# What Is an Object?

Objects are key to understanding *object-oriented* technology. You can look around you now and see many examples of real-world objects: your dog, your desk, your television set, your bicycle.

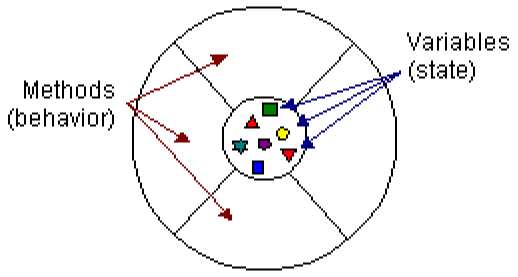
These real-world objects share two characteristics: They all have *state* and *behavior*. For example, dogs have state (name, color, breed, hungry) and behavior (barking, fetching, and wagging tail). Bicycles have state (current gear, current pedal cadence, two wheels, number of gears) and behavior (braking, accelerating, slowing down, changing gears).

Software objects are modeled after real-world objects in that they too have state and behavior. A software object maintains its state in one or more *variables*. A variable is an item of data named by an identifier. A software object implements its behavior with *methods*. A method is a function (subroutine) associated with an object.

**Definition:** An object is a software bundle of variables and related methods.

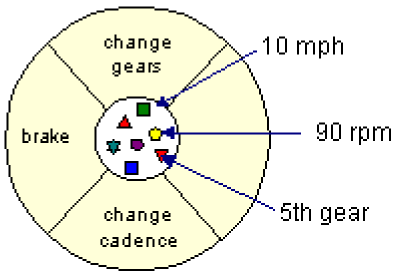
You can represent real-world objects by using software objects. You might want to represent real-world dogs as software objects in an animation program or a real-world bicycle as a software object in the program that controls an electronic exercise bike. You can also use software objects to model abstract concepts. For example, an *event* is a common object used in GUI window systems to represent the action of a user pressing a mouse button or a key on the keyboard.

The following illustration is a common visual representation of a software object:



Everything that the software object knows (state) and can do (behavior) is expressed by the variables and the methods within that object. A software object that modeled your real-world bicycle would have variables that indicated the bicycle's current state: its speed is 10 mph, its pedal cadence is 90 rpm, and its current gear is the 5th gear. These variables are formally known as *instance variables* because they contain the state for a particular bicycle object, and in object-oriented terminology, a particular object is called an *instance*.

The following figure illustrates a bicycle modeled as a software object.



In addition to its variables, the software bicycle would also have methods to brake, change the pedal cadence, and change gears. (The bike would not have a method for changing the speed of the bicycle, as the bike's speed is just a side effect of what gear it's in, how fast the rider is pedaling, whether the brakes are on, and how steep the hill is.)

These methods are formally known as *instance methods* because they inspect or change the state of a particular bicycle instance.

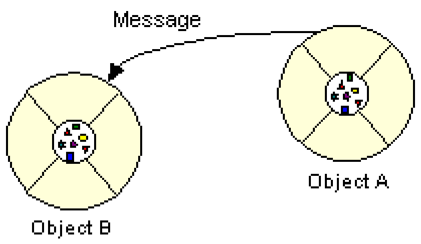
The object diagrams show that the object's variables make up the center, or nucleus, of the object. Methods surround and hide the object's nucleus from other objects in the program. Packaging an object's variables within the protective custody of its methods is called *encapsulation* This conceptual picture of an object-a nucleus of variables packaged within a protective membrane of methods-is an ideal representation of an object and is the ideal that designers of object-oriented systems strive for. However, it's not the whole story. Often, for practical reasons, an object may wish to expose some of its variables or hide some of its methods. In the Java programming language, an object can specify one of four access levels for each of its variables and methods. The access level determines which other objects and classes can access that variable or method. Variable and method access in Java is covered in Controlling Access to Members of a Class. Encapsulating related variables and methods into a neat software bundle is a simple yet powerful idea that provides two primary benefits to software developers:

- **Modularity:** The source code for an object can be written and maintained independently of the source code for other objects. Also, an object can be easily passed around in the system. You can give your bicycle to someone else, and it will still work.
- **Information hiding:** An object has a public interface that other objects can use to communicate with it. The object can maintain private information and methods that can be changed at any time without affecting the other objects that depend on it. You don't need to understand the gear mechanism on your bike to use it.

## What Is a Message?

A single object alone is generally not very useful. Instead, an object usually appears as a component of a larger program or application that contains many other objects. Through the interaction of these objects, programmers achieve higher-order functionality and more complex behavior. Your bicycle hanging from a hook in the garage is just a bunch of titanium alloy and rubber; by itself, the bicycle is incapable of any activity. The bicycle is useful only when another object (you) interacts with it (pedal).

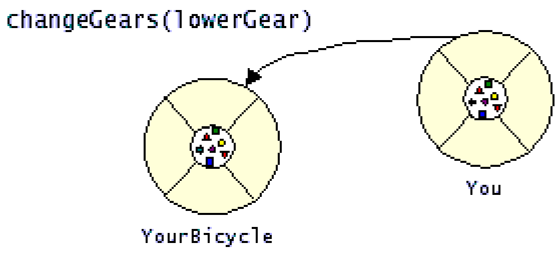
Software objects interact and communicate with each other by sending *messages* to each other. When object A wants object B to perform one of B's methods, object A sends a message to object B



Sometimes, the receiving object needs more information so that it knows exactly what to do; for example, when you want to change gears on your bicycle, you have to indicate which gear you want. This information is passed along with the message as *parameters*.

