

Introduction to Objects

This chapter is an introduction to objects. What is an *object*? Simply, it is any thing. That is, if you can name it, it is an object. Car is an object. House is an object. Love is an object. Anything with a name is an object. This is a very profound concept. Until you understand this concept, do not try to learn Java. A specific thing is an *instance* of an object. My car is an instance of the object Car. An instance can be derived from several objects. I am an instance of the object Person, the object Man and the object Author. The relationships between objects and instances will be discussed in great length later on.

Introduction – programming as model of “real world”

Programming has always attempted to model the “real world”. The first application of a computer was to calculate artillery tables. The question was, what angle do you need to set for a cannon in order for a shell to land at a specific distance? The computer was programmed to create a table that had the firing angle needed for a list of distances for a specific cannon with a specified shell and powder charge. This was the first computer model. It is a model because soldiers or sailors could have gone out and fired real cannons. They would have had to carefully measure the angles the cannons were fired at and the distances the shells traveled. They would have had to fire a lot of expensive shells and hoped that the weather and wind remained constant. And whenever the gunpowder got a little stronger or the shells a little heavier, the whole, expensive process would have to be repeated. As has happened many times, military programs have paid for technology development that has subsequently resulted in tremendous civilian products.

Today, computer models are everywhere. A word processor is a model of pen and paper. An inventory control system models a warehouse. Even the programs that you will write to control a robot are models. Explaining it now is hard; very soon you will have enough experience to see that the concept of computer programs as models is obvious.

Relationship of programming objects to physical objects

Object-oriented programming creates computer objects to represent real objects. What that means is simple. When you look at the world, you see objects. Everything is an object of some sort. Trees are objects. Books are objects. Thoughts are objects. Anything that can be named is an object. OOP recognizes the object nature of things and uses that nature as an organizing mechanism. Objects can be alike or different. Two spoons share the same characteristics. A spoon and a fork have a lot in common but are distinctly different. A car and a pig are not alike at all. When we program a computer, we look at the characteristics of the things we want to model. Then we try to impose some sort of order to make our programs better.

Most of you have studied your natural or first language. Since I am familiar with English class, I'll describe what I learned there and hope that you learned the same things. The study of English starts with the concepts of parts of speech. The parts of speech include nouns, verbs, adjectives, and proper nouns, among other things. In programming terms, a noun is a *class* and a proper noun is an *instance*. That is, a class describes the characteristics of a generic object. An instance describes a specific object. Adjectives help describe a specific object and are, in programming terms, *properties*. Verbs are action words and are called *methods*.

For example, dog is a class. Shelby, my dog, is an instance. Weight is a property of dog and in Shelby's case the value of weight is 42 lbs (18 kg). Feed is a method of dog; one that Shelby really enjoys being used. Shelby is a Basset hound. Basset hound is a *derived class* of dog. Basset hound *inherits* all the properties and methods of dog. But by being a derived class Basset hound *extends* dog by adding additional properties and methods, by creating new constraints on existing properties or by changing what the existing methods do. Shelby, like most Bassets, likes to eat a lot. Obesity is a real problem with Bassets. So, the Basset hound class may *override* the feed method of dog with its own feed method. The Basset hound feed method may limit the amount of food that the instance can receive at any time.

Sometimes I want to work with a group of dogs. I have had three dogs in households over the years, Blacky, a mostly German Shepard mixed breed, Ginger, a Dachshund, and Shelby, a Basset hound. All three are dogs, so they all have the feed method and the weight property that comes from being a dog. But they are also all instances of their respective breeds. Shelby is a Basset hound but she is also a dog because Basset hound extends dog. When I go through my list of dogs and feed each one, Shelby will be fed using the limit imposed because she is a Basset hound. Ginger and Blacky will be fed using the generic dog method unless, of course, their respective breed overrode the feed method. The ability to deal with instances through their generic class, dog, and their specific class, Basset hound, is called *polymorphism*. As you will see later, this is a very useful ability.

There are two kinds of properties and methods: *class* and *instance*. Class properties and methods are global while instance properties and methods apply to a specific instance. The feed method is an instance method. It is called to feed one specific dog. Likewise, weight is an instance property; it is the weight of a particular dog. An example of a class property would be maximum weight. This property does not apply to a particular instance, it limits the weight of all dogs.

