Introduction to Interactive Programming
by Lynn Andrea Stein
A Rethinking CS101 Project

Intelligent Objects and Implicit Dispatch

Chapter Overview

- How can I exploit method "ownership" to make objects do what I want?
- How do I pass behavior around?
- How do I know which method will be invoked?

Methods belong to objects. In some cases, as when getter and setter methods allows access to an object's internal state, the reason for housing methods in objects is clear. But in many cases, it may be less obvious why a method ought to be affiliated with a particular object. In this chapter, we look at several cases in which methods are used in concert with their owning objects to accomplish tasks that might not be obvious.

Methods can be used as a way to create implicit dispatch. Many objects, belonging to many different classes, can each be given a method of the same name (and footprint). In this case, dispatching to the correct code is as simple as asking the object to perform this method for you.

Fixing the name of the method but leaving the owning object to vary allows you to do a wide range of things. You can, in effect, pass a method as an argument (by passing its containing object), return a method from a procedure (by returning its containing object), or store it in a name or other structure (by storing its containing object). You can remember who called you and arrange to call that object back; you can build complex homogeneous structures by exploiting the fact that one object is associated with other, equally intelligent, objects that can cooperatively solve problems that none could solve individually.

Each of these mechanisms works because every method is associated with an object. If the method name is fixed at the time that the program is written, its target object can be allowed to vary, allowing a runtime decision as to which piece of code -- which instructions, which method body -- should actually be executed.

Objectives of this Chapter

1. To understand Java's method dispatch mechanism.
2. To be able to use the same-named method in different classes of objects to create an implicit dispatch.
3. To appreciate how Runnables can be used to encapsulate procedures.
4. To learn how to set up and use callbacks so that a method can convey information to its calling object without returning.
5. To recognize various forms of recursion and to be able to use structural recursion as a problem-solving technique.

6. To understand, recognize, learn how, increase familiarity, master details, appreciate, discover, be able to ....

Procedural Encapsulation and Object Encapsulation

In the previous chapters, we saw how a central control loop can be used as a dispatcher, invoking different methods at different times depending on circumstances such as the value of a particular piece of state. We also saw how the different responses can be packaged up inside methods and how these methods in turn can be encapsulated inside objects. In this chapter, we will take these ideas one step further and use Java's method dispatch mechanism (plus some clever design) to determine what response is appropriate under various circumstances.

Before we turn to the use of objects as a dispatch mechanism, let's briefly review some of the properties of methods and of objects.

A method is a set of instructions to be followed. The method instructions are executed when an instruction-follower evaluates a corresponding method invocation expression, i.e., a call to the method. The method instructions may require some information to be able to execute; these are the arguments to the method. The method instructions may also produce some information; this is the method's return value.

Every method belongs to a particular object; there are no methods "just floating around" in Java. Each method body is textually contained in a class definition. Regular methods belong to individual instances of the class in which they are textually contained. (Static methods belong to the class object itself.)

For example, if `yesBox` is a Checkbox and you want to find out whether `yesBox` is currently selected, you can ask `yesBox` to supply you with that information by using the method invocation expression `yesBox.isSelected()`. There's no way to just ask `isSelected()`, though: you have to know whose `isSelected()` method it is.

Methods encapsulate behavior, but they do not by themselves encapsulate state. This is the role of objects. An object typically contains both methods -- sets of instructions -- and persistent information. For example, the Checkbox named by `yesBox` has a method called `isSelected()`, which provides instructions for how to determine whether `yesBox` is currently checked. When the expression `yesBox.isSelected()` is evaluated, those instructions are executed and the desired information is produced. But when the method is not being invoked, the method itself doesn't have any information or action. In contrast, even in the absence of any method invocation, the Checkbox `yesBox` contains state indicating whether it is currently selected, perhaps in the form of a private boolean field.

Objects, then, package both behavior (in the form of methods that can be invoked) and persistent state that provides a background context for that behavior. (Presumably `yesBox.isSelected()` behaves differently depending on whether the hypothetical private boolean field is true.) An object exists even when none of its methods is being invoked, and its fields persist between method invocations. An object is thus a powerful mechanism for modeling parts of the world. By making that state internal to the object, hiding it from external access, and providing a set of methods that give selective access to that state, objects can be used to encapsulate the coherent behavioral aspects of real-world things. The method `isSelected()` by itself would have little meaning. The object `yesBox` provides a context for the `isSelected()` method, so that it legitimately models coherent persistent behavior.

The name of a method to be invoked must be chosen at the time that you are writing your program. In contrast, the particular object whose method will be invoked need not be known until the time that you actually run the program. For example, when the expression `yesBox.isSelected()` is written, the method name -- `isSelected` -- and even its footprint -- no arguments -- is already known. No other method can be invoked with this expression. But at the time that the expression is written, it may not be possible to tell to which object the name `yesBox` will refer. It may, in fact, not even be possible to tell the exact type of the object to which `yesBox` refers, although we know that it will be some type of Checkbox. (It could be any subtype of Checkbox.)

