Java Tutorial for Beginners

from http://java.sun.com/

This material is obtained java tutorial as is from http://java.sun.com. This material covers everything needed for a beginner.

Lesson: Object-Oriented Programming Concepts

If you've never used an object-oriented programming language before, you'll need to learn a few basic concepts before you can begin writing any code. This lesson will introduce you to objects, classes, inheritance, interfaces, and packages. Each discussion focuses on how these concepts relate to the real world, while simultaneously providing an introduction to the syntax of the Java programming language.

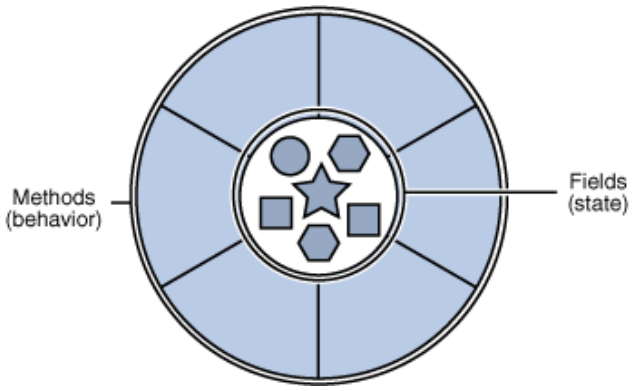
What Is an Object?

An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life. This lesson explains how state and behavior are represented within an object, introduces the concept of data encapsulation, and explains the benefits of designing your software in this manner.

Objects are key to understanding object-oriented technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

Real-world objects share two characteristics: They all have state and behavior. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

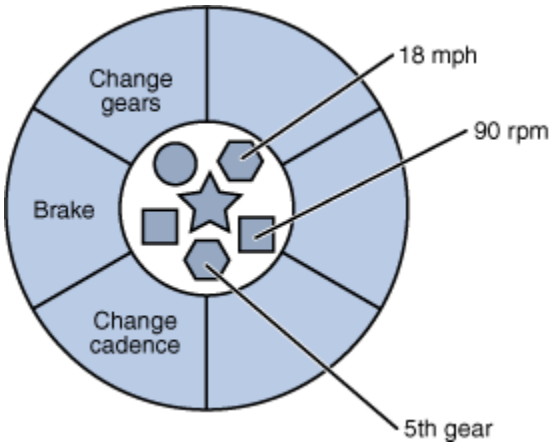
Take a minute right now to observe the real-world objects that are in your immediate area. For each object that you see, ask yourself two questions: "What possible states can this object be in?" and "What possible behavior can this object perform?". Make sure to write down your observations. As you do, you'll notice that real-world objects vary in complexity; your desktop lamp may have only two possible states (on and off) and two possible behaviors (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). You may also notice that some objects, in turn, will also contain other objects. These real-world observations all translate into the world of object-oriented programming.



A software object.

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

Consider a bicycle, for example:



A bicycle modeled as a software object.

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

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

1. Modularity: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.

2. Information-hiding: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.

3. Code re-use: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.

4. Pluggability and debugging ease: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace it, not the entire machine.

What Is a Class?

A class is a blueprint or prototype from which objects are created. This section defines a class that models the state and behavior of a real-world object. It intentionally focuses on the basics, showing how even a simple class can cleanly model state and behavior.

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.

The following Bicycle class is one possible implementation of a bicycle:

```
class Bicycle {

       int cadence = 0;
       int speed = 0;
       int gear = 1;

       void changeCadence(int newValue) {
            cadence = newValue;
       }

       void changeGear(int newValue) {
            gear = newValue;
       }

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

       void applyBrakes(int decrement) {
            speed = speed - decrement;
       }

       void printStates() {
            System.out.println("cadence:"+cadence+" speed:"+speed+"
gear:"+gear);
       }
}
```

The syntax of the Java programming language will look new to you, but the design of this class is based on the previous discussion of bicycle objects. The fields cadence, speed, and gear represent the object's state, and the methods (changeCadence, changeGear, speedUp etc.) define its interaction with the outside world.

You may have noticed that the Bicycle class does not contain a main method. That's because it's not a complete application; it's just the blueprint for bicycles that might be used in an application. The responsibility of creating and using new Bicycle objects belongs to some other class in your application.

Here's a BicycleDemo class that creates two separate Bicycle objects and invokes their methods:

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

          // Create two different Bicycle objects
          Bicycle bike1 = new Bicycle();
          Bicycle bike2 = new Bicycle();

          // Invoke methods on those objects
          bike1.changeCadence(50);
          bike1.speedUp(10);
          bike1.changeGear(2);
          bike1.printStates();

          bike2.changeCadence(50);
          bike2.speedUp(10);
          bike2.changeGear(2);
          bike2.changeCadence(40);
          bike2.speedUp(10);
          bike2.changeGear(3);
          bike2.printStates();
     }
}
```

The output of this test prints the ending pedal cadence, speed, and gear for the two bicycles:

cadence:50 speed:10 gear:2

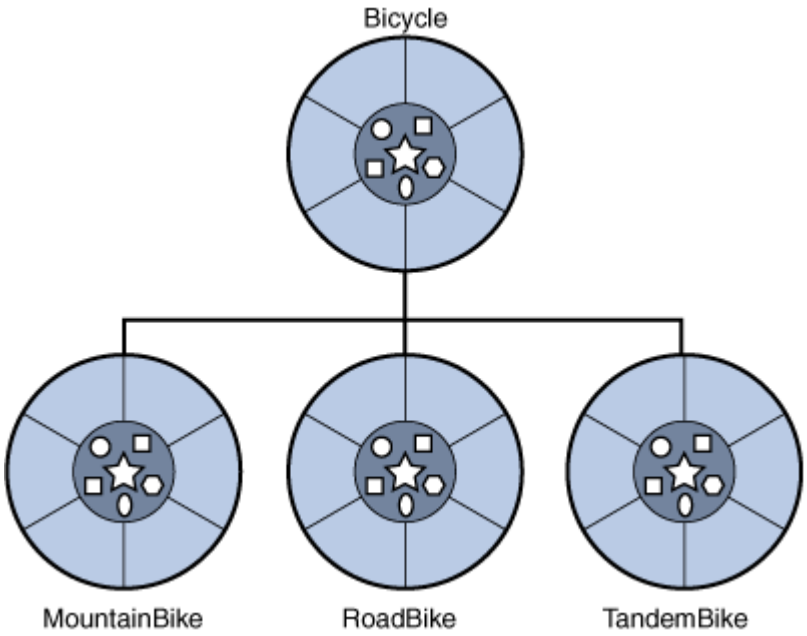
cadence:40 speed:20 gear:3

What Is Inheritance?

Inheritance provides a powerful and natural mechanism for organizing and structuring your software. This section explains how classes inherit state and behavior from their superclasses, and explains how to derive one class from another using the simple syntax provided by the Java programming language.

Different kinds of objects often have a certain amount in common with each other. Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear). Yet each also defines additional features that make them different: tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars; some mountain bikes have an additional chain ring, giving them a lower gear ratio.

Object-oriented programming allows classes to inherit commonly used state and behavior from other classes. In this example, Bicycle now becomes the superclass of MountainBike, RoadBike, and TandemBike. In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of subclasses:



A hierarchy of bicycle classes.

The syntax for creating a subclass is simple. At the beginning of your class declaration, use the extends keyword, followed by the name of the class to inherit from:

```
class MountainBike extends Bicycle {

    // new fields and methods defining a mountain bike would go here

}
```

This gives MountainBike all the same fields and methods as Bicycle, yet allows its code to focus exclusively on the features that make it unique. This makes code for your subclasses easy to read. However, you must take care to properly document the state and behavior that each superclass defines, since that code will not appear in the source file of each subclass.

What Is an Interface?

An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface. This section defines a simple interface and explains the necessary changes for any class that implements it.

As you've already learned, objects define their interaction with the outside world through the methods that they expose. Methods form the object's interface with the outside world; the buttons

on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to turn the television on and off.

In its most common form, an interface is a group of related methods with empty bodies. A bicycle's behavior, if specified as an interface, might appear as follows:

```
interface Bicycle {

        void changeCadence(int newValue);   // wheel revolutions per minute

        void changeGear(int newValue);

        void speedUp(int increment);

        void applyBrakes(int decrement);
}
```

To implement this interface, the name of your class would change (to a particular brand of bicycle, for example, such as ACMEBicycle), and you'd use the implements keyword in the class declaration:

```
class ACMEBicycle implements Bicycle {

  // remainder of this class implemented as before

}
```

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

Note: To actually compile the ACMEBicycle class, you'll need to add the public keyword to the beginning of the implemented interface methods. You'll learn the reasons for this later in the lessons on Classes and Objects and Interfaces and Inheritance.

What Is a Package?

A package is a namespace for organizing classes and interfaces in a logical manner. Placing your code into packages makes large software projects easier to manage. This section explains why this is useful, and introduces you to the Application Programming Interface (API) provided by the Java platform.

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

The Java platform provides an enormous class library (a set of packages) suitable for use in your own applications. This library is known as the "Application Programming Interface", or "API" for short. Its packages represent the tasks most commonly associated with general-purpose programming. For example, a String object contains state and behavior for character strings; a File object allows a programmer to easily create, delete, inspect, compare, or modify a file on the filesystem; a Socket object allows for the creation and use of network sockets; various GUI objects control buttons and checkboxes and anything else related to graphical user interfaces. There are literally thousands of classes to choose from. This allows you, the programmer, to focus on the design of your particular application, rather than the infrastructure required to make it work.

The Java Platform API Specification contains the complete listing for all packages, interfaces, classes, fields, and methods supplied by the Java Platform 6, Standard Edition. Load the page in your browser and bookmark it. As a programmer, it will become your single most important piece of reference documentation.

Questions and Exercises: Object-Oriented Programming Concepts

Use the questions and exercises presented in this section to test your understanding of objects, classes, inheritance, interfaces, and packages.

Questions

1. Real-world objects contain ___ and ___.
2. A software object's state is stored in ___.
3. A software object's behavior is exposed through ___.
4. Hiding internal data from the outside world, and accessing it only through publicly exposed methods is known as data ___.
5. A blueprint for a software object is called a ___.
6. Common behavior can be defined in a ___ and inherited into a ___ using the ___ keyword.
7. A collection of methods with no implementation is called an ___.

8. A namespace that organizes classes and interfaces by functionality is called a ___.
9. The term API stands for ___?

Exercises

1. Create new classes for each real-world object that you observed at the beginning of this trail. Refer to the Bicycle class if you forget the required syntax.
2. For each new class that you've created above, create an interface that defines its behavior, then require your class to implement it. Omit one or two methods and try compiling. What does the error look like?

Answers to Questions

1. Real-world objects contain **state** and **behavior**.
2. A software object's state is stored in **fields**.
3. A software object's behavior is exposed through **methods**.
4. Hiding internal data from the outside world, and accessing it only through publicly exposed methods is known as data **encapsulation**.
5. A blueprint for a software object is called a **class**.
6. Common behavior can be defined in a **superclass** and inherited into a **subclass** using the **extends** keyword.
7. A collection of methods with no implementation is called an **interface**.
8. A namespace that organizes classes and interfaces by functionality is called a **package**.
9. The term API stands for **Application Programming Interface**.

Answers to Exercises

1. Your answers will vary depending on the real-world objects that you are modeling.
2. Your answers will vary here as well, but the error message will specifically list the required methods that have not been implemented.

Lesson: Language Basics

Variables

You've already learned that objects store their state in fields. However, the Java programming language also uses the term "variable" as well. This section discusses this relationship, plus variable naming rules and conventions, basic data types (primitive types, character strings, and arrays), default values, and literals.

As you learned in the previous lesson, an object stores its state in fields.

int **cadence** = 0;
int **speed** = 0;
int **gear** = 1;

The What Is an Object? discussion introduced you to fields, but you probably have still a few questions, such as: What are the rules and conventions for naming a field? Besides int, what other data types are there? Do fields have to be initialized when they are declared? Are fields assigned a default value if they are not explicitly initialized? We'll explore the answers to such questions in this lesson, but before we do, there are a few technical distinctions you must first become aware of. In the Java programming language, the terms "field" and "variable" are both used; this is a common source of confusion among new developers, since both often seem to refer to the same thing.

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); the currentSpeed of one bicycle is independent from the currentSpeed of another.

- **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. The code static int numGears = 6; would create such a static field. Additionally, the keyword final could be added to indicate that the number of gears will never change.

- **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** You've already seen examples of parameters, both in the Bicycle class and in the main method of the "Hello World!" application. Recall that the signature for the main method is public static void main(String[] args). Here, the args variable is the parameter to this method. The important thing to remember is that parameters are always classified as "variables" not "fields". This applies to other parameter-accepting constructs as well (such as constructors and exception handlers) that you'll learn about later in the tutorial.

Having said that, the remainder of this tutorial uses the following general guidelines when discussing fields and variables. If we are talking about "fields in general" (excluding local variables and parameters), we may simply say "fields". If the discussion applies to "all of the above", we may simply say "variables". If the context calls for a distinction, we will use specific terms (static field, local variables, etc.) as appropriate. You may also occasionally see the term "member" used as well. A type's fields, methods, and nested types are collectively called its members.

Naming

Every programming language has its own set of rules and conventions for the kinds of names that you're allowed to use, and the Java programming language is no different. 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; fields named cadence, speed, and gear, for example, are much more intuitive than abbreviated versions, such as s, c, and g. 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_GEARS = 6, the convention changes slightly, capitalizing every letter and separating subsequent words with the underscore character. By convention, the underscore character is never used elsewhere.

Primitive Data Types

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 a 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. You'll learn more about the String class in Simple Data Objects

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;
```

The integral types (byte, short, int, and long) can be expressed using decimal, octal, or hexadecimal number systems. Decimal is the number system you already use every day; it's based on 10 digits, numbered 0 through 9. The octal number system is base 8, consisting of the digits 0 through 7. The hexadecimal system is base 16, whose digits are the numbers 0 through 9 and the letters A through F. For general-purpose programming, the decimal system is likely to be

the only number system you'll ever use. However, if you need octal or hexadecimal, the following example shows the correct syntax. The prefix 0 indicates octal, whereas 0x indicates hexadecimal.

```
int decVal = 26;      // The number 26, in decimal
int octVal = 032;     // The number 26, in octal
int hexVal = 0x1a;    // The number 26, in hexadecimal
```

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

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.

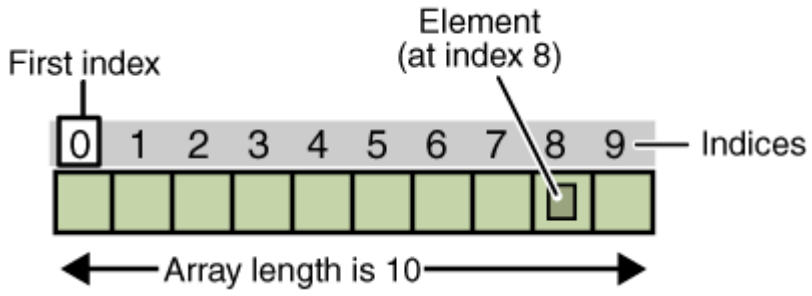
The Java programming language also supports a few special escape sequences for char and String literals: \b (backspace), \t (tab), \n (line feed), \f (form feed), \r (carriage return), \" (double quote), \' (single quote), and \\ (backslash).

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.

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) {
          int[] anArray;              // declares an array of integers

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

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

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

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:

float anArrayOfFloats[]; // this form is discouraged

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.

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

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"}};
        System.out.println(names[0][0] + names[1][0]); //Mr. Smith
        System.out.println(names[0][2] + names[1][1]); //Ms. Jones
    }
}
```

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

Summary of Variables

The Java programming language uses both "fields" and "variables" as part of its terminology. Instance variables (non-static fields) are unique to each instance of a class. Class variables (static fields) are fields declared with the `static` modifier; there is exactly one copy of a class variable, regardless of how many times the class has been instantiated. Local variables store temporary state inside a method. Parameters are variables that provide extra information to a method; both local variables and parameters are always classified as "variables" (not "fields"). When naming your fields or variables, there are rules and conventions that you should (or must) follow.

The eight primitive data types are: byte, short, int, long, float, double, boolean, and char. The `java.lang.String` class represents character strings. The compiler will assign a reasonable default value for fields of the above types; for local variables, a default value is never assigned. A literal is the source code representation of a fixed value. 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.

Questions and Exercises: Variables

Questions

1. The term "instance variable" is another name for ___.
2. The term "class variable" is another name for ___.
3. A local variable stores temporary state; it is declared inside a ___.
4. A variable declared within the opening and closing parenthesis of a method signature is called a ____.
5. What are the eight primitive data types supported by the Java programming language?
6. Character strings are represented by the class ___.
7. An ___ is a container object that holds a fixed number of values of a single type.

Exercises

1. Create a small program that defines some fields. Try creating some illegal field names and see what kind of error the compiler produces. Use the naming rules and conventions as a guide.
2. In the program you created in Exercise 1, try leaving the fields uninitialized and print out their values. Try the same with a local variable and see what kind of compiler errors you can produce. Becoming familiar with common compiler errors will make it easier to recognize bugs in your code.

Answers to Questions on Variables

1. The term "instance variable" is another name for **non-static field**.
2. The term "class variable" is another name for **static field**.
3. A local variable stores temporary state; it is declared inside a **method**.
4. A variable declared within the opening and closing parenthesis of a method is called a **parameter**.
5. What are the eight primitive data types supported by the Java programming language? **byte, short, int, long, float, double, boolean, char**
6. Character strings are represented by the class **java.lang.String**.
7. An **array** is a container object that holds a fixed number of values of a single type.

Answers to Exercises on Variables

1. Create a small program that defines some fields. Try creating some illegal field names and see what kind of error the compiler produces. Use the naming rules and conventions as a guide.

There is no single correct answer here. Your results will vary depending on your code.

2. In the program you created in Exercise 1, try leaving the fields uninitialized and print out their values. Try the same with a local variable and see what kind of compiler errors you can produce. Becoming familiar with common compiler errors will make it easier to recognize bugs in your code.

Again, there is no single correct answer for this exercise. Your results will vary depending on your code.

Operators

Now that you've learned how to declare and initialize variables, you probably want to know how to do something with them. Learning the operators of the Java programming language is a good place to start. Operators are special symbols that perform specific operations on one, two, or three operands, and then return a result.

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	`expr++ expr--`
unary	`++expr --expr +expr -expr ~ !`
multiplicative	`* / %`
additive	`+ -`
shift	`<< >> >>>`
relational	`< > <= >= instanceof`
equality	`== !=`
bitwise AND	`&`
bitwise exclusive OR	`^`
bitwise inclusive OR	`|`
logical AND	`&&`
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.

Assignment, Arithmetic, and Unary Operators

The Simple Assignment Operator

One of the most common operators that you'll encounter is the simple assignment operator "=". You saw this operator in the Bicycle class; it assigns the value on its right to the operand on its left:

```
int cadence = 0;
int speed = 0;
int gear = 1;
```

This operator can also be used on objects to assign object references, as discussed in Creating Objects.

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){

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

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

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

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

          result = result + 8; // result is now 10
          result = result % 7; // result is now 3
          System.out.println(result);

     }
}
```

You can also combine the arithmetic operators with the simple assignment operator to create compound assignments. For example, `x+=1;` and `x=x+1;` both increment the value of `x` by 1.

The `+` operator can also be used for concatenating (joining) two strings together, as shown in the following `ConcatDemo` program:

```
class ConcatDemo {
     public static void main(String[] args){
          String firstString = "This is";
          String secondString = " a concatenated string.";
          String thirdString = firstString+secondString;
          System.out.println(thirdString);
     }
}
```

By the end of this program, the variable `thirdString` contains "This is a concatenated string.", which gets printed to standard output.

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){
          int result = +1; // result is now 1
          System.out.println(result);
          result--;  // result is now 0
          System.out.println(result);
          result++; // result is now 1
          System.out.println(result);
          result = -result; // result is now -1
          System.out.println(result);
          boolean success = false;
          System.out.println(success); // false
          System.out.println(!success); // true
     }
}
```

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++;
          System.out.println(i);       // "4"
          ++i;
          System.out.println(i);       // "5"
          System.out.println(++i);     // "6"
          System.out.println(i++);     // "6"
          System.out.println(i);       // "7"
     }
}
```

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.

```
==      equal to
!=      not equal to
>       greater than
>=      greater than or equal to
<       less than
<=      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.

Bitwise and Bit Shift Operators

The Java programming language also provides operators that perform bitwise and bit shift operations on integral types. The operators discussed in this section are less commonly used.

Therefore, their coverage is brief; the intent is to simply make you aware that these operators exist.

The unary bitwise complement operator "~" inverts a bit pattern; it can be applied to any of the integral types, making every "0" a "1" and every "1" a "0". For example, a `byte` contains 8 bits; applying this operator to a value whose bit pattern is "00000000" would change its pattern to "11111111".

The signed left shift operator "<<" shifts a bit pattern to the left, and the signed right shift operator ">>" shifts a bit pattern to the right. The bit pattern is given by the left-hand operand, and the number of positions to shift by the right-hand operand. The unsigned right shift operator ">>>" shifts a zero into the leftmost position, while the leftmost position after `">>"` depends on sign extension.

The bitwise `&` operator performs a bitwise AND operation.

The bitwise `^` operator performs a bitwise exclusive OR operation.

The bitwise `|` operator performs a bitwise inclusive OR operation.

The following program, `BitDemo`, uses the bitwise AND operator to print the number "2" to standard output.

```
class BitDemo {
     public static void main(String[] args) {
          int bitmask = 0x000F;
          int val = 0x2222;
          System.out.println(val & bitmask);  // prints "2"
     }
}
```

Summary of Operators

The following quick reference summarizes the operators supported by the Java programming language.

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 compliment 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
```

Questions and Exercises: Operators

Questions

Consider the following code snippet.
arrayOfInts[j] > arrayOfInts[j+1]
Which operators does the code contain?

Consider the following code snippet.
int i = 10;
int n = i++%5;

What are the values of i and n after the code is executed?

What are the final values of i and n if instead of using the postfix increment operator (i++), you use the prefix version (++i))?

To invert the value of a boolean, which operator would you use?

Which operator is used to compare two values, = or == ?

Explain the following code sample: result = someCondition ? value1 : value2;

Exercises

Change the following program to use compound assignments:

```
class ArithmeticDemo {

     public static void main (String[] args){

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

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

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

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

          result = result + 8; // result is now 10
          result = result % 7; // result is now 3
          System.out.println(result);

     }
}
```

In the following program, explain why the value "6" is printed twice in a row:

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

Answers to Questions

Consider the following code snippet:
arrayOfInts[j] > arrayOfInts[j+1]

Question: What operators does the code contain?
Answer: >, +

Consider the following code snippet:
int i = 10;
int n = i++%5;

Question: What are the values of i and n after the code is executed?
Answer: i is 11, and n is 0.

Question: What are the final values of i and n if instead of using the postfix increment operator (i++), you use the prefix version (++i))?
Answer: i is 11, and n is 1.

Question: To invert the value of a boolean, which operator would you use?
Answer: The logical complement operator "!".

Question: Which operator is used to compare two values, = or == ?
Answer: The == operator is used for comparison, and = is used for assignment.

Question: Explain the following code sample: result = someCondition ? value1 : value2;
Answer: This code should be read as: "If someCondition is true, assign the value of value1 to result. Otherwise, assign the value of value2 to result."

Exercises

Change the following program to use compound assignments:

```
class ArithmeticDemo {

     public static void main (String[] args){

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

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

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

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

          result = result + 8; // result is now 10
          result = result % 7; // result is now 3
```

```
          System.out.println(result);

     }
}
```

Here is one solution:

```
class ArithmeticDemo {

     public static void main (String[] args){
          int result = 3;
          System.out.println(result);

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

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

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

          result += 8; // result is now 10
          result %= 7; // result is now 3
          System.out.println(result);

     }
}
```

In the following program, explain why the value "6" is printed twice in a row:

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

The code System.out.println(++i); evaluates to 6, because the prefix version of ++ evaluates to the incremented value. The next line, System.out.println(i++); evaluates to the current value (6), then increments by one. So "7" doesn't get printed until the next line.