The next figure shows the three components that comprise a message:

1. The object to which the message is addressed (YourBicycle)
2. The name of the method to perform (changeGears)
3. Any parameters needed by the method (lowerGear)



These three components are enough information for the receiving object to perform the desired method. No other information or context is required.

Messages provide two important benefits.

- An object's behavior is expressed through its methods, so (aside from direct variable access) message passing supports all possible interactions between objects.
- Objects don't need to be in the same process or even on the same machine to send and receive messages back and forth to each other.

## What Is a Class?

In the real world, you often have many objects of the same kind. For example, your bicycle is just one of many bicycles in the world. Using object-oriented terminology, we say that your bicycle object is an *instance* of the class of objects known as bicycles. Bicycles have some state (current gear, current cadence, two wheels) and behavior (change gears, brake) in common. However, each bicycle's state is independent of and can be different from that of other bicycles.

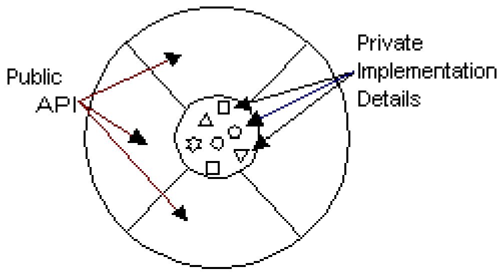
When building bicycles, manufacturers take advantage of the fact that bicycles share characteristics, building many bicycles from the same blueprint. It would be very inefficient to produce a new blueprint for every individual bicycle manufactured.

In object-oriented software, it's also possible to have many objects of the same kind that share characteristics: rectangles, employee records, video clips, and so on. Like the bicycle manufacturers, you can take advantage of the fact that objects of the same kind are similar and you can create a blueprint for those objects. A software blueprint for objects is called a *class*.

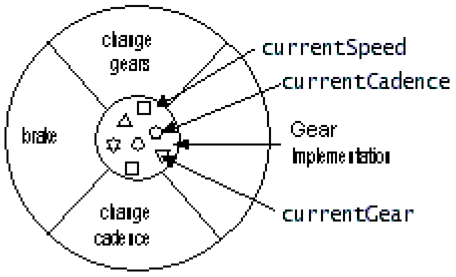
---

**Definition:** A class is a blueprint, or prototype, that defines the variables and the methods common to all objects of a certain kind.

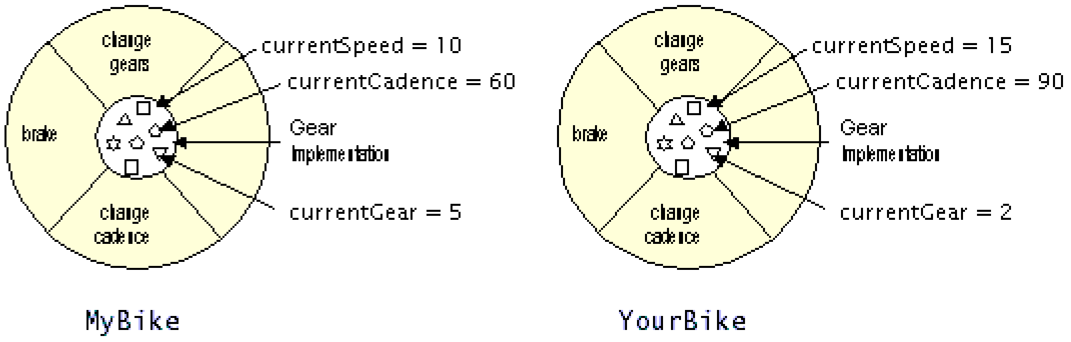
---



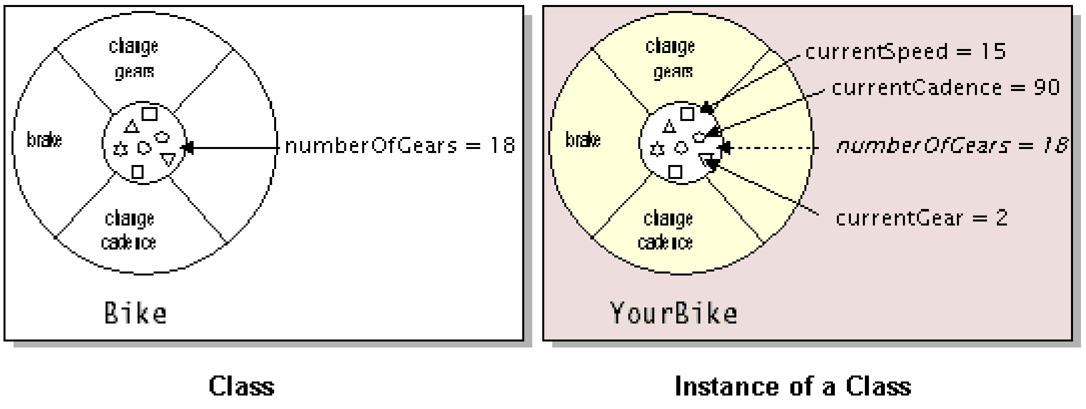
The class for our bicycle example would declare the instance variables necessary to contain the current gear, the current cadence, and so on, for each bicycle object. The class would also declare and provide implementations for the instance methods that allow the rider to change gears, brake, and change the pedaling cadence, as shown in the next figure.



After you’ve created the bicycle class, you can create any number of bicycle objects from the class. When you create an instance of a class, the system allocates enough memory for the object and all its instance variables. Each instance gets its own copy of all the instance variables defined in the class.