In the remainder of this chapter, we will see how fixing the method name and allowing its target object to vary gives the programmer a great deal of additional power. In the first -- central -- example of this technique, we will shift specialized behavior from their previous location in the handler methods within a single object to a new role within separate objects, objects encapsulating both those handler methods and associated state. We will see how this migration of behavior from procedural encapsulation to object encapsulation provides a different model for dispatch, and how it can be used to make object-oriented programming a remarkably powerful technique.

From Dispatch to Objects

Consider the following problem: You are writing code that will retrieve objects, one at a time, and print them out to the user. Some of these objects will be Strings. Some of the objects will be Points, items representing two-dimensional coordinates. A Point object has methods to retrieve its individuate coordinates, called getX() and getY(), each returning an int. And some of the objects will be Dimensions, items representing two-dimensional extents, with int-returning getWidth() and getHeight() methods. Your job is to write the printObject method.

A Straightforward Dispatch

You might implement this by using a simple dispatch mechanism. Since this dispatch is done on the basis of the object's class, you cannot use a switch statement. So we'll try an if:

```
public void printObject( Object o )
{
    if ( o instanceof String )
    {
        Console.println( o );
    }
    else if ( o instanceof Point )
    {
        Point p = (Point) o;
        Console.println( "Point: (" + p.getX()
                         + "," + p.getY() + ")" );
    }
    else if ( o instanceof Dimension )
    {
        Dimension d = (Dimension) o;
        Console.println( "Dimension: (" + d.getWidth()
                         + "," + d.getHeight() + ")" );
    }
}
```

[Footnote: This code suffers from a few problems, not the least of which is that it doesn't do anything about the possibility that o is none of the above. While we'd never write such code in a real application, we'll skip the else error condition clause here for pedagogic succinctness.]

Procedural Encapsulation

Of course, knowing what we do about procedural encapsulation, this looks like a superb opportunity to break out the concisely describable code. There are two relatively obvious routines lurking here:

```
public String pointToString( Point p )
{
    return "Point: (" + p.getX() + "," + p.getY() + ")";
}
```

and

```
public String dimensionToString( Dimension d )
{
    return "Dimension: (" + d.getWidth() + "," + d.getHeight() + ")";
}
```

We might also, for symmetry, add

```
public String stringToString( String s )
{
    return s;
}
```

although it doesn't seem particularly well-motivated at the moment.

Given these routines, we might rewrite printObject as

```
public void printObject( Object o )
{
    if ( o instanceof String )
    {
        Console.println( this.stringToString( (String) o ) );
    }
    else if ( o instanceof Point )
    {
        Console.println( this.pointToString( (Point) o ) );
    }
    else if ( o instanceof Dimension )
    {
        Console.println( this.dimensionToString( (Dimension) o ) );
    }
}
```

Variations

The new printObject still has a certain amount of redundant code. We can pushing the Console.println out of the individual ifs, but then we'll need to remember the String returned by each toString method. We could write

```
public void printObject( Object o )
{
    String s = "";
    if ( o instanceof String )
    {
        s = this.stringToString( (String) o );
    }
    else if ( o instanceof Point )
    {
        s = this.pointToString( (Point) o );
    }
    else if ( o instanceof Dimension )
    {
        s = this.dimensionToString( (Dimension) o );
    }
    Console.println( s );
}
```

In yet another optimization, we could actually transfer the coercion into the individual toString methods, calling them on Objects rather than on specialized types. This makes the methods somewhat less general -- what if they're called on the wrong type of objects? -- but if we can be sure that they'll always be called appropriately, it cleans up our dispatch code further.

```
public String pointToString( Object o )
{
    Point p = (Point) o;
    return "Point: (" + p.getX() + "," + p.getY() + ")";
}

public String dimensionToString( Object o )
{
    Dimension d = (Dimension) o
    return "Dimension: (" + d.getWidth() + "," + d.getHeight() + ")";
}

public String stringToString( Object o )
{
    return (String) s;
}
```

Now the dispatch routine reads

```
public void printObject( Object o )
{
    String s = "";
    if ( o instanceof String )
    {
        s = this.stringToString( o );
    }
    else if ( o instanceof Point )
```

```
        {
            s = this.pointToString( o );
        }
        else if ( o instanceof Dimension )
        {
            s = this.dimensionToString( o );
        }
        Console.println( s );
    }
```

Pushing Methods Into Objects

We can take this whole approach one step further, and in doing so dramatically simplify our dispatcher code. Instead of trying to give this dispatcher object a toString method for each individual type that it might need to know about, we can put the toString methods into the individual types directly. For example, Point might have a method that says

```
public class Point
{

     //...

            public String toString()
     {
         return "Point: (" + this.getX() + "," + this.getY() + ")";
     }

}
```

This is just the old pointToString, with a few modifications. First, note that we've eliminated the argument that pointToString needed. This is because the Point we're converting is `this`, i.e. the particular object whose toString() method is being executed. Second, we don't need a coercion. That's because if this set of instructions is being executed, it is because this (Point) object's toString() method has been called, i.e., we must be dealing with a Point. You simply can't call Point's toString() method on a Dimension (or a String).

