**What is an OBJECT**

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

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

Take a minute right now to observe the real-world objects that are in your immediate area. For each object that you see, ask yourself two questions:

*"What possible states can this object be in?"*

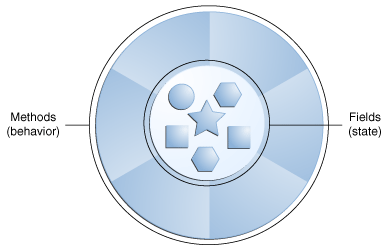
and

*"What possible behaviour can this object perform?”*

Make sure to write down your observations. As you do, you'll notice that real-world objects vary in complexity; your desktop lamp may have only two possible states (on and off) and two possible behaviours (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behaviour (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). You may also notice that some objects, in turn, will also contain other objects.

These real-world observations all translate into the world of object-oriented programming.

A software object.



Software objects are conceptually similar to real-world objects: they too consist of state and related behaviour.

An object stores its state in *fields* (variables in some programming languages) and exposes its behaviour through *methods* (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication.

Hiding internal state and requiring all interaction to be performed through an object's methods is known as ***data encapsulation*** — a fundamental principle of object-oriented programming.

**What is a CLASS**

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

**class** bicycle {

**int** pedalRate = 0;

**int** speed = 0;

**int** gear = 1;

**void** changePedalRate(**int** newValue)

{

pedalRate = newValue;

}

**void** changeGear(**int** newValue)

{

gear = newValue;

}

**void** speedUp(**int** increment)

{

speed = speed + increment;

}

**void** applyBrakes(**int** decrement)

{

speed = speed - decrement;

}

**void** printStates()

{

System.*out*.println("PedalRate:" + pedalRate + " speed:" + speed + " gear:" + gear);

}

} // end of class

The syntax of the Java programming language will look new to you, but the design of this class is based on the previous discussion of bicycle objects.

The fields pedal rate, speed, and gear represent the object's state, and the methods (changePedalRate, changeGear, speedUp etc.) define its interaction with the outside world.

You may have noticed that the *bicycle* class does not contain a main method.

That's because it's not a complete application; it's just the blueprint for bicycles that might be *used* in an application.

The responsibility of creating and using new Bicycle objects belongs to some other class in your application. This class is often called the driver class.

**class** bicycleDemo {

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

// Create two different

// Bicycle objects

bicycle bike1 = **new** bicycle();

bicycle bike2 = **new** bicycle();

// Invoke methods on

// those objects

bike1.changePedalRate(50);

bike1.speedUp(10);

bike1.changeGear(2);

bike1.printStates();

bike2.changePedalRate(50);

bike2.speedUp(10);

bike2.changeGear(2);

bike2.changePedalRate(40);

bike2.speedUp(10);

bike2.changeGear(3);

bike2.printStates();

}

}

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

Pedal Rate: 50 speed: 10 gear: 2

Pedal Rate: 40 speed: 20 gear: 3

As with functions defined earlier in the course methods can be written which can return values. For example rather that ‘hard coding’ the changes in pedal rate or speeding up – the value can be prompted.

**int** getSpeedUpValue()

{

Scanner kboard = **new** Scanner(System.*in*);

**int** speed=0;

System.*out*.println("Please enter new speed value");

speed = kboard.nextInt();

**return**(speed);

}

Would as for a new value to ne entered – and therefore the driver could which was

bike2.speedUp(10);

would now become

bike2.speedUp(bike2.getSpeedUpValue());

but the final output would be the same.

**Declaring Classes**

You've seen classes defined in the following way:

class *MyClass* {

// field, constructor, and

// method declarations

}

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

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

class *MyClass extends MySuperClass* {

// field, constructor, and

// method declarations

}

Means that MyClass is a subclass of MySuperClass (see later INHERITANCE).

You can also add modifiers like *public* or *private* at the very beginning—so you can see that the opening line of a class declaration can become quite complicated.

The modifiers *public* and *private*, which determine what other classes can access MyClass, are discussed later. For the moment you do not need to worry about these extra complications.

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

1. Modifiers such as *public*, *private*, and a number of others that you will encounter later.
2. The class name, with the initial letter capitalized by convention.
3. The name of the class's parent (superclass), if any, preceded by the keyword *extends*. A class can only *extend* (subclass) one parent.
4. A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword *implements*. A class can *implement* more than one interface.
5. The class body, surrounded by braces, {}.

**Declaring Member Variables**

There are several kinds of variables:

* Member variables in a class—these are called *fields*.
* Variables in a method or block of code—these are called *local variables*.
* Variables in method declarations—these are called *parameters*.