In addition to instance variables, classes can define *class variables*◆. A class variable contains information that is shared by all instances of the class. For example, suppose that all bicycles had the same number of gears. In this case, defining an instance variable to hold the number of gears is inefficient; each instance would have its own copy of the variable, but the value would be the same for every instance. In such situations, you can define a class variable that contains the number of gears. All instances share this variable. If one object changes the variable, it changes for all other objects of that type. A class can also declare *class methods*◆. You can invoke a class method directly from the class, whereas you must invoke instance methods on a particular instance.



Understanding Instance and Class Members◆ discusses instance variables and methods and class variables and methods in detail.

## Objects vs. Classes

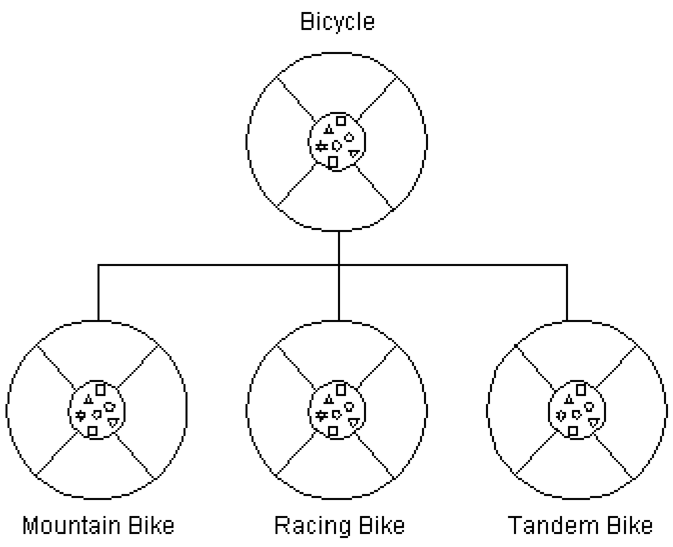
You probably noticed that the illustrations of objects and classes look very similar. And indeed, the difference between classes and objects is often the source of some confusion. In the real world, it’s obvious that classes are not themselves the objects they describe: A blueprint of a bicycle is not a bicycle. However, it’s a little more difficult to differentiate classes and objects in software. This is partially because software objects are merely electronic models of real-world objects or abstract concepts in the first place. But it’s also because the term “object” is sometimes used to refer to both classes and instances.

In the figures, the class is not shaded, because it represents a blueprint of an object rather than an object itself. In comparison, an object is shaded, indicating that the object exists and that you can use it.

## What Is Inheritance?

Generally speaking, objects are defined in terms of classes. You know a lot about an object by knowing its class. Even if you don't know what a penny-farthing is, if I told you it was a bicycle, you would know that it had two wheels, handle bars, and pedals.

Object-oriented systems take this a step further and allow classes to be defined in terms of other classes. For example, mountain bikes, racing bikes, and tandems are all kinds of bicycles. In object-oriented terminology, mountain bikes, racing bikes, and tandems are all *subclasses*◆ of the bicycle class. Similarly, the bicycle class is the *superclass*◆ of mountain bikes, racing bikes, and tandems. This relationship is shown in the following figure.



Each subclass *inherits*◆ state (in the form of variable declarations) from the superclass. Mountain bikes, racing bikes, and tandems share some states: cadence, speed, and the like. Also, each subclass inherits methods from the superclass. Mountain bikes, racing bikes, and tandems share some behaviors: braking and changing pedaling speed, for example.

However, subclasses are not limited to the state and behaviors provided to them by their superclass. Subclasses can add variables and methods to the ones they inherit from the superclass. Tandem bicycles have two seats and two sets of handle bars; some mountain bikes have an extra set of gears with a lower gear ratio.

Subclasses can also *override*◆ inherited methods and provide specialized implementations for those methods. For example, if you had a mountain bike with an extra set of gears, you would override the "change gears" method so that the rider could use those new gears.

You are not limited to just one layer of inheritance. The inheritance tree, or class *hierarchy*◆, can be as deep as needed. Methods and variables are inherited down through the levels. In general, the farther down in the hierarchy a class appears, the more specialized its behavior.

The Object class is at the top of class hierarchy, and each class is its descendant (directly or indirectly). A variable of type Object can hold a reference to any object, such as an instance of a class or an array. Object provides behaviors that are required of all objects running in the Java Virtual Machine. For example, all classes inherit Object's toString method, which returns a string representation of the object.

Inheritance offers the following benefits:

- Subclasses provide specialized behaviors from the basis of common elements provided by the superclass. Through the use of inheritance, programmers can reuse the code in the superclass many times.
- Programmers can implement superclasses called *abstract classes* that define “generic” behaviors. The abstract superclass defines and may partially implement the behavior, but much of the class is undefined and unimplemented. Other programmers fill in the details with specialized subclasses.

## What Is an Interface?

In English, an interface is a device or a system that unrelated entities use to interact. According to this definition, a remote control is an interface between you and a television set, the English language is an interface between two people, and the protocol of behavior enforced in the military is the interface between people of different ranks. Within the Java programming language, an *interface* is a device that unrelated objects use to interact with each other. An interface is probably most analogous to a protocol (an agreed on behavior). In fact, other object-oriented languages have the functionality of interfaces, but they call their interfaces protocols.

The bicycle class and its class hierarchy defines what a bicycle can and cannot do in terms of its “bicycleness.” But bicycles interact with the world on other terms. For example, a bicycle in a store could be managed by an inventory program. An inventory program doesn’t care what class of items it manages as long as each item provides certain information, such as price and tracking number. Instead of forcing class relationships on otherwise unrelated items, the inventory program sets up a protocol of communication. This protocol comes in the form of a set of constant and method definitions contained within an interface. The inventory interface would define, but not implement, methods that set and get the retail price, assign a tracking number, and so on.

