# Lesson: Object-Oriented Programming Concepts

If you've never programmed using an object-oriented language before, you need to understand what objects and classes are and how these concepts translate to code.

This lesson provides the conceptual basis for object-oriented languages in general. The last section shows you how these general concepts translate into code for an applet written in the Java programming language.

## What Is an Object?

An object is a software bundle of related variables and methods. Software objects are often used to model real-world objects you find in everyday life.

## What Is a Message?

Software objects interact and communicate with each other using messages.

## What Is a Class?

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

## What Is Inheritance?

A class inherits state and behavior from its superclass. Inheritance provides a powerful and natural mechanism for organizing and structuring software programs.

## What Is an Interface?

An interface is a contract in the form of a collection of method and constant declarations. When a class implements an interface, it promises to implement all of the methods declared in that interface.

## How Do These Concepts Translate into Code?

This section looks at a small applet, and shows you the code that creates objects, implements classes, sends messages, establishes a superclass, and implements an interface.

# Questions and Exercises: Object-Oriented Concepts

Test your understanding of objects, classes, messages, and so on by doing some exercises and answering some questions.


Start of Tutorial > Start of Trail

Search
Feedback Form

# What Is an Object?

As the name *object-oriented* implies, *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 they all have *behavior*. For example, dogs have state (name, color, breed, hungry) and dogs have behavior (barking, fetching, and slobbering on your newly cleaned slacks). 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 *variables* and implements its behavior with *methods*.

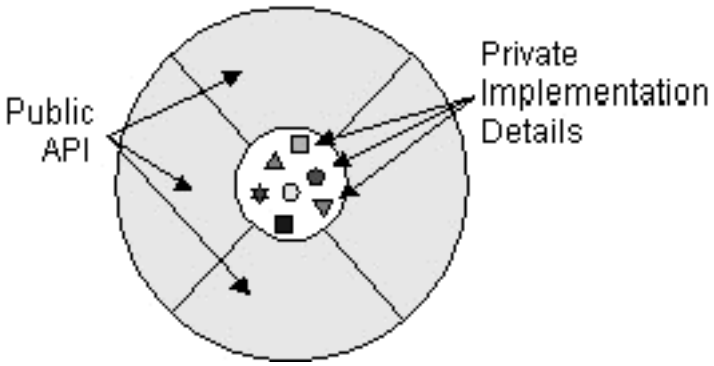
---

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

---

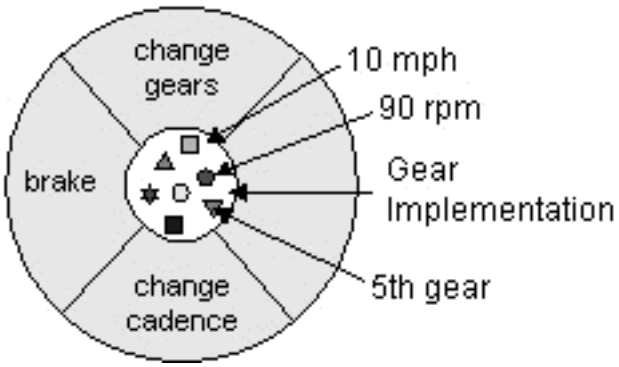
You can represent real-world objects using software objects. You might want to represent real-world dogs as software objects in an animation program or a real-world bicycle as a software object in the program that controls an electronic exercise bike. However, you can also use software objects to model 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 methods within that object. A software object that modelled 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 and methods

are formally known as *instance variables* and *instance methods* to distinguish them from class variables and class methods (described later in What Is a Class?). The following figure illustrates a bicycle modeled as a software object.




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 really 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.)

Anything that an object does not know or cannot do is excluded from the object. For example, your bicycle (probably) doesn't have a name, and it can't run, bark, or fetch. Thus there are no variables or methods for those states and behaviors in the bicycle class.

As you can see from the diagrams, 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*. Typically, encapsulation is used to hide unimportant implementation details from other objects. When you want to change gears on your bicycle, you don't need to know how the gear mechanism works, you just need to know which lever to move. Similarly in software programs, you don't need to know how a class is implemented, you just need to know which methods to invoke. Thus, the implementation details can change at any time without affecting other parts of the program.

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 implementation or efficiency reasons, an object may wish to expose some of its variables or hide some of its methods.