A similar modification gives us Dimension's toString() method:

```
public class Dimension
{

     //...

            public String toString()
     {
         return "Dimension: (" + this.getWidth()
                + "," + this.getHeight() + ")";
     }

}
```

And finally String's toString method is quite simple:

```
public class String
{

     //...

           public String toString()
     {
         return this;
     }

}
```

Now, if `origin` names the Point with coordinates (0,0) and `square` names the Dimension with height 25 and width 25, `origin.toString()` returns the String "Point: (0,0)", while `extentless.toString()` returns the String "Dimension: (25,25)". Each object knows how to turn itself into a String using the toString() method provided by its class.

In point of fact, the Java class java.lang.Object has a toString() method, and so any Java object necessarily has a toString() method. In many cases, the toString() method is inherited from Object and so prints a rather ugly representation of the object. You may wish to override the toString() method of any class you expect to be printing out a lot. For example, there is a real class called java.awt.Point, but its toString() method isn't quite as succinct as the one we've given here.

What Happens to the Central Loop?

We have seen that writing the methods inside their respective classes makes them considerably more succinct. After all, the toString() method of Point just has to give instructions for how to print `this`, i.e., the particular Point whose toString() method is being invoked. At the time that the method is invoked, all of the relevant information is present in the **target** -- the object whose method is invoked, i.e., `this`. But we haven't come to the best part yet.

Suppose that our types each implement their own toString() method. What, then, does the dispatcher look like?

The new dispatch code is

```
public void printObject( Object o )
{
    Console.println( o.toString() );
}
```

Where did the conditional go? The answer is that it is hidden inside Java's method dispatch mechanism. Java decides which toString() method to invoke by looking at the target's type.

Whenever an instruction-follower evaluates a method invocation expression, Java does a quick calculation to determine what kind of object the target -- the method's owner object -- is. Depending on the class of that object, Java looks up the appropriate method to invoke. (The argument types also play a role in selecting the method invoked, specifically by selecting a method whose footprint is appropriate.) This dispatch based upon the type of the target object is a simple form of **polymorphism**. In general, polymorphism means doing different things with different types of objects.

If we move the dispatchee methods out to their respective classes, we give each kind of object its own type-specific way to respond to the request. Here, a particular -- known, fixed -- method name and footprint is polymorphic with respect to the target object to which it belongs. (Instances of many classes support the same method footprint. Each class provides a different implementation.) By allowing the target object to vary, we cause the same expression to invoke different pieces of code.

This approach has several benefits. First, the dispatcher becomes significantly more succinct. Second, the code that actually does the work is associated with a specific type, meaning that it doesn't have to worry about verifying type or coercion. Java does both dispatch and coercion automatically. The method is necessarily invoked on a target of the appropriate type, because the target helps to determine which method is invoked. Finally, if a new object type is to be added (e.g., to the printObject method), the particular instructions for converting it to a String can be added in the definition of the object's class; printObject no longer needs to worry about which types it is suited to handle. In fact, since toString is a method defined in the class java.lang.Object, printObject can handle any kind of Object at all.

The Use of Interfaces

In the example above, we gained great power from pushing the conversion to a String into each specific object type. Of course, any object type not supplied with its own toString() method simply inherits one from its superclass. Since java.lang.Object is the root of the class inheritance hierarchy, each class is guaranteed to have a toString() method, if only the one defined for Object. But sometimes you will want to use polymorphism to dispatch to a method that isn't defined on java.lang.Object. What do you do then?

Consider the Calculator buttons of an earlier chapter. In that example, number buttons are supposed to display themselves on the Calculator screen, while arithmetic operator buttons are supposed to perform calculations and the clear button is supposed to erase whatever happens to be displayed. The central dispatcher of that program checked which button had been pressed and called the appropriate helper method, contained within the dispatcher object.

Precisely the same sort of logic that we applied to the object printer would work here. First, we need to define a series of object types. For example, we might have a NumberButton class whose ten instances represent the number keys, from 0 to 9. We might have an OperatorButton class, one of whose instances would represent the addition function of the calculator. And we might have a ClearButton class with a single instance corresponding to the calculator's clear key.

Each of these classes might be endowed with a buttonPressed method, to be invoked by the dispatcher when the corresponding calculator button is pressed. For example, ClearButton's buttonPressed method might say resetCalculator, while a NumberButton's buttonPressed method would invoke displayDigit. Whose resetCalculator and displayDigit methods are these? They belong to the calculator. In order to do its job, the buttonPressed method will need to be given access to the CalculatorState -- an object representing what's going on inside the Calculator -- as an argument.

```
public class ClearButton
{
```

```
          public void buttonPressed( CalculatorState calc )
          {
              calc.resetCalculator();
          }

     }
```

When the individual clear button's buttonPressed method is invoked, it will in turn ask the calculator to reset itself.

```
     public class NumberButton
     {

          private final int whichDigit;

          public NumberButton( int which )
          {
              this.whichDigit = which;
          }

          public void buttonPressed( CalculatorState calc )
          {
              calc.displayDigit( this.whichDigit );
          }

     }
```

Note that there are ten different NumberButton instances, and each instance will need to remember which digit it represents.[Footnote: Once assigned, this digit doesn't change; hence, the field is declared final.] When, for example, the 0 button's buttonPressed method is invoked, it asks its calculator to display its digit, i.e., 0. The code for other button types is similar.

