Object-Oriented Programming Basics With Java

In his keynote address to the 11th World Computer Congress in 1989, renowned computer scientist Donald Knuth said that one of the most important lessons he had learned from his years of experience is that **software is hard to write**!

Computer scientists have struggled for decades to design new languages and techniques for writing software. Unfortunately, experience has shown that writing large systems is virtually impossible. Small programs seem to be no problem, but scaling to large systems with large programming teams can result in $100M projects that never work and are thrown out. The only solution seems to lie in writing small software units that communicate via well-defined interfaces and protocols like computer chips. The units must be small enough that one developer can understand them entirely and, perhaps most importantly, the units must be protected from interference by other units so that programmers can code the units in isolation.

The object-oriented paradigm fits these guidelines as designers represent complete concepts or real world entities as objects with approved interfaces for use by other objects. Like the outer membrane of a biological cell, the interface hides the internal implementation of the object, thus, isolating the code from interference by other objects. For many tasks, object-oriented programming has proven to be a very successful paradigm. Interestingly, the first object-oriented language (called Simula, which had even more features than C++) was designed in the 1960's, but object-oriented programming has only come into fashion in the 1990's.

This module is broken down into three sections. First, you will find a high-level overview that shows object-oriented programming to be a very natural concept since it mirrors how your hunter-gatherer mind views the outside world. Second, you will walk through object-oriented programming by example; learning to use a simple object, examining the definition, extending the definition, and then designing your own object. Finally, you will explore the most important concepts in object-oriented programming: encapsulation, data hiding, messages, and inheritance.

Executive Summary

Summary

- Object-oriented programming takes advantage of our perception of world
- An object is an encapsulated completely-specified data aggregate containing attributes and behavior
- Data hiding protects the implementation from interference by other objects and defines approved interface
- An object-oriented program is a growing and shrinking collection of objects that interact via messages
- You can send the same message to similar objects--the target decides how to implement or respond to a message at run-time
- Objects with same characteristics are called instances of a class
- Classes are organized into a tree or hierarchy.
- Two objects are similar if they have the same ancestor somewhere in the class hierarchy
- You can define new objects as they differ from existing objects
- Benefits of object-oriented programming include:
    - reduced cognitive load (have less to think about and more natural paradigm)
    - isolation of programmers (better team programming)
    - less propagation of errors
    - more adaptable/flexible programs

- faster development due to reuse of code

You are used to observing the world around you through the eyes of a hunter-gatherer: mainly animals acting upon other animals and objects. There must have been tremendous selection pressure for brains that were adept at reasoning about entities, their attributes, their behavior, and the relationships among them. Is that object edible, ready to eat me, or going to mate with me? When writing software, one can easily argue that you are at your best when designing and implementing software in a manner that parallels the way your brain perceives the real world. This section attempts to explain and motivate object-oriented design concepts by drawing parallels to our natural way of thinking.

Encapsulation and data hiding

The first and most important design principle we can derive from our perception of the real world is called *encapsulation*. When you look at an animal, you consider it to be a complete entity--all of its behavior and attributes arise strictly from that animal. It is an independent, completely-specified, and self-sufficient actor in the world. You do not have to look behind a big rock looking for another bit of functionality or another creature that is remotely controlling the animal.

Closely associated with encapsulation is the idea of *data hiding*. Most animals have hidden attributes or functionality that are inaccessible or are only indirectly accessible by other animals. The inner construction and mechanism of the human body is not usually available to you when conversing with other humans. You can only interact with human beings through the approved voice-recognition interface. Bookkeeping routines such as those controlled by the autonomic nervous system like breathing may not be invoked by other humans. Without bypassing the approved interface, you cannot directly measure attributes such as internal body temperature and so on.

One can conclude that we perceive objects in the world as encapsulated (self-contained) entities with approved interfaces that hide some implementation behavior and attributes. From a design perspective, this is great because it limits what you have to think about at once and makes it much easier for multiple programmers to collaborate on a program. You can think about and design each object independently as well as force other programmers to interact with your objects only in a prescribed manner; that is, using only the approved interface. You do not have to worry about other programmers playing around with the inner workings of your object and at the same time you can isolate other programmers from your internal changes.

