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

Inheritance

Chapter Overview

- How do I simplify the program design task by reusing existing code?
- How do I create variants on things I already have?
- When is it not appropriate to reuse code?

This chapter covers class-based inheritance as a way to reuse implementation. Inheritance allows you to define a new class by specifying only the ways in which it differs from an existing class. Those differences can include: additional (or alternative) contracts that it satisfies, behaviors that it provides, internal information that it stores, or startup instructions. Inheritance means that existing code can be adapted and reused, with some modification, in new contexts.

The mechanism by which inheritance works involves extending the parent class definition either by augmenting or overriding behavior defined there. Most of this chapter concentrates on how these mechanisms work. Not every instance of similar behavior is an appropriate context for inheritance. The chapter concludes with a discussion of the limitations of inheritance.

This chapter includes sidebars on the details of method and field lookup. It is supplemented by reference charts on the syntax and semantics of java methods, fields, and class declarations.

Objectives of this Chapter

1. To understand how one class can build on behavior supplied by another.
2. To be able to extend and modify existing definitions
3. To recognize when to use mechanisms other than inheritance to extend behavior.

Derived Factories

We have so far seen several cases in which we wanted to build multiple kinds of things that shared a basic similarity. When this similarity was largely in the contract implemented -- as with Counters and Timers -- we abstracted this similarity into an interface. The interface allowed us to deal with objects without knowing the details of their implementations, i.e., to treat them solely in light of the contracts that they provided.

In this chapter, we are more concerned with situations in which two kinds of objects share not only the same contract but almost the same implementation. For example, the BasicCounter and the Resettable Counter contained almost precisely the same code. In fact, the BasicCounter's code was (except for the

class and constructor name) a proper subset of the Resettable Counter's code. Similarly, the code for AnimateObject was contained in the code for AnimateTimer and the code for CountingMonitor. And almost every StringTransformer simply elaborates on the generic StringTransformer, simply providing a specialized version of the transform() method.

In cases where code really matches at the level of wholesale textual reuse of a class, Java provides a mechanism to allow one type of object to build on the behavior specified by another. This is a relationship between one class and another. Since classes are essentially object factories, we can think of this as a situation in which one factory produces its widgets by buying widgets wholesale from another factory, then adding its own minor tweaks (bells and whistles) to the widgets before claiming to have produced them.

The mechanism by which this is accomplished in Java is called inheritance, and it applies to a relationship between two classes. There is a similar relationship between two interfaces, described below. Inheritance is *not* ever a relationship between a class and an interface (or between an interface and a class). Inheritance really means an almost literal subsuming of one thing by another.

Simple Inheritance

Consider, for example, the AnimateObject class from the previous chapter and its near relative, the CountingMonitor. The AnimateObject class says:

```
public class AnimateObject implements Animate
{

     private AnimatorThread mover;

     public AnimateObject()
     {
          this.mover = new AnimatorThread( this );
          this.mover.startExecution();
     }

     public void act()
     {
         // what the Animate Object should do repeatedly
     }

}
```

In implementing the CountingMonitor class, we really only want to change the underlined things:

```
public class CountingMonitor implements Animate
{

     private Counting whoToMonitor;

     private AnimatorThread mover;

     public CountingMonitor( Counting whoToMonitor )
     {
         this.whoToMonitor = whoToMonitor;
```

```
            this.mover = new AnimatorThread( this );
            this.mover.startExecution();
        }

        public void act()
        {
            Console.println( "The timer says "
                             + this.whoToMonitor.getValue() );
        }

    }
```

It would be really nice only to have to write the underlined information, not the rest. In fact, we can do almost exactly that. The following definition is *almost* equivalent to the Counter definition above:

```
    public class CountingMonitor extends AnimateObject
    {

        private Counting whoToMonitor;

        public CountingMonitor( Counting whoToMonitor )
        {
            this.whoToMonitor = whoToMonitor;
        }

        public void act()
        {
            Console.println( "The timer says "
                             + this.whoToMonitor.getValue() );
        }

    }
```

We have preserved the underlining, and you can see that almost the entire new class is underlined. One of the few non-underlined items is the phrase extends AnimateObject. This is the phrase that does almost all of the work. It means, roughly, a CountingMonitor *is* an AnimateObject, it just provides the additional specified behavior.

