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(){...}
}
```

This class definition is analogous to the blueprint for a car versus the car instances themselves.

The data fields are called *instance variables* and the functions embedded within a class are called *methods* to distinguish them from functions or procedures in non-object-oriented languages. Both the data fields and methods are called *members* of the class. A class definition explicitly relates data and the methods that operate on it.

Instance variables and methods

Every instance of a class has a copy of the instance variables. For example, every instance of a Car has a color, number of doors, an engine, a body, and a set of four wheels. If you have 10 cars in your program, you will have 10 strings, 10 integers, 10 engines, etc... Methods that operate on these instance variables are instance methods. Method `start()` manipulates the engine of a single car instance per invocation. Which car instance is determined by the method invocation site as you will see later.

Class variables and methods

How would you count the number of cars created during the execution of the program? In a non-object-oriented language, you would use a global variable and increment it every time you created a new car. You cannot have global variables in Java, but you can have *class variables*, which are like global variables scoped within a class definition. There is exactly one copy of each class variable no matter how many instances of a class you create. If you have 10 cars, they all share a single copy of a class variable. Class variables are defined just like instance variables, but are modified with the `static` keyword. The constructor for a `Car` would increment the count:

```
class Car {
  // class variables
  static int count = 0;

  // instance variables; properties
  String color;
  int numberOfDoors;
  ...

  Car() {
    count = count + 1;
    ...
  }
}
```

Reference class variables as *ClassName.var* as in `Car.count` because they exist as variables of the class not any particular object. In fact, class variables may be referenced even if you have not created any instances of that class.

Methods that operate on class variables only are called class methods and are also modified with the static keyword. For example, here is a method called `resetCount()` that resets the car object creation count:

```
class Car {
  // class variables
  static int count = 0;

  static void resetCount() {
    count = 0;
  }
  ...
}
```

Encapsulation and Data Hiding

Summary

Classes encapsulate state and behavior

Data hiding isolates internals/implementation, providing an approved interface

Encapsulation and data hiding:

> make team programming much easier (you are protected from them and they are protected from themselves)

> reduce what you have to think about at once

> limit propagation of errors

Encapsulation

One of the best metaphors for the software object is the cell, the fundamental biological building block. Loosely speaking, the surface of a cell is a membrane that physically wraps the contents (*encapsulation*), restricts chemical exchange with the outside environment (*data hiding*), and has receptors (*message interface*) that receive chemical messages from other cells. Cells interact by sending chemical messages not by directly accessing the internals of other cells.

This metaphor extends nicely to designing programs. Imagine designing a program that tracks your company's vehicle pool. The first thing you might do is write down all of the types of vehicles in your company and then start to describe their characteristics and behavior. You might have non-vehicle objects, such as the vehicle pool itself, that do not correspond directly to physical objects. If you consider each object in your design a cell, then you will think of it as an independent entity with an internal implementation and external interface for interacting with other objects.

Good programmers have always tried to keep the data and the functions that manipulate that data together, for example in the same file. Object-oriented languages formalize this strategy by forcing you to encapsulate state and behavior within classes.

Data hiding and interface versus implementation

Closely associated with encapsulation is the idea of *data hiding* where certain variables or methods are hidden from other objects. You may prevent foreign objects from directly manipulating the data or other implementation details of an object just as the membrane of a cell hides its internal mechanism. Methods labeled as public are considered part of the approved *interface* for using the object; assume for now that any other members are not part of the interface. For example, the methods of the Car class above should be part of the interface, but the variables should not:

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

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

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

The code within the methods and any of the member variables could be rewritten without affecting the interface and, hence, without forcing changes in other objects.

So, encapsulation and data hiding work together as a barrier that separates the contractual interface of an object from its implementation. This barrier is important for two reasons:

1. The interface/implementation separation protects your object from other objects. Also, when looking for bugs, you can usually limit your search to the misbehaving object.

2. The separation also protects other objects from themselves. Changes in implementation of one object will not require changes in the other objects that use it.

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 because objects may be implemented independently.

Sending Messages

Summary

> Objects communicate via messages
>
> A target object receives a message and implements it with a method
>
> You may overload method names
>
> Binding occurs at run-time

Objects in a running program collaborate by sending messages back and forth to elicit responses, changes of state, or more message sends. An object's behavior is characterized by how it reacts to received messages. Message sends are of the form:

```
targetObject.message(arguments);
```

At run-time, the `targetObject` receives the `message`, looks up the `message` name in its method list, and executes the method with the matching *signature*; the signature includes the name of the method plus the number and type of the arguments. Sending `start` to a `Car` object such as:

```
aCar.start();
```

means that `aCar` would look up `start` in its method table, find method `start()`,

and execute it.

A method name may be reused, or *overloaded*, if the arguments differ in type or number. For example, you could add another `start()` method to class `Car` as long as the signature was different:

```
class Car {
  ...
  public void start() {...}
  public void start(float acceleration) {...}
  ...
}
```

Sending message `start` to `aCar` would now choose between the `start()` and `start(float)` methods of `Car` depending on the argument list of the message send:

```
aCar.start();              // call empty-arg start method
aCar.start(9.08);   // call acceleration start method
```

Method overloading makes an object choose between methods with different signatures within a class and, while very convenient, is not a core principle of object-oriented programming. Overloading should be distinguished from polymorphism, which is described in the next section, where a message forces a choice between methods with the same signature, but in different classes.

Polymorphism

Summary

> Polymorphism is the ability for a single object reference to refer to more than a single concrete type
>
> The benefit of polymorphism is that the introduction of a new type does not force widespread code changes

The more specific your program statements have to be, the less they can deal with change and the shorter their useful lifespan. The primary adaptability of object-oriented programming derives from the message sends between objects. Consider that, when you call a function in C or other procedural language, you are specifying exactly the code you want to execute. While easy to follow, such exactness renders your program extremely brittle. Sending a message to an object, on the other hand, relies on the receiver object to react to the message by executing

an appropriate method. You can swap out the receiver object and replace it with any other object as long as that object answers the same message. No change is required to the message send code. For example, if someone hands you an object and you know it is one of 5 objects that can respond to message `start`, there are 5 possible `start()` methods that could be executed in response to the `start` message. This nonspecificity and flexibility is called *polymorphism* because the object may take one of many forms.

If there are 5 different classes that can answer `start`, there are 5 possible methods that could be executed by this one statement if `v` is an instance of one of those 5 classes:

```
v.start();
```

At run-time the exact type of object, such as `Car`, is known and the target object binds the `start` message to its `start()` method.

The equivalent code written in C would use a function call instead of a message send and would have to specify exactly which function to call. If there is only one data type that you can `start`, a single function call suffices:

```
startCar(v);
```

Adding a new data type, say, `Truck`, would mean adding an if-then-else statement to deal with the newly-introduced type; you would have to do something like this:

```
// C fragment
if ( v.type == CAR_TYPE ) { // assume vehicle stores type
  startCar(v);
}
else if ( v.type == TRUCK_TYPE ) {
  startTruck(v);
}
```

A non-object-oriented language simply has no mechanism for expressing the similarity of a truck and a car and that `start` is understood by both. (*Typically, procedural programmers will make a single `struct` or record that can hold state for any similar type of thing and then they make an integer variable that indicates what the struct is supposed to be--truck or car or whatever*). The big difference between the approaches is that in C you have to handle the various types at the call site manually whereas an object-oriented language does this "switch on type" for you.

Some object-oriented languages, such as SmallTalk, allow you to send any message you want to any other object, which can lead to annoying run-time errors like

"object does not implement that message." Further, while both `Rectangle` and `GunFighter` might understand message `draw`, the same message is surely interpreted in two different ways. It would be nice if the compiler informed you at compile-time that you were sending the same message to very different objects.

Java provides this service via strong typing. For each object you refer to, you have to specify the type of object, kind of object, or behavior of the object. You cannot perform assignments between dissimilar objects, which prevents you from sending a message to a radically different object inadvertently.

```
GunFighter gf = ...;
Rectangle r = gf; // INVALID!!! compile error!
r.draw();
```

So how does Java know which objects are similar and which are unrelated? Java uses the notion of inheritance to express similarity between classes. The next section explores inheritance and revisits polymorphism in more detail.

Inheritance

Summary

Inheritance specifies the commonality or similarity relationship between classes

It is a relationship of identify

A subclass inherits state and behavior from the superclass

Any reference to an object can be given a reference to a subclass instead

The inheritance relationships together form a class hierarchy, which helps to organize your program

Classes whose primary job is to group similar classes is often called an abstract class

Inheritance lets you define by difference: augment, override, or refine members inherited from superclass

Polymorphism uses the inheritance relationship

> as its definition of similarity; Java restricts a
> message send to similar objects

Procedural languages like C are less flexible than object-oriented languages because they cannot treat items of different, but compatible types in a similar manner. For example, two types, `Manager` and `Developer`, may be similar in the sense that they are both employees of a company, but there is no mechanism for you to exploit this similarity. You cannot pass a `Manager` to a function that, for example, prints out the name of a `Developer` you pass in even though the process of asking both types to print their names could be identical.

Object-oriented languages support a simple, but potent idea called *inheritance* that lets you express the similarity between different class types. A class (the *subclass*) that inherits from another acquires all of the properties and behavior of the parent class (*superclass*). For this reason any two classes that inherit from the same superclass share a common set of properties and behavior, namely that of the superclass, and you can treat the two subclasses similarly.

Inheritance therefore expresses a relationship of identity or commonality, sometimes called an "*is kind of*" relationship. For example, to express that `Manager` and `Developer` are similar, make them share a superclass (perhaps `Employee`) that encapsulates their common properties and behavior. This indicates that managers and developers are kinds of employees. Java uses the `extends` keyword to specify this inheritance relationship. Here are the skeletons for these three class types:

```
class Employee {
  ...
}

class Manager extends Employee {
  ...
}

class Developer extends Employee {
  ...
}
```

Because inheritance is an expression of similarity, you can:

1. organize your program as a hierarchy of related classes

2. define new classes as they differ from existing classes

3. refer to many similar classes as a group rather than individually resulting in

more flexible programs

Class hierarchies

Many procedural languages are just "bags" of variables, data types, and functions as there is no language construct to group them or show the relationships between them. Some languages have modules or packages that can group things but, still, the relationship between functions and data and between data types remains unspecified. Often, the proximity of one function to another is due to the timing of when you typed in the code.

The class mechanism of object-oriented languages lets you encapsulate data and behavior associated with each conceptual entity and the class inheritance relationship lets you specify the relationship between the classes. The combined result of all inheritance relationships is a tree, or *class hierarchy*. In Java, this class hierarchy has a single root, class `Object`, from which all classes directly or indirectly inherit. You can interpret this to mean that all objects are related in that they all share the characteristics of a basic `Object`.

When designing an object-oriented program, you decide what objects you will need in the system and then you look for similarities between objects that you can exploit. For example, given objects of types `Manager`, `Developer`, `Marketeer`, `VP`, `CEO`, and `Guru`, you would probably define the following inheritance relationships:
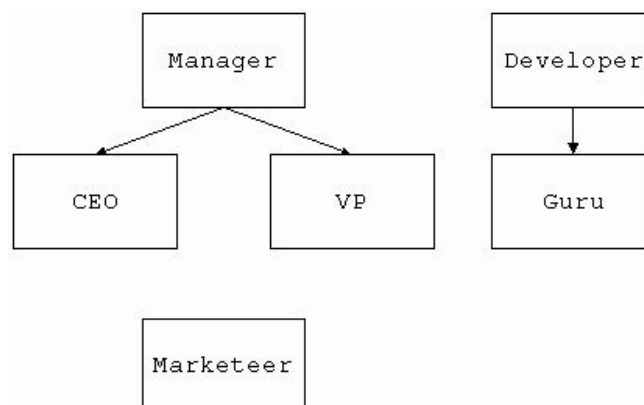
When looking for similarities, you will often create *abstract* classes that exist purely to group related classes and to contain the factored data and methods; abstract classes are organizational classes that have no run-time instances. Creating class `Employee` helps to organize your classes as it shows the relationship now between all classes: they are all employees. Further, `Employee`

---

would probably contain common data and behavior such as method `getName()`, automatically extending that functionality to all employees, that is all subclasses of `Employee`.
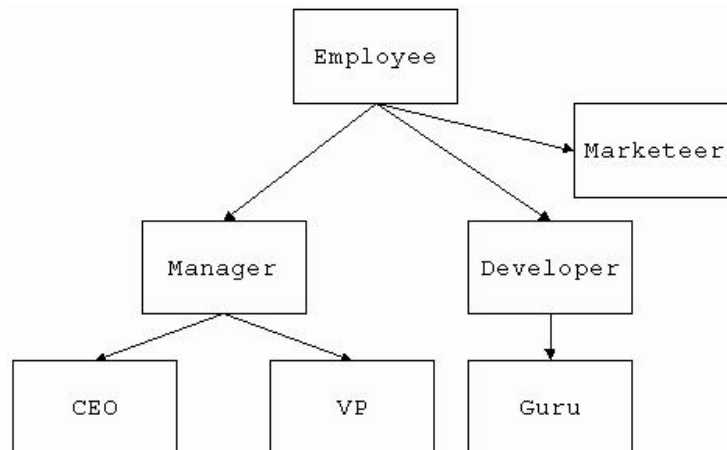
For small sets of classes, a decent class hierarchy is usually obvious, but this is not true for large programs. Different designers will analyze the same situation differently yielding different classes and class organizations. Following this, different programmers will add different "helper" classes to actually implement the design. The classes and their relationships depends on the perspective and experience of the designers and developers.

Defining by difference

Another way to exploit the class inheritance relationship is to define new classes as they differ from existing classes. For example, to define `SalesPerson`, you could extend `Employee` and to define `Analyst` you could extend `Developer` (assuming all analysts know how to develop software).

Organizing your program into a hierarchy, then, means that you have less work to do to augment a program. Inexperienced developers often take this to mean that you extend a class if you need to reuse or borrow some or all of its code. Unfortunately, this practice results in poorly organized and specified programs. Use the class hierarchy to show the appropriate relationships between the classes and code reuse occurs naturally as a side effect. A good rule of thumb is that if the phrase "x is a kind of y" sounds correct in English, x is probably a appropriate subclass of y. For example, reconsider the `Point` and `Rectangle` classes from above. A `Rectangle` has two points so you might be tempted to have `Rectangle` extend `Point` so you could reuse all the `Point` code, but a `Rectangle` **contains** two `Point` objects it is not a kind of `Point`.

When you extend a class you have the opportunity to add data or method
members to *specialize* or augment the characteristics of the super class. For
example, you might define Employee as:

```
class Employee {
  String name;
  int id;
  public Employee(String name, int id) {// constructor
    this.name = name;
    this.id = id;
  }
  public void setName(String name) {        // "setter"
    this.name = name;
  }
  public String getName() {          // "getter"
    return name;
  }
}
```

Then, you can then augment Employee with, say, an office number (plus the usual
"getter/setter") methods to define Manager:

```
class Manager extends Employee {
  int officeNumber;
  public Manager(String name, int id) {
    super(name, id);
  }
  public void setOfficeNumber(int n) {
    officeNumber = n;
  }
  public int getOfficeNumber() {
    return officeNumber;
  }
}
```

The constructor does nothing in this case but delegate to the constructor of
Employee.

Another thing you can do with inheritance is *override* methods inherited from
your superclass to specialize the behavior of a subclass. For example, you might
want to redefine getName() in Manager so that it always returns "Boss" instead
of the manager's name:

```
class Manager extends Employee {
  ...
  public String getName() {
    return "Boss";
  }
}
```

Now, you will get the following behavior:

```
Manager m = new Manager("Jimbo", 213002);
String mname = m.getName(); // mname is "Boss"
```

A `Developer` on the other hand, would still respond with the employee's name unless it too overrode `getName()`:

```
Developer d = ...;
String dname = d.getName(); // dname is developer's name
```

One final thing you can do with inheritance is *refine* the behavior of the superclass. If you wanted a `Manager` to respond to `getName` with the person's name prefixed with "Boss", the following definition would suffice:

```
class Manager extends Employee {
  ...
  public String getName() {
    // return name prefixed with Boss
    return "Boss" + super.getName();
  }
}
```

The expression `super.getName()` is a reference to the definition of `getName()` in the superclass. Simply referencing `getName()` within `Manager`'s `getName()` would, naturally, result in an infinite-recursion loop. You need the scope override reference "`super.`" to disambiguate which `getName()` method to call.

A tricky situation can arise when a subclass delegates functionality to a superclass that in turns calls a method that is overridden in the subclass. For example, assume `Manager` does not override `getName()`, delegating it to superclass `Employee`. But, `Employee.getName()` simply returns the result of calling `toString()`. What value does `name` get in the following code?

```
Manager m = new Manager("Jimbo", 213002);
String name = m.getName();
```

It depends. Scenario 1: `Employee.toString()` returns "An Employee" and there is no `Manager.toString()` method. Variable `name` will be "An Employee" because target `m` delegates response to `getName` in `Employee`, which calls `Employee`'s `toString()`. Scenario 2: `Manager.toString()` returns "A Manager". Now, `name` will be "A Manager" because `Employee.getName()` calls `toString()` which you knows is really `this.toString()`. The target, `this` (which points at the same thing as `m`), is a manager and, hence, `Manager`'s `toString()` is called.

You might wonder if you may remove or hide functionality inherited from an

---

object's superclass? No, because the subclass could no longer behave like the superclass and, hence, the subclass would not be the same kind of object. If you hid `getName()` in `Manager`, a `Manager` could not act like a generic `Employee` since you could not ask it to respond with its name.

Inheritance and type equivalence

Java does not let you send any message to any object--the objects must be similar by some definition. Class inheritance is the primary definition of similarity for polymorphism in strongly typed systems like Java. Two objects are similar if they inherit from the same parent directly or indirectly; that is, they are both kinds of the same object.

Java restricts message sends to dissimilar objects by preventing assignment of completely dissimilar types. You may assign an instance of any subclass to its superclass, but not the other way around:

```
Manager m;
Employee e;
Developer d;

e = m; // OK: a Manager is a kind of Employee
e = d; // OK: a Developer is a kind of Employee
m = e; // NO: an Employee is not necessarily a Manager
```

Nor can you assign instances of classes that are at the same level in the hierarchy:

```
m = d; // NO: a Manager is not a kind of Developer
d = m; // NO: a Developer is not a kind of Manager
```

The type of the object on the left-hand-side must be a superclass of the type on the right-hand-side. This prevents you from sending subsequent messages to an object that might not respond to that message. For example, you cannot create a generic employee and then treat it like a manager:

```
Employee e = ...;
Manager boss = e;            // compile-time type error!
boss.setOfficeNumber(210); // run-time error!
```

Inheritance and polymorphism

If you have an array of `Employee` objects such as developers, managers, gurus, and marketeers, can you print out each object's name with the following loop?