To work in the inventory program, the bicycle class must agree to this protocol by implementing the interface. When a class implements an interface, the class agrees to implement all the methods defined in the interface. Thus, the bicycle class would provide the implementations for the methods that set and get retail price, assign a tracking number, and so on.

You use an interface to define a protocol of behavior that can be implemented by any class anywhere in the class hierarchy. Interfaces are useful for the following:

- Capturing similarities among unrelated classes without artificially forcing a class relationship.
- Declaring methods that one or more classes are expected to implement.
- Revealing an object’s programming interface without revealing its class.

## Controlling Access to Members of a Class

One of the benefits of classes is that classes can protect their member variables and methods from access by other objects. Why is this important? Well, consider this. You're writing a class that represents a query on a database that contains all kinds of secret information, say employee records or income statements for your startup company.

Certain information and queries contained in the class, the ones supported by the publicly accessible methods and variables in your query object, are OK for the consumption of any other object in the system. Other queries contained in the class are there simply for the personal use of the class. They support the operation of the class but should not be used by objects of another type--you've got secret information to protect. You'd like to be able to protect these personal variables and methods at the language level and disallow access by objects of another type.

In Java, you can use access specifiers to protect both a class's variables and its methods when you declare them. The Java language supports four distinct access levels for member variables and methods: private, protected, public, and, if left unspecified, package.

---

**Note:** The 1.0 release of the Java language supported five access levels: the four listed above plus private protected. The private protected access level is not supported in versions of Java higher than 1.0; you should no longer be using it in your Java programs.

---

The following chart shows the access level permitted by each specifier.

| Specifier | class | subclass | package | world |
|---|---|---|---|---|
| private | X | | | |
| protected | X | X* | X | |
| public | X | X | X | X |
| package | X | | X | |

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

Note that the protected/subclass intersection has an '*' . This particular access case has a special caveat discussed in detail later.

Let's look at each access level in more detail.

## Private

The most restrictive access level is private. A private member is accessible only to the class in which it is defined. Use this access to declare members that should only be used by the class. This includes variables that contain information that if accessed by an outsider could put the object in an inconsistent state, or methods that, if invoked by an outsider, could jeopardize the state of the object or the program in which it's running. Private members are like secrets you never tell anybody.

To declare a private member, use the private keyword in its declaration. The following class contains one private member variable and one private method:

```
class Alpha {
    private int iamprivate;
    private void privateMethod() {
        System.out.println("privateMethod");
    }
}
```

Objects of type Alpha can inspect or modify the iamprivate variable and can invoke privateMethod, but objects of other types cannot. For example, the Beta class defined here:

```
class Beta {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iamprivate = 10;      // illegal
        a.privateMethod();      // illegal
    }
}
```

cannot access the iamprivate variable or invoke privateMethod on an object of type Alpha because Beta is not of type Alpha.

When one of your classes is attempting to access a member varible to which it does not have access, the compiler prints an error message similar to the following and refuses to compile your program:

```
Beta.java:9: Variable iamprivate in class Alpha not
accessible from class Beta.
        a.iamprivate = 10;     // illegal
         ^
1 error
```

Also, if your program is attempting to access a method to which it does not have access, you will see a compiler error like this:

```
Beta.java:12: No method matching privateMethod()
found in class Alpha.
        a.privateMethod();         // illegal
1 error
```

New Java programmers might ask if one Alpha object can access the private members of another Alpha object. This is illustrated by the following example. Suppose the Alpha class contained an instance method that compared the current Alpha object (this) to another object based on their iamprivate variables:

```
class Alpha {
    private int iamprivate;
    boolean isEqualTo(Alpha anotherAlpha) {
        if (this.iamprivate == anotherAlpha.iamprivate)
            return true;
        else
            return false;
    }
}
```

This is perfectly legal. Objects of the same type have access to one another's private members. This is because access restrictions apply at the class or type level (all instances of a class) rather than at the object level (this particular instance of a class).

---

**Note:** this is a Java language keyword that refers to the current object. For more information about how to use this see The Method Body.

---

## Protected

The next access level specifier is protected, which allows the class itself, subclasses (with the caveat that we referred to earlier), and all classes in the same package to access the members. Use the protected access level when it's appropriate for a class's subclasses to have access to the member, but not unrelated classes. Protected members are like family secrets--you don't mind if the whole family knows, and even a few trusted friends but you wouldn't want any outsiders to know.

To declare a protected member, use the keyword protected. First, let's look at how the protected specifier affects access for classes in the same package. Consider this version of the Alpha class which is now declared to be within a package named Greek and which has one protected member variable and one protected method declared in it:

```
package Greek;

public class Alpha {
    protected int iamprotected;
    protected void protectedMethod() {
        System.out.println("protectedMethod");
    }
}
```

Now, suppose that the class Gamma was also declared to be a member of the Greek package (and is not a subclass of Alpha). The Gamma class can legally access an Alpha object's iamprotected member variable and can legally invoke its protectedMethod:

```
package Greek;

class Gamma {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iamprotected = 10;    // legal
        a.protectedMethod();    // legal
    }
}
```

That's pretty straightforward. Now, let's investigate how the protected specifier affects access for subclasses of Alpha.

Let's introduce a new class, Delta, that derives from Alpha but lives in a different package--Latin. The Delta class can access both iamprotected and protectedMethod, but only on objects of type Delta or its subclasses. The Delta class cannot access iamprotected or protectedMethod on objects of type Alpha. accessMethod in the following code sample attempts to access the iamprotected member variable on an object of type Alpha, which is illegal, and on an object of type Delta, which is legal. Similarly, accessMethod attempts to invoke an Alpha object's protectedMethod which is also illegal:

```
package Latin;

import Greek.*;

class Delta extends Alpha {
    void accessMethod(Alpha a, Delta d) {
        a.iamprotected = 10;    // illegal
        d.iamprotected = 10;    // legal
        a.protectedMethod();    // illegal
        d.protectedMethod();    // legal
    }
}
```

If a class is both a subclass of and in the same package as the class with the protected member, then the class has access to the protected member.