1. It has its own private field, whoToMonitor, suitable for labelling a Counting.
2. It has a constructor that takes one argument, a Counting, and holds on to it.
3. Its act() method has a much more interesting body than AnimateObject's.

This code is equivalent to the original definition of CountingMonitor. It is much shorter to write. To use it, simply begin with the instructions for AnimateObject and add the pieces that CountingMonitor provides, **extending** the behavior of the AnimateObject (in the absence of conflicting instructions) to do these additional things.

In essence each CountingMonitor instance has an AnimateObject instance inside of it. Whenever the CoutingMonitor can't figure out how to do something, it simply defaults to the behavior of its AnimateObject. That way, the CountingMonitor doesn't have to provide all of the behavior that an AnimateObject already has; it can just rely on the existing implementation.

The remainder of this chapter deals with the details of this proposition.

java.lang.Object

There is actually a single built-in type called `Object`, and all other object types (directly or indirectly) `extend Object`. In other words, anything which is not one of the built in types is an `Object` of some sort or another.

```
class Cat extends Animal
{
    ....
}
```

A class declaration is followed by an optional `extends` clause, then a pair of braces around the body of the class definition. If the `extends` clause is missing (e.g., `class Widget {...}`), the default clause `extends Object` is assumed. Thus, ***every class*** (implicitly or explicitly, directly or indirectly) *`extends Object`*.

The class Object provides some basic functionality that every other class necessarily inherits. This means that you can guarantee that every Java object has, e.g., a toString() method. See the sidebar on The class Object for details.

The class Object

The class java.lang.Object is the root of the inheritance hierarchy, i.e., the class of which all other classes are subclasses. Every Java object is guaranteed to implement each of the methods provided by Object (though their implementations may vary).

`boolean equals( Object )` returns true exactly when the argument Object is the same Object as the one whose method is invoked. This is exactly the same thing that == would do on two Objects. You may override equals to do something somewhat more interesting.

`String toString()` returns a String ostensibly suitable for printing. It contains a lot of useful information in a generally illegible format, so if you are interested in being able to read your objects, you may wish to override this method to print something more easily human-readable.

`class getClass()` returns the class object (i.e., factory) from which this instance was created.

`Object clone()` is a peculiar method of Object because although every object implements it, it can only be used with instances of classes that also implement the `Cloneable` interface. If a class implements the `Cloneable` interface, the inherited version of `clone()` simply creates a new object of the same type as the original and whose fields have the same values as the fields of the original. You may override `clone()` to do whatever you wish.[Footnote: If you call the clone() method of an object that doesn't implement Cloneable, it will throw CloneNotSupportedException. See the next chapter for more on Exceptions.]

Object also provides other methods (finalize, hashcode, wait, notify, and notifyAll) that are beyond the scope of the material covered here.

Superclass Membership

When one class extends another -- as in the CountingMonitor/AnimateObject example above, we say that the extending class (CountingMonitor) is a **subclass** of the extended class (AnimateObject), and that the extended class is a **superclass** of the extending class. Neither subclass nor superclass is an absolute description; instead, both describe relationships between two classes.

When we say that one class is a subclass of another, what we mean is that we can treat instances of the subclass in all respects as though they were members of the superclass. For example, we can use a CountingMonitor anywhere we can use an AnimateObject. We can assign a CountingMonitor to a name whose type makes it appropriate for labelling AnimateObjects. (After all, a CountingMonitor *is* an AnimateObject.) We can return a CountingMonitor from a method that expects to return an AnimateObject, or pass one as an argument to a method expecting an AnimateObject parameter. A CountingMonitor is simply a special kind of AnimateObject.

