

OOP Basic concepts with Java

Source: <http://docs.oracle.com/javase/tutorial/java/concepts/>

I recommend going through this doc at minimum for Android dev. For more detailed tutorials, follow link above.

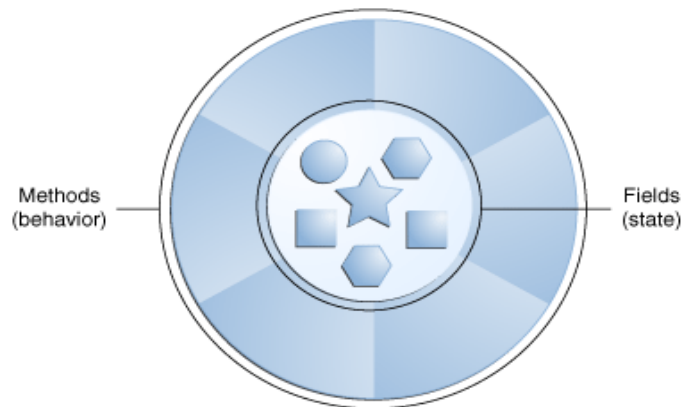
– Fahim Uddin.

What Is an Object?

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.

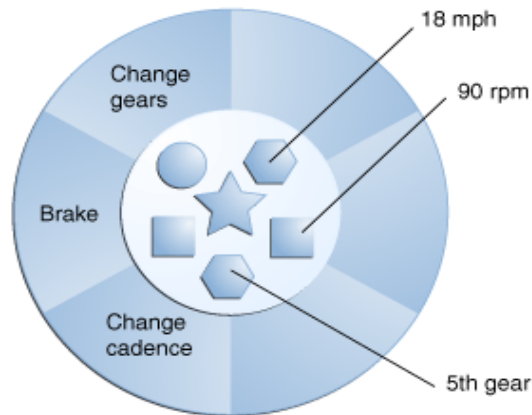


A software object.

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?

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();  
}  
}
```

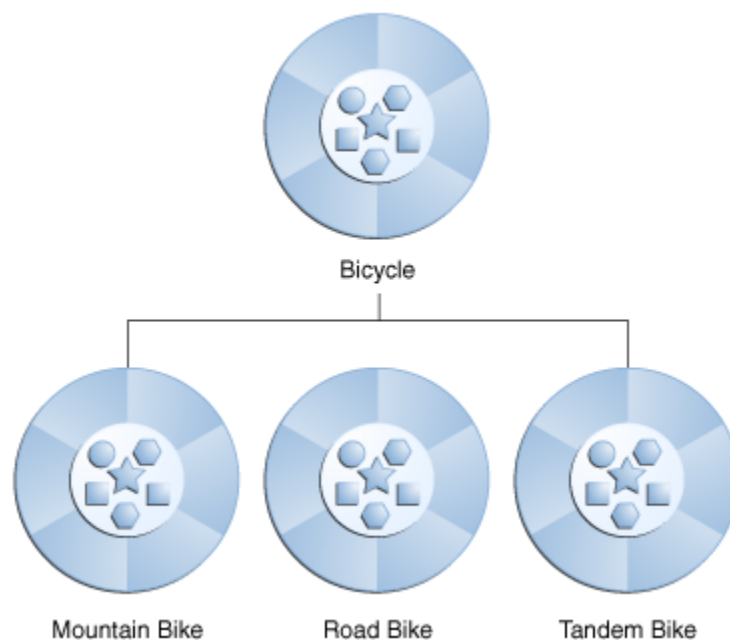
The output of this test prints the ending pedal cadence, speed, and gear for the two bicycles:

```
cadence:50 speed:10 gear:2  
cadence:40 speed:20 gear:3
```

What Is Inheritance?

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?

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 {  
  
    // wheel revolutions per minute  
    void changeCadence(int newValue);  
  
    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 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 SE platform. Load the page in your browser and bookmark it. As a programmer, it will become your single most important piece of reference documentation.

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):

`originOne` 

Instantiating a Class

The `new` operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The `new` operator also invokes the object constructor.

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

The `new` operator requires a single, postfix argument: a call to a constructor. The name of the constructor provides the name of the class to instantiate.

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

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

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

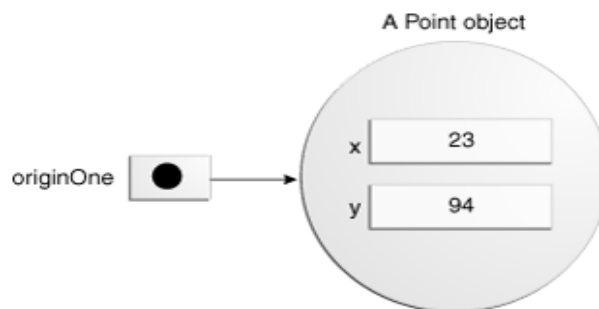
Here's the code for the Point class:

```
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;
}

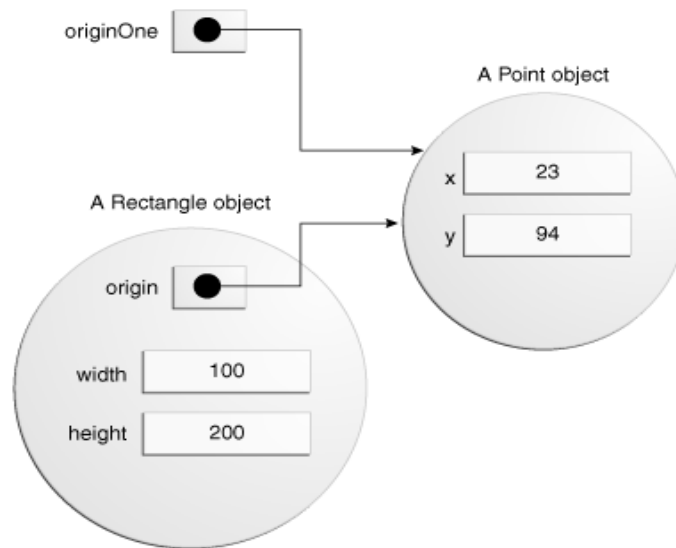
// a method for computing the area
// of the rectangle
public int getArea() {
    return width * height;
}
}

```

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 `getArea` 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;
}
```

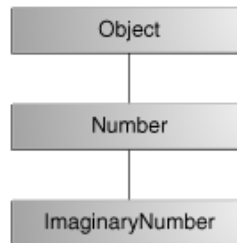


```
}
```

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.

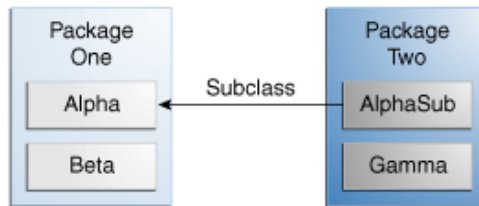
Access Levels				
Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
<i>no modifier</i>	Y	Y	N	N
private	Y	N	N	N

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
<i>no modifier</i>	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 {

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

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

```

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

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

Static Initialization Blocks

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

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

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

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

```
class Whatever {  
    public static varType myVar = initializeClassVariable();  
  
    private static varType initializeClassVariable() {  
  
        // initialization code goes here  
    }  
}
```

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

Initializing Instance Members

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

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

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

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

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

```
class Whatever {  
    private varType myVar = initializeInstanceVariable();  
  
    protected final varType initializeInstanceVariable() {  
  
        // initialization code goes here  
    }  
}
```

}

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

2. `public class IdentifyMyParts {`
3. `public static int x = 7;`
4. `public int y = 3;`
5. `}`
 - a. What are the class variables?
 - b. What are the instance variables?
 - c. What is the output from the following code:
 - d. `IdentifyMyParts a = new IdentifyMyParts();`
 - e. `IdentifyMyParts b = new IdentifyMyParts();`
 - f. `a.y = 5;`
 - g. `b.y = 6;`
 - h. `a.x = 1;`
 - i. `b.x = 2;`
 - j. `System.out.println("a.y = " + a.y);`
 - k. `System.out.println("b.y = " + b.y);`
 - l. `System.out.println("a.x = " + a.x);`
 - m. `System.out.println("b.x = " + b.x);`
 - n. `System.out.println("IdentifyMyParts.x = " +`
 - o. `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. 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 and Exercises: Objects

Questions

1. What's wrong with the following program?
2.