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:

```
x + y / 100

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

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.

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

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:

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

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
     }
}
```

Questions and Exercises: Expressions, Statements, and Blocks

Questions

1. Operators may be used in building ___, which compute values.
2. Expressions are the core components of ___.
3. Statements may be grouped into ___.
4. The following code snippet is an example of a ___ expression.
5. 1 * 2 * 3
6. Statements are roughly equivalent to sentences in natural languages, but instead of ending with a period, a statement ends with a ___.
7. A block is a group of zero or more statements between balanced ___ and can be used anywhere a single statement is allowed.

Exercises

Identify the following kinds of expression statements:

- aValue = 8933.234;
- aValue++;
- System.out.println("Hello World!");
- Bicycle myBike = new Bicycle();

Answers to Questions and Exercises: Expressions, Statements, and Blocks

Questions

1. Operators may be used in building **expressions**, which compute values.
2. Expressions are the core components of **statements**.
3. Statements may be grouped into **blocks**.
4. The following code snippet is an example of a **compound** expression.
   ```
   5.  1 * 2 * 3
   ```
6. Statements are roughly equivalent to sentences in natural languages, but instead of ending with a period, a statement ends with a **semicolon**.
7. A block is a group of zero or more statements between balanced **braces** and can be used anywhere a single statement is allowed.

Exercises

Identify the following kinds of expression statements:

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

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

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

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 >= 70` and `76 >= 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`, the `switch` statement allows for any 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 Classes and Inheritance) and a few special classes that "wrap" certain primitive types: `Character`, `Byte`, `Short`, and `Integer` (discussed in Simple Data Objects).

The following program, `SwitchDemo`, declares an `int` named `month` whose value represents a month out of the year. The program displays the name of the month, based on the value of month, using the `switch` statement.

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

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

In this case, "August" is printed to standard output.

The body of a `switch` statement is known as a switch block. Any statement immediately contained by the `switch` block may be labeled with one or more `case` or `default` labels. The `switch` statement evaluates its expression and executes the appropriate `case`.

Of course, you could also implement the same thing 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 sometimes a judgment call. You can decide which one to use based on readability and other factors. An `if-then-else` statement can be used to make decisions based on ranges of values or conditions, whereas a `switch` statement can make decisions based only on a single integer or enumerated value.

Another point of interest is the `break` statement after each `case`. 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, `case` statements fall through; that is, without an explicit `break`, control will flow sequentially through subsequent `case` statements. The following program, `SwitchDemo2`, illustrates why it might be useful to have `case` statements fall through:

```
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 program.

```
    Number of Days = 29
```

Technically, the final `break` is not required because flow would fall out of the `switch` statement anyway. However, we recommend using a `break` so that modifying the code is easier and less error-prone. The `default` section handles all values that aren't explicitly handled by one of the `case` sections.

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:

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

     // 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
```

We recommend using this 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 print statement at the end of the program. 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 occurences 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;
```

The Classes and Objects lesson will cover everything you need to know about writing methods.

Summary of Control Flow Statements

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.

Questions and Exercises: Control Flow Statements

Questions

1. The most basic control flow statement supported by the Java programming language is the ___ statement.
2. The ___ statement allows for any number of possible execution paths.
3. The ___ statement is similar to the while statement, but evaluates its expression at the ___ of the loop.
4. How do you write an infinite loop using the for statement?
5. How do you write an infinite loop using the while statement?

Exercises

1. Consider the following code snippet.
2. if (aNumber >= 0)
3. if (aNumber == 0) System.out.println("first string");
4. else System.out.println("second string");
5. System.out.println("third string");
    a. What output do you think the code will produce if aNumber is 3?

b. Write a test program containing the previous code snippet; make aNumber 3. What is the output of the program? Is it what you predicted? Explain why the output is what it is; in other words, what is the control flow for the code snippet?
c. Using only spaces and line breaks, reformat the code snippet to make the control flow easier to understand.
d. Use braces, { and }, to further clarify the code.

Answers to Questions

1. The most basic control flow statement supported by the Java programming language is the **if-then** statement.
2. The **switch** statement allows for any number of possible execution paths.
3. The **do-while** statement is similar to the while statement, but evaluates its expression at the **bottom** of the loop.
4. **Question:** How do you write an infinite loop using the for statement?

Answer:

```
for ( ; ; ) {

}
```

5. **Question:** How do you write an infinite loop using the while statement?

Answer:

```
while (true) {
}
```

Exercises

1. Consider the following code snippet.
2. if (aNumber >= 0)
3. if (aNumber == 0) System.out.println("first string");
4. else System.out.println("second string");
5. System.out.println("third string");
    a. **Exercise:** What output do you think the code will produce if aNumber is 3?

Solution:

second string

third string

b. **Exercise:** Write a test program containing the previous code snippet; make aNumber 3. What is the output of the program? Is it what you predicted? Explain

why the output is what it is. In other words, what is the control flow for the code snippet?

Solution: NestedIf

second string

third string

3 is greater than or equal to 0, so execution progresses to the second if statement. The second if statement's test fails because 3 is not equal to 0. Thus, the else clause executes (since it's attached to the second if statement). Thus, second string is displayed. The final println is completely outside of any if statement, so it always gets executed, and thus third string is always displayed.

c. **Exercise:** Using only spaces and line breaks, reformat the code snippet to make the control flow easier to understand.

Solution:

```
if (aNumber >= 0)
    if (aNumber == 0)
        System.out.println("first string");
    else
        System.out.println("second string");

System.out.println("third string");
```

d. **Exercise:** Use braces { and } to further clarify the code and reduce the possibility of errors by future maintainers of the code.

Solution:

```
if (aNumber >= 0) {
    if (aNumber == 0) {
        System.out.println("first string");
    } else {
        System.out.println("second string");
    }
}
System.out.println("third string");
```

Lesson: Classes and Objects

With the knowledge you now have of the basics of the Java programming language, you can learn to write your own classes. In this lesson, you will find information about defining your own classes, including declaring member variables, methods, and constructors.

You will learn to use your classes to create objects, and how to use the objects you create. This lesson also covers nesting classes within other classes, enumerations, and annotations.

Classes

This section shows you the anatomy of a class, and how to declare fields, methods, and constructors.

Objects

This section covers creating and using objects. You will learn how to instantiate an object, and, once instantiated, how to use the `dot` operator to access the object's instance variables and methods.

More on Classes

This section covers more aspects of classes that depend on using object references and the `dot` operator that you learned about in the preceding section: returning values from methods, the `this` keyword, class vs. instance members, and access control.

Nested Classes

Static nested classes, inner classes, anonymous inner classes, and local classes are covered.

Enum Types

This section covers enumerations, specialized classes that allow you to define and use sets of constants.

Annotations

Annotations allow you to add information to your program that is not actually part of the program. This section describes three built-in annotations that you should know about.

Classes

The introduction to object-oriented concepts in the lesson titled Object-oriented Programming Concepts used a bicycle class as an example, with racing bikes, mountain bikes, and tandem bikes as subclasses. Here is sample code for a possible implementation of a `Bicycle` class, to give you an overview of a class declaration. Subsequent sections of this lesson will back up and explain class declarations step by step. For the moment, don't concern yourself with the details.

```
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 has 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 has 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 (mountain bikes have seats that can be moved up and down as the terrain demands).

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.

The preceding class declaration is a minimal one—it contains only those components of a class declaration that are required. You can provide more information about the class, such as the name of its superclass, whether it implements any interfaces, and so on, at the start of the class declaration. For example,

```
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. The lesson on interfaces and inheritance will explain how and why you would use the extends and implements keywords in a class declaration. For the moment you do not need to worry about these extra complications.

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

```
public int cadence;
public int gear;
public int speed;
```

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 `Bicycle` are named `cadence`, `gear`, and `speed` 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.

Access Modifiers

The first (left-most) modifier used lets you control what other classes have access to a member field. For the moment, consider only `public` and `private`. Other access modifiers will be discussed later.

- `public` modifier—the field is accessible from all classes.
- `private` modifier—the field is accessible only within its own class.

In the spirit of encapsulation, it is common to make fields private. This means that they can only be directly accessed from the Bicycle class. We still need access to these values, however. This can be done indirectly by adding public methods that obtain the field values for us:

```
public class Bicycle {

        private int cadence;
        private int gear;
        private int speed;

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

        public int getCadence() {
                return cadence;
        }

        public void setCadence(int newValue) {
                cadence = newValue;
        }

        public int getGear() {
```

```
                return gear;
        }

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

        public int getSpeed() {
                return speed;
        }

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

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

}
```

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 Language Basics lesson, Variables—Naming .

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 lists (there are some qualifications to this that will be discussed in the lesson titled "Interfaces and Inheritance").

Suppose that you have a class that can use calligraphy to draw various types of data (strings, integers, and so on) and that contains a method for drawing each data type. It is cumbersome to use a new name for each method—for example, `drawString`, `drawInteger`, `drawFloat`, 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 data drawing class might declare four methods named `draw`, each of which has a different parameter list.

```
public class DataArtist {
        ...
        public void draw(String s) {
                ...
        }
        public void draw(int i) {
                ...
        }
        public void draw(double f) {
                ...
        }
        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, `draw(String s)` and `draw(int 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.

Note: Overloaded methods should be used sparingly, as they can make code much less readable.

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 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 a superclass constructor yourself. The `MountainBike` class at the beginning of this lesson did just that. This will be discussed later, in the lesson on interfaces and inheritance.

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

Note : If another class cannot call a `MyClass` constructor, it cannot directly create `MyClass` objects.

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
}
```

Note: The Java programming language doesn't let you pass methods into methods. But you can pass an object into a method and then invoke the object's methods.

Arbitrary Number of Arguments

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

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

```
public Polygon polygonFrom(Point... corners) {
    int numberOfSides = corners.length;
    double squareOfSide1, lengthOfSide1;
    squareOfSide1 = (corners[1].x - corners[0].x)*(corners[1].x -
corners[0].x)
                    + (corners[1].y - corners[0].y)*(corners[1].y -
corners[0].y) ;
    lengthOfSide1 = Math.sqrt(squareOfSide1);
    // more method body code follows that creates
    // and returns a polygon connecting the Points
```

```
}
```

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

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

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

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

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

or like this

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

or with yet a different number of arguments.

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.

Objects

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

Here's a small program, called `CreateObjectDemo`, that creates three objects: one `Point` object and two `Rectangle` objects. You will need all three source files to compile this program.

```
public class CreateObjectDemo {

    public static void main(String[] args) {

        //Declare and create a point object
        //and two rectangle objects.
        Point originOne = new Point(23, 94);
        Rectangle rectOne = new Rectangle(originOne, 100, 200);
        Rectangle rectTwo = new Rectangle(50, 100);

        //display rectOne's width, height, and area
        System.out.println("Width of rectOne: " +
                rectOne.width);
        System.out.println("Height of rectOne: " +
                rectOne.height);
        System.out.println("Area of rectOne: " + rectOne.getArea());

        //set rectTwo's position
        rectTwo.origin = originOne;

        //display rectTwo's position
        System.out.println("X Position of rectTwo: "
                + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: "
                + rectTwo.origin.y);

        //move rectTwo and display its new position
        rectTwo.move(40, 72);
        System.out.println("X Position of rectTwo: "
                + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: "
                + rectTwo.origin.y);
    }
}
```

This program creates, manipulates, and displays information about various objects. Here's the output:

```
Width of rectOne: 100
Height of rectOne: 200
Area of rectOne: 20000
X Position of rectTwo: 23
Y Position of rectTwo: 94
```

```
X Position of rectTwo: 40
Y Position of rectTwo: 72
```

The following three sections use the above example to describe the life cycle of an object within a program. From them, you will learn how to write code that creates and uses objects in your own programs. You will also learn how the system cleans up after an object when its life has ended.

Creating Objects

As you know, a class provides the blueprint for objects; you create an object from a class. Each of the following statements taken from the `CreateObjectDemo` program creates an object and assigns it to a variable:

```
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;
```

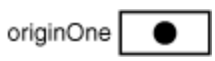
This notifies the compiler that you will use name to refer to data whose type is type. With a primitive variable, this declaration also reserves the proper amount of memory for the variable.

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):



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);
```

The reference returned by the `new` operator does not have to be assigned to a variable. It can also be used directly in an expression. For example:

```
int height = new Rectangle().height;
```

This statement will be discussed in the next section.

Initializing an Object

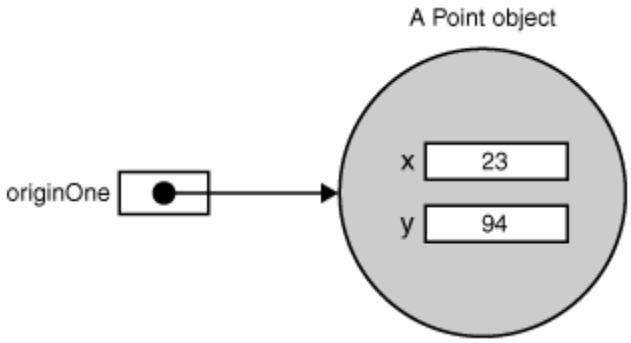
Here's the code for the `Point` class:

```
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:



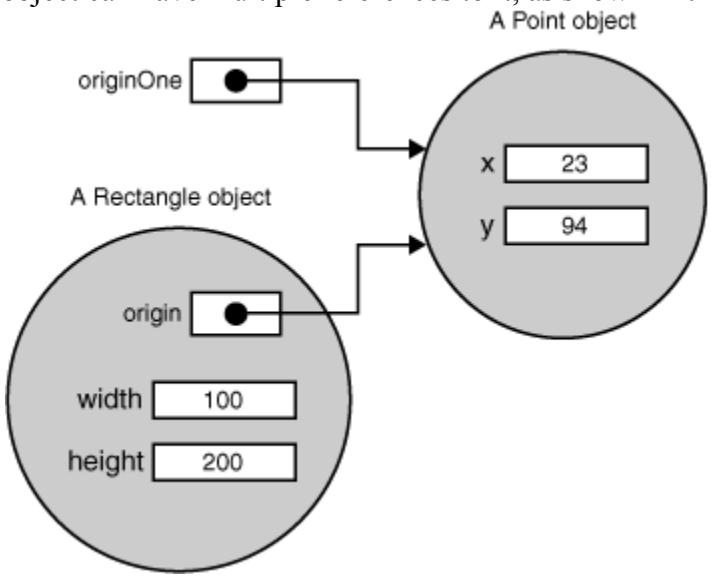
Here's the code for the `Rectangle` class, which contains four constructors:

```
public class Rectangle {
    public int width = 0;
    public int height = 0;
    public Point origin;

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

Each constructor lets you provide initial values for the rectangle's size and width, using both primitive and reference types. If a class has multiple constructors, they must have different signatures. The Java compiler differentiates the constructors based on the number and the type of the arguments. When the Java compiler encounters the following code, it knows to call the constructor in the `Rectangle` class that requires a `Point` argument followed by two integer arguments:

```
Rectangle rectOne = new Rectangle(originOne, 100, 200);
```

This calls one of `Rectangle`'s constructors that initializes `origin` to `originOne`. Also, the constructor sets `width` to 100 and `height` to 200. Now there are two references to the same `Point object`— an object can have multiple references to it, as shown in the next figure:



The following line of code calls the `Rectangle` constructor that requires two integer arguments, which provide the initial values for `width` and `height`. If you inspect the code within the constructor, you will see that it creates a new `Point` object whose `x` and `y` values are initialized to 0:

```
Rectangle rectTwo = new Rectangle(50, 100);
```

The `Rectangle` constructor used in the following statement doesn't take any arguments, so it's called a no-argument constructor:

```
Rectangle rect = new Rectangle();
```

All classes have at least one constructor. If a class does not explicitly declare any, the Java compiler automatically provides a no-argument constructor, called the default constructor. This default constructor calls the class parent's no-argument constructor, or the `Object` constructor if the class has no other parent. If the parent has no constructor (`Object` does have one), the compiler will reject the program.

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);
```

Attempting to use the simple names `width` and `height` from the code in the `CreateObjectDemo` class doesn't make sense — those fields exist only within an object — and results in a compiler error.

Later, the program uses similar code to display information about `rectTwo`. Objects of the same type have their own copy of the same instance fields. Thus, each `Rectangle` object has fields named `origin`, `width`, and `height`. When you access an instance field through an object reference, you reference that particular object's field. The two objects `rectOne` and `rectTwo` in the `CreateObjectDemo` program have different `origin`, `width`, and `height` fields.

To access a field, you can use a named reference to an object, as in the previous examples, or you can use any expression that returns an object reference. Recall that the `new` operator returns a reference to an object. So you could use the value returned from new to access a new object's fields:

```
int height = new Rectangle().height;
```

This statement creates a new `Rectangle` object and immediately gets its height. In essence, the statement calculates the default height of a `Rectangle`. Note that after this statement has been executed, the program no longer has a reference to the created `Rectangle`, because the program never stored the reference anywhere. The object is unreferenced, and its resources are free to be recycled by the Java Virtual Machine.

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.

Some methods, such as `getArea()`, return a value. For methods that return a value, you can use the method invocation in expressions. You can assign the return value to a variable, use it to make decisions, or control a loop. This code assigns the value returned by `getArea()` to the variable `areaOfRectangle`:

```
int areaOfRectangle = new Rectangle(100, 50).getArea();
```

Remember, invoking a method on a particular object is the same as sending a message to that object. In this case, the object that `getArea()` is invoked on is the rectangle returned by the constructor.

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.

An object is eligible for garbage collection when there are no more references to that object. References that are held in a variable are usually dropped when the variable goes out of scope. Or, you can explicitly drop an object reference by setting the variable to the special value `null`. Remember that a program can have multiple references to the same object; all references to an object must be dropped before the object is eligible for 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.

More on Classes

This section covers more aspects of classes that depend on using object references and the dot operator that you learned about in the preceding sections on objects:

- Returning values from methods.
- The this keyword.
- Class vs. instance members.
- Access control.

Returning a Value from a Method

A method returns to the code that invoked it when it

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

whichever occurs first.

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

Any method declared `void` doesn't return a value. It does not need to contain a `return` statement, but it may do so. In such a case, a `return` statement can be used to branch out of a control flow block and exit the method and is simply used like this:

```
return;
```

If you try to return a value from a method that is declared `void`, you will get a compiler error.

Any method that is not declared `void` must contain a `return` statement with a corresponding return value, like this:

```
return returnValue;
```

The data type of the return value must match the method's declared return type; you can't return an integer value from a method declared to return a boolean.

The `getArea()` method in the `Rectangle` Rectangle class that was discussed in the sections on objects returns an integer:

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

This method returns the integer that the expression `width*height` evaluates to.

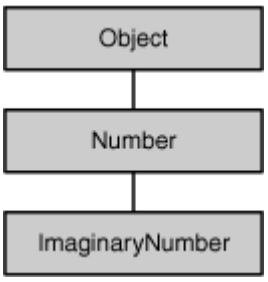
The `area` method returns a primitive type. A method can also return a reference type. For example, in a program to manipulate `Bicycle` objects, we might have a method like this:

```
public Bicycle seeWhosFastest(Bicycle myBike, Bicycle yourBike, Environment
env) {
  Bicycle fastest;
  // code to calculate which bike is faster, given
  // each bike's gear and cadence and given
  // the environment (terrain and wind)
  return fastest;
}
```

Returning a Class or Interface

If this section confuses you, skip it and return to it after you have finished the lesson on interfaces and inheritance.

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



The class hierarchy for ImaginaryNumber

Now suppose that you have a method declared to return a `Number`:

```
public Number returnANumber() {
```

```
    ...
}
```
The `returnANumber` method can return an `ImaginaryNumber` but not an `Object`. `ImaginaryNumber` is a `Number` because it's a subclass of `Number`. However, an `Object` is not necessarily a `Number` — it could be a `String` or another type.

You can override a method and define it to return a subclass of the original method, like this:

```
public ImaginaryNumber returnANumber() {
    ...
}
```
This technique, called covariant return type, means that the return type is allowed to vary in the same direction as the subclass.

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

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, with a different implementation from the one in the Objects section.

```
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.

Controlling Access to Members of a Class

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

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

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

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

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

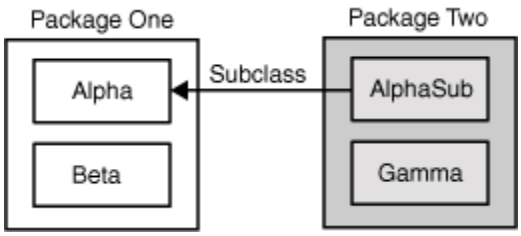
Access Levels

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

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

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

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



Classes and Packages of the Example Used to Illustrate Access Levels

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

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

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

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

Understanding Instance and Class Members

In this section, we discuss the use of the `static` keyword to create fields and methods that belong to the class, rather than to an instance of the class.

Class Variables

When a number of objects are created from the same class blueprint, they each have their own distinct copies of instance variables. In the case of the `Bicycle` class, the instance variables are `cadence`, `gear`, and `speed`. Each `Bicycle` object has its own values for these variables, stored in different memory locations.

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

For example, suppose you want to create a number of `Bicycle` objects and assign each a serial number, beginning with 1 for the first object. This ID number is unique to each object and is therefore an instance variable. At the same time, you need a field to keep track of how many `Bicycle` objects have been created so that you know what ID to assign to the next one. Such a field is not related to any individual object, but to the class as a whole. For this you need a class variable, `numberOfBicycles`, as follows:

```
public class Bicycle{

    private int cadence;
    private int gear;
    private int speed;

    // add an instance variable for the object ID
    private int id;

    // add a class variable for the number of Bicycle objects instantiated
    private static int numberOfBicycles = 0;
        ......
}
```

Class variables are referenced by the class name itself, as in

```
    Bicycle.numberOfBicycles
```

This makes it clear that they are class variables.