```
Employee[] employees = ... ;
for (int i=0; i<employees.length; i=i+1) {
  System.out.println(employees[i].getName());
```

```
}
```

Sure. The beauty of inheritance is that you can treat any subclasses of `Employee` just like an `Employee`. Each time through the loop, the message `getName` is sent to the `i`th object in the `employees` array. The `employees[i].getName()` expression does not uniquely specify what code to run. It only specifies that the `getName()` method of one of `Employee` subclasses will be executed. The exact method cannot be determined until run-time when the concrete object type of each `employees[i]` is known. Therein lies the flexibility of polymorphism. Ten years from now, if someone adds another kind of employee, this loop would still work without modification.

You can see polymorphism at work everyday in the real world. Programming instructors use polymorphism to teach, for example. Each student that walks into the classroom is treated strictly as a developer and, since the instructor presumably does not know the students, that is all the instructor can assume. Any other attributes or specialties of the person are irrelevant and hidden such as their nationality, skill level, political affiliation, etc... As long as they are developers, the instructor can teach them. The instructor teaches by sending messages that developers understand to the students such as "open a new project called DBQuery."

In Java, the analogous situation might be represented as a `teach(Developer)` method within an `Instructor` class:

```
class Instructor {
  void teach(Developer d) {
    d.openNewProject("DBQuery");
    ...
  }
  ...
}
```

A company might have 5 different kinds of developers, but any kind can be passed to method `teach(Developer)` if the developers are defined as subclasses of `Developer`. If a nondeveloper happened to walk into the classroom, they would not understand what was being said. Similarly, if a nondeveloper was somehow passed to `teach(Developer)` a run-time error would occur indicating that the object does not know how to respond to the `openNewProject` message.

The instructor could be introduced to a random person in the hall, so another method, `greet()`, would make sense:

```
class Instructor {
  void teach(Developer d) {
```

```
      d.openNewProject("DBQuery");
      ...
   }
   void greet(Employee e) {
     String name = e.getName();
     ...
   }
   ...
}
```

Method `greet(Employee)` is much more widely applicable than `teach(Developer)` and would work for any kind of employee including developers:

```
Instructor instr = ...;
Developer jim = ...;
Marketeer pat = ...;
instr.greet(jim);
instr.greet(pat);
instr.teach(jim); // only jim can be taught programming
```

On the other hand, the instructor can assume much less about the incoming objects. The instructor is limited to asking the `Employee` objects for their name.

Polymorphism and Interface Inheritance

Summary

> Class inheritance specifies identity and identity implies behavior, but behavior does not imply identity
>
> Encapsulation and data hiding team up to separate the internal implementation of an object from its external approved interface
>
> Java formalizes the notion of an interface so that you may refer to objects by their partial behavior rather than their identity, resulting in more flexible and adaptable programs

Procedural languages like C force you to specify exact types like "*this is a car*." Object-oriented languages, because of class inheritance, let you be more flexible: "*this is a kind of vehicle*." Java lets you be even more flexible; you can say "*this object can turn left and right*." The secret to writing flexible code is putting the fewest restrictions possible on the types of objects you refer to. If you can get away with specifying the kind of object instead of a concrete type, do it. If you

can get away with specifying a partial behavior of an object instead of the kind of object, do it.

Class inheritance implies identity and identity implies a behavior, but similar behavior does not necessarily imply the same identity. For example, broccoli and corn share the vegetable identity and, hence, both behave like vegetables:

```
class Vegetable {
  public void die() {...}
  void growRoots() {...}
  ...
}
class Broccoli extends Vegetable {...}
class Corn extends Vegetable {...}
```

Cows and chickens share an animal identity and behave like animals:

```
class Animal {
  public void die() {...}
  void eat() {...}
  ...
}
class Cow extends Animal {...}
class Chicken extends Animal {...}
```

While all of these flora and fauna objects may share a common message such as die, being mortal does not indicate vegetable or animal. If all you care about is an object's mortality, you need a type system that lets you specify behavior rather than just identity:
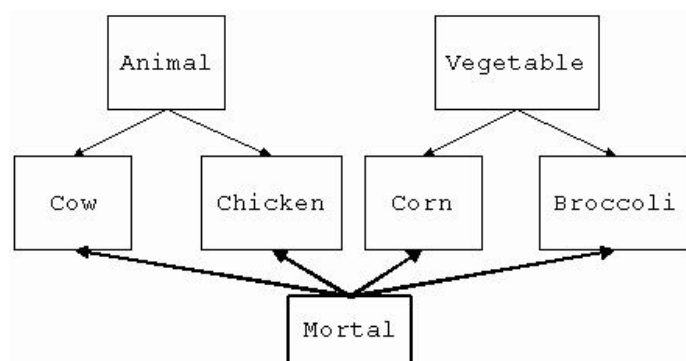
Java provides a construct called an *interface* that looks exactly like a class, but without any method implementations or data variables. Here is an interface that defines the mortality behavior:

```
// all Mortal objects implement method die().
interface Mortal {
```

```
  public void die();
}
```

If a class of objects can answer die, you say that the class *implements* Mortal. You can then rewrite Animal and Vegetable to indicate their mortality:

```
class Animal implements Mortal {
  public void die() {...}
  void eat() {...}
  ...
}
```

```
class Vegetable implements Mortal {
  public void die() {...}
  void growRoots() {...}
  ...
}
```

The power of interfaces comes when you reference objects by their behavior as in the following method that accepts a Mortal object:

```
void kill(Mortal o) {
  o.die(); // could be an Animal or Vegetable
}
```

As a more realistic example, consider the List behavior:

```
interface List {
  public void add(Object o);
  public Object getElementAt(int i);
}
```

Two totally different classes can implement the behavior:

```
class LinkedList implements List {
  public void add(Object o) {
    ...
  }
  public Object getElementAt(int i) {
    ...
  }
  ...
}
```

```
class DynamicArray implements List {
  public void add(Object o) {
    ...
  }
  public Object getElementAt(int i) {
    ...
  }
  ...
}
```

You can create either a `LinkedList` object or a `DynamicArray` object and refer to either by their shared `List` behavior.