```
public class SomethingIsWrong {
```
3.

```
    public static void main(String[] args) {
```
4.

```
        Rectangle myRect;
```
5.

```
        myRect.width = 40;
```
6.

```
        myRect.height = 50;
```
7.

```
        System.out.println("myRect's area is "
```
8.

```
            + myRect.area());
```
9.

```
    }
```
10.

```
}
```
11. 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?
12. ...
13.

```
String[] students = new String[10];
```
14.

```
String studentName = "Peter Parker";
```
15.

```
students[0] = studentName;
```
16.

```
studentName = null;
```
17. ...
18. 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.
3.

```
public class NumberHolder {
```
4.

```
    public int anInt;
```
5.

```
    public float aFloat;
```
6.

```
}
```

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();
```

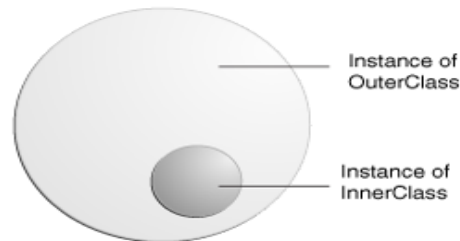
Inner Classes

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 `DataSet` class below consists of:

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

    public DataSet() {
        // 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
        DataSet ds = new DataSet();
        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?

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 tellItLikeIts() {  
        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.

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";
    // Note use of array
    String[] reviewers();
}
```

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",
// Note array notation
reviewers = {"Alice", "Bob", "Cindy"}
)
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 this to use @Documented
import java.lang.annotation.*;

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

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?
2. `public interface House {`
3. `@Deprecated`
4. `void open();`
5. `void openFrontDoor();`
6. `void openBackDoor();`
7. `}`
8. Consider this implementation of the House interface, shown in Question 1.
9. `public class MyHouse implements House {`
10. `public void open() {}`
11. `public void openFrontDoor() {}`
12. `public void openBackDoor() {}`
13. `}`

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?

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

// An enum with values RIGHT, LEFT
int turn(Direction direction,
         double radius,
         double startSpeed,
         double endSpeed);
int changeLanes(Direction direction,
                double startSpeed,
                double endSpeed);
int signalTurn(Direction direction,
               boolean signalOn);
int getRadarFront(double distanceToCar,
                  double speedOfCar);
int getRadarRear(double distanceToCar,
                  double speedOfCar);
.....
// more method signatures
}

```

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

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

```

public class OperateBMW760i implements OperateCar {

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

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

```

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  
  
    // base of natural logarithms  
    double E = 2.718282;  
  
    // method signatures  
    void doSomething (int i, double x);  
    int doSomethingElse(String s);  
}
```

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

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

The Interface Body

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

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

Implementing an Interface

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

A Sample Interface, `Relatable`

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

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

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

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

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

Implementing the `Relatable` Interface

Here is the `Rectangle` class 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

1. What methods would a class that implements the `java.lang.CharSequence` interface have to implement?
2. What is wrong with the following interface?
3.

```
public interface SomethingIsWrong {
```
4.

```
    void aMethod(int aValue){
```
5.

```
        System.out.println("Hi Mom");
```
6.

```
    }
```
7.

```
}
```
8. Fix the interface in question 2.
9. Is the following interface valid?
10.

```
public interface Marker {
```
11.