Properties can be simple things, like weight or name which are numbers or strings, or more complex things, like feeding schedule or nutrition requirements. These more complex things are objects in their own right. Creating objects that have complex properties is called *composition*.

Java's concept of an object - `java.lang.Object`

Object	
Properties	none
Methods	+clone() +equals() +finalize() +getClass() +hashCode() +notify() +notifyAll() +toString +wait() +wait() +wait()

At left is a *UML (Universal Modeling Language)* diagram. UML is a visual language. That is, it uses text in diagrams to convey more information than text alone could convey. UML is a fundamental tool of computer science education and has radically transformed the way computer science is taught since 1990. UML is specific to object-oriented programming in a general way. This includes the languages of Smalltalk, C++, C# and, most prominently, Java. UML is not a programming environment or a programming language. UML is a conceptual and visual tool. UML includes a variety of diagrams that cover different aspects of programming. These include class, use case, state, activity, and implementation diagrams, among others. Covering these topics in even a very minimal way is beyond the scope of this work. We will use only the UML class diagram in this work. However, you are well advised to understand that class diagrams are only a very small part of a very rich programming methodology.

A *class diagram* is a way of visually organizing information about an object. It has three parts: a *name*, a list of *properties* or *attributes* and a list of *methods* or *operations*. The name is usually the same as the physical thing the programming object represents. In the previous section, we discussed the class `dog` and “Dog” would be an appropriate name for the programming object. Properties, Java term, or attributes, UML term, are characteristics of an instance of the object, the “blanks” that you need to “fill-in” to describe a specific instance. For example, `dog` would have attributes of weight, age and color that would need to be filled in for a specific dog. Methods, Java term, or operations, UML term,

are things that an object can do or have done to it. `Dog` would have a `feed` method that would set the `happy` property if `feed` were called often and `unhappy` if `feed` were called infrequently.

In the previous section, we discussed derived classes. `BassetHound` and `GermanShepard` were derived classes of `Dog`. An instance of `BassetHound` has all the properties and methods of `Dog` plus a few that are specific to Basset hounds. That raises the question, is `Dog` derived from another class? In biological terms, the answer is yes. Dogs are mammals and mammals are invertebrates and invertebrates are living things, etc. So we could create a hierarchy that has these classes and more. However, in programming, we model only things that we decide are relevant to the problem at hand. So, we can decide that a `Mammals` class is beyond the scope of our model. So we can safely omit that complication. However in programming terms, all things are objects so, in Java, all objects are derived from `Object`, either directly or, as in the case of `BassetHound` and `Dog`, indirectly.

The UML diagram shown is the class diagram for the Java class Object. Object is the base class for all classes in Java. That is, every object in Java has these properties and methods. So what properties and methods did the designers of Java believe to be so fundamental that they are in every class?

As it turns out, they found that there was no property that was so fundamental that it was needed for every object. Remember that properties are things that describe a specific instance of an object, or in rare cases, describe the object itself.

They did decide that 11 methods are so fundamental that they were required for all objects. The diagram lists them in alphabetic order; we will describe them in functional groups.

The methods `getClass`, `toString` and `hashCode` are used to identify a specific instance of an object. `getClass` returns the class of an instance. `toString` takes the contents of an instance and converts it to a string that in some way identifies the instance. `hashCode` returns a number, the hash code, that identifies, but not uniquely, the instance. Hash codes are used extensively to make programs run faster. `getClass` is a method that should never be overridden. `toString`, on the other hand, is frequently overridden to provide more meaningful information about an object. `hashCode` is in the middle, it should be overridden only by programmers who understand how to create a good hash method and who have determined that they need to override the default.