In fact, subclasses have all of the type-relational properties of classes and the interfaces that they implement. A subclass instance can be assigned to a name of the superclass type. It answers true to the `instanceof` predicate on the superclass. It can even be automatically coerced **up-cast** to its superclass type. This is the same kind of automatic coercion that happens from int to long, and it is similarly guaranteed always to succeed and never to lose information.

Treating a CountingMonitor as an AnimateObject doesn't actually change the CountingMonitor, though. The CountingMonitor is still a CountingMonitor, with its extended act() method and its Counting to keep track of. This is the same situation as when an object is treated according to its interface type: this narrows the view of the object, but it doesn't change the underlying object.

If you are currently holding what looks like a superclass instance (e.g., an AnimateObject), and you suspect that it is actually an instance of a subclass, you can attempt to do a **down-cast** coercion on it. As with primitive types, a narrowing conversion is one that may not work or may lose information.

For example, if AnimateObject ao has some value that you think might be a CountingMonitor, you can try the expression

```
(CountingMonitor) ao
```

(e.g., in an assignment statement or in a method invocation). However, if you're wrong and this AnimateObject is not a CountingMonitor, this will cause your program serious problems. (See the next chapter for information about how these problems arise and what you can do about them.) So you may want to test whether this is an OK thing to do first, using a **guard** expression:

```
CountingMonitor cm;
if ( ao instanceof CountingMonitor )
{
```

```
        cm = (CountingMonitor) ao;
    }
```

This first checks to see whether it's OK to treat the AnimateObject as a CountingMonitor.

So far, we have seen that instances have several types: the type of the class from which the instance was created, the types of any interfaces that class implemented, and the types of any superclass that this class extends. This may mean many interface types (since a class can implement many interfaces). A class can only extend a single superclass, but this does not limit the number of legal class types because the superclass may itself extend another class, and so on. Where does this end?

We can use the idea of superclass membership to create very powerful abstractions, but not without the help of casting. For example, Java provides a class, Vector, that allows us to hold on to a collection of Objects; it behaves sort-of like a whole bunch of names, but indexed by number. Vector provides an addElement() method that takes any Object as an argument. This means that any Object can be inserted into a Vector. For example, you can insert a String into a Vector, and an AnimateObject as well:

```
Vector v = new Vector();
v.addElement( "Silly string" );
v.addElement( new Timer() );
```

However, when we retrieve the elements we've inserted, we discover that Vector's elementAt() method doesn't know the type of the Object we've inserted. Instead, elementAt() returns an Object; it is up to us to figure out what kind of thing we've gotten back. For example, the first thing in the Vector (at element 0) is the String "Silly string". So we can say

```
Object o = v.elementAt( 0 );
```

or

```
String s = ( String ) v.elementAt( 0 );
```

but not

```
String s = v.elementAt( 0 );
```

because this is an illegal attempt to assign a value of type Object (v.elementAt(0)) to a name of type String. The explicit cast expression of the previous line is needed to make this statement legal.

Overriding

The examples of inheritance in the previous section demonstrated that a subclass can extend the functionality of its superclass. The subclass can also modify superclass functionality by **overriding**, or redefining, methods provided by the superclass. In fact, CountingMonitor overrode the act() method provided by AnimateObject. This just wasn't a very interesting example because AnimateObject's act() method didn't do anything.

Consider the following classes:

```
public class Super
{
    public void doit()
    {
        Console.println( "super method" );
    }

    public void doitAgain()
    {
        this.doit();
    }

}

public class OverridingSub extends Super
{
    public void doit()
    {
        Console.println( "overridingSub method" );
    }
}
```

Now suppose that we create an instance of OverridingSub and ask it to doit():

```
OverridingSub over = new OverridingSub();
over.doit();
```

As expected, this prints `overridingSub method`. What if we labelled the OverridingSub with a Super name?

```
Super supe = new OverridingSub();
supe.doit();
```

The same thing: `overridingSub method` Recall that using a different type of name doesn't change the underlying object.

super.