Encapsulation and data hiding are **allowed** to varying degrees in non-object-oriented languages and programmers have used these principles for decades. For example, in C, you can group related variables and functions in a single file, making some invisible to functions in other files by labeling them as `static`. Conversely, object-oriented languages **support** these design principles. In Java, for example, you will use an actual language construct called a `class` definition to group variables and functions. You can use access modifiers like `private` and `public` to indicate which class members are visible to functions in other objects.

The interaction of objects using polymorphism

Encapsulation and data hiding are used to define objects and their interfaces, but what about the mechanism by which objects interact? In the real world, animals are self-contained and, therefore, do not physically share brains. Animals must communicate by sending signals. Animals send signals, depending on the species, by generating sound waves such as a voice, images such as a smile, and chemicals such as pheromones. There is no way for an animal to communicate with another by directly manipulating the internal organs or brain of another because those components are hidden within the other animal. Consequently, our brain is hardwired to communicate by sending signals.

If we view the world as a collection of objects that send and receive messages, what programming principle can we derive? At first you may suspect that a signal or message is just a function call. Rather than manipulate the internals of an object, you might call a function that corresponded to the signal you wanted to send.

Unfortunately, function calls are poor analogs to real world messaging for two main reasons. First, function calls do things backwards. You pass objects to functions whereas you send messages to an object. You have to pass objects to functions for them to operate on because they are not associated with a particular object. Second, function calls are unique in that the function's name uniquely identifies what code to run whereas messages are more generic. The receiver of a message determines how to implement it. For example, you can tell a man and a woman both to shave and yet they respond to the exact same message by doing radically different things.

The truly powerful idea behind message sending lies in its flexibility--you do not even need to know what sex the human is to tell them to shave. All you need to know is that the receiver of the message is human. The notion that similar, but different, objects can respond to the same message is technically called *polymorphism* (literally "multiple-forms"). Polymorphism is often called *late-binding* because the receiver object binds the message to an appropriate implementation function (*method* in Java terminology) at run-time when the message is sent rather than at compile-time as functions are.

Polymorphism's flexibility is derived from not having to change the code that sends a message when you define new objects. Imagine a manager that signals employees when to go home at night. A so-called micro-manager must know all sorts of details about each employee (such as where they live) to get them home at night. A manager that delegates responsibility well will simply tell each employee to go home and let them take care of the details. Surely, adding a new kind of employee such as a "summer intern" should not force changes in the manager code. Alas, a micro-manager would in fact have to change to handle the details of the new employee type, which is exactly what happens in the function-centric, non-object-oriented programming model.

Without polymorphism, encapsulation's value is severely diminished because you cannot effectively delegate, that is, you cannot leave all the details within a self-contained object. You would need to know details of an object to interact with it rather than just the approved communication interface.

The relationship between objects and inheritance

Humans rely on their ability to detect similarities between objects to survive new situations. If an animal has many of the characteristics of a snake, it is best to leave it alone for fear of a venomous bite. In fact, some animals take advantage of similarity detectors in other animals by mimicking more dangerous creatures; some kingsnakes have colored bands like the deadly coral snake, although in a different order. Similarly, we learn most easily when shown new topics in terms of how they relate or differ from our current knowledge base. Our affinity for detecting and using similarity supports two important ideas in the object-oriented design model: polymorphism requires a definition of similarity to be meaningful and we can design new objects as they differ from existing objects.

Behavior versus identity

Sending the same message to two different objects only makes sense when the objects are similar by some measure. For example, it makes sense to tell a bird and an airplane to fly because they share behavior. On the other hand, telling a dog to sit makes sense, but telling a cat to sit makes no sense; well, it is a waste of time anyway. Telling a human to shave makes sense, but telling an airplane to shave does not. One way to define similarity is to simply say that the objects implement or respond to the same message or set of messages; that is to say, they share at least a partial *interface* (common subset of public methods). This is the approach of early object-oriented languages such as SmallTalk.

Another similarity measure corresponds to the relationship between the objects themselves rather than just their interfaces. If two objects are the same kind of object then it makes sense that they also share a partial interface. For example, males and females are kinds of humans and, hence, share a common interface (things that all humans can do like shave, sleep, sit and so on). Java uses object relationships to support polymorphism.