Note: You can also refer to static fields with an object reference like

```
    myBike.numberOfBicycles
```

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

You can use the `Bicycle` constructor to set the `id` instance variable and increment the `numberOfBicycles` class variable:

```
public class Bicycle{

    private int cadence;
    private int gear;
    private int speed;
    private int id;
    private static int numberOfBicycles = 0;

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

```
        // increment number of Bicycles and assign ID number
        id = ++numberOfBicycles;
    }

    // new method to return the ID instance variable
    public int getID() {
        return id;
    }
        .....
}
```

Class Methods

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

Note: You can also refer to static methods with an object reference like
`instanceName.methodName(args)`
but this is discouraged because it does not make it clear that they are class methods.

A common use for static methods is to access static fields. For example, we could add a static method to the `Bicycle` class to access the `numberOfBicycles` static field:

```
public static int getNumberOfBicycles() {
    return numberOfBicycles;
}
```

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

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

Constants

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

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

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

Constants defined in this way cannot be reassigned, and it is a compile-time error if your program tries to do so. By convention, the names of constant values are spelled in uppercase

letters. If the name is composed of more than one word, the words are separated by an underscore (_).

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

The `Bicycle` Class

After all the modifications made in this section, the `Bicycle` class is now:

```
public class Bicycle{

    private int cadence;
    private int gear;
    private int speed;

    private int id;

    private static int numberOfBicycles = 0;


    public Bicycle(int startCadence, int startSpeed, int startGear){
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;

        id = ++numberOfBicycles;
    }

    public int getID() {
        return id;
    }

    public static int getNumberOfBicycles() {
        return numberOfBicycles;
    }

    public int getCadence(){
        return cadence;
    }

    public void setCadence(int newValue){
        cadence = newValue;
    }

    public int getGear(){
    return gear;
    }

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

```
    }

    public int getSpeed(){
        return speed;
    }

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

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

}
```

Initializing Fields

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

```
public class BedAndBreakfast {

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

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

This works well when the initialization value is available and the initialization can be put on one line. However, this form of initialization has limitations because of its simplicity. If initialization requires some logic (for example, error handling or a `for` loop to fill a complex array), simple assignment is inadequate. Instance variables can be initialized in constructors, where error handling or other logic can be used. To provide the same capability for class variables, the Java programming language includes static initialization blocks.

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

Static Initialization Blocks

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

```
static {

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

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

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

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

    private static varType initializeClassVariable() {

        //initialization code goes here
    }
}
```

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

Initializing Instance Members

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

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

```
{

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

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

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

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

    protected final varType initializeInstanceVariable() {

        //initialization code goes here
    }
}
```

This is especially useful if subclasses might want to reuse the initialization method. The method is final because calling non-final methods during instance initialization can cause problems. Joshua Bloch describes this in more detail in Effective Java.

Summary of Creating and Using Classes and Objects

A class declaration names the class and encloses the class body between braces. The class name can be preceded by modifiers. The class body contains fields, methods, and constructors for the class. A class uses fields to contain state information and uses methods to implement behavior. Constructors that initialize a new instance of a class use the name of the class and look like methods without a return type.

You control access to classes and members in the same way: by using an access modifier such as `public` in their declaration.

You specify a class variable or a class method by using the `static` keyword in the member's declaration. A member that is not declared as `static` is implicitly an instance member. Class variables are shared by all instances of a class and can be accessed through the class name as well as an instance reference. Instances of a class get their own copy of each instance variable, which must be accessed through an instance reference.

You create an object from a class by using the `new` operator and a constructor. The new operator returns a reference to the object that was created. You can assign the reference to a variable or use it directly.

Instance variables and methods that are accessible to code outside of the class that they are declared in can be referred to by using a qualified name. The qualified name of an instance variable looks like this:

```
objectReference.variableName
```

The qualified name of a method looks like this:

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

or

```
objectReference.methodName()
```

The garbage collector automatically cleans up unused objects. An object is unused if the program holds no more references to it. You can explicitly drop a reference by setting the variable holding the reference to `null`.

Questions and Exercises: Classes

Questions

1. Consider the following class:

```
public class IdentifyMyParts {
    public static int x = 7;
    public int y = 3;
}
```

a. What are the class variables?

b. What are the instance variables?

c. What is the output from the following code:

```
IdentifyMyParts a = new IdentifyMyParts();
IdentifyMyParts b = new IdentifyMyParts();
a.y = 5;
b.y = 6;
a.x = 1;
b.x = 2;
System.out.println("a.y = " + a.y);
System.out.println("b.y = " + b.y);
System.out.println("a.x = " + a.x);
System.out.println("b.x = " + b.x);
System.out.println("IdentifyMyParts.x = " + IdentifyMyParts.x);
```

Exercises

1. Write a class whose instances represent a single playing card from a deck of cards. Playing cards have two distinguishing properties: rank and suit. Be sure to keep your solution as you will be asked to rewrite it in Enum Types.

Hint: You can use the `assert` statement to check your assignments. You write:

```
assert (boolean expression to test);
```
If the boolean expression is false, you will get an error message. For example,

```
assert toString(ACE) == "Ace";
```
should return `true`, so there will be no error message.

If you use the `assert` statement, you must run your program with the `ea` flag:

```
java -ea YourProgram.class
```

---

2. Write a class whose instances represent a **full** deck of cards. You should also keep this solution.

3. Write a small program to test your deck and card classes. The program can be as simple as creating a deck of cards and displaying its cards.

Questions

Question 1: Consider the following class:

```
public class IdentifyMyParts {
    public static int x = 7;
    public int y = 3;
}
```
Question 1a. What are the class variables?

Answer 1a: x

Question 1b. What are the instance variables?
Answer 1b: y

Question 1c. What is the output from the following code:

```
IdentifyMyParts a = new IdentifyMyParts();
IdentifyMyParts b = new IdentifyMyParts();
a.y = 5;
b.y = 6;
a.x = 1;
b.x = 2;
System.out.println("a.y = " + a.y);
System.out.println("b.y = " + b.y);
System.out.println("a.x = " + a.x);
System.out.println("b.x = " + b.x);
System.out.println("IdentifyMyParts.x = " + IdentifyMyParts.x);
```

Answer 1c: Here is the output:

```
a.y = 5
b.y = 6
a.x = 2
b.x = 2
IdentifyMyParts.x = 2
```

Because `x` is defined as a `public static int` in the class `IdentifyMyParts`, every reference to `x` will have the value that was last assigned because `x` is a static variable (and therefore a class variable) shared across all instances of the class. That is, there is only one `x`: when the value of `x` changes in any instance it affects the value of `x` for all instances of `IdentifyMyParts`.

This is covered in the Class Variables section of Understanding Instance and Class Members.

Exercises

Exercise 1: Write a class whose instances represent a single playing card from a deck of cards. Playing cards have two distinguishing properties: rank and suit. Be sure to keep your solution as you will be asked to rewrite it in Enum Types.
Answer 1: Card.java

Exercise 2: Write a class whose instances represents a **full** deck of cards. You should also keep this solution.
Answer 2: See Deck.java.

Exercise 3: Write a small program to test your deck and card classes. The program can be as simple as creating a deck of cards and displaying its cards.
Answer 3: See DisplayDeck.java.

Questions and Exercises: Objects

Questions

What's wrong with the following program?

```
public class SomethingIsWrong {
    public static void main(String[] args) {
      Rectangle myRect;
        myRect.width = 40;
        myRect.height = 50;
        System.out.println("myRect's area is " + myRect.area());
    }
}
```

The following code creates one array and one string object. How many references to those objects exist after the code executes? Is either object eligible for garbage collection?

```
...
String[] students = new String[10];
String studentName = "Peter Parker";
students[0] = studentName;
studentName = null;
...
```

How does a program destroy an object that it creates?

Exercises

1. Fix the program called SomethingIsWrong shown in Question 1.
2. Given the following class, called NumberHolder, write some code that creates an instance of the class, initializes its two member variables, and then displays the value of each member variable.

```
public class NumberHolder {
   public int anInt;
   public float aFloat;
}
```

Questions

Question 1: What's wrong with the following program?

```
public class SomethingIsWrong {
    public static void main(String[] args) {
        Rectangle myRect;
        myRect.width = 40;
        myRect.height = 50;
        System.out.println("myRect's area is " + myRect.area());
    }
}
```

Answer 1: The code never creates a `Rectangle` object. With this simple program, the compiler generates an error. However, in a more realistic situation, `myRect` might be initialized to `null` in

one place, say in a constructor, and used later. In that case, the program will compile just fine, but will generate a `NullPointerException` at runtime.

Question 2: The following code creates one array and one string object. How many references to those objects exist after the code executes? Is either object eligible for garbage collection?

```
...
String[] students = new String[10];
String studentName = "Peter Parker";
students[0] = studentName;
studentName = null;
...
```

Answer 2: There is one reference to the `students` array and that array has one reference to the string `Peter Parker`. Neither object is eligible for garbage collection.

Question 3: How does a program destroy an object that it creates?
Answer 3: A program does not explicitly destroy objects. A program can set all references to an object to `null` so that it becomes eligible for garbage collection. But the program does not actually destroy objects.

Exercises

Exercise 1: Fix the program called `SomethingIsWrong` shown in Question 1.
Answer 1: See `SomethingIsRight`

```
public class SomethingIsRight {
    public static void main(String[] args) {
        Rectangle myRect = new Rectangle();
        myRect.width = 40;
        myRect.height = 50;
        System.out.println("myRect's area is " + myRect.area());
    }
}
```

Exercise 2: Given the following class, called `NumberHolder`, write some code that creates an instance of the class, initializes its two member variables, and then displays the value of each member variable.

```
public class NumberHolder {
    public int anInt;
    public float aFloat;
}
```

Answer 2: See `NumberHolderDisplay`

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

```
        NumberHolder aNumberHolder = new NumberHolder();
        aNumberHolder.anInt = 1;
        aNumberHolder.aFloat = 2.3f;
        System.out.println(aNumberHolder.anInt);
        System.out.println(aNumberHolder.aFloat);
    }
}
```

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:

```
class OuterClass {
    ...
    class NestedClass {
        ...
    }
}
```

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.

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

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.

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

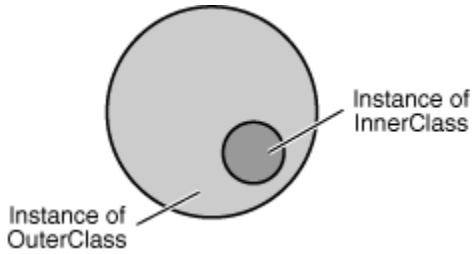
As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields. Also, because an inner class is associated with an instance, it cannot define any static members itself.

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.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.

Note: If you want more information on the taxonomy of the different kinds of classes in the Java programming language (which can be tricky to describe concisely, clearly, and correctly), you might want to read Joseph Darcy's blog: Nested, Inner, Member and Top-Level Classes.

Inner Class Example

To see an inner class in use, let's first consider an array. In the following example, we will create an array, fill it with integer values and then output only values of even indices of the array in ascending order.

The `DataStructure` class below consists of:

- The `DataStructure` outer class, which includes methods to add an integer onto the array and print out values of even indices of the array.
- The `InnerEvenIterator` inner class, which is similar to a standard Java iterator. Iterators are used to step through a data structure and typically have methods to test for the last element, retrieve the current element, and move to the next element.
- A `main` method that instantiates a `DataStructure` object (`ds`) and uses it to fill the `arrayOfInts` array with integer values (0, 1, 2, 3, etc.), then calls a `printEven` method to print out values of even indices of `arrayOfInts`.

```
public class DataStructure {
```

```
    //create an array
    private final static int SIZE = 15;
    private int[] arrayOfInts = new int[SIZE];

    public DataStructure() {
        //fill the array with ascending integer values
        for (int i = 0; i < SIZE; i++) {
            arrayOfInts[i] = i;
        }
    }

    public void printEven() {
        //print out values of even indices of the array
        InnerEvenIterator iterator = this.new InnerEvenIterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.getNext() + " ");
        }
    }

//inner class implements the Iterator pattern
    private class InnerEvenIterator {
        //start stepping through the array from the beginning
        private int next = 0;

        public boolean hasNext() {
            //check if a current element is the last in the array
            return (next <= SIZE - 1);
        }

        public int getNext() {
            //record a value of an even index of the array
            int retValue = arrayOfInts[next];
            //get the next even element
            next += 2;
            return retValue;
        }
    }

    public static void main(String s[]) {
        //fill the array with integer values and print out only values of
even indices
        DataStructure ds = new DataStructure();
        ds.printEven();
    }
}
```

The output is:

```
0 2 4 6 8 10 12 14
```

Note that the `InnerEvenIterator` class refers directly to the `arrayOfInts` instance variable of the `DataStructure` object.

Inner classes can be used to implement helper classes like the one shown in the example above. If you plan on handling user-interface events, you will need to know how to use inner classes because the event-handling mechanism makes extensive use of them.

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. You will encounter such classes in advanced Java programming.

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.

Summary of Nested Classes

A class defined within another class is called a nested class. Like other members of a class, a nested class can be declared static or not. A nonstatic nested class is called an inner class. An instance of an inner class can exist only within an instance of its enclosing class and has access to its enclosing class's members even if they are declared private.

The following table shows the types of nested classes:

Types of Nested Classes		
Type	**Scope**	**Inner**
static nested class	member	no
inner [non-static] class	member	yes
local class	local	yes
anonymous class	only the point where it is defined	yes

Questions and Exercises: Nested Classes

Questions

1. The program `Problem.java` doesn't compile. What do you need to do to make it compile? Why?

2. Use the Java API documentation for the `Box` class (in the `javax.swing` package) to help you answer the following questions.

a. What static nested class does `Box` define?
b. What inner class does `Box` define?
c. What is the superclass of `Box`'s inner class?
d. Which of `Box`'s nested classes can you use from any class?
e. How do you create an instance of `Box`'s `Filler` class?

Exercises

1. Get the file `Class1.java`.

Compile and run `Class1`. What is the output?

Answers to Questions and Exercises: Nested Classes

Questions

Question 1: The program `Problem.java` doesn't compile. What do you need to do to make it compile? Why?
Answer 1: Delete `static` in front of the declaration of the `Inner` class. An static inner class does not have access to the instance fields of the outer class. See `ProblemSolved.java`.

Question 2: Use the Java API documentation for the `Box` class (in the `javax.swing` package) to help you answer the following questions.

a. What static nested class does `Box` define?
Answer 2a: `Box.Filler`

b. What inner class does `Box` define?
Answer 2b: `Box.AccessibleBox`

c. What is the superclass of `Box`'s inner class?
Answer 2c: `[java.awt.]Container.AccessibleAWTContainer`

d. Which of `Box`'s nested classes can you use from any class?
Answer 2d: `Box.Filler`

e. How do you create an instance of `Box`'s `Filler` class?
Answer 2e: `new Box.Filler(minDimension, prefDimension, maxDimension)`

Exercises

Exercise 1: Get the file `Class1.java`.
a. Compile and run `Class1`. What is the output?
Answer:

```
InnerClass1: getString invoked.
InnerClass1: getAnotherString invoked.
```

Enum Types

An enum type is a type whose fields consist of a fixed set of constants. Common examples include compass directions (values of NORTH, SOUTH, EAST, and WEST) and the days of the week.

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

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

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

You should use enum types any time you need to represent a fixed set of constants. That includes natural enum types such as the planets in our solar system and data sets where you know all possible values at compile time—for example, the choices on a menu, command line flags, and so on.

Here is some code that shows you how to use the `Day` enum defined above:

```
public class EnumTest {
        Day day;

        public EnumTest(Day day) {
                this.day = day;
        }

        public void tellItLikeItIs() {
                switch (day) {
                        case MONDAY: System.out.println("Mondays are bad.");
                                        break;

                        case FRIDAY: System.out.println("Fridays are better.");
                                        break;

                        case SATURDAY:
                        case SUNDAY: System.out.println("Weekends are best.");
                                        break;

                        default:        System.out.println("Midweek days are
so-so.");
                                        break;
                }
        }

        public static void main(String[] args) {
                EnumTest firstDay = new EnumTest(Day.MONDAY);
                firstDay.tellItLikeItIs();
                EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);
                thirdDay.tellItLikeItIs();
                EnumTest fifthDay = new EnumTest(Day.FRIDAY);
                fifthDay.tellItLikeItIs();
                EnumTest sixthDay = new EnumTest(Day.SATURDAY);
                sixthDay.tellItLikeItIs();
                EnumTest seventhDay = new EnumTest(Day.SUNDAY);
                seventhDay.tellItLikeItIs();


        }
```

```
}
```

The output is:

```
Mondays are bad.
Midweek days are so-so.
Fridays are better.
Weekends are best.
Weekends are best.
```

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

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

Note: All enums implicitly extend `java.lang.Enum`. Since Java does not support multiple inheritance, an enum cannot extend anything else.

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

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

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

In addition to its properties and constructor, `Planet` has methods that allow you to retrieve the surface gravity and weight of an object on each planet. Here is a sample program that takes your weight on earth (in any unit) and calculates and prints your weight on all of the planets (in the same unit):

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
```

```
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7);

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

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

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

If you run `Planet.class` from the command line with an argument of 175, you get this output:

```
$ java Planet 175
Your weight on MERCURY is 66.107583
Your weight on VENUS is 158.374842
Your weight on EARTH is 175.000000
Your weight on MARS is 66.279007
Your weight on JUPITER is 442.847567
Your weight on SATURN is 186.552719
Your weight on URANUS is 158.397260
Your weight on NEPTUNE is 199.207413
```

Questions and Exercises: Enum Types

Exercises

1. Rewrite the class `Card` from the exercise in Questions and Exercises: Classes so that it represents the rank and suit of a card with enum types.
2. Rewrite the `Deck` class.

Answers to Questions and Exercises: Enum Types

Exercises

Exercise 1: Rewrite the class `Card` from the exercise in Questions and Exercises: Classes so that it represents the rank and suit of a card with enum types.
Answer 1: See `Card3.java`, `Suit.java`, and `Rank.java`.

Exercise 2: Rewrite the `Deck` class.
Answer 2: See `Deck3.java`.

Annotations

Annotations provide data about a program that is not part of the program itself. They have no direct effect on the operation of the code they annotate.

Annotations have a number of uses, among them:

- **Information for the compiler** — Annotations can be used by the compiler to detect errors or suppress warnings.
- **Compiler-time and deployment-time processing** — Software tools can process annotation information to generate code, XML files, and so forth.
- **Runtime processing** — Some annotations are available to be examined at runtime.

Annotations can be applied to a program's declarations of classes, fields, methods, and other program elements.

The annotation appears first, often (by convention) on its own line, and may include elements with named or unnamed values:

```
@Author(
   name = "Benjamin Franklin",
   date = "3/27/2003"
)
class MyClass() { }
```

or

```
@SuppressWarnings(value = "unchecked")
void myMethod() { }
```

If there is just one element named "value," then the name may be omitted, as in:

```
@SuppressWarnings("unchecked")
void myMethod() { }
```

Also, if an annotation has no elements, the parentheses may be omitted, as in:

```
@Override
void mySuperMethod() { }
```

Documentation

Many annotations replace what would otherwise have been comments in code.

Suppose that a software group has traditionally begun the body of every class with comments providing important information:

```
public class Generation3List extends Generation2List {

   // Author: John Doe
   // Date: 3/17/2002
   // Current revision: 6
   // Last modified: 4/12/2004
   // By: Jane Doe
   // Reviewers: Alice, Bill, Cindy

   // class code goes here

}
```

To add this same metadata with an annotation, you must first define the annotation type. The syntax for doing this is:

```
@interface ClassPreamble {
   String author();
   String date();
   int currentRevision() default 1;
   String lastModified() default "N/A";
   String lastModifiedBy() default "N/A";
   String[] reviewers();  // Note use of array
}
```

The annotation type definition looks somewhat like an interface definition where the keyword `interface` is preceded by the @ character (@ = "AT" as in Annotation Type). Annotation types are, in fact, a form of interface, which will be covered in a later lesson. For the moment, you do not need to understand interfaces.

The body of the annotation definition above contains annotation type element declarations, which look a lot like methods. Note that they may define optional default values.

Once the annotation type has been defined, you can use annotations of that type, with the values filled in, like this:

```
@ClassPreamble (
   author = "John Doe",
   date = "3/17/2002",
   currentRevision = 6,
   lastModified = "4/12/2004",
   lastModifiedBy = "Jane Doe",
   reviewers = {"Alice", "Bob", "Cindy"} // Note array notation
)
public class Generation3List extends Generation2List {

// class code goes here

}
```

Note: To make the information in `@ClassPreamble` appear in Javadoc-generated documentation, you must annotate the `@ClassPreamble` definition itself with the `@Documented` annotation:

```
import java.lang.annotation.*; // import this to use @Documented
```

```
@Documented
@interface ClassPreamble {

   // Annotation element definitions

}
```

Annotations Used by the Compiler

There are three annotation types that are predefined by the language specification itself: `@Deprecated`, `@Override`, and `@SuppressWarnings`.

@Deprecated—the `@Deprecated` annotation indicates that the marked element is deprecated and should no longer be used. The compiler generates a warning whenever a program uses a method, class, or field with the `@Deprecated` annotation. When an element is deprecated, it should also be documented using the Javadoc `@deprecated` tag, as shown in the following example. The use of the "@" symbol in both Javadoc comments and in annotations is not coincidental—they are related conceptually. Also, note that the Javadoc tag starts with a lowercase "d" and the annotation starts with an uppercase "D".

```
   // Javadoc comment follows
    /**
     * @deprecated
     * explanation of why it was deprecated
     */
    @Deprecated
    static void deprecatedMethod() { }
}
```

@Override—the `@Override` annotation informs the compiler that the element is meant to override an element declared in a superclass (overriding methods will be discussed in the the lesson titled "Interfaces and Inheritance").

```
   // mark method as a superclass method
   // that has been overridden
   @Override
   int overriddenMethod() { }
```

While it's not required to use this annotation when overriding a method, it helps to prevent errors. If a method marked with `@Override` fails to correctly override a method in one of its superclasses, the compiler generates an error.

@SuppressWarnings—the `@SuppressWarnings` annotation tells the compiler to suppress specific warnings that it would otherwise generate. In the example below, a deprecated method is used and the compiler would normally generate a warning. In this case, however, the annotation causes the warning to be suppressed.

```
   // use a deprecated method and tell
   // compiler not to generate a warning
   @SuppressWarnings("deprecation")
```

```
    void useDeprecatedMethod() {
        objectOne.deprecatedMethod(); //deprecation warning - suppressed
    }
```

Every compiler warning belongs to a category. The Java Language Specification lists two categories: "deprecation" and "unchecked." The "unchecked" warning can occur when interfacing with legacy code written before the advent of generics (discussed in the lesson titled "Generics"). To suppress more than one category of warnings, use the following syntax:

```
@SuppressWarnings({"unchecked", "deprecation"})
```

Annotation Processing

The more advanced uses of annotations include writing an annotation processor that can read a Java program and take actions based on its annotations. It might, for example, generate auxiliary source code, relieving the programmer of having to create boilerplate code that always follows predictable patterns. To facilitate this task, release 5.0 of the JDK includes an annotation processing tool, called `apt`. In release 6 of the JDK, the functionality of `apt` is a standard part of the Java compiler.

To make annotation information available at runtime, the annotation type itself must be annotated with `@Retention(RetentionPolicy.RUNTIME)`, as follows:

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@interface AnnotationForRuntime {

   // Elements that give information
   // for runtime processing

}
```

Questions and Exercises: Annotations

Questions

1. What is wrong with the following interface?

```
public interface House {
    @Deprecated
    void open();
    void openFrontDoor();
    void openBackDoor();
}
```

2. Consider this implementation of the `House` interface, shown in Question 1.

```
public class MyHouse implements House {
    public void open() {}
    public void openFrontDoor() {}
    public void openBackDoor() {}
}
```

If you compile this program, the compiler complains that `open` has been deprecated (in the interface). What can you do to get rid of that warning?

Answers to Questions and Exercises: Annotations

Questions

Question 1: What is wrong with the following interface:

```
public interface House {
    @Deprecated
    public void open();
    public void openFrontDoor();
    public void openBackDoor();
}
```

Answer 1:The documentation should reflect why `open` is deprecated and what to use instead. For example:

```
public interface House {
    /**
     * @deprecated use of open is discouraged, use
     * openFrontDoor or openBackDoor instead.
     */
    @Deprecated
    public void open();
    public void openFrontDoor();
    public void openBackDoor();
}
```

Question 2: Consider this implementation of the `House` interface, shown in Question 1.

```
public class MyHouse implements House {
    public void open() {}
    public void openFrontDoor() {}
    public void openBackDoor() {}
}
```

If you compile this program, the compiler complains that `open` has been deprecated (in the interface). What can you do to get rid of that warning?

Answer 2: You can deprecate the implementation of `open`:

```
public class MyHouse implements House {
    //The documentation is inherited from the interface.
    @Deprecated
    public void open() {}
    public void openFrontDoor() {}
    public void openBackDoor() {}
}
```

Alternatively, you can suppress the warning:

```
public class MyHouse implements House {
    @SuppressWarnings("deprecation")
    public void open() {}
    public void openFrontDoor() {}
    public void openBackDoor() {}
}
```

Lesson: Interfaces and Inheritance

Interfaces

You saw an example of implementing an interface in the previous lesson. You can read more about interfaces here—what they are for, why you might want to write one, and how to write one.

Inheritance

This section describes the way in which you can derive one class from another. That is, how a subclass can inherit fields and methods from a superclass. You will learn that all classes are derived from the `Object` class, and how to modify the methods that a subclass inherits from superclasses. This section also covers interface-like abstract classes.

Interfaces

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

For example, imagine a futuristic society where computer-controlled robotic cars transport passengers through city streets without a human operator. Automobile manufacturers write software (Java, of course) that operates the automobile—stop, start, accelerate, turn left, and so forth. Another industrial group, electronic guidance instrument manufacturers, make computer systems that receive GPS (Global Positioning System) position data and wireless transmission of traffic conditions and use that information to drive the car.

The auto manufacturers must publish an industry-standard interface that spells out in detail what methods can be invoked to make the car move (any car, from any manufacturer). The guidance manufacturers can then write software that invokes the methods described in the interface to command the car. Neither industrial group needs to know how the other group's software is implemented. In fact, each group considers its software highly proprietary and reserves the right to modify it at any time, as long as it continues to adhere to the published interface.