When we are done writing these button types, we will need to add code to the calculator dispatcher (or to some other part of the system) that creates all of the necessary instances of these classes. We might, for example, stick these instances into an array indexed by the buttonID ints described in chapter 12. This would be a field of our animate calculator object:

```
     private Object[] buttonObjects = new Object[ Calculator.LAST_BUTTON_ID ];
```

And then, inside the constructor for that object, we need initialization code:

```
          for ( int buttonID = 0; buttonID < 10; buttonID = buttonID + 1 )
          {
              this.buttonObjects[ buttonID ] = new NumberButton( buttonID );
          }

                // and so on for operators, clear....
```

Once we have instantiated these button types, what does the dispatcher look like? Its job will simply be to invoke the appropriate button object's buttonPressed method.

```
public void act()
{
    int buttonID = this.gui.getButton();
    this.buttonObjects[ buttonID ].buttonPressed( this.calcState );
}
```

There is just one problem: this code won't compile. The array buttonObjects is an array of Objects. But most Objects don't have a buttonPressed(CalculatorState) method.

Why wasn't this a problem for the toString method of the object printer? Because each Object has a toString() method, we didn't have to do anything special to make the corresponding line of code -- the invocation of the object's toString() method -- work. However, if we try this trick with a method that isn't possessed by every object, we will find that our code won't compile. We can resolve this by using an interface that specifies this contract.

```
public interface CalculatorButton
{
    public void buttonPressed( CalculatorState calc );
}
```

This interface gives just the information we need -- the presence of a buttonPressed method that requires a CalculatorState -- without saying anything about how a particular CalculatorState should respond to a button's being pressed. It leaves those aspects of the method to each class that provides an implementation for CalculatorButton's buttonPressed method.

We will also need to go back and add this interface to each of the individual calculator button classes. For example:

```
public class ClearButton implements CalculatorButton
___    {

     public void buttonPressed( CalculatorState calc )
     {
         calc.resetCalculator();
     }

}
```

Now, we can rewrite our declaration of the buttonObjects array.

```
private CalculatorButton[] buttonObjects
                = new CalculatorButton[ Calculator.LAST_BUTTON_ID ];
```

Finally, our code will compile!

The calculator button is a more general example than the object printer, but both illustrate the same set of ideas. By pushing methods out of the central dispatcher object and into the classes representing distinct types of objects, we can package up the methods with the information that they need to do their jobs. We can also largely eliminate the explicit dispatcher of the chapter 12, using Java's method dispatch mechanism in its place. This approach is very much in keeping with the philosophy of object-oriented design: keep behavior together with state encapsulated in objects.

Runnables as First Class Procedures

We have actually seen a special case of this kind of target-polymorphism-as-dispatch in our use of Animates as the instructions for AnimatorThreads. In that case, an AnimatorThread does very different things depending on the class of the particular object whose act() method it executes. In other words, AnimatorThread uses its constructor argument -- the object whose act() method it is supposed to execute -- to determine what it is supposed to do. The method footprint -- act() -- is fixed by the Animate contract. Naming this method there allows the programmer to write it explicitly into code. Remember, method names cannot be deduced and runtime, though their target objects can.

There is a similar situation in Java involving the interface Runnable (with a single method, run()) and the class Thread. A Thread is started on a particular object, and the Thread follows the instructions supplied by that object's run() method. By starting them on instances of different classes of Runnable objects, Threads can be induced to behave in very different ways. Like act(), run() exploit's Java's target-based dispatch mechanism to create different kinds of behavior.

But Runnables and run() can be used even without starting a new Thread, simply because they are fixed names for executable behavior that takes no arguments.[Footnote: Everything said here for run() could be done with another method with a different name, but that name, too, would have to be fixed when the program is written. For no-arguments executable code, run() and Runnable make a convenient convention. If you wish to pass arguments to this procedure, you will need to define your own interface and your own method signature, as Java offers no standard conventions.] Suppose that you want to pass a procedure around from one object to another. For example, suppose that you want to create a secret message and later, you will give that message to a decoder that will print out your secret message. One way to do this is to make the secret message a Runnable object and to use the secret message's run() method as a way for the decoder to get the message out.

```
public class SecretMessage implements Runnable
{

     private String message;

     public SecretMessage( String message )
     {
         this.message = message;
     }

     public void run()
     {
         Console.println( this.message );
     }

}

public class SecretDecoder
{

     public void decode( Runnable secret )
     {
         secret.run();
     }
}
```

Now, if we have

```
SecretMessage message = new SecretMessage( "Meet me at midnight." );
```

and

```
SecretDecoder decoder = new SecretDecoder();
```

then we can try

```
decoder.decode( message );
```

which will print

```
Meet me at Midnight.
```

to the Java console. The message stays safe inside the SecretMessage as the SecretMessage is passed from method to method, stored in fields, returned from methods, and otherwise passed around the system. Because it has a run() method, that method can eventually be invoked to get the desired behavior from of the object.