The inheritance relationship

What metaphor does Java use to specify object relationships? Java uses *inheritance*. That is, if man and woman objects are kinds of humans then they are said to inherit from human and therefore share the human interface. Consequently, we are able to treat them generically as humans and do not have to worry about their actual type. If we had a list of humans, we could walk down the list telling everyone to shave without concern for the objects' concrete type (either man or woman). Late-binding ensures that the appropriate method is invoked in response to the shave message depending on the concrete type at run-time.

All objects have exactly one parent, inheriting all of the data and method members from the parent. The inheritance relationships among objects thus form a tree, with a single predefined root called `Object` representing the generic object.

Defining by difference

The second important feature of an object-oriented design model inspired by similarity detection is "defining by difference." If I want to define a new object in my system, an efficient way to do so is to describe how it differs from an existing object. Since man and woman objects are very similar, the human object presumably contains most of the behavior. The man object, for example, only has to describe what distinguishes a man from the abstract notion of a human, such as buying lots of electronic toys.

Background

Imagine a portable, object-oriented (classes, data hiding, inheritance, polymorphism), statically-typed ALGOL-derived language with garbage collection, threads, lots of support utilities. Thinking about Java or Objective-C or C++? Actually, the description fits Simula67 as in 1967--over thirty years ago. The complete object-oriented mechanism found in Java has been around a long time and there have been many languages developed between Simula67 and Java.

Excerpt from *The History of Simula*
by Jan Rune Holmevik, jan@utri.no (`mailto:jan@utri.no`)
`http://www.javasoft.com/people/jag/SimulaHistory.html`

The SIMULA programming language was designed and built by Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Centre (NCC) in Oslo between 1962 and 1967. It was originally designed and implemented as a language for discrete event simulation, but was later expanded and reimplemented as a full scale general purpose programming language. Although SIMULA never became widely used, the language has been highly influential on modern programming methodology. Among other things SIMULA introduced important object-oriented programming concepts like classes and objects, inheritance, and dynamic binding.

Over the past 40 years, four main computation paradigms have developed:

- *Imperative* (procedural); C, Pascal, Fortran
- *Functional*; Lisp, ML
- *Logic* (declarative); Prolog
- *Object-oriented*; Simula67, SmallTalk, CLOS, Objective-C, Eiffel, C++, Java

There are also application-specific languages like SQL, SETL (set language), AWK/PERL, XML (structured data description), ANTLR (grammars), PostScript, and so on. The idea is that you should use the "right tool for the right job."

Object-oriented programming is not a "silver bullet" that should be used in every situation, but many modern programming problems are readily mapped to object-oriented languages.

Object-Oriented Programming By Example

This section is a starting point for learning object-oriented programming. It parallels the process you would follow to become a car designer: first you learn to drive a car, then you look under the hood, next you learn to modify and repair engines, finally you design new cars. Specifically, in this section, you will:

1. learn how to use a trivial object
2. study the object definition
3. learn to modify an object definition
4. learn how to define new objects

"Learning to drive"

We all know how to add two numbers, for example "3+4=7". In a programming language such as C, you might have the values in variables like this:

```
int a,b,c;
a=3;
b=4;
c=a+b; // c is 7
```

Now imagine that you plan on doing some graphics work and your basic unit will be a two-dimensional point not a single number. Languages do not have built in definitions of point, so you have to represent points as objects (similar to structs in C and records in Pascal). The plus operator "+" also is not defined for point so you must use a method call to add points together. In the following code fragment, type `Point` holds x and y coordinates and knows how to add another point to itself yielding another `Point`. Just like in C or other procedural languages, you have to define the type of your variables. Here, the variables are objects of type `Point`. To create a new point, use the `new` operator and specify the coordinates in parentheses like a function call.

```
Point a, b, c;
a = new Point(2,4);
b = new Point(3,5);
```

To add two points together, use the `plus()` method:

```
c = a.plus(b); // c is 5,9
```

In object-oriented terminology, you would interpret this literally as, "send the message `plus` to point `a` with an argument of `b`." In other words, tell `a` to add `b` to itself, yielding a new point. The field access operator, "dot", is used for methods just like you use it in C or Pascal to access `struct` or record fields.