The methods `clone`, `equals` and `finalize` are used to manipulate instances of objects. `clone` is used to make a copy of an instance. `equals` is used to determine if two instances are equal. `finalize` is called by the JVM when an instance has been thrown away. All three methods should be overridden when needed. For example, `clone` is used to make a copy. But what a copy is depends upon the object. You could, for example, create an object `Pack` that contains a collection of instances of `Dog`. What does it mean to clone an instance of `Pack`? Does cloning a pack clone the dogs in that pack or not? That is, do you wind up with two packs with the same dogs in each or do you wind up with two packs, each with its own set of dogs? So, if you intend to use `clone`, you should probably override it. Likewise, what `equal` means depends a lot upon the object. In the real world, twenty pounds of all-purpose flour is probably equal to any other 20 pounds of all-purpose flour. But any 20 karats of diamonds is probably not equal to any other 20 karats of diamonds. In the `Dog` class, `equal` could mean that it is the same dog or it could mean two dogs are the same age or weight or breed. If you use the `equals` method, you almost certainly want to override it. `finalize`, on the other hand, is hardly ever overridden. Try and come up with an example of when you might need to.

The last five methods, `notify`, `notifyAll` and the three `wait`s are only used in multi-threaded situations. `notify` and `notifyAll` are used to tell a thread that an instance is available while the `wait`s are used to suspend a thread until an instance is available. The three `wait` methods differ only in that one is a `wait forever` while the other two provide two different ways to specify a maximum time to wait. What, by the way, is a thread? Threads are independent execution paths. Remember in the RIS programming where you set up sensor watchers. That is, you had a block that said wait for touch sensor 1 to be pressed and then do this code fragment and another block that

said wait for touch sensor 3 and it had another code fragment. Each sensor watcher is a thread. The thread runs whenever its sensor is touched. Each sensor watcher is waiting for its signal. Effectively, each thread has called the wait method on their respective sensors. But there is a third thread that you don't see and it, essentially, owns both instances of the sensors. What it does is constantly check to see if a sensor has been touched. When one is touched, it calls the notify or notifyAll method of the sensor and the sensor watcher executes. There are many details that go along with this that will be discussed in later sections. Note that these methods are usually not overridden.

Before leaving this introduction to the UML class diagram, there are a couple of details that should be clarified. Note that all the methods in Object start with a "+" sign. A plus sign indicates a public method or property. A public method or property is one that can be used by any other object. A "-", minus, sign indicates a private method or property. Private methods and properties can only be used by the class itself; even derived classes cannot use private methods or properties. A "#", sharp, sign indicates a protected property or method. Protected properties and methods are halfway between public and private; they can be used and overridden in derived classes but not in other classes. UML uses an underline (dog) to indicate class, as opposed to instance, properties and methods. Object has none but they will appear in later class diagrams.

Examples of Lego motor as object

A Lego Mindstorms kits contains over 700 individual pieces or physical objects. Two of the pieces are motors. You used motors on the Roverbot and programmed it to move around. What are the attributes and methods of a motor?

Two obvious methods are turn on and turn off. Equally obvious are methods to tell the motor to run forward or backward. As for attributes, being able to tell which power source the motor is connected to also important. What other attributes and methods would be useful? The next section contains the object model for motor from the LeJOS environment. Before reviewing it, write up a list and see how your list compares to the creators of LeJOS.

Motor	
Properties	-iId -iMode -iPower +Motor A +Motor B +Motor C
Methods	-Motor() +getId() +setPower () +forward() +isForward() +backward() +isBackward() +reverseDirection() +getPower() +isMoving() +isFloating() +stop() +isStopped() +flt() +controlMotor

The diagram to the left is a UML class diagram (UML will be described in a later section). This is how the authors of LeJOS decided to describe a Lego motor. The diagram is divided into three parts: the top cell is the name, the middle cell is a list of properties and the bottom cell is a list of methods. The plus, “+”, sign indicates a publicly accessible property or method, one that you can use in programming, while a minus, “-“, sign indicates a private method, something that you are not allowed to use. Underlining indicates that an entry is static or applies to the class. No underline indicates an instance method or property.

Looking at the list of methods, there are six basic commands that should be on your list: setPower, forward, backward, reverseDirection, stop and flt (float – remove power but do not set the brake). There are five boolean methods: isForward, isBackward, isMoving, isFloating and isStopped. Booleans return simply true or false. So these methods ask about the state of the motor. Two other methods, getId and getPower, ask about properties of the motor. Note that these 13 methods are instance methods. They ask about or set properties of a specific motor. Notice the naming convention used. Action verbs are used for commands while these verbs are combined with is or get to form questions that inquire about the state of the motor.