```
}
```

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.

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.

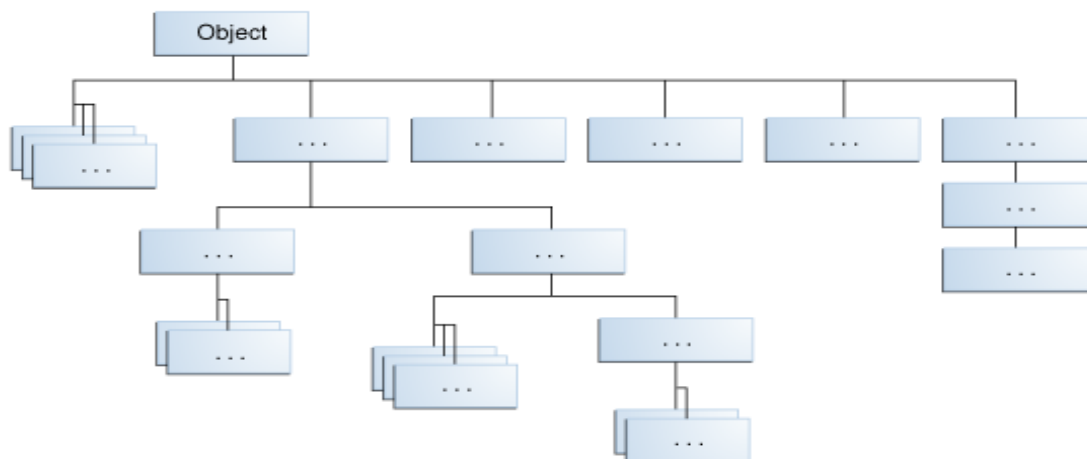
Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, Object. Such a class is said to be *descended* from all the classes in the inheritance chain stretching back to Object.

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

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

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

```

    }

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

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

```

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

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

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

```

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

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

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

```

The following is the output from the test program:

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

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

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

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

Hiding Fields

Within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if their types are different. Within the subclass, the field in the superclass cannot be referenced by its simple name. Instead, the field must be accessed through `super`, which is covered in the next section. Generally speaking, 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 {  
    // overrides printMethod in Superclass  
    public void printMethod() {  
        super.printMethod();  
        System.out.println("Printed in Subclass");  
    }  
    public static void main(String[] args) {  
        Subclass s = new Subclass();  
        s.printMethod();  
    }  
}
```

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

```
Printed in Superclass.  
Printed in Subclass
```

Subclass Constructors

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

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

```
    seatHeight = startHeight;
}
```

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

The syntax for calling a superclass constructor is

```
super();
or:
```

```
super(parameter list);
```

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

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

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

Object as a Superclass

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

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

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

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

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

The `clone()` Method

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

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

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

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

or:

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

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

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

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

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

The equals() Method

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

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

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

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

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

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

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

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

The finalize() Method

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

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

The getClass() Method

You cannot override getClass.

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

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

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

The hashCode() Method

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

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

The toString() Method

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

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

You can use toString() along with System.out.println() to display a text representation of an object, such as an instance of Book:

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

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

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

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. A class that is declared final cannot be subclassed. This is particularly useful, for example, when creating an immutable class like the `String` class.

Abstract Methods and Classes

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

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

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

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

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

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

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

Abstract Classes versus Interfaces

Unlike interfaces, abstract classes can contain fields that are not static and final, and they can contain implemented methods. Such abstract classes are similar to interfaces, except that they provide a partial

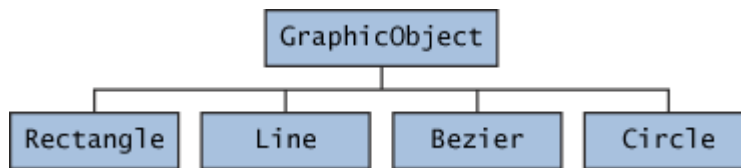
implementation, leaving it to subclasses to complete the implementation. If an abstract class contains *only* abstract method declarations, it should be declared as an interface instead.

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

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

An Abstract Class Example

In an object-oriented drawing application, you can draw circles, rectangles, lines, Bezier curves, and many other graphic objects. These objects all have certain states (for example: position, orientation, line color, fill color) and behaviors (for example: moveTo, rotate, resize, draw) in common. Some of these states and behaviors are the same for all graphic objects—for example: position, fill color, and moveTo. Others require different implementations—for example, resize or draw. All GraphicObjects must know how to draw or resize themselves; they just differ in how they do it. This is a perfect situation for an abstract superclass. You can take advantage of the similarities and declare all the graphic objects to inherit from the same abstract parent object—for example, GraphicObject, as shown in the following figure.



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

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

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

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

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

```

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

```

When an Abstract Class Implements an Interface

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

```

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

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

```

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

Class Members

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

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.