A comment about object-oriented method call syntax

The object-oriented syntax

```
target.method(arguments);
```

may look a bit odd (backwards) since you are used to calling functions not invoking methods on objects; functions are called with both operands as arguments:

```
c = plus(a,b);
```

On the contrary, object-oriented syntax is closer to mathematics and English syntax where the "operator" appears in the middle of the statement. For example, you say "Robot, move left 3 units":

```
robot.moveLeft(3); /* Java style */
```

not "Move left Robot, 3":

```
moveLeft(robot,3); /* C style */
```

The "dot" field access operator is consistent with field access. For example, in most procedural languages like C, you say `p.x` to get field `x` from struct or record `p`. As you will see, in object-oriented languages, you group methods as well as data into objects--the field access operator is consistently used for both.

The `Point` type is called a *class* because it represents a template for a class of objects. In the example above, `a`, `b`, and `c` are all point objects and, hence, they are *instances* of class `Point`. You can think of a class as blueprints for a house whereas instances are actual houses made from those blueprints. Instances exist only at run-time and an object-oriented program is just a bunch of object instances sending messages to each other. Also, when you hear someone talking about the methods of an object, they are, strictly speaking, referring to the methods defined in the object's class definition.

"Looking under the hood"

Imagine that you wanted to duplicate the simple point addition example above, but in a procedural programming language. You almost certainly would make a data *aggregate* such as the following C struct:

```
struct Point {
  int x, y;
};
```

Then, to define two points, you would do:

```
struct Point a = {2,4};
struct Point b = {3,5};
struct Point c;
```

Adding two points together would mean that you need an `add_points` function that returned the point sum like this:

```
c = plus_points(a,b);
```

A Comment on C syntax

Variable definitions in C are like Java and of the form:

```
type name;
```

or

type name = *init-value*;

For example,

```
int x = 0;
```

defines an integer called x and initializes it to 0.

C data aggregates are called structs and have the form:

```
struct Name {
  data-field(s);
};
```

Defining a variable of struct type allocates memory space to hold something that big; specifically, `sizeof(Name)`, in C terminology. You can initialize these struct variables using a curly-brace-enclosed list of values. For example,

```
struct Point a = {1,2};
```

makes memory space to hold a point of size `sizeof(Point)` (normally two 32-bit words) and initializes its `x` and `y` values to 1 and 2, respectively.

To access the fields within a struct, use the "dot" operator. For example, if you printed out the value of `a.x` after you defined variable `a` as above, you would see the value 1.

Functions in C are defined as follows:

```
return-type name(arguments)
{
  statement(s);
}
```

For example, here is a function that adds two integers and returns the result:

```
int add_ints(int i, int j)
{
  return i+j;
}
```

Functions with no return type (that is, procedures) use type void:

```
void foo() {...}
```

You could define `plus_points` as follows.

```
struct Point plus_points(struct Point a, struct Point b)
{
  struct Point result = {a.x+b.x, a.y+b.y};
  return result;
}
```

At this point, you have the data defined for a point and a function to operate on it. How does someone reading your code know that function `plus_points` is used to manipulate points? That person would have to look at the name, arguments, and statements to decide the function's relevance to the `Point` struct. Invert the question. Where is all the code that manipulates a point? The bad news, of course, is that you have to search the entire program!

Object-oriented programming offers a better solution. If humans naturally see the elements in the real world as self-contained objects, why not program the same way? If you move the functions that manipulate a data aggregate physically into the definition of that aggregate, you have *encapsulated* all functionality and state into single program entity called a class. Anybody that wants to examine everything to do with a point, for example, just has to look in a single place. Here is a starting version of class `Point` in Java:

```
class Point {
  int x,y;
  Point plus(Point p) {
    return new Point(x+p.x,y+p.y);
  }
}
```

It contains both the *state* and *behavior* (data and functionality) of `Point`. The syntax is very similar to the C version, except for the method hanging around inside. Notice that the name of function `plus_points` becomes method `plus`; since plus is inside `Point`, the "`_points`" is redundant. The difference between a function and a method will be explored in more detail in the section on polymorphism. For now, it is sufficient to say that C functions or Pascal procedures become methods inside class definitions.