That leaves two other methods, Motor and controlMotor.

Motor is special. Notice that its name is the same as the class. This method is called the *constructor*. The constructor is the method called when you create a new instance of a class. Typically, a constructor will set up everything so that you can use an instance once you create it. This constructor is very unusual because it is private. That means that you can only create an instance from inside the

class. How is this done? It is done through either a class method or class property initialization. If you look into the properties section of the diagram you can see that there are three instances of motor, A, B and C. Since an RCX has three and only three motor outputs, Motor sets them up from the beginning. There is never a need to create another motor object; doing so only wastes memory in the RCX.

The other special method in Motor is controlMotor. It is a static or class method. It is used to directly control the RCX motor outputs. All of the action commands in Motor use controlMotor to do their jobs. A question to ponder is, is there any reason for controlMotor to be public given the other methods in Motor? Another question, is there any reason for controlMotor to be an instance method instead of a class method?

It may be a little disappointing but the answer to both questions is, maybe. It's left as an exercise for you to determine why.

Finally, the middle box is the properties of motor. The three motors are static and public so that they can be used anywhere in the program. The three remaining properties are Id, mode and power. Id is simply the name, A, B or C of the instance. Power contains the power level of the motor while mode indicates the state of the motor, forward, reverse, stopped or floating. These three properties are all private, which is quite common in Java programming. The reason for being private is straightforward. Being public means that your program could get or set these properties. For Motor, getting them poses no problem. Each of the get methods simply returns the value of the property. But setting these properties poses a real problem. Suppose a program could set the Id property and proceeded to set the Id on all three motors to "D". This would be bad. And so, to make bad things harder to do, most programmers make properties private unless there is a good reason not to.

Packages

A single object is not very useful when it comes to modeling the real world. For example, if you want to model a forest, you need a tree object. But tree also needs component parts like leaves, roots, stems, etc. Given that the number of objects in the world is extremely large, it is a good idea to somehow organize objects in a reasonable manner.

In Java, objects are organized into *packages*. Packages have two attributes: a name and a list of objects that the package contains. There are rules, more properly conventions, for naming packages. Package names are hierarchical, with the first word being the top-level name. The next level name describes the function. For example, java

Package *josx.platform.rcx*

Access to RCX sensors, motors, etc.

See:

[Description](#)

Interface Summary	
<u>ButtonListener</u>	Abstraction for receiver of button events.
<u>LCDConstants</u>	LCD constants.
<u>ListenerCaller</u>	Interface for calling calling lejos listeners.
<u>Opcodes</u>	Opcodes constants.
<u>Segment</u>	LCD segment constants.
<u>SensorConstants</u>	Constants for Sensor methods.
<u>SensorListener</u>	Listener of sensor events.
<u>SerialListener</u>	Listener of incoming serial data.

Class Summary

<u>Battery</u>	Provides access to Battery.
<u>Button</u>	Abstraction for an RCX button.
<u>LCD</u>	LCD routines.
<u>Memory</u>	Provides access to memory.
<u>MinLCD</u>	Only the most basic APIs from LCD.
<u>MinSound</u>	Only the most basic APIs from Sound.
<u>MinuteTimer</u>	Provides access to Battery.
<u>Motor</u>	Abstraction for a motor.
<u>PersistentMemoryArea</u>	A memory area for persistent storage.
<u>Poll</u>	Provides blocking access to events from the RCX.
<u>ProximitySensor</u>	A 'sensor' to detect object proximity.
<u>ROM</u>	Provides access to ROM routines.
<u>Sensor</u>	Abstraction for a sensor (<i>considerably changed since alpha5</i>).
<u>Serial</u>	Low-level API for infra-red (IR) communication between an RCX and the IR tower or between two RCXs.
<u>Servo</u>	Implmentation of a servo using a Motor and a Rotation Sensor.
<u>Sound</u>	RCX sound routines.
<u>TextLCD</u>	Display text on the LCD screen.