```
LinkedList llist = ...;
DynamicArray darray = ...;
List alist;
alist = llist;
alist = darray;
```

Given some method `foo()` that accepts an object behaving like a `List`:

```
void foo(List a) {
  a.add("Joe");
  a.getElementAt(0);
}
```

you can can pass either data structure to the same method:

```
foo(llist);
foo(darray);
```

The benefit is that you can pass the linked list if you need fast insertions or pass the dynamic array if you need fast access to the ith element.

A Start-to-Finish Example

Introduction

To gain a better understanding of object-oriented programming, it is a good idea to walk through a complete example from analysis of the problem to its implementation and execution. In this section, you will build a set of classes representing a book, which clearly demonstrates inheritance and containment relationships as well as polymorphic references.

Analysis and Design

First, think about what your goals. You would like a `Book` class that you can add chapters to and that you can tell to `print()` out. Next, think about the players in your "simulation". What does a book look like? It has a title, author, a table of contents, a number of chapters, and an index. A chapter has sections and paragraphs. A section has sections and paragraphs too. You might decide that your program then has the following kinds of objects:

Book

Title

Author

```
TOC

Index

Chapter

Section

Paragraph
```

The next step is often to design the containment relationship. In other words, what objects refers to what objects? Who is composed of what other objects? In this case, you can represent the containment relationships by imagining a sample book with a few chapters, sections, and paragraphs. Indentation in the following "book" implies containment. For example, a book has a title, an author, set of chapters, etc... Chapters have sections and so on:

```
Book
  Title: "JDK User's Guide"
  Author: "MageLang"
  TOC: "..."
  Paragraph: "The Preface"

    1.  Chapter: "Overview of JDK"

          1.  Section: "Introduction"

                1.  Paragraph: "1st para of intro section of chapter 1"

                2.  Paragraph: "2nd para of intro section of chapter 1"

          2.  Section: "Features"

                1.  Paragraph: "1st para of features section of chapter 1"

          3.  Section: "Help"

                1.  Paragraph: "1st para of help section of chapter 1"

                2.  Paragraph: "2nd para of help section of chapter 1"

    2.  Chapter: Getting Started

          1.  Section: "Installation"

                1.  Paragraph: "1st para of install section of chapter 2"
```
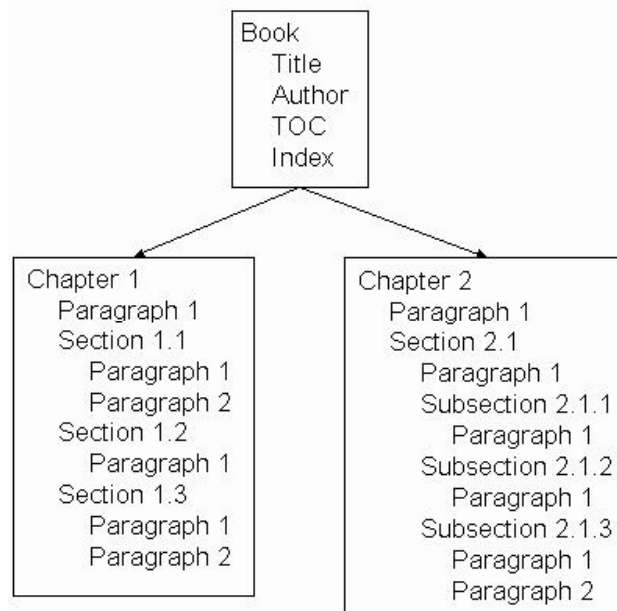
2. Section: "Unzipping"

    1. Paragraph: "1st para of unzip subsection of install section chapter 2"

3. Section: "Setting CLASSPATH"

    1. Paragraph: "1st para of classpath subsection of install section chapter 2"

4. Section: "Setting PATH"

    1. Paragraph: "1st para of setting PATH section of chapter 2"

    2. Paragraph: "2nd para of setting PATH section of chapter 2"

Index: "..."

Visually, you might show a rough overview for this book containing 2 chapters:

```
                    Book
                      Title
                      Author
                      TOC
                      Index