How are objects initialized? In other words, how do the arguments of "`new Point(1,2)`" get stored in the `x` and `y` fields? In an object-oriented language, you can define a constructor method that sets the class data members according to the constructor arguments (arriving from the `new` expression). The constructor takes the form of another method with the same name as the class definition and no return type. For example, here is an augmented version of `Point` containing a constructor.

```
class Point {
  int x,y;
  Point(int i, int j) {
    x=i;
    y=j;
  }
  Point plus(Point p) {
    return new Point(x+p.x,y+p.y);
  }
}
```

The constructor is called by the Java virtual machine once memory space has been allocated for a `Point` instance during the `new` operation.

What would happen if you defined the constructor with arguments named the same thing as the instance variables:

```
  Point(int x, int y) {
    x=x;
    y=y;
  }
```

Ooops. Reference to variables `x` and `y` are now ambiguous because you could mean the parameter or the instance variable. Java resolves `x` to the closest definition, in this case the parameter. You must use a scope override to make this constructor work properly:

```
  Point(int x, int y) {
    this.x=x; // set this object's x to parameter x
    this.y=y;
  }
```

This class definition is now adequate to initialize and add points as in the following fragment.

```
Point a=new Point(2,4);
Point b=new Point(3,5);
Point c=a.plus(b); // c is 5,9
```

Look at the method invocation:

```
c=a.plus(b);
```

What is this really doing and what is the difference between it and

```
c=d.plus(b);
```

for some other point, `d`? You are supposed to think of `a.plus(b)`as telling `a` to add `b` to itself and return the result. The difference between the two method invocations is the target object, but how does method `plus` know about this target; that is, how does `plus` know which object to add `b` to--there is only one argument to the method? Remember that computers cannot "tell" objects to do things, computers only know (in hardware) how to call subroutines and pass arguments. It turns out that the two `plus` invocations are actually implemented as follows by the computer:

```
c=plus(a,b);
c=plus(d,b);
```

where the method targets `a` and `d` are moved to arguments, thus, converting them from message sends to function calls. Because you may have many objects with a method called `plus` such as `Point3D`, the compiler will actually convert your method named `plus` to a unique function name, perhaps `Point_plus`. The method could be converted to a function something like:

```
Point Point_plus(Point this, Point p) {
  return new Point(this.x+p.x,this.y+p.y);
}
```

The "`this`" argument will be the target object referenced in the method invocation expression as the translation above implies.

So, for the moment, you can imagine that the compiler converts the class definition, variable definitions, and method invocations into something very much like what you would do in C or other procedural languages. The main difference at this point is that the class definition encapsulates the data of and the functions for a point into a single unit whereas C leaves the data and functions as disconnected program elements.

Encapsulation promotes team programming efforts because each programmer can work on an object without worrying that someone else will interfere with the functionality of that object. A related concept enhances the sense of isolation. *Data hiding* allows you to specify which parts of your object are visible to the outside world. The set of visible methods provides the "user interface" for your class, letting everyone know how you want them to interact with that object. Sometimes helper methods and usually your data members are hidden from others. For example, you can modify class `Point` to specify visibility with a few Java keywords:

```
class Point {
  protected int x,y;
  public Point(int i, int j) {
    x=i;
    y=j;
  }

  public Point plus(Point p) {
    return new Point(x+p.x,y+p.y);
  }
}
```

The data is protected from manipulation, but the constructor and `plus()` method must be publicly visible for other objects to construct and add points.

"Modifying the engine"

You know how to use objects and what class definitions look like. The next step is to augment a class definition with additional functionality. To illustrate this, consider how you would print out a point. To print strings to standard output in Java, you call method `System.out.println()`. You would like to say:

```
System.out.println(p);
```

for some point `p`. Java has a convention that objects are automatically converted to strings via the `toString()` method, therefore, the above statement is interpreted as:

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

Naturally, the functionality for converting a `Point` to a string will be in class `Point`:

```
class Point {
  Point(int i, int j) {
    x=i;
    y=j;
  }
  Point plus(Point p) {
    return new Point(x+p.x,y+p.y);
  }
  String toString() {
    return "("+x+","+y+")";
  }
}
```

"Designing new cars"

Once you have decided upon a set of objects and their state/behavior in your design, defining the Java classes is relatively straightforward.