In many languages, including Java, an object can choose to expose its variables to other objects allowing those other objects to inspect and even modify the variables. Also, an object can choose to hide methods from other objects forbidding those objects from invoking the methods. An object has complete control over whether other objects can access its variables and methods and in fact, can specify which other objects have access. Variable and method access in Java is covered in Controlling Access to Members of a Class.

# The Benefits of Encapsulation

Encapsulating related variables and methods into a neat software bundle is a simple yet powerful idea that provides two primary benefits to software developers:

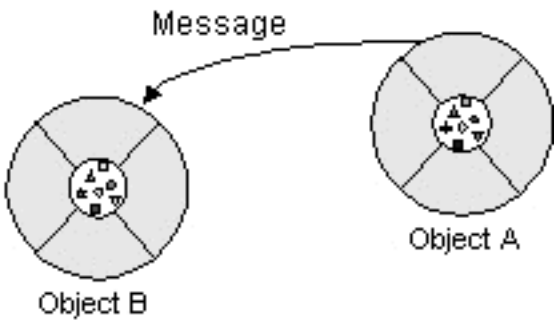
- Modularity--The source code for an object can be written and maintained independently of the source code for other objects. Also, an object can be easily passed around in the system. You can give your bicycle to someone else and it will still work.
- Information hiding--An object has a public interface that other objects can use to communicate with it. But 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 in order to use it.

**Trail**: Learning the Java Language
**Lesson**: Object-Oriented Programming Concepts

# What Is a Message?

A single object alone is generally not very useful and 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 (starts pedaling).

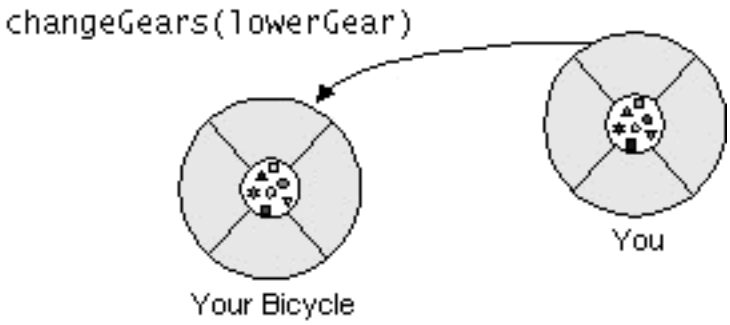
Software objects interact and communicate with each other by sending *messages* to each other. When object A wants object B to perform one of B's methods, object A sends a message to object B.




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*.

Three components comprise a message:

1. The object to whom the message is addressed (Your Bicycle)
2. The name of the method to perform (`changeGears`)
3. Any parameters needed by the method (lower gear)




These three components are enough information for the receiving object to perform the

desired method. No other information or context is required.

## The Benefits of Messages

- 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 other bicycles.

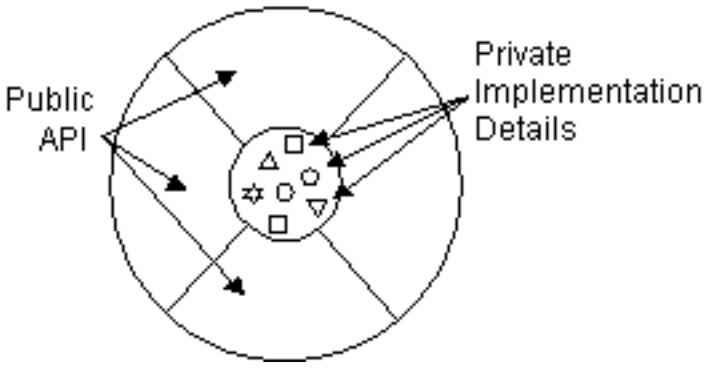
When building bicycles, manufacturers take advantage of the fact that bicycles share characteristics by building many bicycles from the same blueprint--it would be very inefficient to produce a new blueprint for every individual bicycle they manufactured.

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. Software "blueprints" for objects are called *classes*.

