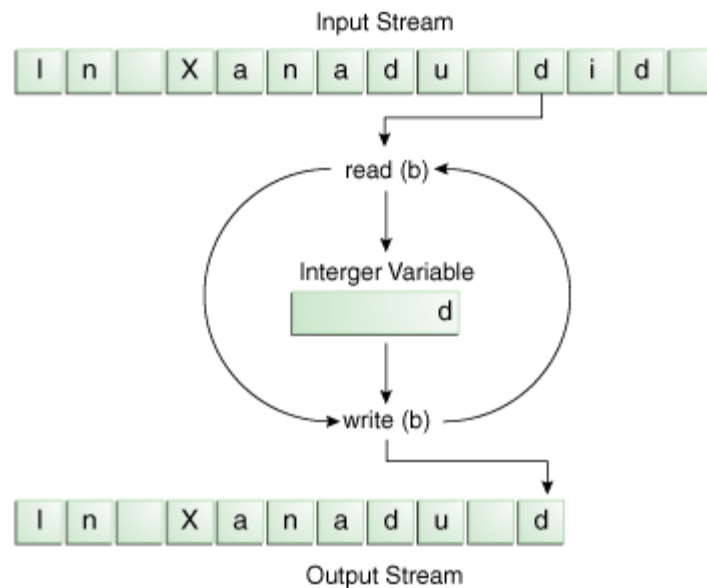
public class CopyBytes {
    public static void main(String[] args) throws IOException {

        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();
            }
        }
    }
}
```

`CopyBytes` spends most of its time in a simple loop that reads the input stream and writes the output stream, one byte at a time, as shown in the following figure.



Simple byte stream input and output.

Notice that `read()` returns an `int` value. If the input is a stream of bytes, why doesn't `read()` return a `byte` value? Using a `int` as a return type allows `read()` to use `-1` to indicate that it has reached the end of the stream.

Always Close Streams

Closing a stream when it's no longer needed is very important — so important that `CopyBytes` uses a `finally` block to guarantee that both streams will be closed even if an error occurs. This practice helps avoid serious resource leaks.

One possible error is that `CopyBytes` was unable to open one or both files. When that happens, the stream variable corresponding to the file never changes from its initial `null` value. That's why `CopyBytes` makes sure that each stream variable contains an object reference before invoking `close`.

When Not to Use Byte Streams

`CopyBytes` seems like a normal program, but it actually represents a kind of low-level I/O that you should avoid. Since `xanadu.txt` contains character data, the best approach is to use [character streams](#), as discussed in the next section. There are also streams for more complicated data types. Byte streams should only be used for the most primitive I/O.

Character Streams

The Java platform stores character values using Unicode conventions. Character stream I/O automatically translates this internal format to and

from the local character set. In Western locales, the local character set is usually an 8-bit superset of ASCII.

For most applications, I/O with character streams is no more complicated than I/O with byte streams. Input and output done with stream classes automatically translates to and from the local character set. A program that uses character streams in place of byte streams automatically adapts to the local character set and is ready for internationalization — all without extra effort by the programmer.

If internationalization isn't a priority, you can simply use the character stream classes without paying much attention to character set issues. Later, if internationalization becomes a priority, your program can be adapted without extensive recoding. See the [Internationalization](#) trail for more information.

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. The [sockets lesson](#) in the [networking trail](#) shows how to create character streams from the byte streams provided by socket classes.