In fact, by the time that this object makes it to the decoder, we might have lost track of the fact that it is a SecretMessage. Suppose that we have an object `toBeRun`, and all that we know about it is that it is a Runnable. We can still ask

```
decoder.decode( toBeRun );
```

And now we might find out, for example, that someone has replaced our message with some Fireworks:

```
public class Fireworks implements Runnable
{

     private Color color;

     public Fireworks( Color color )
     {
         this.color = color;
     }

     public void run()
     {
         Console.println( "Crash!  Bang!  You see "
                          + this.color.toString() );
     }

}
```

Polymorphic dispatch ensures that toBeRun will print its message if it is a SecretMessage, and will explode colorfully if it is Fireworks. You do not need to know what kind of thing it is to arrange to send it to the right method; instead, Java's dispatch mechanism ensures that even when you don't know exactly what type of thing you have, the right method will be invoked.

Callbacks

A particular circumstance in which this "do the right thing" aspect of Java's method dispatch is important is called **callbacks**. A callback is a situation in which one object has invoked a method of another, and the second object needs to get some information back to the first without returning from the method invocation. There are a few prerequisites for callbacks:

1. The invoking object must pass a reference to itself into the original invocation, or must otherwise indicate whose method is to be "called back."
2. The invoking method and the invoked method must agree upon the name of the callback method.
3. The invoked method must record the reference to the invoking object -- the callback target -- e.g., as a parameter to the original invocation or as a field.
4. At the appropriate occasion, the invoked method must invoke the callback method on the callback target. The fixed method name is used in this expression; the reference to the callback target is a variable.

Suppose, for example, that we have an object whose purpose is to create many separate "web spiders", simple programs that traverse the Internet looking for interesting information.[Footnote: Such programs can be very useful, but you must be extremely careful in writing them. Serious disasters have been caused by web spiders that got out of control, for example creating so many spiders that the network filled up with spiders and couldn't sustain its regular traffic.] Your original object will want to know when the spider finds interesting information. But the spider won't want to stop executing when it finds the first interesting piece of information. Instead, the spider should take the address of its sponsor with it when it goes crawling through the web, and any time it finds an interesting piece of information it should "call back" the sponsor object, giving it that information without stopping its execution.

The actual situation for a web spider is a little bit more complicated than this description because web spiders often don't run on the same computer as their sponsor and so can't make direct method calls. But we can use this idea as the framework for some code that illustrates callbacks.

```
public class SpiderStarter
{

     private String interestingStuff = "";

     public void startSpider()
     {
         new Spider( this ); // give invoked method a reference
           }              // to the invoker, i.e., the callback target


           /* informationFound is the callback method.
      * It simply records the information...
      */
           public void informationFound( String interestingItem )
     {
         if ( this.interestingStuff == null )
         {
             this.interestingStuff = interestingItem;
         }
         else
         {
             this.interestingStuff = this.interestingStuff
                                 + " and also "
                                 + interestingItem;
         }
     }

           /* This is a simple utility method.
      */
           public void printInfoSoFar()
     {
         Console.println( "I heard " + this.interestingStuff );

}
```

This class provides three methods. The first starts up a Spider, telling the Spider who its sponsor is. The second provides a way for the Spider to call it back (when it finds information). The third provides a way for other objects to ask the SpiderStarter to let it know what information it has collected.[Footnote: Strictly speaking, this code might be subject to problems if we start up more than one Spider. We really need to protect the interestingStuff using synchronization, as described in part 5 of this book. These issues don't affect the main point of this chapter, but you should be aware of them if you want to run a code example like this one.]

The definition for Spider might read

```
    public class Spider extends AnimateObject
    {

        // where to record the callback target
              private SpiderStarter sponsor;

        public Spider( SpiderStarter who )
        {
            this.sponsor = who;  // record the callback target
              }

        public void act()
        {
            // Some code that looks for interesting stuff.
            // if you find it, call back

                this.sponsor.informationFound( interestingInfo );
        }


    }
```

Now, we might say

```
SpiderStarter mamaTarantula = new SpiderStarter();
mamaTarantula.startSpider();
```

This starts a spider going. The "looking for interesting stuff" part of the Spider is missing, but we can still see how a Spider might take advantage of the callback mechanism. Since a Spider is an AnimateObject, its act() method will be executed over and over again. Each time, if it finds some interesting information, it will invoke its sponsor's informationFound method with the interesting information. But SpiderStarter's informationFound method just adds the new information to its information store and returns, so the AnimatorThread that runs the Spider AnimateObject is free to call its act() object again.

Consider trying to write Spider without the callback. SpiderStarter doesn't call a method of Spider's directly, so Spider can't return a String that way. Even if SpiderStarter did call Spider directly, mamaTarantula presumably wants the Spiders to keep going even after they find their first piece of interesting information. So it is very important that the individual Spiders have a way to get information back without stopping their own execution. This is precisely the kind of situation in which a callback is useful.

Callbacks are a very general mechanism that can be used any time one object needs to get information to its invoker without returning the information directly. They require agreement on the name of a method -- perhaps specified by an interface contract -- that will be used to produce the callback. Callbacks take advantage of the idea that Java's dispatch mechanism will call the appropriate piece of code. Good object encapsulation ensures that the information supplied in a callback gets to the appropriate place.