      Chapter 1                      Chapter 2
        Paragraph 1                    Paragraph 1
        Section 1.1                    Section 2.1
          Paragraph 1                    Paragraph 1
          Paragraph 2                    Subsection 2.1.1
        Section 1.2                        Paragraph 1
          Paragraph 1                    Subsection 2.1.2
        Section 1.3                        Paragraph 1
          Paragraph 1                    Subsection 2.1.3
          Paragraph 2                      Paragraph 1
                                           Paragraph 2
```

In more detail, you could show that chapter 2 contains a paragraph and a section with three subsections, which contain paragraphs:
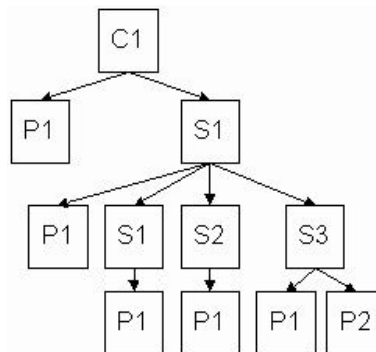
Now that you have a decent idea of what objects refer to what kinds of objects, you need to start looking for polymorphism so you can start designing object properties. In other words, where do you need to reference objects of more than a single type as if they were the same? Notice that `Chapter` can reference both a paragraphs and sections. If you have a list of the elements property within `Chapter`, the list elements can be either `Paragraph` or `Section` objects. This ambiguity is the source of object-oriented programming's power: you can deal with multiple elements as if they were the same by some definition. In other words, the same code works in more than one situation. Notice that `Section` also must reference paragraphs and sections. It also has a polymorphic reference.

Revisit the stated goal of printing out a book. From a design point of view, you should be able to tell any element within a book to print itself out, which would tell any contained elements to print out and so on. The benefit of polymorphism is that you can treat all the elements the same in the sense that they can all answer `print`. A piece of code that says

```
element.print();
```

will print out any element, regardless of element's actual type at run-time. Consequently, `Chapter` and `Section` will need code like this to handle their polymorphic list properties.

So, what should the stated compile-time type of `element` be? This question brings us to object-oriented programming's definition of *similarity*. Recall that you use inheritance to show the relationship between classes (and also to organize your code). Any two classes that are subclasses of the same parent are considered related by identity, implying a common set of characteristics. In this case, all book elements are similar in that they answer message `print`. Then, to show the similarity between elements, create an abstract class that embodies the commonality, say, `TextEntity`:

```
abstract class TextEntity {
  public abstract void print();
}
```

Because this object is generic, it does not actually know how to print anything out--the subclasses will have to implement that functionality, hence, `print()` is `abstract` here. At this point, the classes in your design are related as follows:
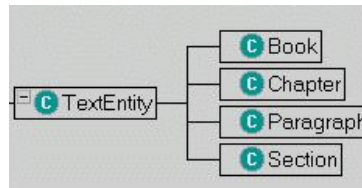

The type of `element` is, therefore, `TextEntity` and it may refer to any object whose type is a subclass of `TextEntity`. While some objects will have a more complicated procedure for printing than others (for example, `Paragraph` vs `Section`), you will leave it up to the target of message `print` to do the right thing.

The needs of polymorphism have forced the initial specification of the inheritance relationships, however, another reason to look for similarities is to factor out common functionality.

Take another look at the inheritance hierarchy. Class `Paragraph`'s `print()` method will be trivial (it will probably just print out a text property), however, all the other classes are lists, or containers, of other text entities. That similarity is enough to force a restructuring of the class hierarchy. How do you show similarity? Define another class that abstracts or factors out the commonality from `Book`, `Chapter`, and `Section`:

```
class TextContainer extends TextEntity {
  public void print() {
    // walk list of contained elements and tell
    // them to print themselves out.
  }
  public void add(TextEntity e) {
    // add e to the list of contained elements
  }
}
```

In this case, we are factoring the common functionality of walking and printing a list of text entities so the `print()` method has an implementation and is not `abstract`. Also, text containers share a need to add entities to themselves for construction purposes.
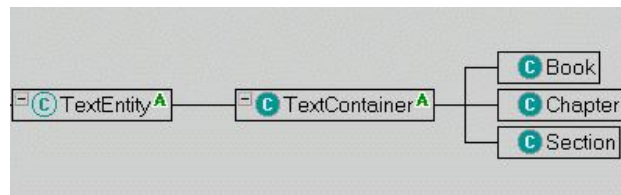
`TextContainer` extends `TextEntity` because it can answer `print`:
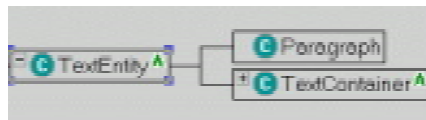
This relationship allows you to nest sections--a `Section` contains a list of
`TextEntity` objects, which can be another `TextContainer`. This also means,
unfortunately, that you can put a `Chapter` under a section, which is bogus. The
routines that construct a book must prevent illegally nested book structures.

Now that the relationships between classes known, finish off the infrastructure
within the classes by specifying the properties:

```
class Book extends TextContainer {
  private String TOC;
  private String index;
  private String title;
  private String author;
  ...
}

class Chapter extends TextContainer {
  private String title;
  ...
}

class Section extends TextContainer {
  private String title;
  ...
}

class Paragraph extends TextEntity {
  private String text;
  ...
}
```

At this point, you have specified how objects interact (they send `print` messages
around) and have only to implement `print()` where appropriate, implement
`add()` in `TextContainer` and define constructors for the various classes.

Coding

If you are going to walk through the complete coding of this problem, you may find the following coding strategy helpful when implementing the design:

1. Define all the classes for the actors without inheritance

2. Add the inheritance relationships

3. Add the properties

4. Implement `print()` where appropriate

5. Implement the constructors and `add()` methods used for construction

6. Make a `TestHarness` class with a `main()` that constructs a book and does tells it to print itself out

7. Walk through the execution of the program with a debugger to see how the objects get constructed and how you can look at the contents of the overall book containment "tree".

8. Walk through the execution of the program with a debugger, tracing methods and watching how it prints out and recurses the containment relationships.

Code Listing

```
class Book extends TextContainer {
    private String TOC;
    private String index;
    private String title;
    private String author;

    public Book(String title, String author) {
    this.title = title;
    this.author= author;
    }

    public void print() {
    // do title page
    System.out.println();
    System.out.println();
    System.out.println();
    System.out.println(title);
    System.out.println();
    System.out.println("BY");
    System.out.println();
    System.out.println(author);
```

```java
        System.out.println();
        System.out.println(TOC);

        super.print();

        System.out.println();
        System.out.println(index);
        }

        public void setIndex(String i) {
        index = i;
        }

        public void setTOC(String t) {
        TOC = t;
        }
}

class Chapter extends TextContainer {
    private String title;

    public Chapter(String t) {
        title = t;
    }

    public void print() {
        System.out.println();
        System.out.println(title);
        System.out.println();
        super.print();
    }
}

class Section extends TextContainer {
    private String title;

    public Section(String title) {
    this.title = title;
    }

    public void print() {
    System.out.println();
    System.out.println(title);
    super.print();
    }
}

class Paragraph extends TextEntity {
    private String text;

    public Paragraph(String t) {
    text = t;
    }
```

```java
    public void print() {
    System.out.println(text);
    }
}

abstract class TextEntity {
    public abstract void print();
}

abstract class TextContainer extends TextEntity {
    private static final int MAX_CONTAINED = 20;
    private int n=0;
    private TextEntity[] elements;

    public void add(TextEntity e) {
    if ( elements==null ) {
        elements = new TextEntity[MAX_CONTAINED];
    }
    elements[n] = e;
    n = n + 1;
    }

    public void print() {
    for (int i=0; i<n; i=i+1) {
        elements[i].print();
    }
    }
}

class TestHarness {
public static void main(String[] args) {
    Book b = new Book("JDK User's Guide", "MageLang");
    b.setTOC("Preface\n"+
            "1. Overview of JDK\n"+
            "2. Getting Started\n");
    b.setIndex("the index");
    Paragraph preface = new Paragraph("The Preface");
    Chapter overview = new Chapter("1. Overview of JDK");
    Chapter started = new Chapter("2. Getting Started");
    b.add(preface);
    b.add(overview);
    b.add(started);

    // chapter 1
    Paragraph hello =
     new Paragraph("Intro paragraph of chapter 1");
    Section intro =
     new Section("Introduction");
    intro.add(
     new Paragraph("1st para of intro section of chapter 1")
        );
    intro.add(
        new Paragraph("2nd para of intro section of chapter 1")
        );
    Section features = new Section("Features");
    features.add(
        new Paragraph("1st para of features section of chapter 1")
```

```
            );
      Section help = new Section("Help");
      help.add(
            new Paragraph("1st para of help section of chapter 1")
            );
      help.add(
            new Paragraph("2nd para of help section of chapter 1")
      );
      overview.add(hello);
      overview.add(intro);
      overview.add(features);
      overview.add(help);

      // chapter 2
      Section install = new Section("Installation");
      install.add(
       new Paragraph("1st para of install section of chapter 2")
            );
      // 2 subsections of section 1
      Section unzip = new Section("Unzipping");
      unzip.add(
       new Paragraph(
        "1st para of unzip subsection of install section chapter 2")
            );
      Section classpath = new Section("Setting CLASSPATH");
      classpath.add(
            new Paragraph(
            "1st para of classpath subsection of install section chapter 2")
            );
      Section path = new Section("Setting PATH");
      path.add(
            new Paragraph(
            "1st para of setting PATH subsection of install chapter 2")
            );
      path.add(
            new Paragraph(
            "2nd para of setting PATH subsection of install chapter 2")
            );
      install.add(unzip);
      install.add(classpath);
      install.add(path);

      started.add(install);

      // NOW: PRINT THE BOOK!
      b.print();
   }
}
```

[*MML: 1.03*]
[*Version: $ /OOProgWithJava/OOProgWithJava.mml#5 $*]