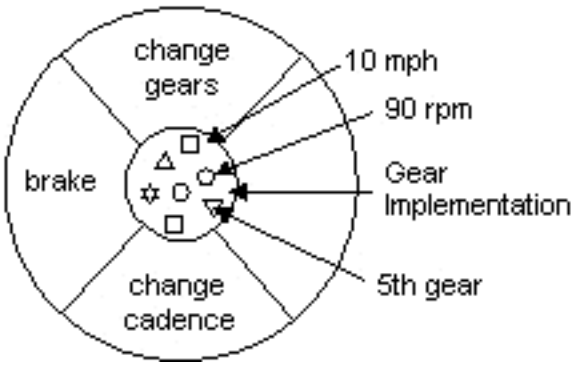
---

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

---




For example, you could create a bicycle class that declares several instance variables 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.




The values for instance variables are provided by each instance of the class. So, after you've created the bicycle class, you must *instantiate* it (create an instance of it) before you can use it. When you create an instance of a class, you create an object of that type and the system allocates memory for the instance variables declared by the class. Then you can invoke the object's instance methods to make it do something. Instances of the same class share the same instance method implementations (method implementations are not duplicated on a per object basis), which reside in the class itself.

In addition to instance variables and methods, classes can also define *class variables* and *class methods*. You can access class variables and methods from an instance of the class or directly from a class--you don't have to instantiate a class to use its class variables and methods. Class methods can only operate on class variables--they do not have access to instance variables or instance methods.

The system creates a single copy of all class variables for a class the first time it encounters the class in a program--all instances of that class share its class variables. For example, suppose that all bicycles had the same number of gears. In this case defining an instance variable for 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 situations such as this, you could 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.

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 to one another. 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 that 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 many people use the term "object" inconsistently and use it 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 actually

exists and you can use it.

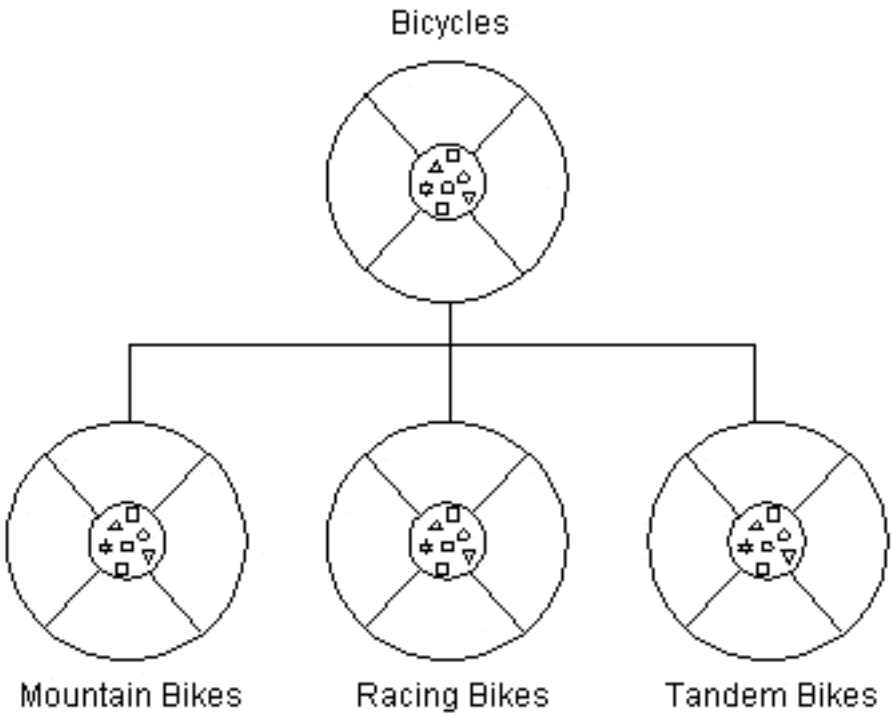
## The Benefit of Classes

Objects provide the benefit of modularity and information hiding. Classes provide the benefit of reusability. Bicycle manufacturers reuse the same blueprint over and over again to build lots of bicycles. Software programmers use the same class, and thus the same code, over and over again to create many objects.

# 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 different 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.




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. What would be the point in that? 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 actually 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 further down in the hierarchy a class appears, the more specialized its behavior.

## The Benefits of Inheritance

- 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.