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

private int pedalRate;

private int gear;

private int speed;

Field declarations are composed of three components, in order:

1. Zero or more modifiers, such as public or private.
2. The field's type.
3. The field's name.

The fields of Bicycle are named cadence, gear, and speed and are all of data type integer (int). The public keyword identifies these fields as public members, accessible by any object that can access the class.

# Defining Methods

Here is an example of a typical method declaration:

public double calculateAnswer(double wingSpan, int numberOfEngines,

double length, double grossTons) {

//do the calculation here

}

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

More generally, method declarations have six components, in order:

1. Modifiers—such as public, private, and others you will learn about later.
2. The return type—the data type of the value returned by the method, or void if the method does not return a value.
3. The method name—the rules for field names apply to method names as well, but the convention is a little different.
4. The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.
5. An exception list—to be discussed later.
6. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

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

The signature of the method declared above is:

calculateAnswer(double, int, double, double)

## Naming a Method

Although a method name can be any legal identifier, code conventions restrict method names. By convention, method names should be a verb in lowercase or a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, etc. In multi-word names, the first letter of each of the second and following words should be capitalized. Here are some examples:

run

runFast

getBackground

getFinalData

compareTo

setX

isEmpty

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

## Overloading Methods

The Java programming language supports *overloading* methods, and Java can distinguish between methods with different *method signatures*. This means that methods within a class can have the same name if they have different parameter lists (there are some qualifications to this that will be discussed in the lesson titled "Interfaces and Inheritance").

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

public class DataArtist {

...

public void draw(String s) {

...

}

public void draw(int i) {

...

}

public void draw(double f) {

...

}

public void draw(int i, double f) {

...

}

}

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. In the code sample, draw(String s) anddraw(int i) are distinct and unique methods because they require different argument types.

You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

# Providing Constructors for Your Classes

A class contains constructors that are invoked to create objects from the class blueprint. Constructor declarations look like method declarations—except that they use the name of the class and have no return type. For example, Bicycle has one constructor:

public Bicycle(int startCadence, int startSpeed, int startGear) {

gear = startGear;

cadence = startCadence;

speed = startSpeed;

}

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

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

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

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

public Bicycle() {

gear = 1;

speedRate = 10;

speed = 0;

}

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

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

You don't have to provide any constructors for your class, but you must be careful when doing this. The compiler automatically provides a no-argument, default constructor for any class without constructors. This default constructor will call the no-argument constructor of the superclass. In this situation, the compiler will complain if the superclass doesn't have a no-argument constructor so you must verify that it does. If your class has no explicit superclass, then it has an implicit superclass of Object, which *does* have a no-argument constructor.

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

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

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

# Passing Information to a Method or a Constructor

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

public double computePayment(

double **loanAmt**,

double **rate**,

double **futureValue**,

int **numPeriods**) {

double interest = **rate** / 100.0;

double partial1 = Math.pow((1 + interest),

- **numPeriods**);

double denominator = (1 - partial1) / interest;

double answer = (-**loanAmt** / denominator)

- ((**futureValue** \* partial1) / denominator);

return answer;

}

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

**Note:** *Parameters* refers to the list of variables in a method declaration. *Arguments* are the actual values that are passed in when the method is invoked. When you invoke a method, the arguments used must match the declaration's parameters in type and order.

## Parameter Types

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

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

public Polygon polygonFrom(Point[] corners) {

// method body goes here

}

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

## Parameter Names

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

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

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

public class Circle {

private int x, y, radius;

public void setOrigin(int x, int y) {

...

}

}

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

## Passing Primitive Data Type Arguments

Primitive arguments, such as an int or a double, are passed into methods *by value*. This means that any changes to the values of the parameters exist only within the scope of the method. When the method returns, the parameters are gone and any changes to them are lost. Here is an example:

public class PassPrimitiveByValue {

public static void main(String[] args) {

int x = 3;

// invoke passMethod() with

// x as argument

passMethod(x);

// print x to see if its

// value has changed

System.out.println("After invoking passMethod, x = " + x);

}

// change parameter in passMethod()

public static void passMethod(int p) {

p = 10;

}

}

When you run this program, the output is:

After invoking passMethod, x = 3

## Passing Reference Data Type Arguments

Reference data type parameters, such as objects, are also passed into methods *by value*. This means that when the method returns, the passed-in reference still references the same object as before. *However*, the values of the object's fields *can* be changed in the method, if they have the proper access level.

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

public void moveCircle(Circle circle, int deltaX, int deltaY) {

// code to move origin of circle to x+deltaX, y+deltaY

circle.setX(circle.getX() + deltaX);

circle.setY(circle.getY() + deltaY);

// code to assign a new reference to circle

circle = new Circle(0, 0);

}

Let the method be invoked with these arguments:

moveCircle(myCircle, 23, 56)

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