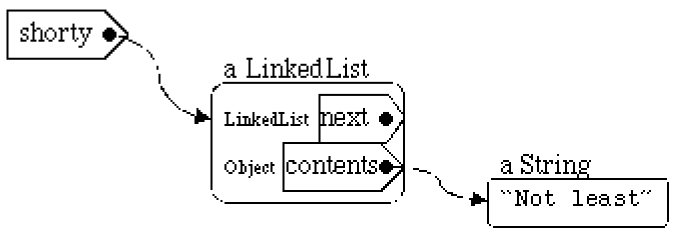
Recursion

One final example of how Java's method dispatch mechanisms work is the idea of **recursion**. Recursion is the name for a technique in which the same named method is called over and over again, doing something slightly different each time. There are two kinds of recursion: structural recursion, which is quite common in Java and other object-oriented programming languages, and functional recursion, which is much more prevalent in functional programming languages.

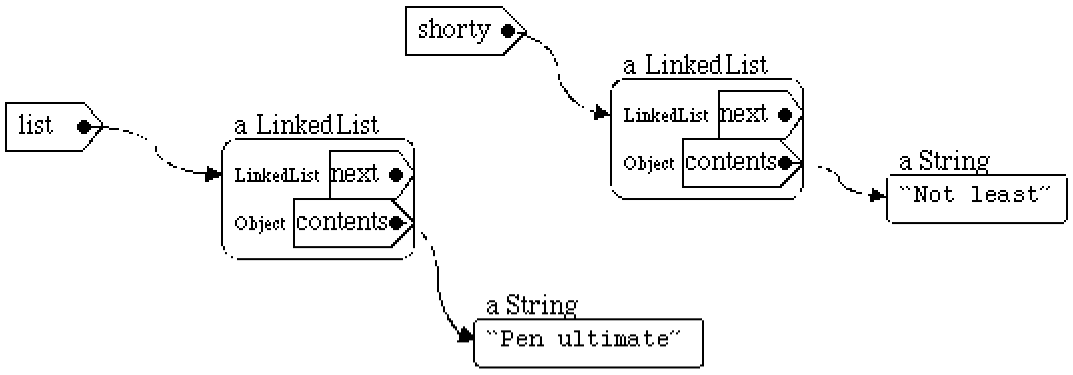
Structural Recursion

Structural recursion is a natural extension of method dispatch to a uniform collection of objects. It is really just the idea that an object can act on its own behalf -- i.e. provides methods specifying its own behavior -- coupled with the idea that one object can contain -- or have fields that are -- other objects. For example, the calculator had (access to) many CalculatorButton objects, and it relied on them to each provide the appropriate behavior. Structural recursion is just like this, except that the object doing the relying and the component object on which it relies are instances of the same class.

A.



B.



C.

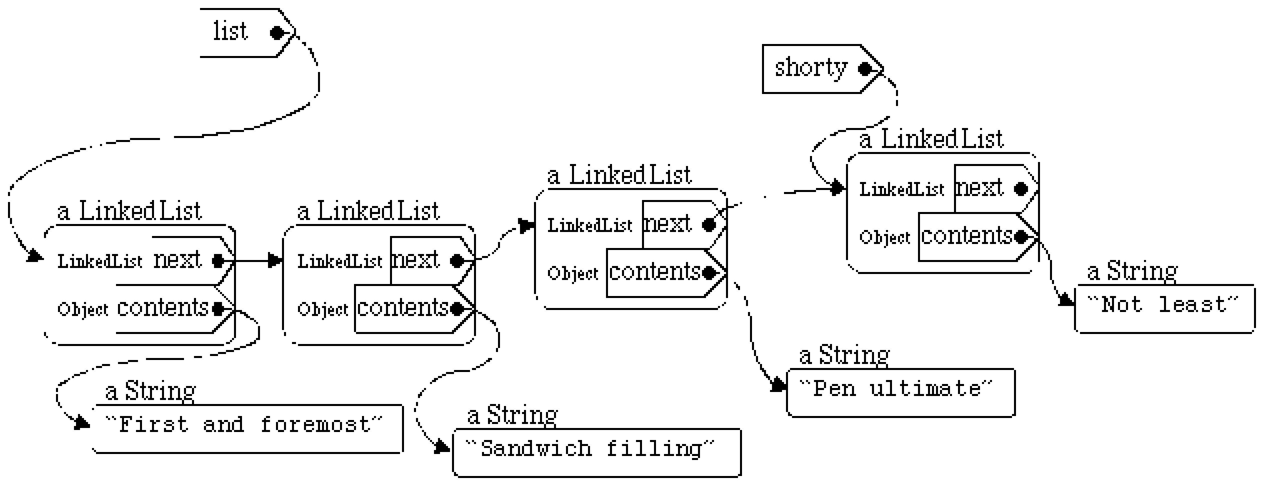


Figure #. Various linked lists (following code in text).
A. After defining shorty. **B.** After defining list. **C.** After assigning to list.

A Recursive Class Definition

Suppose, for example, that we have a class called LinkedList:

```
public class LinkedList
{

     private LinkedList  next;
     private Object      contents;

     public LinkedList( Object what, LinkedList next )
     {
         this.contents = what;
         this.next = next;
     }

           // maybe some methods....

}
```

To begin with, this definition is recursive. That is, the LinkedList type is defined in terms of itself. Note that this isn't at all the same thing as saying that a *particular* LinkedList is defined in terms of itself; it just means that a LinkedList consists of its contents (some arbitrary object) and its next element, which is either nothing (i.e., this is the last element) or also a LinkedList.