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.

Let's modify the `CopyCharacters` example to use line-oriented I/O. To do this, we have to use two classes we haven't seen before, `BufferedReader` and `PrintWriter`. We'll explore these classes in greater depth in [Buffered I/O](#) and [Formatting](#). Right now, we're just interested in their support for line-oriented I/O.

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.

Flushing Buffered Streams

It often makes sense to write out a buffer at critical points, without waiting for it to fill. This is known as *flushing* the buffer.

Some buffered output classes support *autoflush*, specified by an optional constructor argument. When autoflush is enabled, certain key events cause the buffer to be flushed. For example, an autoflush `PrintWriter` object flushes the buffer on every invocation of `println` or `format`.

To flush a stream manually, invoke its `flush` method. The `flush` method is valid on any output stream, but has no effect unless the stream is buffered.

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	<code>DataOutputStream.writeDouble</code>	<code>DataInputStream.readDouble</code>	19.99
2	int	Unit count	<code>DataOutputStream.writeInt</code>	<code>DataInputStream.readInt</code>	12
3	String	Item description	<code>DataOutputStream.writeUTF</code>	<code>DataInputStream.readUTF</code>	"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.

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 [BigDecimal](#) objects, 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.

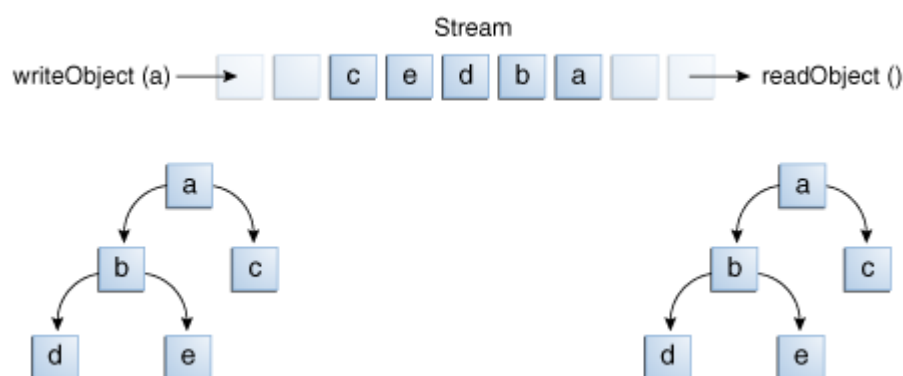
Output and Input of Complex Objects

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.

Serialization API

We all know the Java platform allows us to create reusable objects in memory. However, all of those objects exist only as long as the Java virtual machine¹ remains running. It would be nice if the objects we create could exist beyond the lifetime of the virtual machine, wouldn't it? Well, with object serialization, you can flatten your objects and reuse them in powerful ways.

Object serialization is the process of saving an object's state to a sequence of bytes, as well as the process of rebuilding those bytes into a live object at some future time. The Java Serialization API provides a standard mechanism for developers to handle object serialization. The API is small and easy to use, provided the classes and methods are understood.

Let's start with the basics. To persist an object in Java, we must have a persistent object. An object is marked serializable by implementing the `java.io.Serializable` interface, which signifies to the underlying API that the object can be flattened into bytes and subsequently inflated in the future.

Let's look at a persistent class we'll use to demonstrate the serialization mechanism:

```
10 import java.io.Serializable;  
20 import java.util.Date;  
30 import java.util.Calendar;  
40 public class PersistentTime implements Serializable  
50 {  
60     private Date time;  
70  
80     public PersistentTime()  
90 {  
100     time = Calendar.getInstance().getTime();
```

```

110 }
120
130 public Date getTime()
140 {
150     return time;
160 }
170 }

```

As you can see, the only thing we had to do differently from creating a normal class is implement the `java.io.Serializable` interface on line 40. The completely empty `Serializable` is only a *marker* interface -- it simply allows the serialization mechanism to verify that the class is able to be persisted. Thus, we turn to the first rule of serialization:

Rule #1: The object to be persisted must implement the `Serializable` interface or inherit that implementation from its object hierarchy.

The next step is to actually persist the object. That is done with the `java.io.ObjectOutputStream` class. That class is a *filter stream*--it is wrapped around a lower-level byte stream (called a *node stream*) to handle the serialization protocol for us. Node streams can be used to write to file systems or even across sockets. That means we could easily transfer a flattened object across a network wire and have it be rebuilt on the other side!

Take a look at the code used to save the `PersistentTime` object:

```

10 import java.io.ObjectOutputStream;
20 import java.io.FileOutputStream;
30 import java.io.IOException;
40 public class FlattenTime
50 {
60     public static void main(String [] args)
70     {
80         String filename = "time.ser";
90         if(args.length > 0)
100         {
110             filename = args[0];
120         }
130         PersistentTime time = new PersistentTime();
140         FileOutputStream fos = null;
150         ObjectOutputStream out = null;
160         try
170         {
180             fos = new FileOutputStream(filename);
190             out = new ObjectOutputStream(fos);
200             out.writeObject(time);
210             out.close();
220         }
230         catch(IOException ex)
240         {

```

```

250     ex.printStackTrace();
260 }
270 }
280 }

```

The real work happens on line 200 when we call the `ObjectOutputStream.writeObject()` method, which kicks off the serialization mechanism and the object is flattened (in that case to a file).

To restore the file, we can employ the following code:

```

10 import java.io.ObjectInputStream;
20 import java.io.FileInputStream;
30 import java.io.IOException;
40 import java.util.Calendar;
50 public class InflateTime
60 {
70     public static void main(String [] args)
80     {
90         String filename = "time.ser";
100        if(args.length > 0)
110        {
120            filename = args[0];
130        }
140        PersistentTime time = null;
150        FileInputStream fis = null;
160        ObjectInputStream in = null;
170        try
180        {
190            fis = new FileInputStream(filename);
200            in = new ObjectInputStream(fis);
210            time = (PersistentTime)in.readObject();
220            in.close();
230        }
240        catch(IOException ex)
250        {
260            ex.printStackTrace();
270        }
280        catch(ClassNotFoundException ex)
290        {
300            ex.printStackTrace();
310        }
320        // print out restored time
330        System.out.println("Flattened time: " + time.getTime());
340        System.out.println();
350        // print out the current time
360        System.out.println("Current time: " + Calendar.getInstance().getTime());
370    }
380}

```

In the code above, the object's restoration occurs on line 210 with the `ObjectInputStream.readObject()` method call. The method call reads in the raw bytes that we previously persisted and creates a live object that is an exact replica of the original. Because `readObject()` can read any serializable object, a cast to the correct type is required. With that in mind, the class file must be accessible from the system in which the restoration occurs. In other words, the object's class file and methods are not saved; only the object's *state* is saved.

Later, on line 360, we simply call the `getTime()` method to retrieve the time that the original object flattened. The flatten time is compared to the current time to demonstrate that the mechanism indeed worked as expected.

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](#).) To see how scanning works, let's look at [ScanXan](#), a program that reads the individual words in `xanadu.txt` and prints them out, one per line.

```
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();
            }
        }
    }
}
```

Notice that `ScanXan` invokes `Scanner`'s `close` method when it is done with the scanner object. Even though a scanner is not a stream, you need to close it to indicate that you're done with its underlying stream.

The output of `ScanXan` looks like this:

```
In
Xanadu
did
Kubla
Khan
A
statelý
pleasure-dome
...
```

To use a different token separator, invoke `useDelimiter()`, specifying a regular expression. For example, suppose you wanted the token separator to be a comma, optionally followed by white space. You would invoke,

```
s.useDelimiter(",\\s*");
```

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.

We have to mention the locale, because thousands separators and decimal symbols are locale specific. So, the following example would not work correctly in all locales if we didn't specify that the scanner should use the `US` locale. That's not something you usually have to worry about, because your input data usually comes from sources that use the same locale as you do. But this example is part of the Java Tutorial and gets distributed all over the world.

The [ScanSum](#) example reads a list of `double` values and adds them up. Here's the source:

```
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);
}
}

```

And here's the sample input file, [usnumbers.txt](#)

```

8.5
32,767
3.14159
1,000,000.1

```

The output string is "1032778.74159". The period will be a different character in some locales, because `System.out` is a `PrintStream` object, and that class doesn't provide a way to override the default locale.

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. For more information, refer to the documentation for your command line interpreter.

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.

The [Password](#) example is a prototype program for changing a user's password. It demonstrates several `Console` methods.

```
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:
 - a. Invoke `Console.ReadPassword` twice to prompt for and read a new password.
 - b. If the user entered the same password both times, invoke `change` to change it. (Again, `change` is a dummy method.)
 - c. Overwrite both passwords with blanks.
6. Overwrite the old password with blanks.