Composition

Consider defining a `Rectangle` object composed of an upper left corner and a lower right corner. The data portion of `Rectangle` is pretty simple:

```
class Rectangle {
  Point ul;
  Point lr;
}
```

What should the constructor for `Rectangle` look like? Well, how do you want to construct them? Construction should look something like:

```
Rectangle r = new Rectangle(10,10, 20,40); // size 10x30
```

or

```
Point p = new Point(10,10);
Point q = new Point(20,40);
Rectangle r = new Rectangle(p,q); // size 10x30
```

The first `Rectangle` construction statement uses a constructor that takes the four coordinates for the upper left and lower right coordinates:

```
  public Rectangle(int ulx, int uly, int lrx, int lry) {
    ul = new Point(ulx,uly);
    lr = new Point(lrx,lry);
  }
```

The second constructor takes two points:

```
  public Rectangle(Point ul, Point lr) {
    this.ul = ul;
    this.lr = lr;
  }
```

If your design calls for a method to return the width and height of a `Rectangle`, you could define method, say `getDimensions()`, as follows.

```
class Rectangle {
  protected Point ul;
  protected Point lr;

  /** Return width and height */
  public Dimension getDimensions() {
    return new Dimension(lr.x-ul.x, lr.y-ul.y);
  }
}
```

Because Java does not allow multiple return values, you must encapsulate the width and height into an object, called `Dimension` as returned by `getDimensions()` in the example:

```
class Dimension {
  public int width, height;

  public void Dimension(int w, int h) {
    width=w;
    height=h;
  }
}
```

The complete `Rectangle` class looks like:

```
class Rectangle {
  protected Point ul;
  protected Point lr;

  /** Return width and height */
  public Dimension getDimensions() {
    return new Dimension(lr.x-ul.x, lr.y-ul.y);
  }

  public Rectangle(int ulx, int uly, int lrx, int lry) {
    ul = new Point(ulx,uly);
    lr = new Point(lrx,lry);
  }

  public Rectangle(Point ul, Point lr) {
    this.ul = ul;
    this.lr = lr;
  }
}
```

You might notice that we have two constructors, which naturally must have the same name--that of the surrounding class! As long as the arguments different in type, order, and/or number, you can reuse constructor and regular method names. When you reuse a method name, you are *overloading* a method. At compile time, the compiler matches up the arguments to decide which method to execute just as if the constructors were named differently.

Inheritance

A `Rectangle` object contains two `Point` objects. A `Rectangle` is not a kind of `Point` nor is a `Point` a kind of `Rectangle`; their relationship is one of containment. There is another kind of object relationship called *inheritance* in which one object is a specialization of another. You say that a class *extends* or *derives* from another class and may add data or behavior. The original class is called the *superclass* and the class derived from it is called the *subclass*.

The first important concept behind inheritance is that the subclass can behave just like any instance of the superclass. For example, if you do not add any data members nor extend the behavior of `Rectangle`, you have simply defined another name for a `Rectangle`:

```
class AnotherNameForRectangle extends Rectangle {
}
```

You can refer to `AnotherNameForRectangle` as a `Rectangle`:

```
AnotherNameForRectangle ar = ...;
Rectangle r = ar; // OK: ar is a kind of Rectangle
```

If instance `ar` is a kind of rectangle, you can ask for its dimensions:

```
Dimension d = ar.getDimensions();
```

The definition of class `AnotherNameForRectangle` is empty, so how can you ask for its dimensions? The subclass inherits all data and behavior of `Rectangle`, therefore, object `ar` can masquerade as a plain `Rectangle`. Just as you are born with nothing, but inherit money and characteristics from your parents, `AnotherNameForRectangle` inherits data and behavior from its parent: `Rectangle`. One way to look at inheritance of data and behavior is to consider it a language construct for a "live" cut-n-paste. As you change the definition of a class, all subclasses automatically change as well.