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. Extension is discussed later in this lesson.

Defining an interface is similar to creating a new class:

```
public interface OperateCar {
```

```
    // constant declarations, if any

    // method signatures
    int turn(Direction direction,   // An enum with values RIGHT, LEFT
                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

}
```

In the robotic car example above, it is the automobile manufacturers who will implement the interface. Chevrolet's implementation will be substantially different from that of Toyota, of course, but both manufacturers will adhere to the same interface. The guidance manufacturers, who are the clients of the interface, will build systems that use GPS data on a car's location, digital street maps, and traffic data to drive the car. In so doing, the guidance systems will invoke the interface methods: turn, change lanes, brake, accelerate, and so forth.

Interfaces as APIs

The robotic car example shows an interface being used as an industry standard Application Programming Interface (API). APIs are also common in commercial software products. Typically, a company sells a software package that contains complex methods that another company wants to use in its own software product. An example would be a package of digital image processing methods that are sold to companies making end-user graphics programs. The image processing company writes its classes to implement an interface, which it makes public to its customers. The graphics company then invokes the image processing methods using the signatures and return types defined in the interface. While the image processing company's API is made public (to its customers), its implementation of the API is kept as a closely guarded

secret—in fact, it may revise the implementation at a later date as long as it continues to implement the original interface that its customers have relied on.

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 (inheritance is discussed later in this lesson), 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. This is discussed later in this lesson, in the section titled "Using an Interface as a Type."

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
   double E = 2.718282;  // base of natural logarithms

   // 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 that was presented in the Creating Objects section, 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`.

Using an Interface as a Type

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

As an example, here is a method for finding the largest object in a pair of objects, for any objects that are instantiated from a class that implements `Relatable`:

```
public Object findLargest(Object object1, Object object2) {
   Relatable obj1 = (Relatable)object1;
   Relatable obj2 = (Relatable)object2;
   if ( (obj1).isLargerThan(obj2) > 0)
      return object1;
   else
      return object2;
}
```

By casting `object1` to a `Relatable` type, it can invoke the `isLargerThan` method.

If you make a point of implementing `Relatable` in a wide variety of classes, the objects instantiated from any of those classes can be compared with the `findLargest()` method—provided that both objects are of the same class. Similarly, they can all be compared with the following methods:

```
public Object findSmallest(Object object1, Object object2) {
   Relatable obj1 = (Relatable)object1;
   Relatable obj2 = (Relatable)object2;
   if ( (obj1).isLargerThan(obj2) < 0)
      return object1;
   else
      return object2;
}

public boolean isEqual(Object object1, Object object2) {
   Relatable obj1 = (Relatable)object1;
   Relatable obj2 = (Relatable)object2;
   if ( (obj1).isLargerThan(obj2) == 0)
      return true;
   else
      return false;
}
```

These methods work for any "relatable" objects, no matter what their class inheritance is. When they implement `Relatable`, they can be of both their own class (or superclass) type and a `Relatable` type. This gives them some of the advantages of multiple inheritance, where they can have behavior from both a superclass and an interface.

Rewriting Interfaces

Consider an interface that you have developed called `DoIt`:

```
public interface DoIt {
   void doSomething(int i, double x);
   int doSomethingElse(String s);
}
```

Suppose that, at a later time, you want to add a third method to `DoIt`, so that the interface now becomes:

```
public interface DoIt {

   void doSomething(int i, double x);
   int doSomethingElse(String s);
   boolean didItWork(int i, double x, String s);

}
```

If you make this change, all classes that implement the old `DoIt` interface will break because they don't implement the interface anymore. Programmers relying on this interface will protest loudly.

Try to anticipate all uses for your interface and to specify it completely from the beginning. Given that this is often impossible, you may need to create more interfaces later. For example, you could create a `DoItPlus` interface that extends `DoIt`:

```
public interface DoItPlus extends DoIt {

   boolean didItWork(int i, double x, String s);

}
```

Now users of your code can choose to continue to use the old interface or to upgrade to the new interface.

Summary of 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.

Questions and Exercises: Interfaces

Questions

What methods would a class that implements the java.lang.CharSequence interface have to implement?
What is wrong with the following interface?

```
public interface SomethingIsWrong {
    void aMethod(int aValue){
        System.out.println("Hi Mom");
    }
}
```

Fix the interface in question 2.
Is the following interface valid?
public interface Marker {
}

Exercises

1. Write a class that implements the CharSequence interface found in the java.lang package. Your implementation should return the string backwards. Select one of the sentences from this book to use as the data. Write a small main method to test your class; make sure to call all four methods.
2. Suppose you have written a time server that periodically notifies its clients of the current date and time. Write an interface the server could use to enforce a particular protocol on its clients.

Answers to Questions and Exercises: Interfaces

Questions

Question 1: What methods would a class that implements the `java.lang.CharSequence` interface have to implement?
Answer 1: `charAt`, `length`, `subSequence`, and `toString`.

Question 2: What is wrong with the following interface?

```
public interface SomethingIsWrong {
    void aMethod(int aValue) {
        System.out.println("Hi Mom");
    }
}
```

Answer 2: It has a method implementation in it. It should just have a declaration.

Question 3: Fix the interface in Question 2.
Answer 3:

```
public interface SomethingIsWrong {
    void aMethod(int aValue);
}
```

Question 4: Is the following interface valid?

```
public interface Marker {
}
```

Answer 4: Yes. Methods are not required. Empty interfaces can be used as types and to mark classes without requiring any particular method implementations. For an example of a useful empty interface, see `java.io.Serializable`.

Exercises

Exercise 1: Write a class that implements the `CharSequence` interface found in the `java.lang` package. Your implementation should return the string backwards. Select one of the sentences from this book to use as the data. Write a small `main` method to test your class; make sure to call all four methods.
Answer 1: See `CharSequenceDemo.java`

Exercise 2: Suppose that you have written a time server, which periodically notifies its clients of the current date and time. Write an interface that the server could use to enforce a particular protocol on its clients.
Answer 2: See `TimeClient.java`

Inheritance

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`.

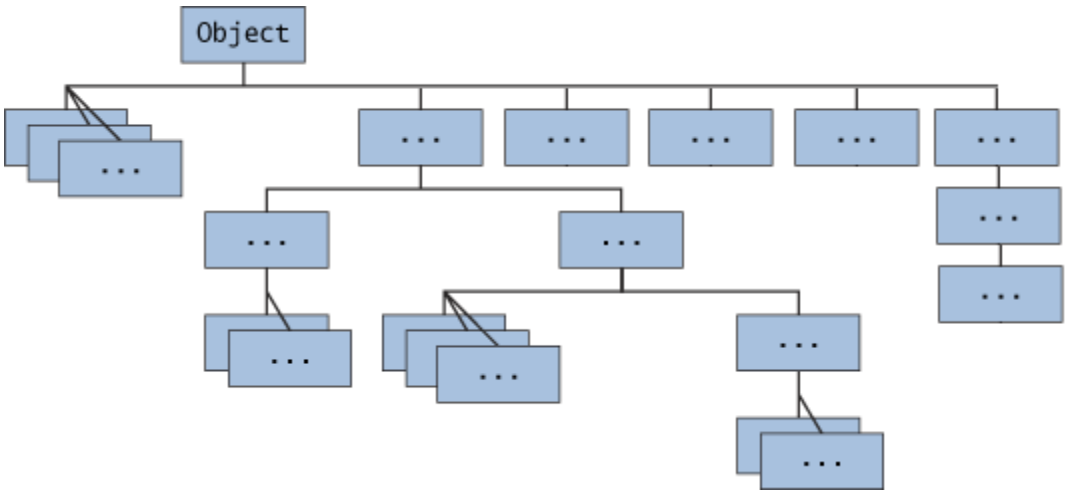
Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, `Object`. Such a class is said to be descended from all the classes in the inheritance chain stretching back to `Object`.

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 behavior.

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`.

The following sections in this lesson will expand on these topics.

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{
  private int tireWidth; // In millimeters (mm)

  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, we don'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 {

    public void printMethod() { //overrides printMethod in Superclass
        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:

```
Book firstBook  = new Book("0201914670"); //Swing Tutorial, 2nd edition
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 JFC Swing Tutorial; A Guide to Constructing GUIs, 2nd
Edition
```

Writing Final Classes and Methods

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

You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object. 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 — this prevents the class from being subclassed. This is particularly useful, for example, when creating an immutable class like the `String` class.

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

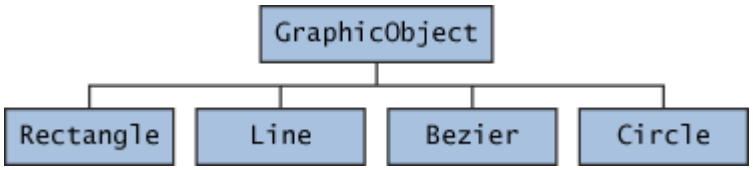
Unlike interfaces, abstract classes can contain fields that are not `static` and `final`, and they can contain implemented methods. Such abstract classes are similar to interfaces, except that they provide a partial implementation, leaving it to subclasses to complete the implementation. If an abstract 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 `GraphicObject`s 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.

Summary of 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.

Questions and Exercises: Inheritance

Questions

1. Consider the following two classes:

```
public class ClassA {
    public void methodOne(int i) {
    }
    public void methodTwo(int i) {
    }
    public static void methodThree(int i) {
    }
    public static void methodFour(int i) {
    }
}

public class ClassB extends ClassA {
    public static void methodOne(int i) {
    }
    public void methodTwo(int i) {
    }
    public void methodThree(int i) {
    }
    public static void methodFour(int i) {
    }
}
```

a. Which method overrides a method in the superclass?
b. Which method hides a method in the superclass?
c. What do the other methods do?

2. Consider the `Card`, `Deck`, and `DisplayDeck` classes you wrote in `Questions and Exercises: Classes` . What `Object` methods should each of these classes override?

Exercises

1. Write the implementations for the methods that you answered in question 2.

Answers to Questions and Exercises: Inheritance

Questions

Question 1: Consider the following two classes:

```
public class ClassA {
    public void methodOne(int i) {
    }
    public void methodTwo(int i) {
    }
    public static void methodThree(int i) {
    }
    public static void methodFour(int i) {
    }
}
```

```
public class ClassB extends ClassA {
    public static void methodOne(int i) {
    }
    public void methodTwo(int i) {
    }
    public void methodThree(int i) {
    }
    public static void methodFour(int i) {
    }
}
```

Question 1a: Which method overrides a method in the superclass?
Answer 1a: `methodTwo`

Question 1b: Which method hides a method in the superclass?
Answer 1b: `methodFour`

Question 1c: What do the other methods do?
Answer 1c: They cause compile-time errors.

Question 2: Consider the `Card`, `Deck`, and `DisplayDeck` classes you wrote in the previous exercise. What `Object` methods should each of these classes override?
Answer 2: `Card` and `Deck` should override `equals`, `hashCode`, and `toString`.

Exercises

Exercise 1: Write the implementations for the methods that you answered in question 2.
Answer 1: See `Card2`.

Lesson: Numbers and Strings

Numbers

This section begins with a discussion of the `Number` class (in the `java.lang` package) and its subclasses. In particular, this section talks about the situations where you would use instantiations of these classes rather than the primitive data types. Additionally, this section talks about other classes you might need to work with numbers, such as formatting or using mathematical functions to complement the operators built into the language.

Strings

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects. This section describes using the `String` class to create and manipulate strings. It also compares the `String` and `StringBuilder` classes.

Numbers

This section begins with a discussion of the `Number` class in the `java.lang` package, its subclasses, and the situations where you would use instantiations of these classes rather than the primitive number types.

This section also presents the `PrintStream` and `DecimalFormat` classes, which provide methods for writing formatted numerical output.

Finally, the `Math` class in `java.lang` is discussed. It contains mathematical functions to complement the operators built into the language. This class has methods for the trigonometric functions, exponential functions, and so forth.

The Numbers Classes

When working with numbers, most of the time you use the primitive types in your code. For example:

```
int i = 500;
float gpa = 3.65f;
byte mask = 0xff;
```

There are, however, reasons to use objects in place of primitives, and the Java platform provides wrapper classes for each of the primitive data types. These classes "wrap" the primitive in an object. Often, the wrapping is done by the compiler—if you use a primitive where an object is expected, the compiler boxes the primitive in its wrapper class for you. Similarly, if you use a number object when a primitive is expected, the compiler unboxes the object for you.

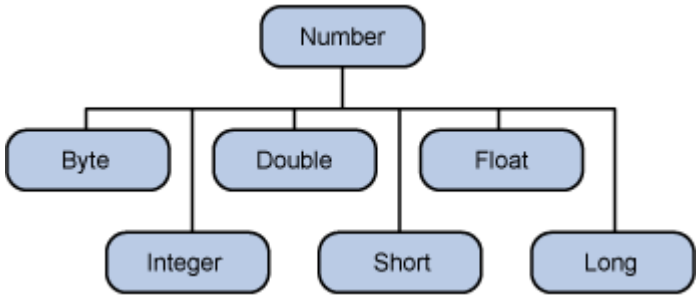
Here is an example of boxing and unboxing:

```
Integer x, y;
x = 12;
y = 15;
```

```
System.out.println(x+y);
```

When `x` and `y` are assigned integer values, the compiler boxes the integers because `x` and `y` are integer objects. In the `println()` statement, `x` and `y` are unboxed so that they can be added as integers.

All of the numeric wrapper classes are subclasses of the abstract class `Number`:



Note: There are four other subclasses of `Number` that are not discussed here. `BigDecimal` and `BigInteger` are used for high-precision calculations. `AtomicInteger` and `AtomicLong` are used for multi-threaded applications.

There are three reasons that you might use a `Number` object rather than a primitive:

1. As an argument of a method that expects an object (often used when manipulating collections of numbers).
2. To use constants defined by the class, such as `MIN_VALUE` and `MAX_VALUE`, that provide the upper and lower bounds of the data type.
3. To use class methods for converting values to and from other primitive types, for converting to and from strings, and for converting between number systems (decimal, octal, hexadecimal, binary).

The following table lists the instance methods that all the subclasses of the `Number` class implement.

Methods Implemented by all Subclasses of Number

Method	Description

`byte byteValue()`
`short shortValue()`
`int intValue()`
`long longValue()`
`float floatValue()`
`double doubleValue()`
	Converts the value of this `Number` object to the primitive data type returned.

`int compareTo(Byte anotherByte)`
`int compareTo(Double anotherDouble)`
`int compareTo(Float anotherFloat)`
`int compareTo(Integer anotherInteger)`
`int compareTo(Long anotherLong)`
`int compareTo(Short anotherShort)`
	Compares this `Number` object to the argument.

`boolean equals(Object obj)`	Determines whether this number object is equal to the argument. The methods return `true` if the argument is not `null` and is an object of the same type and with the same numeric value. There are some extra requirements for `Double` and `Float` objects that are described in the Java API documentation.

Each `Number` class contains other methods that are useful for converting numbers to and from strings and for converting between number systems. The following table lists these methods in the `Integer` class. Methods for the other `Number` subclasses are similar:

Conversion Methods, `Integer` Class

Method	Description
`static Integer decode(String s)`	Decodes a string into an integer. Can accept string representations of decimal, octal, or hexadecimal numbers as input.
`static int parseInt(String s)`	Returns an integer (decimal only).
`static int parseInt(String s, int radix)`	Returns an integer, given a string representation of decimal, binary, octal, or hexadecimal (`radix` equals 10, 2, 8, or 16 respectively) numbers as input.
`String toString()`	Returns a `String` object representing the value of this `Integer`.
`static String toString(int i)`	Returns a `String` object representing the specified integer.
`static Integer valueOf(int i)`	Returns an `Integer` object holding the value of the specified primitive.
`static Integer valueOf(String s)`	Returns an `Integer` object holding the value of the specified string representation.
`static Integer valueOf(String s, int radix)`	Returns an `Integer` object holding the integer value of the specified string representation, parsed with the value of radix. For example, if s = "333" and radix = 8, the method returns the base-ten integer equivalent of the octal number 333.

Formatting Numeric Print Output

Earlier you saw the use of the `print` and `println` methods for printing strings to standard output (`System.out`). Since all numbers can be converted to strings (as you will see later in this lesson), you can use these methods to print out an arbitrary mixture of strings and numbers. The Java programming language has other methods, however, that allow you to exercise much more control over your print output when numbers are included.

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 `printf` and `format` methods are overloaded. Each has a version with the following syntax:

```
public PrintStream format(Locale l, String format, Object... args)
```

To print numbers in the French system (where a comma is used in place of the decimal place in the English representation of floating point numbers), for example, you would use:

```
System.out.format(Locale.FRANCE, "The value of the float variable is %f,
while the value of the " +
                  "integer variable is %d, and the string is %s%n",
floatVar, intVar, stringVar);
```

An Example

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("%tl:%tM %tp%n", c, c, c);   //  -->  "2:34 am"
      System.out.format("%tD%n", c);                 //  -->  "05/29/06"
    }
}
```

Note: The discussion in this section covers just the basics of the `format` and `printf` methods. Further detail can be found in the `Basic I/O` section titled "Formatting". Using `String.format` to create strings is covered in Strings.

The DecimalFormat Class

You can use the `java.text.DecimalFormat` class to control the display of leading and trailing zeros, prefixes and suffixes, grouping (thousands) separators, and the decimal separator. `DecimalFormat` offers a great deal of flexibility in the formatting of numbers, but it can make your code more complex.

The example that follows creates a `DecimalFormat` object, `myFormatter`, by passing a pattern string to the `DecimalFormat` constructor. The `format()` method, which `DecimalFormat` inherits from `NumberFormat`, is then invoked by `myFormatter`—it accepts a `double` value as an argument and returns the formatted number in a string:

Here is a sample program that illustrates the use of `DecimalFormat`:

```
import java.text.*;

public class DecimalFormatDemo {

   static public void customFormat(String pattern, double value ) {
      DecimalFormat myFormatter = new DecimalFormat(pattern);
      String output = myFormatter.format(value);
      System.out.println(value + "  " + pattern + "  " + output);
   }

   static public void main(String[] args) {

      customFormat("###,###.###", 123456.789);
      customFormat("###.##", 123456.789);
      customFormat("000000.000", 123.78);
      customFormat("$###,###.###", 12345.67);
   }
}
```

The output is:

```
123456.789  ###,###.###  123,456.789
123456.789  ###.##  123456.79
123.78  000000.000  000123.780
12345.67  $###,###.###  $12,345.67
```

The following table explains each line of output.

`DecimalFormat.java` Output

Value	Pattern	Output	Explanation
123456.789	###,###.###	123,456.789	The pound sign (#) denotes a digit, the comma is a placeholder for the grouping separator, and the period is a placeholder for the decimal separator.
123456.789	###.##	123456.79	The `value` has three digits to the right of the decimal point, but the `pattern` has only two. The `format` method handles this by rounding up.
123.78	000000.000	000123.780	The `pattern` specifies leading and trailing zeros, because the 0 character is used instead of the pound sign (#).
12345.67	$###,###.###	$12,345.67	The first character in the `pattern` is the dollar sign ($). Note that it immediately precedes the leftmost digit in the formatted `output`.

Beyond Basic Arithmetic

The Java programming language supports basic arithmetic with its arithmetic operators: +, -, *, /, and %. The `Math` class in the `java.lang` package provides methods and constants for doing more advanced mathematical computation.

The methods in the `Math` class are all static, so you call them directly from the class, like this:

```
Math.cos(angle);
```

Note : Using the `static import` language feature, you don't have to write `Math` in front of every math function:

```
import static java.lang.Math.*;
```
This allows you to invoke the `Math` class methods by their simple names. For example:

```
cos(angle);
```

Constants and Basic Methods

The `Math` class includes two constants:

- `Math.E`, which is the base of natural logarithms, and
- `Math.PI`, which is the ratio of the circumference of a circle to its diameter.

The `Math` class also includes more than 40 static methods. The following table lists a number of the basic methods.

Basic Math Methods

Method	Description

`double abs(double d)`
`float abs(float f)`
`int abs(int i)`
`long abs(long lng)`
	Returns the absolute value of the argument.

`double ceil(double d)`	Returns the smallest integer that is greater than or equal to the argument. Returned as a double.
`double floor(double d)`	Returns the largest integer that is less than or equal to the argument. Returned as a double.
`double rint(double d)`	Returns the integer that is closest in value to the argument. Returned as a double.
`long round(double d)` `int round(float f)`	Returns the closest long or int, as indicated by the method's return type, to the argument.

`double min(double arg1, double arg2)`
`float min(float arg1, float arg2)`
`int min(int arg1, int arg2)`
`long min(long arg1, long arg2)`
	Returns the smaller of the two arguments.

`double max(double arg1, double arg2)`
`float max(float arg1, float arg2)`
`int max(int arg1, int arg2)`
`long max(long arg1, long arg2)`
	Returns the larger of the two arguments.

The following program, `BasicMathDemo`, illustrates how to use some of these methods:

```
public class BasicMathDemo {
    public static void main(String[] args) {
        double a = -191.635;
        double b = 43.74;
        int c = 16, d = 45;

        System.out.printf("The absolute value of %.3f is %.3f%n", a,
Math.abs(a));
        System.out.printf("The ceiling of %.2f is %.0f%n", b, Math.ceil(b));
        System.out.printf("The floor of %.2f is %.0f%n", b, Math.floor(b));
        System.out.printf("The rint of %.2f is %.0f%n", b, Math.rint(b));
        System.out.printf("The max of %d and %d is %d%n",c, d, Math.max(c,
d));
        System.out.printf("The min of of %d and %d is %d%n",c, d, Math.min(c,
d));


    }
}
```

Here's the output from this program:

```
The absolute value of -191.635 is 191.635
The ceiling of 43.74 is 44
The floor of 43.74 is 43
The rint of 43.74 is 44
The max of 16 and 45 is 45
The min of 16 and 45 is 16
```

Exponential and Logarithmic Methods

The next table lists exponential and logarithmic methods of the `Math` class.

Exponential and Logarithmic Methods

Method	Description
`double exp(double d)`	Returns the base of the natural logarithms, e, to the power of the argument.
`double log(double d)`	Returns the natural logarithm of the argument.
`double pow(double base, double exponent)`	Returns the value of the first argument raised to the power of the second argument.

`double sqrt(double d)`	Returns the square root of the argument.

The following program, `ExponentialDemo`, displays the value of `e`, then calls each of the methods listed in the previous table on arbitrarily chosen numbers:

```
public class ExponentialDemo {
    public static void main(String[] args) {
        double x = 11.635;
        double y = 2.76;

        System.out.printf("The value of e is %.4f%n", Math.E);
        System.out.printf("exp(%.3f) is %.3f%n", x, Math.exp(x));
        System.out.printf("log(%.3f) is %.3f%n", x, Math.log(x));
        System.out.printf("pow(%.3f, %.3f) is %.3f%n", x, y, Math.pow(x, y));
        System.out.printf("sqrt(%.3f) is %.3f%n", x, Math.sqrt(x));
    }
}
```

Here's the output you'll see when you run `ExponentialDemo`:

```
The value of e is 2.7183
exp(11.635) is 112983.831
log(11.635) is 2.454
pow(11.635, 2.760) is 874.008
sqrt(11.635) is 3.411
```

Trigonometric Methods

The `Math` class also provides a collection of trigonometric functions, which are summarized in the following table. The value passed into each of these methods is an angle expressed in radians. You can use the `toRadians` method to convert from degrees to radians.

Trigonometric Methods

Method	Description
`double sin(double d)`	Returns the sine of the specified double value.
`double cos(double d)`	Returns the cosine of the specified double value.
`double tan(double d)`	Returns the tangent of the specified double value.
`double asin(double d)`	Returns the arcsine of the specified double value.

`double acos(double d)`	Returns the arccosine of the specified double value.
`double atan(double d)`	Returns the arctangent of the specified double value.
`double atan2(double y, double x)`	Converts rectangular coordinates `(x, y)` to polar coordinate `(r, theta)` and returns `theta`.
`double toDegrees(double d)` `double toRadians(double d)`	Converts the argument to degrees or radians.

Here's a program, `TrigonometricDemo`, that uses each of these methods to compute various trigonometric values for a 45-degree angle:

```
public class TrigonometricDemo {
    public static void main(String[] args) {
        double degrees = 45.0;
        double radians = Math.toRadians(degrees);

        System.out.format("The value of pi is %.4f%n", Math.PI);
        System.out.format("The sine of %.1f degrees is %.4f%n", degrees,
                                                  Math.sin(radians));
        System.out.format("The cosine of %.1f degrees is %.4f%n", degrees,
                           Math.cos(radians));
        System.out.format("The tangent of %.1f degrees is %.4f%n", degrees,
                           Math.tan(radians));
        System.out.format("The arcsine of %.4f is %.4f degrees %n",
                          Math.sin(radians),
                          Math.toDegrees(Math.asin(Math.sin(radians))));
        System.out.format("The arccosine of %.4f is %.4f degrees %n",
                          Math.cos(radians),
                          Math.toDegrees(Math.acos(Math.cos(radians))));
        System.out.format("The arctangent of %.4f is %.4f degrees %n",
                          Math.tan(radians),
                          Math.toDegrees(Math.atan(Math.tan(radians))));

    }
}
```

The output of this program is as follows:

```
The value of pi is 3.1416
The sine of 45.0 degrees is 0.7071
The cosine of 45.0 degrees is 0.7071
The tangent of 45.0 degrees is 1.0000
The arcsine of 0.7071 is 45.0000 degrees
The arccosine of 0.7071 is 45.0000 degrees
The arctangent of 1.0000 is 45.0000 degrees
```

Random Numbers

The `random()` method returns a pseudo-randomly selected number between 0.0 and 1.0. The range includes 0.0 but not 1.0. In other words: `0.0 <= Math.random() < 1.0`. To get a number in a different range, you can perform arithmetic on the value returned by the random method. For example, to generate an integer between 0 and 9, you would write:

```
int number = (int)(Math.random() * 10);
```

By multiplying the value by 10, the range of possible values becomes `0.0 <= number < 10.0`.

Using `Math.random` works well when you need to generate a single random number. If you need to generate a series of random numbers, you should create an instance of `java.util.Random` and invoke methods on that object to generate numbers.

Summary of Numbers

You use one of the wrapper classes – `Byte`, `Double`, `Float`, `Integer`, `Long`, or `Short` – to wrap a number of primitive type in an object. The Java compiler automatically wraps (boxes) primitives for you when necessary and unboxes them, again when necessary.

The `Number` classes include constants and useful class methods. The `MIN_VALUE` and `MAX_VALUE` constants contain the smallest and largest values that can be contained by an object of that type. The `byteValue`, `shortValue`, and similar methods convert one numeric type to another. The `valueOf` method converts a string to a number, and the `toString` method converts a number to a string.

To format a string containing numbers for output, you can use the `printf()` or `format()` methods in the `PrintStream` class. Alternatively, you can use the `NumberFormat` class to customize numerical formats using patterns.

The `Math` class contains a variety of class methods for performing mathematical functions, including exponential, logarithmic, and trigonometric methods. `Math` also includes basic arithmetic functions, such as absolute value and rounding, and a method, `random()`, for generating random numbers.

Questions and Exercises: Numbers

Questions

1. Use the API documentation to find the answers to the following questions:
a. What `Integer` method can you use to convert an `int` into a string that expresses the number in hexadecimal? For example, what method converts the integer 65 into the string "41"?

b. What `Integer` method would you use to convert a string expressed in base 5 into the equivalent `int`? For example, how would you convert the string "230" into the integer value 65? Show the code you would use to accomplish this task.

c. What Double method can you use to detect whether a floating-point number has the special value Not a Number (`NaN`)?

2. What is the value of the following expression, and why?
```
Integer.valueOf(1).equals(Long.valueOf(1))
```

Exercises

1. Change `MaxVariablesDemo` to show minimum values instead of maximum values. You can delete all code related to the variables `aChar` and `aBoolean`. What is the output?

2. Create a program that reads an unspecified number of integer arguments from the command line and adds them together. For example, suppose that you enter the following:

```
java Adder 1 3 2 10
```

The program should display `16` and then exit. The program should display an error message if the user enters only one argument. You can base your program on `ValueOfDemo`.

3. Create a program that is similar to the previous one but has the following differences:

- Instead of reading integer arguments, it reads floating-point arguments.
- It displays the sum of the arguments, using exactly two digits to the right of the decimal point.

For example, suppose that you enter the following:
```
java FPAdder 1 1e2 3.0 4.754
```
The program would display `108.75`. Depending on your locale, the decimal point might be a comma (`,`) instead of a period (`.`).

Answers to Questions and Exercises: Numbers

Questions

Question 1: Use the API documentation to find the answers to the following questions:
Question 1a: What `Integer` method can you use to convert an `int` into a string that expresses the number in hexadecimal? For example, what method converts the integer 65 into the string "41"?
Answer 1a: `toHexString`

Question 1b: What `Integer` method would you use to convert a string expressed in base 5 into the equivalent `int`? For example, how would you convert the string "230" into the integer value 65? Show the code you would use to accomplish this task.
Answer 1b: `valueOf`. Here's how:

```
String base5String = "230";
int result = Integer.valueOf(base5String, 5);
```

Question 1c: What Double method can you use to detect whether a floating-point number has the special value Not a Number (`NaN`)?
Answer 1c: `isNaN`

Question 2: What is the value of the following expression, and why?
```
Integer.valueOf(1).equals(Long.valueOf(1))
```
Answer 2: False. The two objects (the `Integer` and the `Long`) have different types.

Exercises

Exercise 1: Change `MaxVariablesDemo` to show minimum values instead of maximum values. You can delete all code related to the variables `aChar` and `aBoolean`. What is the output?

Answer 1: See `MinVariablesDemo`. Here is the output:
```
The smallest byte value is -128
The smallest short value is -32768
The smallest integer value is -2147483648
The smallest long value is -9223372036854775808
The smallest float value is 1.4E-45
The smallest double value is 4.9E-324
```

Exercise 2: Create a program that reads an unspecified number of integer arguments from the command line and adds them together. For example, suppose that you enter the following:

```
java Adder 1 3 2 10
```

The program should display `16` and then exit. The program should display an error message if the user enters only one argument. You can base your program on `ValueOfDemo`.
Answer 2: See `Adder`.

Exercise 3: Create a program that is similar to the previous one but has the following differences:

- Instead of reading integer arguments, it reads floating-point arguments.
- It displays the sum of the arguments, using exactly two digits to the right of the decimal point.

For example, suppose that you enter the following:
```
java FPAdder 1 1e2 3.0 4.754
```
The program would display `108.75`. Depending on your locale, the decimal point might be a comma (`,`) instead of a period (`.`).
Answer 3: See `FPAdder`.

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';
char uniChar = '\u039A'; // Unicode for uppercase Greek omega character
```

```
char[] charArray ={ 'a', 'b', 'c', 'd', 'e' }; // an array of chars
```

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,

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

and here is an example of both boxing and unboxing,

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

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

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` API specification.

Useful Methods in the `Character` Class

Method	Description
`boolean isLetter(char ch)` `boolean isDigit(char ch)`	Determines whether the specified char value is a letter or a digit, respectively.
`boolean isWhitespace(char ch)`	Determines whether the specified char value is white space.
`boolean isUpperCase(char ch)` `boolean isLowerCase(char ch)`	Determines whether the specified char value is uppercase or lowercase, respectively.
`char toUpperCase(char ch)` `char toLowerCase(char ch)`	Returns the uppercase or lowercase form of the specified char value.
`toString(char ch)`	Returns a `String` object representing the specified character value—that is, a one-character string.

Escape Sequences

A character preceded by a backslash (\) is an escape sequence and has special meaning to the compiler. The following table shows the Java escape sequences:

Escape Sequences

Escape Sequence	Description
\t	Insert a tab in the text at this point.
\b	Insert a backspace in the text at this point.
\n	Insert a newline in the text at this point.
\r	Insert a carriage return in the text at this point.
\f	Insert a formfeed in the text at this point.
\'	Insert a single quote character in the text at this point.
\"	Insert a double quote character in the text at this point.
\\	Insert a backslash character in the text at this point.

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.");
```

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 i^{th} 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 Between Numbers and Strings

Converting Strings to Numbers

Frequently, a program ends up with numeric data in a string object—a value entered by the user, for example.

The `Number` subclasses that wrap primitive numeric types (`Byte`, `Integer`, `Double`, `Float`, `Long`, and `Short`) each provide a class method named `valueOf` that converts a string to an object of that type. Here is an example, `ValueOfDemo` , that gets two strings from the command line, converts them to numbers, and performs arithmetic operations on the values:

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

            //this program requires two arguments on the command line
        if (args.length == 2) {
                    //convert strings to numbers
            float a = (Float.valueOf(args[0]) ).floatValue();
            float b = (Float.valueOf(args[1]) ).floatValue();

            //do some arithmetic
            System.out.println("a + b = " + (a + b) );
```

```
            System.out.println("a - b = " + (a - b) );
            System.out.println("a * b = " + (a * b) );
            System.out.println("a / b = " + (a / b) );
            System.out.println("a % b = " + (a % b) );
        } else {
           System.out.println("This program requires two command-line
arguments.");
        }
    }
}
```

The following is the output from the program when you use `4.5` and `87.2` for the command-line arguments:

```
a + b = 91.7
a - b = -82.7
a * b = 392.4
a / b = 0.0516055
a % b = 4.5
```

Note: Each of the `Number` subclasses that wrap primitive numeric types also provides a `parseXXXX()` method (for example, `parseFloat()`) that can be used to convert strings to primitive numbers. Since a primitive type is returned instead of an object, the `parseFloat()` method is more direct than the `valueOf()` method. For example, in the `ValueOfDemo` program, we could use:

```
float a = Float.parseFloat(args[0]);
float b = Float.parseFloat(args[1]);
```

Converting Numbers to Strings

Sometimes you need to convert a number to a string because you need to operate on the value in its string form. There are several easy ways to convert a number to a string:

```
int i;
String s1 = "" + i; //Concatenate "i" with an empty string;
                    //conversion is handled for you.
```

or

```
String s2 = String.valueOf(i);  //The valueOf class method.
```

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

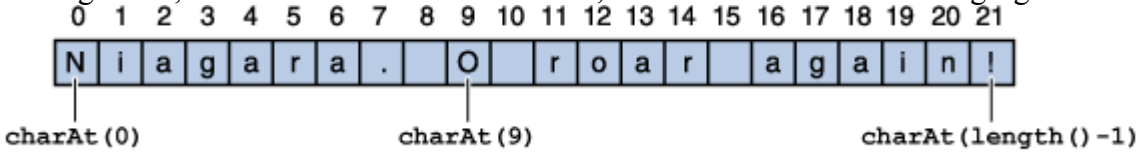
The `String` class has a number of methods for examining the contents of strings, finding characters or substrings within a string, changing case, and other tasks.

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:



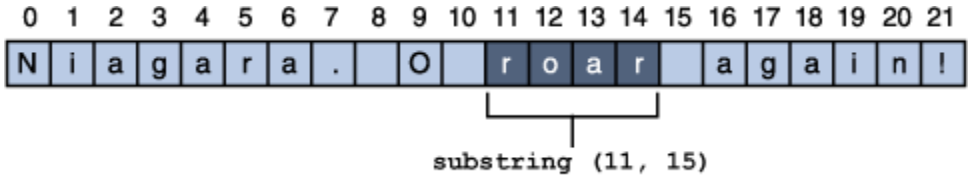
If you want to get more than one consecutive character from a string, you can use the `substring` method. The `substring` method has two versions, as shown in the following table:

The substring Methods in the String Class

Method	Description
`String substring(int beginIndex, int endIndex)`	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.
`String substring(int beginIndex)`	Returns a new string that is a substring of this string. The integer argument specifies the index of the first character. Here, the returned substring extends to the end of the original string.

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)`	Returns the index of the first (last) occurrence of the specified

`int lastIndexOf(int ch)`	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 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);
    }

    public String filename() {  // gets filename without extension
        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:

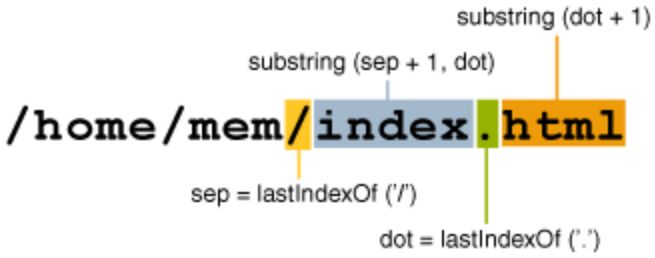
```
public class FilenameDemo {
    public static void main(String[] args) {
        final String FPATH = "/home/mem/index.html";
        Filename myHomePage = new Filename(FPATH,
                                           '/', '.');
        System.out.println("Extension = " +
             myHomePage.extension());
        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)`	Tests whether the specified region of this string matches the specified region of the String argument. Region is of length `len` and begins at the index `toffset` for this string and `ooffset` for the other string. The boolean argument indicates whether case should be ignored; if true, case is ignored when comparing characters.
`boolean matches(String regex)`	Tests whether this string matches the specified regular expression. Regular expressions are discussed in the lesson titled "Regular Expressions."

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.

The StringBuilder Class

`StringBuilder` objects are like `String` objects, except that they can be modified. Internally, these objects are treated like variable-length arrays that contain a sequence of characters. At any point, the length and content of the sequence can be changed through method invocations.

Strings should always be used unless string builders offer an advantage in terms of simpler code (see the sample program at the end of this section) or better performance. For example, if you need to concatenate a large number of strings, appending to a `StringBuilder` object is more efficient.

Length and Capacity

The `StringBuilder` class, like the `String` class, has a `length()` method that returns the length of the character sequence in the builder.

Unlike strings, every string builder also has a capacity, the number of character spaces that have been allocated. The capacity, which is returned by the `capacity()` method, is always greater than or equal to the length (usually greater than) and will automatically expand as necessary to accommodate additions to the string builder.

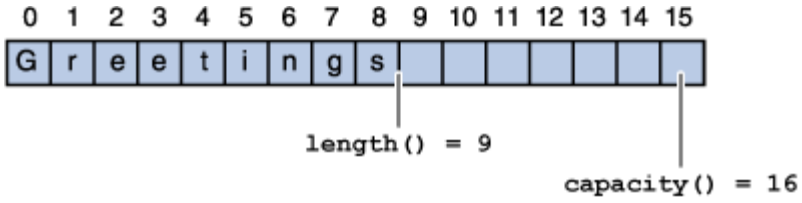
`StringBuilder` Constructors

Constructor	Description
`StringBuilder()`	Creates an empty string builder with a capacity of 16 (16 empty elements).
`StringBuilder(CharSequence cs)`	Constructs a string builder containing the same characters as the specified `CharSequence`, plus an extra 16 empty elements trailing the `CharSequence`.
`StringBuilder(int initCapacity)`	Creates an empty string builder with the specified initial capacity.
`StringBuilder(String s)`	Creates a string builder whose value is initialized by the specified string, plus an extra 16 empty elements trailing the string.

For example, the following code

```
StringBuilder sb = new StringBuilder(); // creates empty builder, capacity 16
```

```
sb.append("Greetings");  // adds 9 character string at beginning
```
will produce a string builder with a length of 9 and a capacity of 16:



The `StringBuilder` class has some methods related to length and capacity that the `String` class does not have:

Length and Capacity Methods

Method	Description
`void setLength(int newLength)`	Sets the length of the character sequence. If `newLength` is less than `length()`, the last characters in the character sequence are truncated. If `newLength` is greater than `length()`, null characters are added at the end of the character sequence.
`void ensureCapacity(int minCapacity)`	Ensures that the capacity is at least equal to the specified minimum.

A number of operations (for example, `append()`, `insert()`, or `setLength()`) can increase the length of the character sequence in the string builder so that the resultant `length()` would be greater than the current `capacity()`. When this happens, the capacity is automatically increased.

StringBuilder Operations

The principal operations on a `StringBuilder` that are not available in `String` are the `append()` and `insert()` methods, which are overloaded so as to accept data of any type. Each converts its argument to a string and then appends or inserts the characters of that string to the character sequence in the string builder. The append method always adds these characters at the end of the existing character sequence, while the insert method adds the characters at a specified point.

Here are a number of the methods of the `StringBuilder` class.

Various `StringBuilder` Methods

Method	Description

`StringBuilder append(boolean b)`
`StringBuilder append(char c)`
`StringBuilder append(char[] str)`
`StringBuilder append(char[] str, int offset, int len)`
`StringBuilder append(double d)`
`StringBuilder append(float f)`
`StringBuilder append(int i)`
`StringBuilder append(long lng)`
`StringBuilder append(Object obj)`
`StringBuilder append(String s)`
	Appends the argument to this string builder. The data is converted to a string before the append operation takes place.

`StringBuilder delete(int start, int end)` `StringBuilder deleteCharAt(int index)`	The first method deletes the subsequence from start to end-1 (inclusive) in the `StringBuilder`'s char sequence. The second method deletes the character located at `index`.

`StringBuilder insert(int offset, boolean b)`
`StringBuilder insert(int offset, char c)`
`StringBuilder insert(int offset, char[] str)`
`StringBuilder insert(int index, char[] str, int offset, int len)`
`StringBuilder insert(int offset, double d)`
`StringBuilder insert(int offset, float f)`
`StringBuilder insert(int offset, int i)`
`StringBuilder insert(int offset, long lng)`
`StringBuilder insert(int offset, Object obj)`
`StringBuilder insert(int offset, String s)`
	Inserts the second argument into the string builder. The first integer argument indicates the index before which the data is to be inserted. The data is converted to a string before the insert operation takes place.

`StringBuilder replace(int start, int end, String s)` `void setCharAt(int index, char c)`	Replaces the specified character(s) in this string builder.
`StringBuilder reverse()`	Reverses the sequence of characters in this string builder.
`String toString()`	Returns a string that contains the character sequence in the builder.

Note: You can use any `String` method on a `StringBuilder` object by first converting the string builder to a string with the `toString()` method of the `StringBuilder` class. Then convert the string back into a string builder using the `StringBuilder(String str)` constructor.

An Example

The `StringDemo` program that was listed in the section titled "Strings" is an example of a program that would be more efficient if a `StringBuilder` were used instead of a `String`.

`StringDemo` reversed a palindrome. Here, once again, is its listing:

```
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 converts the string to an array of characters (first `for` loop), reverses the array into a second array (second `for` loop), and then converts back to a string.

If you convert the `palindrome` string to a string builder, you can use the `reverse()` method in the `StringBuilder` class. It makes the code simpler and easier to read:

```
public class StringBuilderDemo {
    public static void main(String[] args) {
        String palindrome = "Dot saw I was Tod";

        StringBuilder sb = new StringBuilder(palindrome);

        sb.reverse();  // reverse it
```

```
        System.out.println(sb);
    }
}
```
Running this program produces the same output:

```
doT saw I was toD
```
Note that `println()` prints a string builder, as in:

```
System.out.println(sb);
```
because `sb.toString()` is called implicitly, as it is with any other object in a `println()` invocation.

Note: There is also a `StringBuffer` class that is exactly the same as the `StringBuilder` class, except that it is thread-safe by virtue of having its methods synchronized. Threads will be discussed in the lesson on concurrency.

Summary of 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.

In addition to the `String` class, there is also a `StringBuilder` class. Working with `StringBuilder` objects can sometimes be more efficient than working with strings. The

`StringBuilder` class offers a few methods that can be useful for strings, among them `reverse()`. In general, however, the `String` class has a wider variety of methods.

A string can be converted to a string builder using a `StringBuilder` constructor. A string builder can be converted to a string with the `toString()` method.

Questions and Exercises: Characters and Strings

Questions

What is the initial capacity of the following string builder?
StringBuilder sb = new StringBuilder("Able was I ere I saw Elba.");

Consider the following string:
String hannah = "Did Hannah see bees? Hannah did.";

What is the value displayed by the expression hannah.length()?
What is the value returned by the method call hannah.charAt(12)?
Write an expression that refers to the letter b in the string referred to by hannah.

How long is the string returned by the following expression? What is the string?
"Was it a car or a cat I saw?".substring(9, 12)

In the following program, called ComputeResult, what is the value of result after each numbered line executes?

```
public class ComputeResult {
    public static void main(String[] args) {
        String original = "software";
        StringBuilder result = new StringBuilder("hi");
        int index = original.indexOf('a');

/*1*/   result.setCharAt(0, original.charAt(0));
/*2*/   result.setCharAt(1, original.charAt(original.length()-1));
/*3*/   result.insert(1, original.charAt(4));
/*4*/   result.append(original.substring(1,4));
/*5*/   result.insert(3, (original.substring(index, index+2) + " "));

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

Exercises

1. Show two ways to concatenate the following two strings together to get the string "Hi, mom.":
2. String hi = "Hi, ";
3. String mom = "mom.";
4. Write a program that computes your initials from your full name and displays them.

5. An anagram is a word or a phrase made by transposing the letters of another word or phrase; for example, "parliament" is an anagram of "partial men," and "software" is an anagram of "swear oft." Write a program that figures out whether one string is an anagram of another string. The program should ignore white space and punctuation.

Answers to Questions and Exercises: Characters and Strings

Questions

Question 1: What is the initial capacity of the following string builder?
```
StringBuilder sb = new StringBuilder("Able was I ere I saw Elba.");
```
Answer 1: It's the length of the initial string + 16: 26 + 16 = 42.

Question 2: Consider the following string:

```
String hannah = "Did Hannah see bees? Hannah did.";
```

Question 2a: What is the value displayed by the expression `hannah.length()`?
Answer 2a: 32.

Question 2b: What is the value returned by the method call `hannah.charAt(12)`?
Answer 2b: `e`.

Question 2c: Write an expression that refers to the letter `b` in the string referred to by `hannah`.
Answer 2c: `hannah.charAt(15)`.

Question 3: How long is the string returned by the following expression? What is the string?
```
"Was it a car or a cat I saw?".substring(9, 12)
```
Answer 3: It's 3 characters in length: `car`. It does not include the space after car.

Question 4: In the following program, called `ComputeResult`, what is the value of `result` after each numbered line executes?

```
public class ComputeResult {
    public static void main(String[] args) {
        String original = "software";
        StringBuilder result = new StringBuilder("hi");
        int index = original.indexOf('a');

/*1*/   result.setCharAt(0, original.charAt(0));
/*2*/   result.setCharAt(1, original.charAt(original.length()-1));
/*3*/   result.insert(1, original.charAt(4));
/*4*/   result.append(original.substring(1,4));
/*5*/   result.insert(3, (original.substring(index, index+2) + " "));

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

```
}
```

Answer 4:

1. si
2. se
3. swe
4. sweoft
5. swear oft

Exercises

Exercise 1: Show two ways to concatenate the following two strings together to get the string `"Hi, mom."`:

```
String hi = "Hi, ";
String mom = "mom.";
```

Answer 1: `hi.concat(mom)` and `hi + mom`.

Exercise 2: Write a program that computes your initials from your full name and displays them.

Answer 2: `ComputeInitials`

```
public class ComputeInitials {
    public static void main(String[] args) {
        String myName = "Fred F. Flintstone";
        StringBuffer myInitials = new StringBuffer();
        int length = myName.length();

        for (int i = 0; i < length; i++) {
            if (Character.isUpperCase(myName.charAt(i))) {
                myInitials.append(myName.charAt(i));
            }
        }
        System.out.println("My initials are: " + myInitials);
    }
}
```

Exercise 3: An anagram is a word or a phrase made by transposing the letters of another word or phrase; for example, "parliament" is an anagram of "partial men," and "software" is an anagram of "swear oft." Write a program that figures out whether one string is an anagram of another string. The program should ignore white space and punctuation.