**Trail**: Learning the Java Language
**Lesson**: Object-Oriented Programming Concepts

# What Is an Interface?

In English, an interface is a device or 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 one another. Interfaces are probably most analogous to protocols (an agreed-upon 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 bicycles can and cannot do in terms of its "bicycle-ness". But bicycles interact with the world on other terms. For example, a bicycle in a store is an inventory item with a retail price, a part number, a parts list, and so on. To set or get this sort of information from a bicycle object, an inventory program and the bicycle class must agree on a protocol of communication. This protocol comes in the form of an interface, let's call it `InventoryItem`, that contains method definitions. The `InventoryItem` interface would define methods such as `setRetailPrice`, `getRetailPrice`, and so on.

To work within the inventory program, the bicycle class must agree to this protocol by implementing the `InventoryItem` interface. When a class implements an interface, the class agrees to implement all of the methods defined in the interface. Thus, the bicycle class would have to implement `setRetailPrice`, `getRetailPrice`, and so on.

## The Benefit of Interfaces

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 between 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. (Objects such as these are called anonymous objects and can be useful when shipping a package of classes to other developers.)

# 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 the 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.

# Using Objects

Once you've created an object, you probably want to use it for something. You may need information from it, want to change its state, or have it perform some action.

Objects give you two ways to do these things:

1. Manipulate or inspect its variables.
2. Call its methods.

## Referencing an Object's Variables

The `CreateObjectDemo` program creates a rectangle named `rect_two`. The constructor used to create that rectangle initializes the rectangle's origin to 0, 0. Later the program changes the rectangle's origin with this statement:

```
rect_two.origin = origin_one;
```

This statement moves the rectangle by setting its point of origin to a new position. `rect_two.origin` is the name of `rect_two`'s `origin` variable. You can use these kinds of object variable names in the same manner as you use other variables names. Thus, as in the previous example code, you can use the `=` operator to assign a value to `rect_two.origin`.

The `Rectangle` class has two other variables-- `width` and `height`-- that are accessible to objects outside of the class. The program displays them with this code:

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

In general, to refer to an object's variables, append the name of the variable to an object reference with an intervening period (.):

```
objectReference.variable
```

The first part of the variable's name, *`objectReference`*, must be a reference to an object. You can use an object name here just as was done in the previous examples with `rect`. You also 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 variables:

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

This statement creates a new `Rectangle` object and immediately gets its height. Effectively, 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 `Rectangle` that was created because the program never stored the reference in a variable. Thus the object becomes eligible for garbage collection.

Here's a final word about accessing an object's variables to clear up a point of some confusion that beginning Java programmers often have. All objects of the same type have the same variables. All `Rectangle` objects have `origin`, `width`, and `height` variables that they got from the `Rectangle` class. When you access a variable through an object reference, you reference that particular object's variables. Suppose that `bob` is also a rectangle in your drawing program and it has a different height and width than `rect`. The following instruction calculates the area of the rectangle named `bob`, which differs from the previous instruction that calculated the area of `rect`:

```
area = bob.height * bob.width;
```

## Calling an Object's Methods

To move `rect` to a new location using its `move` method, you write this:

```
rect.move(15, 37);
```

This Java statement calls `rect`'s `move` method with two integer parameters, 15 and 37. It moves the `rect` object because the `rect` method assigns new values to `origin.x` and `origin.y` and is equivalent to the assignment statement used previously:

```
rect.origin = new Point(15, 37);
```

The notation used to call an object's method is similar to that used when referring to its variables: You append the method name to an object reference with an intervening period (.). Also, you provide any arguments to the method within enclosing parentheses. If the method does not require any arguments, use empty parentheses.

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

As stated previously in this lesson, *`objectReference`* must be a reference to an object. You can use a variable name here, 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 call a new object's methods:

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

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 call the new `Rectangle`'s `area` method to compute the area of the new rectangle.

Some methods, like `area`, return a value. For methods that return a value, you can use the method call 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 area to a variable:

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

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

## A Word About Variable Access

Ideal object-oriented programming discourages the direct manipulation of an object's variables because it would be possible to set the variables to values that don't make sense. For example, consider the `Rectangle` class from the previous section. Using that class, you can create a rectangle whose width and height are negative, which, for some applications, doesn't make sense.

Instead of allowing direct manipulation of an its variables, an ideal class provides methods through which you can inspect or change its variables. These methods ensure that the values of the variables make sense for objects of that type. So, the `Rectangle` class would provide methods for setting and getting the `width` and the `height`. The methods for setting the variables would report an error, if the caller tried to set the width or height to a negative number.