The second important concept behind inheritance is that you may define new objects as they differ from existing objects, which promotes code reuse. For example, your design may call for a `FilledRectangle` object. Clearly, a filled rectangle is a kind of rectangle, which allows you to define the new class as it differs from `Rectangle`. You may extend `Rectangle` by adding data:

```
class FilledRectangle extends Rectangle {
  protected String fillColor = "black"; // default color
}
```

Unfortunately, `FilledRectangle` does not, by default, know how to initialize itself as a kind of `Rectangle`. You must specify a constructor, but how does a subclass initialize the contributions from its superclass? A subclass constructor can invoke a superclass constructor:

```
class FilledRectangle extends Rectangle {
  protected String fillColor = "black"; // default color
  public FilledRectangle(Point ul, Point lr, String c) {
    super(ul,lr);  // calls constructor: Rectangle(ul,lr)
    fillColor = c; // initialize instance vars of this
  }
}
```

When you construct a `FilledRectangle`, space is allocated for an object the size of a `Rectangle` plus a `String` contributed by subclass `FilledRectangle`. Then Java initializes the `Rectangle` portion of the object followed by the `FilledRectangle` portion. The new operation for `FilledRectangle` takes three arguments, two of which are used to initialize the `Rectangle` component:

```
Point p = new Point(10,10);
Point q = new Point(20,40);
FilledRectangle fr = new FilledRectangle(p,q,"red");
```

You may add behavior to subclasses as well. For example, to allow other objects to set and get the fill color, add a so-called "*getter/setter*" pair:

```
class FilledRectangle extends Rectangle {
  protected String fillColor = "black"; // default color
  public FilledRectangle(Point ul, Point lr, String c) {
    super(ul,lr);  // calls constructor: Rectangle(ul,lr)
    fillColor = c; // initialize instance vars of this
  }
  public void setFillColor(String c) {
    fillColor = c;
  }
  public String getFillColor() {
    return fillColor;
  }
}
```

Another object may then access the added functionality:

```
fr.setFillColor("blue");
String c = fr.getFillColor();
```

Recall that, since `FilledRectangle` is a `Rectangle`, you may refer to it as such:

```
Rectangle r = fr;
```

But, even though `r` and `fr` physically point at the same object in memory, you cannot treat the `Rectangle` as a `FilledRectangle`:

```
fr.setFillColor("blue");// no problem
Rectangle r = fr;       // r and fr point to same object
r.setFillColor("blue"); // ILLEGAL!!!
```

Reference `r` has a restricted perspective of the filled rectangle.

Key Object-Oriented Concepts

Objects, Classes, and Object-Oriented Programs

Summary

- An object-oriented program is a collection of objects
- Objects with same properties and behavior are instances of the same class
- Objects have two components: state and behavior (variables and methods)
- An object may contain other objects
- Instance variables in every object, class variables are like global variables shared by all instances of a class

A running object-oriented program is a collection of objects that interact by sending messages to each other like actors in a theater production. An *object* is an "actor" or self-contained software unit that often corresponds to a real world entity and, therefore, running an object-oriented program is like performing a simulation of the real world.

Many of the objects in a running program will have the same characteristics and conform to the same rules of behavior. Such objects are considered to be *instances* of the same *class*. For example, there may be many instances of the class `Car` in existence at any point in a running traffic simulation. An object-oriented program is a collection of class definitions, each one wrapping up all the data and functionality associated with a single concept or entity specified in the program design.

Consider trying to outline the definition of a car for a computer or person unfamiliar with a car. You would note that a car has certain properties like color and number of doors:

- color
- numberOfDoors

A car is mainly composed of an engine, a body, and four wheels:

- theEngine
- theBody
- theWheels[4]

A car has certain behavior; it can start, stop, turn left, turn right, and so on:

- start
- stop
- turnLeft
- turnRight

By adding a bit of punctuation, you can turn this outline into a Java class definition:

```
class Car {
  // properties
  String color;
  int numberOfDoors;

  // contained objects (Engine,Body,Wheel defined elsewhere)
  Engine theEngine;
  Body theBody;
  Wheel[] theWheels;

  // behavior
  void start() {...}
  void stop()  {...}
  void turnLeft() {...}
  void turnRight(){...}
}
```