Answer 3: `Anagram`

```
public class Anagram {

    public static boolean areAnagrams(String string1,
```

```
                                    String string2) {

        String workingCopy1 = removeJunk(string1);
        String workingCopy2 = removeJunk(string2);

            workingCopy1 = workingCopy1.toLowerCase();
            workingCopy2 = workingCopy2.toLowerCase();

            workingCopy1 = sort(workingCopy1);
            workingCopy2 = sort(workingCopy2);

        return workingCopy1.equals(workingCopy2);
    }
    protected static String removeJunk(String string) {
        int i, len = string.length();
        StringBuilder dest = new StringBuilder(len);
               char c;

            for (i = (len - 1); i >= 0; i--) {
                c = string.charAt(i);
                if (Character.isLetter(c)) {
                        dest.append(c);
                }
            }

        return dest.toString();
    }

    protected static String sort(String string) {
            char[] charArray = string.toCharArray();

            java.util.Arrays.sort(charArray);

        return new String(charArray);
    }
    public static void main(String[] args) {
        String string1 = "Cosmo and Laine:";
        String string2 = "Maid, clean soon!";

        System.out.println();
        System.out.println("Testing whether the following "
                         + "strings are anagrams:");
        System.out.println("    String 1: " + string1);
        System.out.println("    String 2: " + string2);
        System.out.println();

        if (areAnagrams(string1, string2)) {
            System.out.println("They ARE anagrams!");
        } else {
            System.out.println("They are NOT anagrams!");
        }
        System.out.println();
    }
}
```

Lesson: Generics

Generics are a built-in language feature that will make your software more reliable. This lesson discusses the following topics:

Introduction

This section explains some common shortcomings associated with non-generic code. Specifically, it shows how certain kinds of bugs will crash an application at runtime, since they are not detectable by the compiler.

Generic Types

This section explains generic type declarations, type variables, type parameters, and type arguments. It also describes the naming conventions that are specific to generics.

Generic Methods and Constructors

This section shows how type parameters can be used to define generic methods and constructors.

Bounded Type Parameters

This section describes how type parameters can specify an upper bound that limits the kind of types that can be passed in.

Subtyping

This section describes how generic subtyping differs from non-generic subtyping.

Wildcards

This section continues the discussion of subtyping by describing bounded and unbounded wildcards.

Type Erasure

This section describes type erasure, raw types, and unchecked warnings.

Introduction

In any nontrivial software project, bugs are simply a fact of life. Careful planning, programming, and testing can help reduce their pervasiveness, but somehow, somewhere, they'll always find a way to creep into your code. This becomes especially apparent as new features are introduced and your code base grows in size and complexity.

Fortunately, some bugs are easier to detect than others. Compile-time bugs, for example, tell you immediately that something is wrong; you can use the compiler's error messages to figure out what the problem is and fix it, right then and there. Runtime bugs, however, can be much more problematic; they don't always surface immediately, and when they do, it may be at a point in time that's far removed from the actual cause of the problem.

Generics add stability to your code by making more of your bugs detectable at compile time. Some programmers choose to learn generics by studying the Java Collections Framework; after all, generics are heavily used by those classes. However, since we haven't yet covered collections, this chapter will focus primarily on simple "collections-like" examples that we'll design from scratch. This hands-on approach will teach you the necessary syntax and terminology while demonstrating the various kinds of problems that generics were designed to solve.

A Simple Box Class

Let's begin by designing a nongeneric `Box` class that operates on objects of any type. It need only provide two methods: `add`, which adds an object to the box, and `get`, which retrieves it:

```
    public class Box {

        private Object object;

        public void add(Object object) {
            this.object = object;
        }

        public Object get() {
            return object;
        }
    }
```

Since its methods accept or return `Object`, you're free to pass in whatever you want, provided that it's not one of the primitive types. However, should you need to restrict the contained type to something specific (like `Integer`), your only option would be to specify the requirement in documentation (or in this case, a comment), which of course the compiler knows nothing about:

```
public class BoxDemo1 {

    public static void main(String[] args) {

        // ONLY place Integer objects into this box!
        Box integerBox = new Box();

        integerBox.add(new Integer(10));
        Integer someInteger = (Integer)integerBox.get();
        System.out.println(someInteger);
    }
}
```

The `BoxDemo1` program creates an `Integer` object, passes it to `add`, then assigns that same object to `someInteger` by the return value of `get`. It then prints the object's value (10) to standard

output. We know that the cast from `Object` to `Integer` is correct because we've honored the "contract" specified in the comment. But remember, the compiler knows nothing about this — it just trusts that our cast is correct. Furthermore, it will do nothing to prevent a careless programmer from passing in an object of the wrong type, such as `String`:

```
public class BoxDemo2 {

    public static void main(String[] args) {

        // ONLY place Integer objects into this box!
        Box integerBox = new Box();

        // Imagine this is one part of a large application
        // modified by one programmer.
        integerBox.add("10"); // note how the type is now String

        // ... and this is another, perhaps written
        // by a different programmer
        Integer someInteger = (Integer)integerBox.get();
        System.out.println(someInteger);
    }
}
```

In `BoxDemo2` we've stored the number 10 as a `String`, which could be the case when, say, a GUI collects input from the user. However, the existing cast from `Object` to `Integer` has mistakenly been overlooked. This is clearly a bug, but because the code still compiles, you wouldn't know anything is wrong until runtime, when the application crashes with a `ClassCastException`:

```
    Exception in thread "main"
         java.lang.ClassCastException:
            java.lang.String cannot be cast to java.lang.Integer
            at BoxDemo2.main(BoxDemo2.java:6)
```

If the `Box` class had been designed with generics in mind, this mistake would have been caught by the compiler, instead of crashing the application at runtime.

Generic Types

Let's update our `Box` class to use generics. We'll first create a generic type declaration by changing the code "`public class Box`" to "`public class Box<T>`"; this introduces one type variable, named `T`, that can be used anywhere inside the class. This same technique can be applied to interfaces as well. There's nothing particularly complex about this concept. In fact, it's quite similar to what you already know about variables in general. Just think of `T` as a special kind of variable, whose "value" will be whatever type you pass in; this can be any class type, any interface type, or even another type variable. It just can't be any of the primitive data types. In this context, we also say that `T` is a formal type parameter of the `Box` class.

```
/**
 * Generic version of the Box class.
 */
public class Box<T> {

    private T t; // T stands for "Type"
```

```
    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}
```

As you can see, we've replaced all occurrences of `Object` with `T`. To reference this generic class from within your own 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're passing a type argument — `Integer` in this case — to the `Box` class itself. 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:

```
integerBox = new Box<Integer>();
```

Or, you can put the entire statement on one line, such as:

```
Box<Integer> integerBox = new Box<Integer>();
```

Once `integerBox` is initialized, you're free to invoke its `get` method without providing a cast, as in `BoxDemo3`:

```
public class BoxDemo3 {

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        Integer someInteger = integerBox.get(); // no cast!
        System.out.println(someInteger);
    }
}
```

Furthermore, if you try adding an incompatible type to the box, such as `String`, compilation will fail, alerting you to what previously would have been a runtime bug:

```
    BoxDemo3.java:5: add(java.lang.Integer) in Box<java.lang.Integer>
    cannot be applied to (java.lang.String)
        integerBox.add("10");
```

```
                ^
    1 error
```

It's important to understand that type variables are not actually types themselves. In the above examples, you won't find `T.java` or `T.class` anywhere on the filesystem. Furthermore, `T` is not a part of the `Box` class name. In fact during compilation, all generic information will be removed entirely, leaving only `Box.class` on the filesystem. We'll discuss this later in the section on Type Erasure

Also note that a generic type may have multiple type parameters, but each parameter must be unique within its declaring class or interface. A declaration of `Box<T,T>`, for example, would generate an error on the second occurrence of `T`, but `Box<T,U>`, however, would be allowed.

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 tutorial.

Generic Methods and Constructors

Type parameters can also be declared within method and constructor signatures to create generic methods and generic constructors. This is similar to declaring a generic type, but the type parameter's scope is limited to the method or constructor in which it's declared.

```
/**
 * This version introduces a generic method.
 */
public class Box<T> {

    private T t;

    public void add(T t) {
        this.t = t;
    }

    public T get() {
```

```
        return t;
    }

    public <U> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        integerBox.inspect("some text");
    }
}
```

Here we've added one generic method, named `inspect`, that defines one type parameter, named `U`. This method accepts an object and prints its type to standard output. For comparison, it also prints out the type of `T`. For convenience, this class now also has a `main` method so that it can be run as an application.

The output from this program is:

```
T: java.lang.Integer
U: java.lang.String
```

By passing in different types, the output will change accordingly.

A more realistic use of generic methods might be something like the following, which defines a static method that stuffs references to a single item into multiple boxes:

```
public static <U> void fillBoxes(U u, List<Box<U>> boxes) {
    for (Box<U> box : boxes) {
        box.add(u);
    }
}
```

To use this method, your code would look something like the following:

```
Crayon red = ...;
List<Box<Crayon>> crayonBoxes = ...;
```

The complete syntax for invoking this method is:

```
Box.<Crayon>fillBoxes(red, crayonBoxes);
```

Here we've explicitly provided the type to be used as `U`, but more often than not, this can be left out and the compiler will infer the type that's needed:

```
Box.fillBoxes(red, crayonBoxes); // compiler infers that U is Crayon
```

This feature, known as type inference, allows you to invoke a generic method as you would an ordinary method, without specifying a type between angle brackets.

Bounded Type Parameters

There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of `Number` or its subclasses. This is what bounded type parameters are for.

To declare a bounded type parameter, list the type parameter's name, followed by the `extends` keyword, followed by its upper bound, which in this example is `Number`. Note that, in this context, `extends` is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

```
/**
 * This version introduces a bounded type parameter.
 */
public class Box<T> {

    private T t;

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        integerBox.inspect("some text"); // error: this is still String!
    }
}
```

By modifying our generic method to include this bounded type parameter, compilation will now fail, since our invocation of `inspect` still includes a `String`:

```
Box.java:21: <U>inspect(U) in Box<java.lang.Integer> cannot
  be applied to (java.lang.String)
                        integerBox.inspect("10");
                                   ^
1 error
```

To specify additional interfaces that must be implemented, use the `&` character, as in:

```
<U extends Number & MyInterface>
```

Subtyping

As you already know, it's possible to assign an object of one type to an object of another type provided that the types are compatible. For example, you can assign an `Integer` to an `Object`, since `Object` is one of `Integer`'s supertypes:

```
    Object someObject = new Object();
    Integer someInteger = new Integer(10);
```

```
someObject = someInteger; // OK
```
In object-oriented terminology, this is called an "is a" relationship. Since an `Integer` is a kind of `Object`, the assignment is allowed. But `Integer` is also a kind of `Number`, so the following code is valid as well:
```
public void someMethod(Number n){
    // method body omitted
}

someMethod(new Integer(10)); // OK
someMethod(new Double(10.1)); // OK
```

The same is also true with generics. You can perform a generic type invocation, passing `Number` as its type argument, and any subsequent invocation of `add` will be allowed if the argument is compatible with `Number`:

```
Box<Number> box = new Box<Number>();
box.add(new Integer(10)); // OK
box.add(new Double(10.1)); // OK
```

Now consider the following method:

```
public void boxTest(Box<Number> n){
    // method body omitted
}
```
What type of argument does it accept? By looking at its signature, we can see that it accepts a single argument whose type is `Box<Number>`. But what exactly does that mean? Are you allowed to pass in `Box<Integer>` or `Box<Double>`, as you might expect? Surprisingly, the answer is "no", because `Box<Integer>` and `Box<Double>` are not subtypes of `Box<Number>`.

Understanding why becomes much easier if you think of tangible objects — things you can actually picture — such as a cage:

```
// A cage is a collection of things, with bars to keep them in.
interface Cage<E> extends Collection<E>;
```

Note: The `Collection` interface is the root interface of the collection hierarchy; it represents a group of objects. Since a cage would be used for holding a collection of objects (the animals), it makes sense to include it in this example.

A lion is a kind of animal, so `Lion` would be a subtype of `Animal`:

```
interface Lion extends Animal {}
Lion king = ...;
```

Where we need some animal, we're free to provide a lion:

```
Animal a = king;
```

A lion can of course be put into a lion cage:

```
Cage<Lion> lionCage = ...;
lionCage.add(king);
```

and a butterfly into a butterfly cage:

```
interface Butterfly extends Animal {}
Butterfly monarch = ...;
Cage<Butterfly> butterflyCage = ...;
butterflyCage.add(monarch);
```

But what about an "animal cage"? English is ambiguous, so to be precise let's assume we're talking about an "all-animal cage":

```
Cage<Animal> animalCage = ...;
```

This is a cage designed to hold all kinds of animals, mixed together. It must have bars strong enough to hold in the lions, and spaced closely enough to hold in the butterflies. Such a cage might not even be feasible to build, but if it is, then:

```
animalCage.add(king);
animalCage.add(monarch);
```

Since a lion is a kind of animal (`Lion` is a subtype of `Animal`), the question then becomes, "Is a lion cage a kind of animal cage? Is `Cage<Lion>` a subtype of `Cage<Animal>`?". By the above definition of animal cage, the answer must be "no". This is surprising! But it makes perfect sense when you think about it: A lion cage cannot be assumed to keep in butterflies, and a butterfly cage cannot be assumed to hold in lions. Therefore, neither cage can be considered an "all-animal" cage:

```
animalCage = lionCage; // compile-time error
 animalCage = butterflyCage; // compile-time error
```

Without generics, the animals could be placed into the wrong kinds of cages, where it would be possible for them to escape.

Wildcards

Earlier we mentioned that English is ambiguous. The phrase "animal cage" can reasonably mean "all-animal cage", but it also suggests an entirely different concept: a cage designed not for any kind of animal, but rather for some kind of animal whose type is unknown. In generics, an unknown type is represented by the wildcard character "`?`".

To specify a cage capable of holding some kind of animal:

```
Cage<? extends Animal> someCage = ...;
```

Read "`? extends Animal`" as "an unknown type that is a subtype of `Animal`, possibly `Animal` itself", which boils down to "some kind of animal". This is an example of a bounded wildcard, where `Animal` forms the upper bound of the expected type. If you're asked for a cage that simply holds some kind of animal, you're free to provide a lion cage or a butterfly cage.

Note: It's also possible to specify a lower bound by using the `super` keyword instead of `extends`. The code `<? super Animal>`, therefore, would be read as "an unknown type that is a

supertype of `Animal`, possibly `Animal` itself". You can also specify an unknown type with an unbounded wildcard, which simply looks like `<?>`. An unbounded wildcard is essentially the same as saying `<? extends Object>`.

While `Cage<Lion>` and `Cage<Butterfly>` are not subtypes of `Cage<Animal>`, they are in fact subtypes of `Cage<? extends Animal>`:

```
someCage = lionCage; // OK
someCage = butterflyCage; // OK
```

So now the question becomes, "Can you add butterflies and lions directly to `someCage`?". As you can probably guess, the answer to this question is "no".

```
someCage.add(king);    // compiler-time error
someCage.add(monarch); // compiler-time error
```

If `someCage` is a butterfly cage, it would hold butterflies just fine, but the lions would be able to break free. If it's a lion cage, then all would be well with the lions, but the butterflies would fly away. So if you can't put anything at all into `someCage`, is it useless? No, because you can still read its contents:

```
void feedAnimals(Cage<? extends Animal> someCage) {
    for (Animal a : someCage)
        a.feedMe();
}
```

Therefore, you could house your animals in their individual cages, as shown earlier, and invoke this method first for the lions and then for the butterflies:

```
feedAnimals(lionCage);
feedAnimals(butterflyCage);
```

Or, you could choose to combine your animals in the all-animal cage instead:

```
feedAnimals(animalCage);
```

Type Erasure

When a generic type is instantiated, the compiler translates those types by a technique called type erasure — a process where the compiler removes all information related to type parameters and type arguments within a class or method. Type erasure enables Java applications that use generics to maintain binary compatibility with Java libraries and applications that were created before generics.

For instance, Box<String> is translated to type Box, which is called the raw type — a raw type is a generic class or interface name without any type arguments. This means that you can't find out what type of Object a generic class is using at runtime. The following operations are not possible:

```
public class MyClass<E> {
    public static void myMethod(Object item) {
        if (item instanceof E) {  //Compiler error
            ...
        }
        E item2 = new E();   //Compiler error
        E[] iArray = new E[10]; //Compiler error
        E obj = (E)new Object(); //Unchecked cast warning
    }
}
```

The operations shown in bold are meaningless at runtime because the compiler removes all information about the actual type argument (represented by the type parameter `E`) at compile time.

Type erasure exists so that new code may continue to interface with legacy code. Using a raw type for any other reason is considered bad programming practice and should be avoided whenever possible.

When mixing legacy code with generic code, you may encounter warning messages similar to the following:

```
Note: WarningDemo.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

This can happen when using an older API that operates on raw types, as shown in the following `WarningDemo` program:

```
public class WarningDemo {
    public static void main(String[] args){
        Box<Integer> bi;
        bi = createBox();
    }

    static Box createBox(){
        return new Box();
    }
}
```

Recompiling with `-Xlint:unchecked` reveals the following additional information:

```
WarningDemo.java:4: warning: [unchecked] unchecked conversion
found   : Box
required: Box<java.lang.Integer>
        bi = createBox();
                     ^
1 warning
```

Summary of Generics

This chapter described the following problem: We have a `Box` class, written to be generally useful so it deals with `Object`s. We need an instance that takes only `Integer`s. The comments say that only `Integer`s go in, so the programmer knows this (or should know it), but the compiler doesn't know it. This means that the compiler can't catch someone erroneously adding a `String`. When we read the value and cast it to an `Integer` we'll get an exception, but that's not ideal since the exception may be far removed from the bug in both space and time:

1. Debugging may be difficult, as the point in the code where the exception is thrown may be far removed from the point in the code where the error is located.
2. It's always better to catch bugs when compiling than when running.

Specifically, you learned that generic type declarations can include one or more type parameters; you supply one type argument for each type parameter when you use the generic type. You also learned that type parameters can be used to define generic methods and constructors. Bounded

type parameters limit the kinds of types that can be passed into a type parameter; they can specify an upper bound only. Wildcards represent unknown types, and they can specify an upper or lower bound. During compilation, type erasure removes all generic information from a generic class or interface, leaving behind only its raw type. It is possible for generic code and legacy code to interact, but in many cases the compiler will emit a warning telling you to recompile with special flags for more details.

For additional information on this topic, see Generics by Gilad Bracha.

Questions and Exercises: Generics

Questions

1. Consider the following classes:

```
public class AnimalHouse<E> {
    private E animal;
    public void setAnimal(E x) {
        animal = x;
    }
    public E getAnimal() {
        return animal;
    }
}

public class Animal{
}

public class Cat extends Animal {
}

public class Dog extends Animal {
}
```

For the following code snippets, identify whether the code:

- fails to compile,
- compiles with a warning,
- generates an error at runtime, or
- none of the above (compiles and runs without problem.)

a. AnimalHouse<Animal> house = new AnimalHouse<Cat>();

b. AnimalHouse<Dog> house = new AnimalHouse<Animal>();

c. AnimalHouse<?> house = new AnimalHouse<Cat>();

house.setAnimal(new Cat());

d. AnimalHouse house = new AnimalHouse();

house.setAnimal(new Dog());

Exercises

1. Design a class that acts as a library for the following kinds of media: book, video, and newspaper. Provide one version of the class that uses generics and one that does not. Feel free to use any additional APIs for storing and retrieving the media.

Answers to Questions and Exercises: Generics

Questions

Question 1. Consider the following classes:

```
public class AnimalHouse<E> {
    private E animal;
    public void setAnimal(E x) {
        animal = x;
    }
    public E getAnimal() {
        return animal;
    }
}

public class Animal{
}

public class Cat extends Animal {
}

public class Dog extends Animal {
}
```

For the following code snippets, identify whether the code:

- fails to compile,
- compiles with a warning,
- generates an error at runtime, or
- none of the above (compiles and runs without problem.)

```
Question 1a. AnimalHouse<Animal> house = new AnimalHouse<Cat>();
```

Answer 1a: 1. fails to compile
`AnimalHouse<Cat>` and `AnimalHouse<Animal>` are not compatible types, even though `Cat` is a subtype of `Animal`.

```
Question 1b. AnimalHouse<Cat> house = new AnimalHouse<Animal>();
```

Answer 1b: 1. fails to compile

Same as 1a: `AnimalHouse<Cat>` and `AnimalHouse<Animal>` are not compatible types, even though `Cat` is a subtype of `Animal`.

```
Question 1c. AnimalHouse<?> house = new AnimalHouse<Cat>();
             house.setAnimal(new Cat());
```

Answer 1c: 1. fails to compile

While the first line is acceptable — it is OK to define an instance of unknown type — the compiler doesn't know the type of animal stored in `house` so the `setAnimal` method cannot be used.

```
Question 1d. AnimalHouse house = new AnimalHouse();
             house.setAnimal(new Dog());
```

Answer 1d: 2. compiles with a warning

The compiler doesn't know what type `house` contains. It will accept the code, but warn that there might be a problem when setting the animal to an instance of `Dog`.

Using a generic type as a raw type might be a way to work around a particular compiler error, but you lose the type checking that generics provides, so it is not recommended.

Exercises

Exercise 1. Design a class that acts as a library for the following kinds of media: book, video, and newspaper. Provide one version of the class that uses generics and one that does not. Feel free to use any additional APIs for storing and retrieving the media.

Answer 1:

Non-Generic Version

```
import java.util.List;
import java.util.ArrayList;

public class Library {
    private List resources = new ArrayList();
    public void addMedia(Media x) {
        resources.add(x);
    }
    public Media retrieveLast() {
        int size = resources.size();
        if (size > 0) {
            return (Media)resources.get(size - 1);
        }
        return null;
    }
}

interface Media {
```

```
}

interface Book extends Media {
}

interface Video extends Media {
}

interface Newspaper extends Media {
}
```

Generic Version

```
import java.util.List;
import java.util.ArrayList;

public class Library<E extends Media> {
    private List<E> resources = new ArrayList<E>();
    public void addMedia(E x) {
        resources.add(x);
    }
    public E retrieveLast() {
        int size = resources.size();
        if (size > 0) {
            return resources.get(size - 1);
        }
        return null;
    }
}
```

Lesson: Packages

This lesson explains how to bundle classes and interfaces into packages, how to use classes that are in packages, and how to arrange your file system so that the compiler can find your source files.

Creating and Using Packages

To make types easier to find and use, to avoid naming conflicts, and to control access, programmers bundle groups of related types into packages.

Definition: A package is a grouping of related types providing access protection and name space management. Note that types refers to classes, interfaces, enumerations, and annotation types. Enumerations and annotation types are special kinds of classes and interfaces, respectively, so types are often referred to in this lesson simply as classes and interfaces.

The types that are part of the Java platform are members of various packages that bundle classes by function: fundamental classes are in java.lang, classes for reading and writing (input and output) are in java.io, and so on. You can put your types in packages too.

Suppose you write a group of classes that represent graphic objects, such as circles, rectangles, lines, and points. You also write an interface, Draggable, that classes implement if they can be dragged with the mouse.