## Public

The easiest access specifier is public. Any class, in any package, has access to a class's public members. Declare public members only if such access cannot produce undesirable results if an outsider uses them. There are no personal or family secrets here; this is for stuff you don't mind anybody else knowing.

To declare a public member, use the keyword public. For example,

```
package Greek;

public class Alpha {
    public int iampublic;
    public void publicMethod() {
        System.out.println("publicMethod");
    }
}
```

Let's rewrite our Beta class one more time and put it in a different package than Alpha and make sure that it is completely unrelated to (not a subclass of) Alpha:

```
package Roman;

import Greek.*;

class Beta {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iampublic = 10;       // legal
        a.publicMethod();       // legal
    }
}
```

As you can see from the above code snippet, Beta can legally inspect and modify the iampublic variable in the Alpha class and can legally invoke publicMethod.

## Package

The package access level is what you get if you don't explicitly set a member's access to one of the other levels. This access level allows classes in the same package as your class to access the members. This level of access assumes that classes in the same package are trusted friends. This level of trust is like that which you extend to your closest friends but wouldn't trust even to your family.

For example, this version of the Alpha class declares a single package-access member variable and a single package-access method. Alpha lives in the Greek package:

```
package Greek;

class Alpha {
    int iampackage;
    void packageMethod() {
        System.out.println("packageMethod");
    }
}
```

The Alpha class has access both to iampackage and packageMethod. In addition, all the classes declared within the same package as Alpha also have access to iampackage and packageMethod. Suppose that both Alpha and Beta were declared as part of the Greek package:

```
package Greek;

class Beta {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iampackage = 10;     // legal
        a.packageMethod();     // legal
    }
}
```

Beta can legally access iampackage and packageMethod as shown.

# Understanding Instance and Class Members

When you declare a member variable such as aFloat in MyClass:

```
class MyClass {
   float aFloat;
}
```

you declare an *instance variable*. Every time you create an instance of a class, the runtime system creates one copy of each class's instance variables for the instance. You can access an object's instance variables from an object as described in Using Objects◆.

Instance variables are in contrast to *class variables* (which you declare using the static modifier). The runtime system allocates class variables once per class regardless of the number of instances created of that class. The system allocates memory for class variables the first time it encounters the class. All instances share the same copy of the class's class variables. You can access class variables through an instance or through the class itself.

Methods are similar: Your classes can have instance methods and class methods. Instance methods operate on the current object's instance variables but also have access to the class variables. Class methods, on the other hand, cannot access the instance variables declared within the class (unless they create a new object and access them through the object). Also, class methods can be invoked on the class, you don't need an instance to call a class method.

By default, unless otherwise specified, a member declared within a class is an instance member. The class defined below has one instance variable--an integer named x--and two instance methods--x and setX--that let other objects set and query the value of x:

```
class AnIntegerNamedX {
   int x;
   public int x() {
      return x;
   }
   public void setX(int newX) {
      x = newX;
   }
}
```

Every time you instantiate a new object from a class, you get a new copy of each of the class's instance variables. These copies are associated with the new object. So, every time you instantiate a new AnIntegerNamedX object from the class, you get a new copy of x that is associated with the new AnIntegerNamedX object.

All instances of a class share the same implementation of an instance method; all instances of AnIntegerNamedX share the same implementation of x and setX. Note that both methods, x and setX, refer to the object's instance variable x by name. "But", you ask, "if all instances of AnIntegerNamedX share the same implementation of x and setX isn't this ambiguous?" The answer is "no." Within an instance method, the name of an instance variable refers to the current object's instance variable, assuming that the instance variable isn't hidden by a method parameter. So, within x and setX, x is equivalent to this.x.

Objects outside of AnIntegerNamedX that wish to access x must do so through a particular instance of AnIntegerNamedX. Suppose that this code snippet was in another object's method. It creates two different objects of type AnIntegerNamedX, sets their x values to different values, then displays them:

```
. . .
AnIntegerNamedX myX = new AnIntegerNamedX();
AnIntegerNamedX anotherX = new AnIntegerNamedX();
myX.setX(1);
```

```
anotherX.x = 2;
System.out.println("myX.x = " + myX.x());
System.out.println("anotherX.x = " + anotherX.x());
. . .
```

Notice that the code used setX to set the x value for myX but just assigned a value to anotherX.x directly. Either way, the code is manipulating two different copies of x: the one contained in the myX object and the one contained in the anotherX object. The output produced by this code snippet is:

```
myX.x = 1
anotherX.x = 2
```

showing that each instance of the class AnIntegerNamedX has its own copy of the instance variable x and each x has a different value.

You can, when declaring a member variable, specify that the variable is a class variable rather than an instance variable. Similarly, you can specify that a method is a class method rather than an instance method. The system creates a single copy of a class variable the first time it encounters the class in which the variable is defined. All instances of that class share the same copy of the class variable. Class methods can only operate on class variables--they cannot access the instance variables defined in the class.

To specify that a member variable is a class variable, use the static keyword. For example, let's change the AnIntegerNamedX class such that its x variable is now a class variable:

```
class AnIntegerNamedX {
    static int x;
    public int x() {
        return x;
    }
    public void setX(int newX) {
        x = newX;
    }
}
```

Now the exact same code snippet from before that creates two instances of AnIntegerNamedX, sets their x values, and then displays them produces this, different, output.

```
myX.x = 2
anotherX.x = 2
```

The output is different because x is now a class variable so there is only one copy of the variable and it is shared by all instances of AnIntegerNamedX, including myX and anotherX. When you invoke setX on either instance, you change the value of x for all instances of AnIntegerNamedX.