However, in practical situations, it sometimes makes sense to allow direct access to an object's variables. For example, both the `Point` class and the `Rectangle` class allow free access to their member variables. This keeps these classes small and simple. Also, it keeps them generally useful. Some applications might allow rectangles with negative widths and heights.

Java provides an access control mechanism whereby classes can determine which other classes can have direct access to its variables and methods. A class should protect variables against direct manipulation by other objects if those manipulations could result in values that don't make sense objects of that type. State changes should then be affected and therefore controlled by method calls. If an object grants access to its variables, you can assume that you can inspect and change them without adverse effects. To learn more about Java's access control mechanism, refer to Controlling Access to Members of a Class.

# The Method Body

In the code sample that follows, the method bodies for the `isEmpty` and `pop` methods are shown in bold:

```
class Stack {
    static final int STACK_EMPTY = -1;
    Object[] stackelements;
    int topelement = STACK_EMPTY;
    . . .
    boolean isEmpty() {
        if (topelement == STACK_EMPTY)
            return true;
        else
            return false;
    }
    Object pop() {
        if (topelement == STACK_EMPTY)
            return null;
        else {
            return stackelements[topelement--];
        }
    }
}
```

Besides regular Java language elements, you can use `this` in the method body to refer to members in the *current object*. The current object is the object whose method is being called. You can also use `super` to refer to members in the superclass that the current object has hidden or overridden. Also, a method body may contain declarations for variables that are local to that method.

## this

Typically, within an object's method body you can just refer directly to the object's member variables. However, sometimes you need to disambiguate the member variable name if one of the arguments to the method has the same name.

For example, the following constructor for the `HSBColor` class initializes some of an object's member variables according to the arguments passed into the constructor. Each

argument to the constructor has the same name as the object's member variable whose initial value the argument contains.

```
class HSBColor {
    int hue, saturation, brightness;
    HSBColor (int hue, int saturation, int brightness) {
        this.hue = hue;
        this.saturation = saturation;
        this.brightness = brightness;
    }
```

You must use `this` in this constructor because you have to disambiguate the argument `hue` from the member variable `hue` (and so on with the other arguments). Writing `hue = hue;` makes no sense. Argument names take precedence and hide member variables with the same name. So to refer to the member variable you must do so through the current object--`this`--explicitly.

Some programmers prefer to always use `this` when referring to a member variable of the object whose method the reference appears. Doing so makes the intent of the code explicit and reduces errors based on name sharing.

You can also use `this` to call one of the current object's methods. Again this is only necessary if there is some ambiguity in the method name and is often used to make the intent of the code clearer.

## super

If your method hides one of its superclass's member variables, your method can refer to the hidden variable through the use of `super`. Similarly, if your method overrides one of its superclass's methods, your method can invoke the overridden method through the use of `super`.

Consider this class:

```
class ASillyClass {
    boolean aVariable;
    void aMethod() {
        aVariable = true;
    }
}
```

and its subclass which hides `aVariable` and overrides `aMethod`:

```
class ASillierClass extends ASillyClass {
    boolean aVariable;
    void aMethod() {
        aVariable = false;
```

```
            super.aMethod();
            System.out.println(aVariable);
            System.out.println(super.aVariable);
        }
    }
```

First `aMethod` sets `aVariable` (the one declared in `ASillierClass` that hides the one declared in `ASillyClass`) to `false`. Next `aMethod` invoked its overridden method with this statement:

```
super.aMethod();
```

This sets the hidden version of the `aVariable` (the one declared in `ASillyClass`) to `true`. Then `aMethod` displays both versions of `aVariable` which have different values:

```
false
true
```

## Local Variables

Within the body of the method you can declare more variables for use within that method. These variables are *local variables* and live only while control remains within the method. This method declares a local variable `i` that it uses to iterate over the elements of its array argument.

```
Object findObjectInArray(Object o, Object[] arrayOfObjects) {
    int i;      // local variable
    for (i = 0; i < arrayOfObjects.length; i++) {
        if (arrayOfObjects[i] == o)
            return o;
    }
    return null;
}
```

After this method returns, `i` no longer exists.