What if we still want to be able to access Super's doit() method from the subclass? To do this, we need a special expression much like `this`. The expression `this` refers to the instance whose code is being executed. The expression `super` refers to the superclass of the object containing the actual executing code.

```
public class ExpandingSub extends Super
{
    public void doit()
    {
        super.doit();
        Console.println( "expandingSub method" );
    }
}
```

In this case, we'll get the effect of executing the superclass method followed by the local println:

```
super method
expandingSub method
```

If we reverse the lines of the method body, we will reverse the order of the printed lines.

Outside-in rule

There is one more trick lurking in this example. This is the doitAgain() method in Super. We know what happens when we ask an instance of Super to doitAgain(): it does the same thing as if we'd asked it to doit(). But what if we ask a subclass instance?

```
over.doitAgain()
```

The first thing that happens is that we have to find the doitAgain() method for OverridingSub. To do this, we start looking at the outermost (sub) class. This is OverridingSub. But it doesn't contain an appropriate method. So we move up the hierarchy, inside the object, to the superclass. Super *does* define doitAgain(), so now we know what code to execute. But the body of Super's doitAgain() method says `this.doit()`. Who is `this`?

The expression `this` always refers to the object on behalf of whom you are executing. At the moment, we're executing some code in the class Super. But we are doing it for an instance of OverridingSub; we just happen to be looking at `over` *as though* it were a Super, just as we did when we labelled it with a Super-type name. Looking at `over` as a Super doesn't make it one, though. So when we call `this.doit()`, we go right back to the outside (OverridingSub) and start working our way in again, looking for a doit() method. So the effect of invoking over.doitAgain() is the same as invoking over's doit(), *not* the Super method.

[Outside in pic]

Problems with Private

It isn't always completely straightforward to extend a class. Consider the BasicCounter and Resettable Counter classes from the chapter on Designing with Objects. Because the BasicCounter wasn't designed with inheritance in mind, there is a problem in extending it. In fact, we have to go back and modify the BasicCounter before we can describe the Resettable version directly in terms of it.

```
class BasicCounter implements Counting
{
     int currentValue = 0;

     void increment()
     {
         this.currentValue = this.currentValue + 1;
     }

     int getValue()
     {
         return this.currentValue;
     }
```

```
}
```

To implement the Resettable Counter class, we would like to be able to write the following:

```
public class Counter extends BasicCounter implements Resettable
{

    public Counter()
    {
        this.reset();

    }

    public void reset()
    {
        this.currentValue = 0;
    }

}
```

We have preserved the underlining, and you can see that almost the entire new class is underlined. This says that a Counter is just like a BasicCounter except:

1. It implements the Resettable interface (in addition to Counting, already implemented by -- and hence inherited from -- BasicCounter).
2. It has a no-args constructor that calls its own reset method.
3. It has a reset method that sets its currentValue field to 0.

But this code is not entirely adequate. In fact, it does not compile as is. The problem is that the currentValue field is not a part of the Counter class any more. The field currentValue is defined in BasicCounter. But BasicCounter's currentValue field is private, meaning that only BasicCounters (and the BasicCounter class, or factory) can access that field. The solution is to change the visibility of the field from private to protected. This allows the Counter subclass to access BasicCounter's currentValue field. Now, the Counter code in this chapter does the same thing as the Counter code in the Chapter on Designing with Objects.

The moral here is that if you want your class to be extensible -- to be able to be inherited from -- you will need to make sure that subclasses can get access to anything that they need to be able to manipulate. This in turn opens those aspects of your class up to manipulation by other classes, since that information is no longer private. The visibility level protected is an intermediate point between private and public, but it does not always provide adequate protection. For details, see the chapter on Abstraction.