The idea of an object that has associates -- or contains components -- of the same type really isn't all that strange. For example, if we have a representation for a person, we might use the same representation for that person's parents. The same "method" for figuring out who your father is should apply equally well to figure out who *his* father is.

To create a LinkedList, you need to give it a LinkedList. To make this work, there needs to be a simple case that is not explicitly recursive. This is called a **base case**. In the case of the LinkedList definition, the base case is null: null is a (non)value that can be associated with a name of type LinkedList that is not defined in terms of a LinkedList. A LinkedList with a null next field is the last element in the list.

So, for example, we can say

```
LinkedList shorty = new LinkedList( "Not least", null );
```

We can also say

```
LinkedList list = new LinkedList( "Pen Ultimate", shorty );
```

or even

```
list = new LinkedList( "First and foremost",
                       new LinkedList( "Sandwich filling", list ) );
```

Each of these LinkedList objects either has a next field that refers to another LinkedList object, or has a next field that is unassigned, i.e., has the value null.

Methods and Recursive Structure

Structural recursion is simply a way in which methods can take advantage of the recursive definition of LinkedList. It relies on the idea that each of the recursively contained objects is itself a full-fledged intelligent entity. For example, suppose that you are providing a LinkedList with a method to convert itself to a String. This method might, e.g., be suitable for printing out all of the elements contained in a LinkedList. Since one LinkedList contains another (through its next field), we can make use of the fact that that next element is also an intelligent LinkedList and will be able to convert itself to a String as well.

In writing the code to convert a particular LinkedList instance to a String, there are two possibilities.

1. Perhaps this is the last element in the list, i.e., this LinkedList object's next field is null. Then we can solve this problem simply: just convert the contents of this object to a String.
2. Otherwise, this is not the last element; this object contains a non-null next field. In this case, converting this LinkedList to a String requires converting the contents of this object, then adding a comma, then converting this object's next (LinkedList) to a String. But that next LinkedList is an intelligent object, too. We can just ask it to convert itself!

It may seem like there's a bit of sleight of hand going on here. This argument may look suspiciously like a circular definition. But it is not. Let's examine the logic here carefully.

The first of these is the simple case in which there is no further recursion. As in the definition, this is called the base case. This condition would apply if we asked the LinkedList labeled shorty to print itself -- i.e., if we invoked `shorty.toString()` -- which would return the String "Not Least". There is only one element in this list, so printing its contents suffices.

The second case is called the **recursive case**, the case that relies on recursion to work. It says, roughly, I know how to convert myself to a String, and my `next` knows how to convert itself to a String, so I will simply combine those two answers. Of course, the way that the `next` LinkedList element converts itself to a String relies on this same code....so here it is. Imagine this definition inside the class LinkedList, where the comment says *maybe some methods....*

```
public String toString()
{
    if ( this.next == null )
    {
        return this.contents.toString();
    }
    else
    {
        return this.contents.toString()
               + ", " + this.next.toString();
    }
}
```

Suppose that we invoke `list.toString()`. In this case, the object referred to by the name `list` has contents "First and foremost", so it would begin its answer with that String. But that's not enough. Because `list`'s `next` field isn't null, it also needs to do something about that `next` field. It can't complete its answer until it knows how to print the LinkedList that is its `next` field. Luckily, `list.next` is also a LinkedList, so it knows how to convert itself to a String. So after "First and foremost", `list` adds in a comma. Then `list` invokes its `next` field's `toString()` method to find out how to end its String.

When `list.next`'s `toString()` method is invoked, it checks to see whether *its* `next` field is null. Since it isn't, it can't use the base case. So it first converts its own contents into a String -- "Sandwich filling" -- and then adds a comma, and then asks *its* `next` field to convert itself to a String.

Once again, the LinkedList has a non-null `next` field, so once again the recursive case is invoked, creating "Pen Ultimate" + ", " plus the value of its `next` field's toString() method.

The `next` field of this LinkedList is the same object referred to by the name shorty. We've already seen how shorty converts itself to a String using the base case -- returning "Not least" -- so now we can finish off "Pen Ultimate" + ", " + "Not least". This is returned to `list.next`, completing "Sandwich filling, Pen Ultimate, Not least". Finally, this String is returned to the LinkedList labelled `list`, and that LinkedList can return its value as a String: "First and foremost, Sandwich filling, Pen Ultimate, Not least".

The Power of Recursive Structure

The power of recursion here comes from the fact that each of the individual LinkedList elements knows how to combine its `next` field's toString() with its own contents. "If only my `next` field could supply its toString()," the LinkedList seems to say, "I could produce my answer. But of course the answer for the `next` field can be constructed out of *its* contents and *its* `next` field, and so on, until we come to the base case: a LinkedList in which the `next` field is null, so there's no need to get its toString().

[Important] Note that it is crucially important that the recursive case invoke the same-named method on a *simpler*object. That is, each recursive step must get a little bit closer to the base case. Imagine instead a situation in which you were printing a circular LinkedList. In this case, there would always be a `next` LinkedList to print, and the process would never end.[Footnote: Actually, to prevent just such situations, the computer may have the ability to detect this circumstance -- an infinite loop -- and to object to it by raising an exception.]