You use class variables for items that you need only one copy of and which must be accessible by all objects inheriting from the class in which the variable is declared. For example, class variables are often used with final to define constants; this is more memory efficient than final instance variables because constants can't change, so you really only need one copy).

Similarly, when declaring a method, you can specify that method to be a class method rather than an instance method. Class methods can only operate on class variables and cannot access the instance variables defined in the class.

To specify that a method is a class method, use the static keyword in the method declaration. Let's change the AnIntegerNamedX class such that its member variable x is once again an instance variable, and its two methods are now class methods:

```
class AnIntegerNamedX {
    int x;
    static public int x() {
        return x;
    }
    static public void setX(int newX) {
        x = newX;
    }
}
```

When you try to compile this version of AnIntegerNamedX, the compiler displays an error like this one:

```
AnIntegerNamedX.java:4: Can't make a static reference to
nonstatic variable x in class AnIntegerNamedX.
        return x;
               ^
```

This is because class methods cannot access instance variables unless the method created an instance of AnIntegerNamedX first and accessed the variable through it.

Let's fix AnIntegerNamedX by making its x variable a class variable:

```
class AnIntegerNamedX {
    static int x;
    static public int x() {
        return x;
    }
    static public void setX(int newX) {
        x = newX;
    }
}
```

Now the class will compile and the same code snippet from before that creates two instances of AnIntegerNamedX, sets their x values, and then prints the x values produces this output:

```
myX.x = 2
anotherX.x = 2
```

Again, changing x through myX also changes it for other instances of AnIntegerNamedX.

Another difference between instance members and class members is that class members are accessible from the class itself. You don't need to instantiate a class to access its class members. Let's rewrite the code snippet from before to access x and setX directly from the AnIntegerNamedX class:

```
. . .
AnIntegerNamedX.setX(1);
System.out.println("AnIntegerNamedX.x = " + AnIntegerNamedX.x());
. . .
```

Notice that you no longer have to create myX and anotherX. You can set x and retrieve x directly from the AnIntegerNamedX class. You cannot do this with instance members, you can only invoke instance methods from an object and can only access instance variables from an object. You can access class variables and methods either from an instance of the class or from the class itself.

## Initializing Instance and Class Members

You can use static initializers and instance initializers to provide initial values for class and instance members when you declare them in a class:

```
class BedAndBreakfast {
    static final int MAX_CAPACITY = 10;
    boolean full = false;
}
```

This works well for members of primitive data type. Sometimes, it even works when creating arrays and objects. But this form of initialization has limitations, as follows:

1. Initializers can perform only initializations that can be expressed in an assignment statement.

2. Initializers cannot call any method that can throw a checked exception.
3. If the initializer calls a method that throws a runtime exception, then it cannot do error recovery.

If you have some initialization to perform that cannot be done in an initializer because of one of these limitations, you have to put the initialization code elsewhere. To initialize class members, put the initialization code in a static initialization block. To initialize instance members, put the initialization code in a constructor.

## Using Static Initialization Blocks

Here's an example of a static initialization block:

```
import java.util.ResourceBundle;
class Errors {
    static ResourceBundle errorStrings;
    static {
        try {
            errorStrings = ResourceBundle.
                                 getBundle("ErrorStrings");
        } catch (java.util.MissingResourceException e) {
            // error recovery code here
        }
    }
}
```

The errorStrings resource bundle must be initialized in a static initialization block. This is because error recovery must be performed if the bundle cannot be found. Also, errorStrings is a class member and it doesn't make sense for it to be initialized in a constructor. As the previous example shows, a static initialization block begins with the static keyword and is a normal block of Java code enclosed in curly braces {}.

A class can have any number of static initialization blocks that appear anywhere in the class body. The runtime system guarantees that static initialization blocks and static initializers are called in the order (left-to-right, top-to-bottom) that they appear in the source code.

## Initializing Instance Members

If you want to initialize an instance variable and cannot do it in the variable declaration for the reasons cited previously, then put the initialization in the constructor(s) for the class. Suppose the errorStrings bundle in the previous example is an instance variable rather than a class variable. Then you'd use the following code to initialize it:

```
import java.util.ResourceBundle;
class Errors {
    ResourceBundle errorStrings;
    Errors() {
        try {
            errorStrings = ResourceBundle.
                                getBundle("ErrorStrings");
        } catch (java.util.MissingResourceException e) {
            // error recovery code here
        }
    }
}
```

The code that initializes errorStrings is now in a constructor for the class.

Sometimes a class contains many constructors and each constructor allows the caller to provide initial values for different instance variables of the new object. For example, java.awt.Rectangle has these three constructors:

```
Rectangle();
Rectangle(int width, int height);
Rectangle(int x, int y, int width, int height);
```

The no-argument constructor doesn't let the caller provide initial values for anything, and the other two constructors let the caller set initial values either for the size or for the origin and size. Yet, all of the instance variables, the origin and the size, for Rectangle must be initialized. In this case, classes often have one constructor that does all of the work. The other constructors call this constructor and provide it either with the values from their parameters or with default values. For example, here are the possible implementations of the three Rectangle constructors shown previously (assume x, y, width, and height are the names of the instance variables to be initialized):

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

The Java language supports instance initialization blocks, which you could use instead. However, these are intended to be used with anonymous classes, which cannot declare constructors.

The approach described here that uses constructors is better for these reasons:

- All of the initialization code is in one place, thus making the code easier to maintain and read.
- Defaults are handled explicitly.