```
//in the Draggable.java file
public interface Draggable {
    . . .
}

//in the Graphic.java file
public abstract class Graphic {
    . . .
}

//in the Circle.java file
public class Circle extends Graphic implements Draggable {
    . . .
}

//in the Rectangle.java file
public class Rectangle extends Graphic implements Draggable {
    . . .
}

//in the Point.java file
public class Point extends Graphic implements Draggable {
    . . .
```

```
}

//in the Line.java file
public class Line extends Graphic implements Draggable {
    . . .
}
```

You should bundle these classes and the interface in a package for several reasons, including the following:

- You and other programmers can easily determine that these types are related.
- You and other programmers know where to find types that can provide graphics-related functions.
- The names of your types won't conflict with the type names in other packages because the package creates a new namespace.
- You can allow types within the package to have unrestricted access to one another yet still restrict access for types outside the package.

Creating a Package

To create a package, you choose a name for the package (naming conventions are discussed in the next section) and put a `package` statement with that name at the top of every source file that contains the types (classes, interfaces, enumerations, and annotation types) that you want to include in the package.

The package statement (for example, `package graphics;`) must be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

Note: If you put multiple types in a single source file, only one can be `public`, and it must have the same name as the source file. For example, you can define `public class Circle` in the file `Circle.java`, define `public interface Draggable` in the file `Draggable.java`, define `public enum Day` in the file `Day.java`, and so forth.

You can include non-public types in the same file as a public type (this is strongly discouraged, unless the non-public types are small and closely related to the public type), but only the public type will be accessible from outside of the package. All the top-level, non-public types will be package private.

If you put the graphics interface and classes listed in the preceding section in a package called `graphics`, you would need six source files, like this:

```
//in the Draggable.java file
package graphics;
public interface Draggable {
    . . .
}
```

```
//in the Graphic.java file
package graphics;
public abstract class Graphic {
    . . .
}

//in the Circle.java file
package graphics;
public class Circle extends Graphic implements Draggable {
    . . .
}

//in the Rectangle.java file
package graphics;
public class Rectangle extends Graphic implements Draggable {
    . . .
}

//in the Point.java file
package graphics;
public class Point extends Graphic implements Draggable {
    . . .
}

//in the Line.java file
package graphics;
public class Line extends Graphic implements Draggable {
    . . .
}
```

If you do not use a `package` statement, your type ends up in an unnamed package. Generally speaking, an unnamed package is only for small or temporary applications or when you are just beginning the development process. Otherwise, classes and interfaces belong in named packages.

Naming a Package

With programmers worldwide writing classes and interfaces using the Java programming language, it is likely that many programmers will use the same name for different types. In fact, the previous example does just that: It defines a `Rectangle` class when there is already a `Rectangle` class in the `java.awt` package. Still, the compiler allows both classes to have the same name if they are in different packages. The fully qualified name of each `Rectangle` class includes the package name. That is, the fully qualified name of the `Rectangle` class in the `graphics` package is `graphics.Rectangle`, and the fully qualified name of the `Rectangle` class in the `java.awt` package is `java.awt.Rectangle`.

This works well unless two independent programmers use the same name for their packages. What prevents this problem? Convention.

Naming Conventions

Package names are written in all lowercase to avoid conflict with the names of classes or interfaces.

Companies use their reversed Internet domain name to begin their package names—for example, `com.example.orion` for a package named `orion` created by a programmer at `example.com`.

Name collisions that occur within a single company need to be handled by convention within that company, perhaps by including the region or the project name after the company name (for example, `com.company.region.package`).

Packages in the Java language itself begin with `java.` or `javax.`

In some cases, the internet domain name may not be a valid package name. This can occur if the domain name contains a hyphen or other special character, if the package name begins with a digit or other character that is illegal to use as the beginning of a Java name, or if the package name contains a reserved Java keyword, such as "int". In this event, the suggested convention is to add an underscore. For example:

Legalizing Package Names	
Domain Name	**Package Name Prefix**
clipart-open.org	org.clipart_open
free.fonts.int	int_.fonts.free
poetry.7days.com	com._7days.poetry

Using Package Members

The types that comprise a package are known as the package members.

To use a `public` package member from outside its package, you must do one of the following:

- Refer to the member by its fully qualified name
- Import the package member
- Import the member's entire package

Each is appropriate for different situations, as explained in the sections that follow.

Referring to a Package Member by Its Qualified Name

So far, most of the examples in this tutorial have referred to types by their simple names, such as `Rectangle` and `StackOfInts`. You can use a package member's simple name if the code you are writing is in the same package as that member or if that member has been imported.

However, if you are trying to use a member from a different package and that package has not been imported, you must use the member's fully qualified name, which includes the package name. Here is the fully qualified name for the `Rectangle` class declared in the `graphics` package in the previous example.

```
graphics.Rectangle
```

You could use this qualified name to create an instance of `graphics.Rectangle`:

```
graphics.Rectangle myRect = new graphics.Rectangle();
```

Qualified names are all right for infrequent use. When a name is used repetitively, however, typing the name repeatedly becomes tedious and the code becomes difficult to read. As an alternative, you can import the member or its package and then use its simple name.

Importing a Package Member

To import a specific member into the current file, put an `import` statement at the beginning of the file before any type definitions but after the `package` statement, if there is one. Here's how you would import the `Rectangle` class from the `graphics` package created in the previous section.

```
import graphics.Rectangle;
```

Now you can refer to the `Rectangle` class by its simple name.

```
Rectangle myRectangle = new Rectangle();
```

This approach works well if you use just a few members from the `graphics` package. But if you use many types from a package, you should import the entire package.

Importing an Entire Package

To import all the types contained in a particular package, use the `import` statement with the asterisk `(*)` wildcard character.

```
import graphics.*;
```

Now you can refer to any class or interface in the `graphics` package by its simple name.

```
Circle myCircle = new Circle();
Rectangle myRectangle = new Rectangle();
```

The asterisk in the `import` statement can be used only to specify all the classes within a package, as shown here. It cannot be used to match a subset of the classes in a package. For example, the following does not match all the classes in the `graphics` package that begin with `A`.

```
import graphics.A*;     //does not work
```

Instead, it generates a compiler error. With the `import` statement, you generally import only a single package member or an entire package.

Note: Another, less common form of `import` allows you to import the public nested classes of an enclosing class. For example, if the `graphics.Rectangle` class contained useful nested classes, such as `Rectangle.DoubleWide` and `Rectangle.Square`, you could import `Rectangle` and its nested classes by using the following two statements.

```
import graphics.Rectangle;
import graphics.Rectangle.*;
```

Be aware that the second import statement will not import `Rectangle`.

Another less common form of `import`, the static import statement, will be discussed at the end of this section.

For convenience, the Java compiler automatically imports three entire packages for each source file: (1) the package with no name, (2) the `java.lang` package, and (3) the current package (the package for the current file).

Apparent Hierarchies of Packages

At first, packages appear to be hierarchical, but they are not. For example, the Java API includes a `java.awt` package, a `java.awt.color` package, a `java.awt.font` package, and many others that begin with `java.awt`. However, the `java.awt.color` package, the `java.awt.font` package, and other `java.awt.xxxx` packages are not included in the `java.awt` package. The prefix `java.awt` (the Java Abstract Window Toolkit) is used for a number of related packages to make the relationship evident, but not to show inclusion.

Importing `java.awt.*` imports all of the types in the `java.awt` package, but it does not import `java.awt.color`, `java.awt.font`, or any other `java.awt.xxxx` packages. If you plan to use the classes and other types in `java.awt.color` as well as those in `java.awt`, you must import both packages with all their files:

```
import java.awt.*;
import java.awt.color.*;
```

Name Ambiguities

If a member in one package shares its name with a member in another package and both packages are imported, you must refer to each member by its qualified name. For example, the `graphics` package defined a class named `Rectangle`. The `java.awt` package also contains a `Rectangle` class. If both `graphics` and `java.awt` have been imported, the following is ambiguous.

```
Rectangle rect;
```

In such a situation, you have to use the member's fully qualified name to indicate exactly which `Rectangle` class you want. For example,

```
graphics.Rectangle rect;
```

The Static Import Statement

There are situations where you need frequent access to static final fields (constants) and static methods from one or two classes. Prefixing the name of these classes over and over can result in cluttered code. The static import statement gives you a way to import the constants and static methods that you want to use so that you do not need to prefix the name of their class.

The `java.lang.Math` class defines the `PI` constant and many static methods, including methods for calculating sines, cosines, tangents, square roots, maxima, minima, exponents, and many more. For example,

```
public static final double PI 3.141592653589793
public static double cos(double a)
```

Ordinarily, to use these objects from another class, you prefix the class name, as follows.

```
double r = Math.cos(Math.PI * theta);
```

You can use the static import statement to import the static members of java.lang.Math so that you don't need to prefix the class name, `Math`. The static members of `Math` can be imported either individually:

```
import static java.lang.Math.PI;
```

or as a group:

```
import static java.lang.Math.*;
```

Once they have been imported, the static members can be used without qualification. For example, the previous code snippet would become:

```
double r = cos(PI * theta);
```

Obviously, you can write your own classes that contain constants and static methods that you use frequently, and then use the static import statement. For example,

```
import static mypackage.MyConstants.*;
```

Note: Use static import very sparingly. Overusing static import can result in code that is difficult to read and maintain, because readers of the code won't know which class defines a particular static object. Used properly, static import makes code more readable by removing class name repetition.

Managing Source and Class Files

Many implementations of the Java platform rely on hierarchical file systems to manage source and class files, although The Java Language Specification does not require this. The strategy is as follows.

Put the source code for a class, interface, enumeration, or annotation type in a text file whose name is the simple name of the type and whose extension is `.java`. For example:

```
//in the Rectangle.java file
package graphics;
public class Rectangle {
   . . .
}
```

Then, put the source file in a directory whose name reflects the name of the package to which the type belongs:

```
.....\graphics\Rectangle.java
```

The qualified name of the package member and the path name to the file are parallel, assuming the Microsoft Windows file name separator backslash (for Unix, use the forward slash).

class name	`graphics.Rectangle`

pathname to file `graphics\Rectangle.java`

As you should recall, by convention a company uses its reversed Internet domain name for its package names. The Example company, whose Internet domain name is `example.com`, would precede all its package names with `com.example`. Each component of the package name corresponds to a subdirectory. So, if the Example company had a `com.example.graphics` package that contained a `Rectangle.java` source file, it would be contained in a series of subdirectories like this:

```
....\com\example\graphics\Rectangle.java
```

When you compile a source file, the compiler creates a different output file for each type defined in it. The base name of the output file is the name of the type, and its extension is `.class`. For example, if the source file is like this

```
//in the Rectangle.java file
package com.example.graphics;
public class Rectangle {
      . . .
}

class Helper{
      . . .
}
```

then the compiled files will be located at:

```
<path to the parent directory of the output
files>\com\example\graphics\Rectangle.class
<path to the parent directory of the output
files>\com\example\graphics\Helper.class
```

Like the `.java` source files, the compiled `.class` files should be in a series of directories that reflect the package name. However, the path to the `.class` files does not have to be the same as the path to the `.java` source files. You can arrange your source and class directories separately, as:

```
<path_one>\sources\com\example\graphics\Rectangle.java

<path_two>\classes\com\example\graphics\Rectangle.class
```

By doing this, you can give the `classes` directory to other programmers without revealing your sources. You also need to manage source and class files in this manner so that the compiler and the Java Virtual Machine (JVM) can find all the types your program uses.

The full path to the `classes` directory, `<path_two>\classes`, is called the class path, and is set with the `CLASSPATH` system variable. Both the compiler and the JVM construct the path to your `.class` files by adding the package name to the class path. For example, if

```
<path_two>\classes
```

is your class path, and the package name is

`com.example.graphics`,
then the compiler and JVM look for `.class files` in
`<path_two>\classes\com\example\graphics`.

A class path may include several paths, separated by a semicolon (Windows) or colon (Unix). By default, the compiler and the JVM search the current directory and the JAR file containing the Java platform classes so that these directories are automatically in your class path.

Setting the CLASSPATH System Variable

To display the current `CLASSPATH` variable, use these commands in Windows and Unix (Bourne shell):

```
In Windows:   C:\> set CLASSPATH
In Unix:      % echo $CLASSPATH
```

To delete the current contents of the `CLASSPATH` variable, use these commands:

```
In Windows:   C:\> set CLASSPATH=
In Unix:      % unset CLASSPATH; export CLASSPATH
```

To set the `CLASSPATH` variable, use these commands (for example):

```
In Windows:   C:\> set CLASSPATH=C:\users\george\java\classes
In Unix:      % CLASSPATH=/home/george/java/classes; export CLASSPATH
```

Summary of Creating and Using Packages

To create a package for a type, put a `package` statement as the first statement in the source file that contains the type (class, interface, enumeration, or annotation type).

To use a public type that's in a different package, you have three choices: (1) use the fully qualified name of the type, (2) import the type, or (3) import the entire package of which the type is a member.

The path names for a package's source and class files mirror the name of the package.

You might have to set your `CLASSPATH` so that the compiler and the JVM can find the `.class` files for your types.

Questions and Exercises: Creating and Using Packages

Questions

Assume you have written some classes. Belatedly, you decide they should be split into three packages, as listed in the following table. Furthermore, assume the classes are currently in the default package (they have no package statements).

Destination Packages	
Package Name	**Class Name**
mygame.server	Server
mygame.shared	Utilities

mygame.client	Client

1. Which line of code will you need to add to each source file to put each class in the right package?
2. To adhere to the directory structure, you will need to create some subdirectories in the development directory and put source files in the correct subdirectories. What subdirectories must you create? Which subdirectory does each source file go in?
3. Do you think you'll need to make any other changes to the source files to make them compile correctly? If so, what?

Exercises

Download the source files as listed here.

- Client
- Server
- Utilities

1. Implement the changes you proposed in questions 1 through 3 using the source files you just downloaded.
2. Compile the revised source files. (Hint: If you're invoking the compiler from the command line (as opposed to using a builder), invoke the compiler from the directory that contains the mygame directory you just created.)

Answers to Questions and Exercises: Creating and Using Packages

Answers

Question 1: Assume that you have written some classes. Belatedly, you decide that they should be split into three packages, as listed in the table below. Furthermore, assume that the classes are currently in the default package (they have no `package` statements).

Package Name	Class Name
`mygame.server`	`Server`
`mygame.shared`	`Utilities`
`mygame.client`	`Client`

a. What line of code will you need to add to each source file to put each class in the right package?
Answer 1a: The first line of each file must specify the package:

In `Client.java` add:
```
package mygame.client;
```
In `Server.java` add:
```
package mygame.server;:
```

In `Utilities.java` add:

```
package mygame.shared;
```

b. To adhere to the directory structure, you will need to create some subdirectories in your development directory, and put source files in the correct subdirectories. What subdirectories must you create? Which subdirectory does each source file go in?
Answer 1b: Within the `mygame` directory, you need to create three subdirectories: `client`, `server`, and `shared`.

In `mygame/client/` place:

```
Client.java
```

In `mygame/server/` place:

```
Server.java
```

In `mygame/shared/` place:

```
Utilities.java
```

c. Do you think you'll need to make any other changes to the source files to make them compile correctly? If so, what?
Answer 1c: Yes, you need to add import statements. `Client.java` and `Server.java` need to import the `Utilities` class, which they can do in one of two ways:

```
import mygame.shared.*;
       --or--
import mygame.shared.Utilities;
```

Also, `Server.java` needs to import the `Client` class:

```
import mygame.client.Client;
```

Exercises

Exercise 1: Download three source files:

- `Client`
- `Server`
- `Utilities`

a. Implement the changes you proposed in question 1, using the source files you just downloaded.
b. Compile the revised source files. (Hint: If you're invoking the compiler from the command line (as opposed to using a builder), invoke the compiler from the directory that contains the `mygame` directory you just created.)

Answer 1: Download this zip file with the solution: mygame.zip
You might need to change your proposed import code to reflect our implementation.

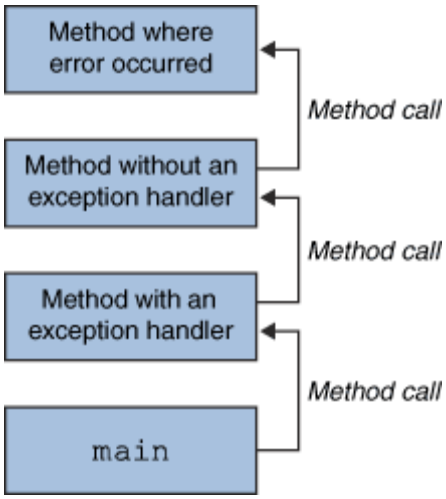
What Is an Exception?

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*.

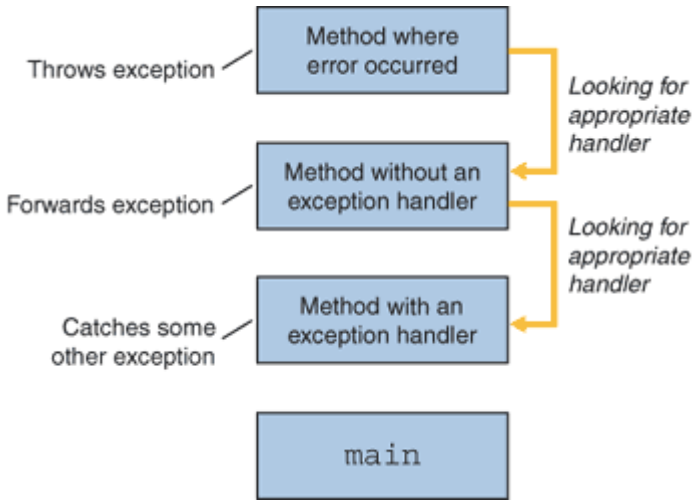
After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the *call stack* (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. You can learn more in the Advantages of Exceptions section.

The Catch or Specify Requirement

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

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

Code that fails to honor the Catch or Specify Requirement will not compile.

Not all exceptions are subject to the Catch or Specify Requirement. To understand why, we need to look at the three basic categories of exceptions, only one of which is subject to the Requirement.

The Three Kinds of Exceptions

The first kind of exception is the *checked exception*. These are exceptional conditions that a well-written application should anticipate and recover from. For example, suppose an application prompts a user for an input file name, then opens the file by passing the name to the constructor for `java.io.FileReader`. Normally, the user provides the name of an existing, readable file, so the construction of the `FileReader` object succeeds, and the execution of the application proceeds

normally. But sometimes the user supplies the name of a nonexistent file, and the constructor throws `java.io.FileNotFoundException`. A well-written program will catch this exception and notify the user of the mistake, possibly prompting for a corrected file name.

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

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

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

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

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

Errors and runtime exceptions are collectively known as *unchecked exceptions*.

Bypassing Catch or Specify

Some programmers consider the Catch or Specify Requirement a serious flaw in the exception mechanism and bypass it by using unchecked exceptions in place of checked exceptions. In general, this is not recommended. The section Unchecked Exceptions — The Controversy talks about when it is appropriate to use 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. 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 a `Vector` 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`, which are covered in Basic I/O.

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

public class ListOfNumbers {

    private Vector vector;
    private static final int SIZE = 10;

    public ListOfNumbers () {
        vector = new Vector(SIZE);
        for (int i = 0; i < SIZE; i++) {
            vector.addElement(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 + " = " +
                        vector.elementAt(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 `Vector` class's `elementAt` method, which throws an `ArrayIndexOutOfBoundsException` 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 `Vector`).

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 `elementAt`. The reason is that the exception thrown by the constructor, `IOException`, is a checked exception, and the one thrown by the `elementAt` method, `ArrayIndexOutOfBoundsException`, 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 Vector vector;
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 + " = "
                    + vector.elementAt(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 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.

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 {

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

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

Important: The `finally` block is a key tool for preventing resource leaks. When closing a file or otherwise recovering resources, place the code in a `finally` block to ensure that resource is *always* recovered.

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. For more information, see the Chained Exceptions section.

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;
}
```

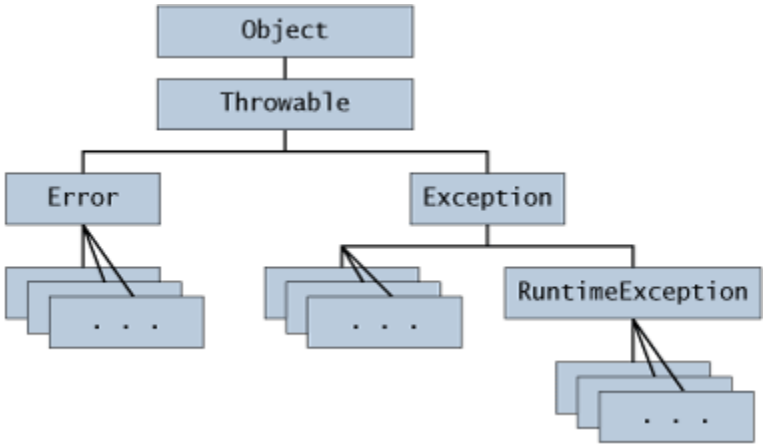
The `pop` method checks to see whether any elements are on the stack. If the stack is empty (its size is equal to `0`), `pop` instantiates a new `EmptyStackException` object (a member of `java.util`) and

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`.



The Throwable class.

Error Class

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

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 `Exception`s as opposed to `Error`s.

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

One `Exception` subclass, `RuntimeException`, is reserved for exceptions that indicate incorrect use of an API. An example of a runtime exception is `NullPointerException`, which occurs when a method tries to access a member of an object through a `null` reference. The section Unchecked Exceptions — The Controversy discusses why most applications shouldn't throw runtime exceptions or subclass `RuntimeException`.

Chained Exceptions

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

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

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

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

The following example shows how to use a chained exception.

```
try {

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

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

Accessing Stack Trace Information

Now let's suppose that the higher-level exception handler wants to dump the stack trace in its own format.

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

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

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

Logging API

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

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

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

Creating Exception Classes

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

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

An Example

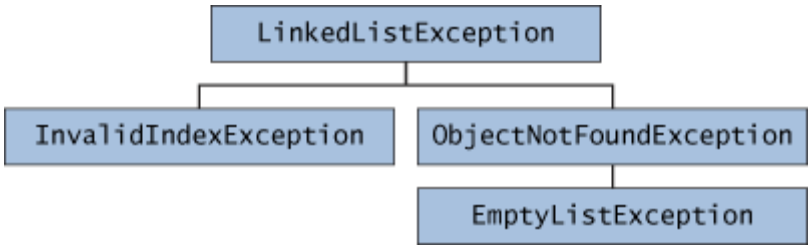
Suppose you are writing a linked list class. The class supports the following methods, among others:

- **`objectAt(int n)`** — Returns the object in the `n`th position in the list. Throws an exception if the argument is less than 0 or more than the number of objects currently in the list.

- **`firstObject()`** — Returns the first object in the list. Throws an exception if the list contains no objects.
- **`indexOf(Object o)`** — Searches the list for the specified `Object` and returns its position in the list. Throws an exception if the object passed into the method is not in the list.

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

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



Example exception class hierarchy.

Choosing a Superclass

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

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

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

Unchecked Exceptions — The Controversy

Because the Java programming language does not require methods to catch or to specify unchecked exceptions (`RuntimeException`, `Error`, and their subclasses), programmers may be tempted to write code that throws only unchecked exceptions or to make all their exception subclasses inherit from `RuntimeException`. Both of these shortcuts allow programmers to write

code without bothering with compiler errors and without bothering to specify or to catch any exceptions. Although this may seem convenient to the programmer, it sidesteps the intent of the `catch` or `specify` requirement and can cause problems for others using your classes.

Why did the designers decide to force a method to specify all uncaught checked exceptions that can be thrown within its scope? Any `Exception` that can be thrown by a method is part of the method's public programming interface. Those who call a method must know about the exceptions that a method can throw so that they can decide what to do about them. These exceptions are as much a part of that method's programming interface as its parameters and `return` value.

The next question might be: "If it's so good to document a method's API, including the exceptions it can throw, why not specify runtime exceptions too?" Runtime exceptions represent problems that are the result of a programming problem, and as such, the API client code cannot reasonably be expected to recover from them or to handle them in any way. Such problems include arithmetic exceptions, such as dividing by zero; pointer exceptions, such as trying to access an object through a null reference; and indexing exceptions, such as attempting to access an array element through an index that is too large or too small.

Runtime exceptions can occur anywhere in a program, and in a typical one they can be very numerous. Having to add runtime exceptions in every method declaration would reduce a program's clarity. Thus, the compiler does not require that you catch or specify runtime exceptions (although you can).

One case where it is common practice to throw a `RuntimeException` is when the user calls a method incorrectly. For example, a method can check if one of its arguments is incorrectly `null`. If an argument is `null`, the method might throw a `NullPointerException`, which is an *unchecked* exception.

Generally speaking, do not throw a `RuntimeException` or create a subclass of `RuntimeException` simply because you don't want to be bothered with specifying the exceptions your methods can throw.

Here's the bottom line guideline: If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

Advantages of Exceptions

Now that you know what exceptions are and how to use them, it's time to learn the advantages of using exceptions in your programs.

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

Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program. In traditional programming, error detection,

reporting, and handling often lead to confusing spaghetti code. For example, consider the pseudocode method here that reads an entire file into memory.

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

At first glance, this function seems simple enough, but it ignores all the following potential errors.

- What happens if the file can't be opened?
- What happens if the length of the file can't be determined?
- What happens if enough memory can't be allocated?
- What happens if the read fails?
- What happens if the file can't be closed?

To handle such cases, the `readFile` function must have more code to do error detection, reporting, and handling. Here is an example of what the function might look like.