A similar kind of structural recursion could be used to find out whether a particular object is contained in a LinkedList. In this case, there are actually two base cases.

1. If `this.contents` is the desired object, then the LinkedList contains that object, i.e., return true.
2. If `this.contents` is not the desired object, but `this.next` is null, then this LinkedList doesn't contain the desired object, i.e., return false.
3. Otherwise, since `this.contents` is not the desired object, this LinkedList contains the desired object exactly when the desired object is contained by the LinkedList `this.next`.

There's a fairly straightforward translation of this into Java code:

```
public boolean contains( Object what )
{
    if ( this.contents == what )
    {
        return true;
    }
    else if ( this.next == null )
    {
        return false;
    }
```

```
                else
                {
                    return this.next.contains( what );
                }
            }
```

[Footnote: Actually, Java's && and || operators are guaranteed to evaluate their operands from left to write, proceeding only until the value of the expression is known. In the case of &&, as soon as one operand is false, no further operands need be evaluated. In the case of ||, evaluation stops as soon as an operand is true. This means that we could rewrite contains as:

```
            public boolean contains( Object what )
            {
                return ( ( this.contents == what )
                         || ( ( this.next != null )
                              && this.next.contains( what ) );
            }
```

]

Structural recursion is an extension of "the object can handle it" to the case in which the method invocation expression is contained within the same method that it invokes. Because the target of the invoked method is a "simpler" object -- one that is somehow closer to the base case -- this approach ultimately produces a satisfactory answer.

Functional Recursion

Functional recursion is a further extension of the idea of recursion. In this case, there is no structure whose inherently recursive nature is exploited by the recursion. Instead, the necessary subsequent simplifications -- steps to get closer to the base case -- happen in one of the method's arguments.

For example, many kinds of numerical calculations can be performed using purely functional recursion. In this case, it is common to define one or more base cases -- e.g., how the function should behave on a simple number such as 1 -- and then to recursively build a solution for one number out of the solution for a smaller number. Factorial is one such function:

1. The factorial of 1 is 1.
2. The factorial of an arbitrary number, n, is n times the factorial of n-1.

The first of these is the base case. It simply produces an answer, with no recursion necessary. The second of these is the recursive case. It wishfully assumes that you know how to calculate the factorial of n-1, then uses that to construct the factorial of n. By "peeling off" one number at a time, it is possible to calculate the factorial of any number. This is really just like structural recursion, but there's no change of the method's target here.

```
            public int factorial( int n )
            {
                if ( n == 1 )
                {
                    return 1;
                }
                else
                {
                    return n * this.factorial( n - 1 );
                }
            }
```

Factorial of 5 is 5*factorial of 4, which is 4*factorial of 3, and so on until factorial of 1, which is 1. So factorial of 2 is 2*1, and factorial of 3 is 3*(2*1), of 4 is 4*(3*2*1), and of 5 is 5*4*3*2*1. This is just like LinkedList's toString() method, except that the accumulation isn't coming from changing the target of the method invocation.

Chapter Summary

- Objects encapsulate information necessary to make methods effective.
- When multiple classes have methods with the same name, Java chooses the method that matches the target's (most specific) type.
- Dispatch can be replaced by empowering objects directly. Depending on the type of the target object, the same textual method invocation will actually call different code. This is called method polymorphism.
- A common superclass or interface, providing the method signature for the polymorphic method, is required for this kind of implicit dispatch.
- Method dispatch based on the target object can be used for other purposes as well:
  - Behavior can be passed to methods, returned from methods, and stored in objects by making it the run method of a Runnable object.
  - An executing method can give information to the object that called it, without returning, by using an explicitly agreed upon callback method.
- Recursion is a situation in which one method name is invoked repeatedly.
  - In structural recursion, the target of the method varies.
  - In functional recursion, at least one of the method's arguments varies.
  - In all recursions, there must be a base case that does not involve recursion.
  - In the recursive case, the recursive call must be to a method/target/argument that is somehow closer to the base case.

Exercises

1. Write toString() methods for an Address object and for a Date object. How would printObject have to change if it might be asked to print an Address or a Date as well as a String, Point, or Dimension?

2. Write clone() methods for Point and Dimension. (A clone() method should create a new copy of its target object.) Write a dispatcher called cloneObject(Object o).

3. Write an animate AlarmedTimer class that counts by itself, as the Timer class of chapter 9 does. In addition, it should have a setAlarm(int interval, Alarmable who) method. When this method is invoked,

the AlarmedTimer should callback the Alarmable's alarmReached() method every int ticks. Here is Alarmable:

```
public interface Alarmable
{
    public void alarmReached();
}
```

4. Using the LinkedList code above, add a method that returns the Object that is the `contents` of the last element in a LinkedList. For example, `list.getLast()` would return "Not least", as would `shorty.getLast()`.

5. Define a recursive structure for a family tree. Each person in the tree should have a father and a mother, which should be either another person or -- e.g., if the information were not available -- null. Give this a method that prints all ancestors of a given individual.

Bonus: Give this structure the ability to print only all female ancestors (using Console.println).

Extra Bonus: Would your female-ancestor-printer print my father's mother?