Constructors are widely understood by the Java community, including relatively new Java programmers, while instance initializers are not and may cause confusion to others reading your code.

## How Do These Concepts Translate into Code?

Now that you have a conceptual understanding of object-oriented programming let's look at how these concepts get translated into code.

Here is an applet named ClickMe. A red spot appears when you click the mouse within the applet's bounds.

**Note:** If you don't see the applet running above, you need to install Java Plug-in, which happens automatically when you install the J2SE JRE or SDK. We strongly recommend that you install the latest version; at least 1.3.1 is required for all our applets. You can find more information in the Java Plug-in home page.

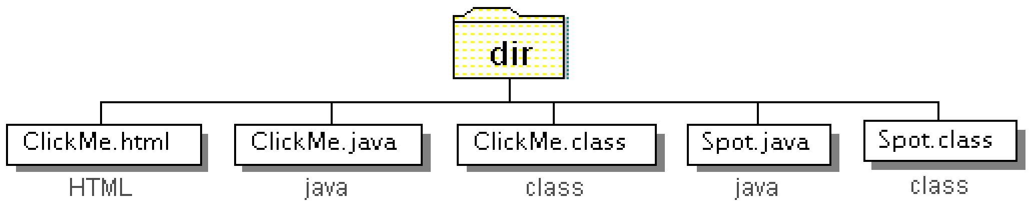
The ClickMe applet is a relatively simple program and the code for it is short. However, if you don't have much experience with programming, you might find the code daunting. We don't expect you to understand everything in this program right away, and this section won't explain every detail. The intent is to expose you to some source code and to associate it with the concepts and terminology you just learned. You will learn the details in later trails.

### The Source Code and the Applet Tag for ClickMe

To compile this applet you need two source files: ClickMe.java and Spot.java. To run the applet you need to create an html file with this applet tag in it:

```
<applet code="ClickMe.class"
width="300" height="150">
</applet>
```

Then load the page into your browser or the appletviewer tool. Make sure all the necessary files are in the same directory.



**Note:** If you're having problems running this example:

- Make sure you download *both* Spot.java and ClickMe.java and place them in the same directory.
- If you can compile one file but not the other, you probably have a CLASSPATH problem. If "." (indicating your current directory) isn't in your class path, the ClickMe class can't find the Spot class, even though they are in the same directory. Try unsetting CLASSPATH and recompiling, like this:
  - set CLASSPATH=
  - javac ClickMe.java

If your problem is fixed, then you should modify your global CLASSPATH variable to add "." to it. See Update the PATH variable (for Microsoft Windows) (or the appropriate documentation for your platform).

### Objects in the ClickMe Applet

Many objects play a part in this applet. The two most obvious ones are the ones you can see: the applet itself and the spot, which is red on-screen.

The browser creates the applet object when it encounters the applet tag in the HTML code containing the applet. The applet tag provides the name of the class from which to create the applet object. In this case, the class name is ClickMe..

The ClickMe.applet in turn creates an object to represent the spot on the screen. Every time you click the mouse in the applet, the applet moves the spot by changing the object's x and y location and repainting itself. The spot does not draw itself; the applet draws the spot, based on information contained within the spot object.

Besides these two obvious objects, other, nonvisible objects play a part in this applet. Three objects represent the three colors used in the applet (black, white, and red); an event object represents the user action of clicking the mouse, and so on.

## Classes in the ClickMe Applet

Because the object that represents the spot on the screen is very simple, let's look at its class, named Spot. It declares three instance variables: size contains the spot's radius, x contains the spot's current horizontal location, and y contains the spot's current vertical location:

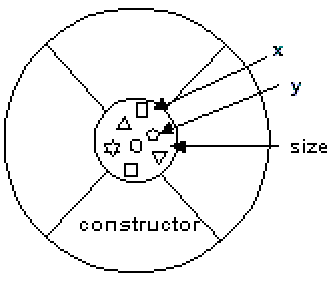
```
public class Spot {
//instance variables
public int size;
public int x, y;

//constructor
public Spot(int intSize) {
size = intSize;
x = -1;
y = -1;
   }
}
```

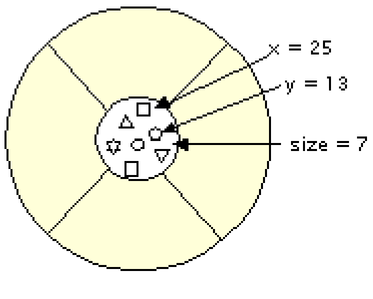
Additionally, the class has a *constructor*-- a subroutine used to initialize new objects created from the class. You can recognize a constructor because it has the same name as the class. The constructor initializes all three of the object's variables. The initial value of size is provided as an argument to the constructor by the caller. The x and y variables are set to -1 indicating that the spot is not on-screen when the applet starts up.

The applet creates a new spot object when the applet is initialized. Here's the relevant code from the applet class:

```
private Spot spot = null;
private static final int RADIUS = 7;
...
spot = new Spot(RADIUS);
```

The first line shown declares a variable named spot whose data type is Spot, the class from which the object is created, and initializes the variable to null. The second line declares an integer variable named RADIUS whose value is 7. Finally, the last line shown creates the object; new allocates memory space for the object. Spot(RADIUS) calls the constructor you saw previously and passes in the value of RADIUS. Thus the spot object's size is set to 7





The figure on the left is a representation of the Spot class. The figure on the right is a spot object.