```
errorCodeType readFile {
    initialize errorCode = 0;

    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
```

```
        errorCode = -5;
    }
    return errorCode;
}
```

There's so much error detection, reporting, and returning here that the original seven lines of code are lost in the clutter. Worse yet, the logical flow of the code has also been lost, thus making it difficult to tell whether the code is doing the right thing: Is the file really being closed if the function fails to allocate enough memory? It's even more difficult to ensure that the code continues to do the right thing when you modify the method three months after writing it. Many programmers solve this problem by simply ignoring it — errors are reported when their programs crash.

Exceptions enable you to write the main flow of your code and to deal with the exceptional cases elsewhere. If the `readFile` function used exceptions instead of traditional error-management techniques, it would look more like the following.

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
       doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Note that exceptions don't spare you the effort of doing the work of detecting, reporting, and handling errors, but they do help you organize the work more effectively.

Advantage 2: Propagating Errors Up the Call Stack

A second advantage of exceptions is the ability to propagate error reporting up the call stack of methods. Suppose that the `readFile` method is the fourth method in a series of nested method calls made by the main program: `method1` calls `method2`, which calls `method3`, which finally calls `readFile`.

```
method1 {
    call method2;
}
```

```
method2 {
    call method3;
}

method3 {
    call readFile;
}
```

Suppose also that `method1` is the only method interested in the errors that might occur within `readFile`. Traditional error-notification techniques force `method2` and `method3` to propagate the error codes returned by `readFile` up the call stack until the error codes finally reach `method1`—the only method that is interested in them.

```
method1 {
    errorCodeType error;
    error = call method2;
    if (error)
        doErrorProcessing;
    else
        proceed;
}

errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;
    else
        proceed;
}

errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error)
        return error;
    else
        proceed;
}
```

Recall that the Java runtime environment searches backward through the call stack to find any methods that are interested in handling a particular exception. A method can duck any exceptions thrown within it, thereby allowing a method farther up the call stack to catch it. Hence, only the methods that care about errors have to worry about detecting errors.

```
method1 {
    try {
        call method2;
    } catch (exception e) {
        doErrorProcessing;
    }
}
```

```
method2 throws exception {
    call method3;
}

method3 throws exception {
    call readFile;
}
```

However, as the pseudocode shows, ducking an exception requires some effort on the part of the middleman methods. Any checked exceptions that can be thrown within a method must be specified in its `throws` clause.

Advantage 3: Grouping and Differentiating Error Types

Because all exceptions thrown within a program are objects, the grouping or categorizing of exceptions is a natural outcome of the class hierarchy. An example of a group of related exception classes in the Java platform are those defined in `java.io` — `IOException` and its descendants. `IOException` is the most general and represents any type of error that can occur when performing I/O. Its descendants represent more specific errors. For example, `FileNotFoundException` means that a file could not be located on disk.

A method can write specific handlers that can handle a very specific exception. The `FileNotFoundException` class has no descendants, so the following handler can handle only one type of exception.

```
catch (FileNotFoundException e) {
    ...
}
```

A method can catch an exception based on its group or general type by specifying any of the exception's superclasses in the `catch` statement. For example, to catch all I/O exceptions, regardless of their specific type, an exception handler specifies an `IOException` argument.

```
catch (IOException e) {
    ...
}
```

This handler will be able to catch all I/O exceptions, including `FileNotFoundException`, `EOFException`, and so on. You can find details about what occurred by querying the argument passed to the exception handler. For example, use the following to print the stack trace.

```
catch (IOException e) {
    e.printStackTrace();  //Output goes to System.err.
    e.printStackTrace(System.out);  //Send trace to stdout.
}
```

You could even set up an exception handler that handles any `Exception` with the handler here.

```
catch (Exception e) {    //A (too) general exception handler
    ...
}
```

The `Exception` class is close to the top of the `Throwable` class hierarchy. Therefore, this handler will catch many other exceptions in addition to those that the handler is intended to catch. You may want to handle exceptions this way if all you want your program to do, for example, is print out an error message for the user and then exit.

In most situations, however, you want exception handlers to be as specific as possible. The reason is that the first thing a handler must do is determine what type of exception occurred before it can decide on the best recovery strategy. In effect, by not catching specific errors, the handler must accommodate any possibility. Exception handlers that are too general can make code more error-prone by catching and handling exceptions that weren't anticipated by the programmer and for which the handler was not intended.

As noted, you can create groups of exceptions and handle exceptions in a general fashion, or you can use the specific exception type to differentiate exceptions and handle exceptions in an exact fashion.

Summary

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.

Questions and Exercises

Questions

1. Is the following code legal?

```
try {

} finally {

}
```

2. What exception types can be caught by the following handler?

```
catch (Exception e) {

}
```

What is wrong with using this type of exception handler?

3. Is there anything wrong with the following exception handler as written? Will this code compile?

```
try {

} catch (Exception e) {

} catch (ArithmeticException a) {

}
```

4. Match each situation in the first list with an item in the second list.
   a. `int[] A;`
      `A[0] = 0;`
   b. The JVM starts running your program, but the JVM can't find the Java platform classes. (The Java platform classes reside in `classes.zip` or `rt.jar`.)
   c. A program is reading a stream and reaches the `end of stream` marker.
   d. Before closing the stream and after reaching the `end of stream` marker, a program tries to read the stream again.
      1. __error
      2. __checked exception
      3. __compile error
      4. __no exception

Exercises

1. Add a readList method to ListOfNumbers.java. This method should read in int values from a file, print each value, and append them to the end of the vector. You should catch all appropriate errors. You will also need a text file containing numbers to read in.

2. Modify the following cat method so that it will compile.

```
public static void cat(File file) {
    RandomAccessFile input = null;
    String line = null;

    try {
        input = new RandomAccessFile(file, "r");
        while ((line = input.readLine()) != null) {
            System.out.println(line);
        }
        return;
    } finally {
        if (input != null) {
```

```
            input.close();
        }
    }
}
```

Answers to Questions and Exercises

Questions

1. Question: Is the following code legal?

```
try {

} finally {

}
```

Answer: Yes, it's legal — and very useful A `try` statement does not have to have a `catch` block if it has a `finally` block. If the code in the `try` statement has multiple exit points and no associated `catch` clauses, the code in the `finally` block is executed no matter how the `try` block is exited. Thus it makes sense to provide a `finally` block whenever there is code that *must always* be executed. This include resource recovery code, such as the code to close I/O streams.

2. Question: What exception types can be caught by the following handler?

```
catch (Exception e) {

}
```

What is wrong with using this type of exception handler?

Answer: This handler catches exceptions of type `Exception`; therefore, it catches any exception. This can be a poor implementation because you are losing valuable information about the type of exception being thrown and making your code less efficient. As a result, your program may be forced to determine the type of exception before it can decide on the best recovery strategy.

3. Question: Is there anything wrong with this exception handler as written? Will this code compile?

```
try {

} catch (Exception e) {

} catch (ArithmeticException a) {

}
```

Answer: This first handler catches exceptions of type `Exception`; therefore, it catches any exception, including `ArithmeticException`. The second handler could never be reached. This code will not compile.

4. Question: Match each situation in the first list with an item in the second list.
    a. `int[] A;`
       `A[0] = 0;`
    b. The JVM starts running your program, but the JVM can't find the Java platform classes. (The Java platform classes reside in `classes.zip` or `rt.jar`.)
    c. A program is reading a stream and reaches the `end of stream` marker.
    d. Before closing the stream and after reaching the `end of stream` marker, a program tries to read the stream again.

    1. __error
    2. __checked exception
    3. __compile error
    4. __no exception

    Answer:

    a. 3 (compile error). The array is not initialized and will not compile.
    b. 1 (error).
    c. 4 (no exception). When you read a stream, you expect there to be an end of stream marker. You should use exceptions to catch unexpected behavior in your program.
    d. 2 (checked exception).

Exercises

1. Exercise: Add a `readList` method to `ListOfNumbers.java`. This method should read in `int` values from a file, print each value, and append them to the end of the vector. You should catch all appropriate errors. You will also need a text file containing numbers to read in.

    Answer: See `ListOfNumbers2.java`.

2. Exercise: Modify the following `cat` method so that it will compile:

```
public static void cat(File file) {
    RandomAccessFile input = null;
    String line = null;

    try {
        input = new RandomAccessFile(file, "r");
        while ((line = input.readLine()) != null) {
            System.out.println(line);
        }
        return;
    } finally {
        if (input != null) {
            input.close();
        }
    }
}
```

Answer: The code to catch exceptions is shown in red:

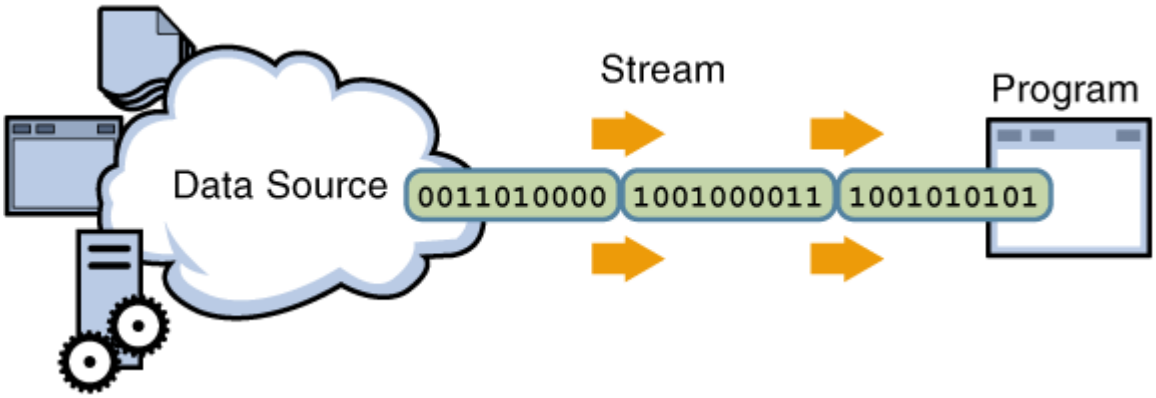
```
public static void cat(File file) {
    RandomAccessFile input = null;
    String line = null;

    try {
        input = new RandomAccessFile(file, "r");
        while ((line = input.readLine()) != null) {
            System.out.println(line);
        }
        return;
    } catch(FileNotFoundException fnf) {
        System.err.format("File: %s not found%n", file);
    } catch(IOException e) {
        System.err.println(e.toString());
    } finally {
        if (input != null) {
            try {
                input.close();
            } catch(IOException io) {
            }
        }
    }
}
```

I/O Streams

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

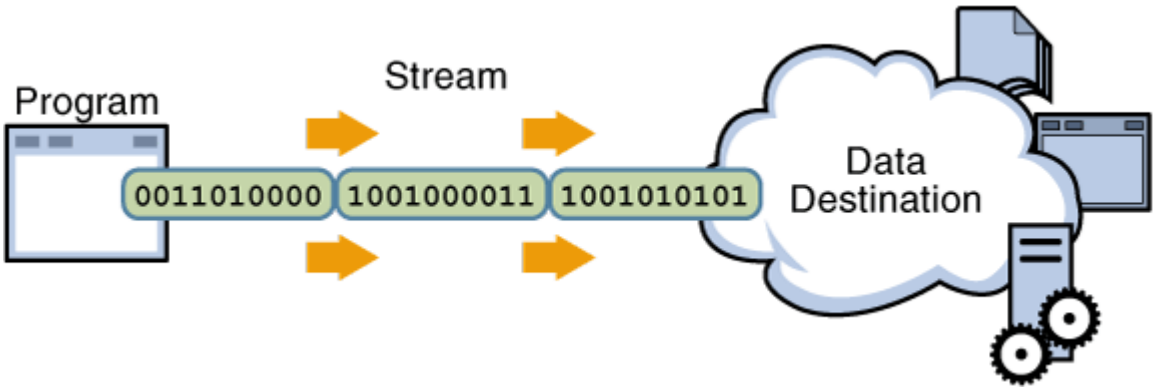
Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Some streams simply pass on data; others manipulate and transform the data in useful ways.

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



Reading information into a program.

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



Writing information from a program.

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

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

In the next section, we'll use the most basic kind of streams, byte streams, to demonstrate the common operations of Stream I/O. For sample input, we'll use the example file `xanadu.txt`, which contains the following verse:

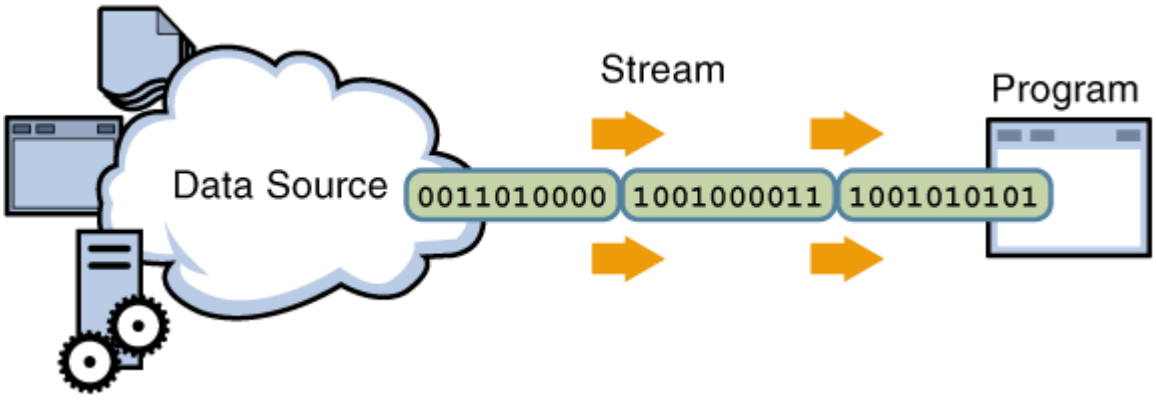
```
In Xanadu did Kubla Khan
A stately pleasure-dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

I/O Streams

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

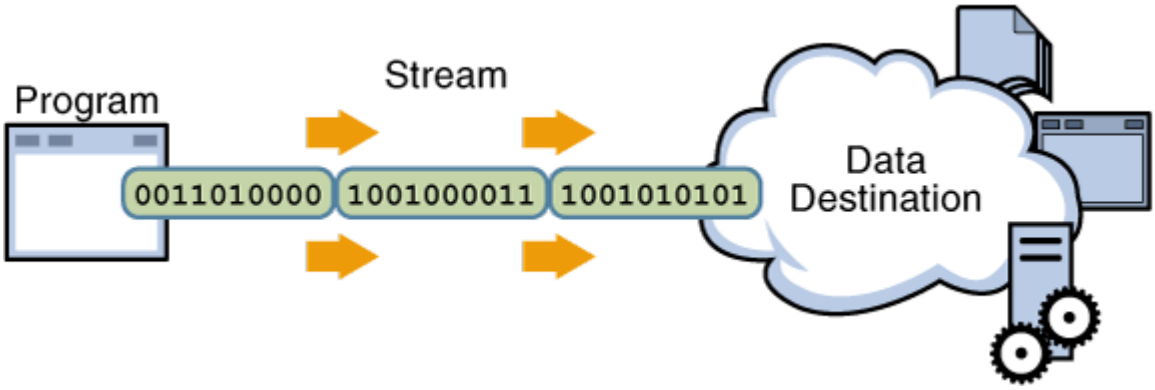
Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Some streams simply pass on data; others manipulate and transform the data in useful ways.

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



Reading information into a program.

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



Writing information from a program.

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

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

In the next section, we'll use the most basic kind of streams, byte streams, to demonstrate the common operations of Stream I/O. For sample input, we'll use the example file `xanadu.txt`, which contains the following verse:

```
In Xanadu did Kubla Khan
A stately pleasure-dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

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`.

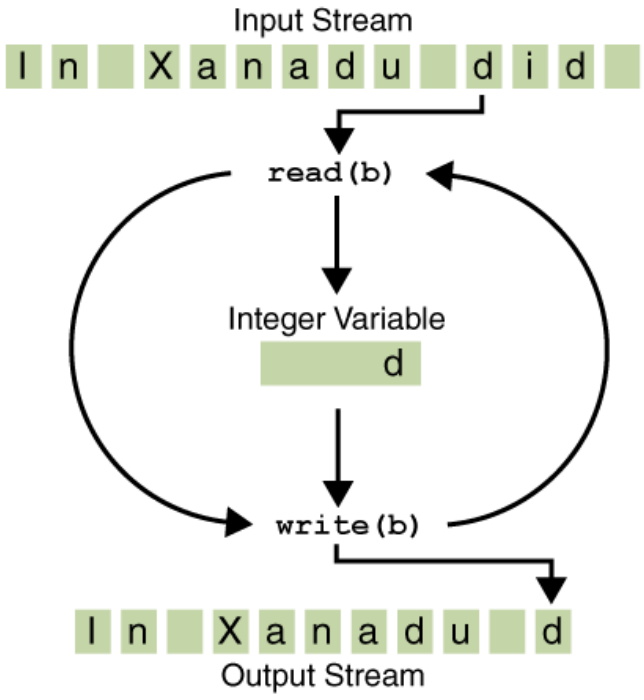
There are many byte stream classes. To demonstrate how byte streams work, we'll focus on the file I/O byte streams, `FileInputStream` and `FileOutputStream`. Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed.

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.

So why talk about byte streams? Because all other stream types are built on byte streams.

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.

There are many ways to structure text input and output beyond characters and lines. For more information, see Scanning and Formatting.

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`. See Formatting for more on these methods.

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.

Scanning and Formatting

Programming I/O often involves translating to and from the neatly formatted data humans like to work with. To assist you with these chores, the Java platform provides two APIs. The scanner API breaks input into individual tokens associated with bits of data. The formatting API assembles data into nicely formatted, human-readable form.

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
...
```

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. We could override the locale for the whole program — or we could just use formatting, as described in the next topic, Formatting.

Formatting

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

Note: The only `PrintStream` objects you are likely to need are `System.out` and `System.err`. (See I/O from the Command Line for more on these objects.) When you need to create a formatted output stream, instantiate `PrintWriter`, not `PrintStream`.

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

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

The `print` and `println` Methods

Invoking `print` or `println` outputs a single value after converting the value using the appropriate `toString` method. We can see this in the `Root` example:

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

        System.out.print("The square root of ");
        System.out.print(i);
        System.out.print(" is ");
        System.out.print(r);
        System.out.println(".");

        i = 5;
        r = Math.sqrt(i);
        System.out.println("The square root of " + i + " is " + r + ".");
    }
```

```
}
```
Here is the output of `Root`:

```
The square root of 2 is 1.4142135623730951.
The square root of 5 is 2.23606797749979.
```
The `i` and `r` variables are formatted twice: the first time using code in an overload of `print`, the second time by conversion code automatically generated by the Java compiler, which also utilizes `toString`. You can format any value this way, but you don't have much control over the results.

The `format` Method

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

Format strings support many features. In this tutorial, we'll just cover some basics. For a complete description, see `format string syntax` in the API specification.

The `Root2` example formats two values with a single `format` invocation:

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

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

```
The square root of 2 is 1.414214.
```
Like the three used in this example, all format specifiers begin with a `%` and end with a 1- or 2-character *conversion* that specifies the kind of formatted output being generated. The three conversions used here are:

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

Here are some other conversions:

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

There are many other conversions.

Note: Except for `%%` and `%n`, all format specifiers must match an argument. If they don't, an exception is thrown.

In the Java programming language, the `\n` escape always generates the linefeed character (`\u000A`). Don't use `\n` unless you specifically want a linefeed character. To get the correct line separator for the local platform, use `%n`.

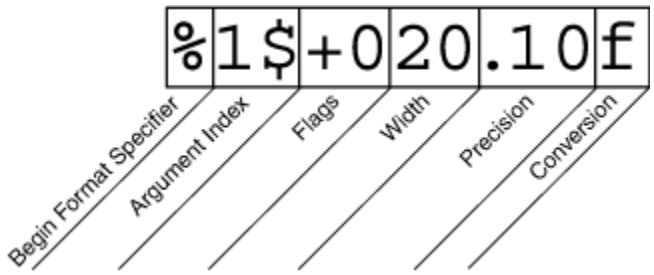
In addition to the conversion, a format specifier can contain several additional elements that further customize the formatted output. Here's an example, `Format`, that uses every possible kind of element.

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

Here's the output:

```
3.141593, +00000003.1415926536
```

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



Elements of a Format Specifier.

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

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

- `System.out.format("%f, %<+020.10f %n", Math.PI);`

I/O from the Command Line

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

Standard Streams

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

The Java platform supports three Standard Streams: *Standard Input*, accessed through `System.in`; *Standard Output*, accessed through `System.out`; and *Standard Error*, accessed through `System.err`. These objects are defined automatically and do not need to be opened. Standard Output and Standard Error are both for output; having error output separately allows the user to divert regular output to a file and still be able to read error messages. 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 verify method.
    static boolean verify(String login, char[] password) {
        return true;
    }

    //Dummy change method.
    static void change(String login, char[] password) {}
}
```

`Password` 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.

Data Streams

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

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

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

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

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

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

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

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

`DataStreams` writes out the records and closes the output stream.

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

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

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

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

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

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

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

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

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

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

The correct type to use for currency values is `java.math.BigDecimal`. Unfortunately, `BigDecimal` is an object type, so it won't work with data streams. However, `BigDecimal` *will* work with object streams, which are covered in the next section.

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`.

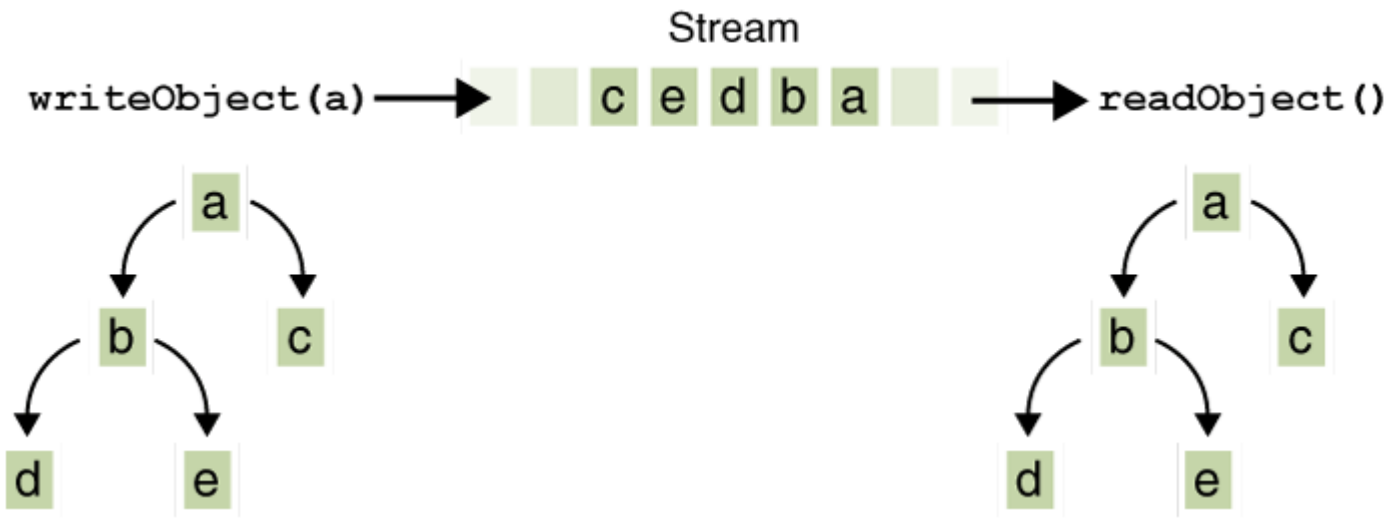
The object stream classes are `ObjectInputStream` and `ObjectOutputStream`. These classes implement `ObjectInput` and `ObjectOutput`, which are subinterfaces of `DataInput` and `DataOutput`. That means that all the primitive data I/O methods covered in Data Streams are also implemented in object streams. 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