## Messages in the ClickMe Applet

As you know, object A can use a message to request that object B do something, and a message has three components:

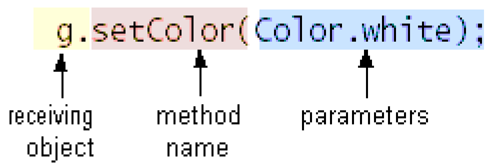
1. The object to which the message is addressed
2. The name of the method to perform
3. Any parameters the method needs

Here are two lines of code from the ClickMe applet:

```
g.setColor(Color.white);
g.fillRect(0, 0, getSize().width - 1, getSize().height - 1);
```

Both are messages from the applet to an object named g—a Graphics object that knows how to draw simple on-screen shapes and text. This object is provided to the applet when the browser instructs the applet to draw itself. The first line sets the color to white; the second fills a rectangle the size of the applet, thus painting the extent of the applet's area white.

The following figure highlights each message component in the first message:



## Inheritance in the ClickMe Applet

To run in a browser, an object must be an applet. This means that the object must be an instance of a class that derives from the Applet class provided by the Java platform.

The ClickMe applet object is an instance of the ClickMe class, which is declared like this:

```
public class ClickMe extends Applet implements MouseListener {
   ...
}
```

The extends Applet clause makes ClickMe a subclass of Applet. ClickMe inherits a lot of capability from its superclass, including the ability to be initialized, started, and stopped by the browser, to draw within an area on a browser page, and to register to receive mouse events. Along with these benefits, the ClickMe class has certain obligations: its painting code must be in a method called paint, its initialization code must be in a method called init, and so on.

```
public void init() {
... // ClickMe's initialization code here
}

public void paint(Graphics g) {
... // ClickMe's painting code here
}
```

## Interfaces in the ClickMe Applet

The ClickMe applet responds to mouse clicks by displaying a red spot at the click location. If an object wants to be notified of mouse clicks, the Java platform event system requires that the object implement the MouseListener interface. The object must also register as a mouse listener.

The MouseListener interface declares five different methods each of which is called for a different kind of mouse event: when the mouse is clicked, when the mouse moves outside of the applet, and so on. Even though the applet is interested only in mouse clicks it must implement all five methods. The methods for the events that it isn't interested in are empty.

The complete code for the ClickMe applet is shown below. The code that participates in mouse event handling is red:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class ClickMe extends Applet implements MouseListener {
private Spot spot = null;
private static final int RADIUS = 7;
public void init() {
addMouseListener(this);
    }

public void paint(Graphics g) {
// draw a black border and a white background
g.setColor(Color.white);
g.fillRect(0, 0, getSize().width - 1, getSize().height - 1);
g.setColor(Color.black);
g.drawRect(0, 0, getSize().width - 1, getSize().height - 1);
// draw the spot
g.setColor(Color.red);
if (spot != null) {
g.fillOval(spot.x - RADIUS,
spot.y - RADIUS,
RADIUS * 2, RADIUS * 2);
        }
    }
public void mousePressed(MouseEvent event) {
if (spot == null) {
spot = new Spot(RADIUS);
        }
spot.x = event.getX();
spot.y = event.getY();
repaint();
    }
public void mouseClicked(MouseEvent event) {}
public void mouseReleased(MouseEvent event) {}
public void mouseEntered(MouseEvent event) {}
public void mouseExited(MouseEvent event) {}
}
```

## API Documentation

The ClickMe applet inherits a lot of capability from its superclass. To learn more about how ClickMe works, you need to learn about its superclass, Applet. How do you find that information? You can find detailed descriptions of every class in the API documentation, which constitute the specification for the classes that make up the Java platform.

The API documentation for the Java 2 Platform is online at java.sun.com. It's helpful to have the API documentation for all releases you use bookmarked in your browser.

- API documents for Java 2 Platform, Standard Edition, v1.5
- API documents for Java 2 Platform, Standard Edition, v1.4
- API documents for Java 2 Platform, Standard Edition, v1.3
- API documents for Java 2 Platform, Standard Edition, v1.2
- API documents for JDK 1.1

To learn more about all the classes and interfaces from the Java platform used by the ClickMe applet, you can look at the API documentation for these classes:

- java.applet.Applet
- java.awt.Graphics
- java.awt.Color
- java.awt.event.MouseListener
- java.awt.event.MouseEvent

## Summary

This discussion glossed over many details and left some things unexplained, but you should have some understanding now of what object-oriented concepts look like in code. You should now have a general understanding of the following:

- That a class is a prototype for objects
- That objects are created from classes
- That an object's class is its type

- How to create an object from a class
- What constructors are
- How to initialize objects
- What the code for a class looks like
- What class variables and methods are
- What instance variables and methods are
- How to find out what a class's superclass is
- That an interface is a protocol of behavior
- What it means to implement an interface

## Questions and Exercises: Object-Oriented Concepts

### Questions

Use the API documentation for the Java 2 Platform to answer these questions:

1. The ClickMe applet uses Color.red to set the drawing color to red. What other colors can you get by name like this?
2. How would you specify other colors, like purple?

### Exercises

Now, use what you learned from the javadocs to make the following modifications to the ClickMe applet. To compile this program, you also need the Spot.java file.

**Note:** This is an exercise in reading code, making guesses, and looking up classes and methods in the javadocs. We haven't yet given you any detailed information about how to write Java code and what it all means. But you might be surprised at how much you can intuit from the code. The main point of this exercise is to get you to use the javadocs and to read code. Do not worry if you cannot get your solution to work. Do not worry if your solutions do not match ours. Do compare your solutions to ours so that you can get more experience reading code and perhaps learn more about the API in the Java platform.

1. Modify the applet to draw a green square instead of a red spot.
2. Modify the applet to display your name in purple instead of a red spot.

**http://java.sun.com/docs/books/tutorial/java/concepts/**