Constructors are Recipes

We already know that constructors give the special instructions for how to create a particular kind of object. How does this interact with inheritance?

this()

When a class has more than one constructor, we can express one constructor in terms of another using the special syntax this(). For example, we might define a Point class that either could be instantiated using specified values for the x and y coordinates or could take on the default value (0,0). We might define the constructors this way:

```
public class Point
{

     private int x, y;

     public Point()
     {
         this( 0, 0 );
         // constructor would continue here....
     }

     public Point( int x, int y )
     {
         ....
```

The line `this( 0, 0 );` in the first (no-args) constructor means "create me using my other constructor and the arguments 0, 0". In other words, when we say `new Point()`, invoking the no-args constructor, this line transfers the responsibility of providing the instructions for the construction of the Point to the two-int constructor, supplying the ints 0 and 0 as values. Now, the second constructor would execute, creating a Point. This new Point's construction process would continue in the first constructor at the comment

```
// constructor would continue here....
```

The point being constructed would be the point resulting from the second constructor's invocation on 0, 0. Since there are in fact no more instructions in the first constructor after the comment, execution of this constructor would terminate and the new point returned would be the point corresponding to (0, 0).

The special buck-passing constructor this() can only be used as the first line of a constructor.

super()

Constructors and inheritance work similarly. Making an inherited object (the "inner object" that belongs to the superclass) is just like passing the buck to a same-class constructor. The first line of any constructor may be an explicit invocation of the superclass constructor, supplying whatever arguments are necessary between the parentheses.

For example, if we wanted to extend the CountingMonitor class, above, to determine whether the reading of its Counting had changed since the previous reading, we could add a field (to keep track of the previous reading) and a conditional in the act() method. But how would we deal with the constructor? The beginning of this class might read:

```
public class ChangeDetectingCountingMonitor extends CountingMonitor
{
```

```
    private int previousReading;

    public ChangeDetectingCountingMonitor( Counting who )
    {
        super( who );
        // ....
```

The first line of this constructor says "create my inner CountingMonitor instance using who as its constructor parameter." When the superclass constructor completes its execution, the remainder of the ChangeDetectingCountingMonitor constructor body is executed, extending the CountingMonitor instance and wrapping it in whatever it needs to be a full-fledged ChangeDetectingCountingMonitor.

implicit super()

We have seen that, when no explicit constructor is supplied, Java blithely inserts a no-args constructor. Java actually has two dirty little secrets about constructors:

1. ***If no constructor is provided for a class, Java automatically adds a no-arguments constructor.***
2. ***Unless a constructor explicitly invokes its superclass constructor or another (`this()`) constructor of the same class, Java automatically inserts `super();` as the first line of the constructor.***

This means that a class that doesn't seem to have a constructor actually has the following one:

```
public ClassName () {
    super();
}
```

What does this do? It means that you can create an instance of the class with `new ClassName()` -- because the constructor has no parameters, so you don't have to give it any arguments -- and it also means that each instance of *`ClassName`* has an instance of the superclass hiding inside it. That is, `super();` is a special incantation that means "Make me an instance of my superclass." (Be careful: there are two readings of this request: "Give me an instance..." and "Turn me into an instance...". The second reading is correct.)

The `BasicCounter` class has such an implicit, automatically inserted constructor, but the Counter class doesn't. Counter does automatically get the implicit call to `super();` though:

```
public BasicCounter () {
    super();
}
```

and

```
public Counter()
{
    super();
    this.reset();
}
```

You can, of course, insert this no-args make-me-an-instance-of-my-superclass constructor into every class definition, and some people like to do so explicitly.

Details:

1. `super();` may only appear as the first line of a constructor.
2. The form `super(`*`args`*`)` may be used if the superclass constructor takes arguments.
3. If a constructor is defined, this constructor is not automatically added. So, for example, `Echo` *does not* have a no-args constructor.
4. If a superclass does not have a no-args constructor, an explicit call to `super(`*`args`*`)` must be used as Java's automatic insertion of `super()` will cause a compile-time error.

What if a class doesn't have a superclass? ***Every class is a subclass except*** `Object`**.** ***If a class doesn't have an*** *`extends`* ***in its declaration, Java automatically inserts*** `extends Object`**.** That means that the automatically-inserted constructor will in general make sense.

Beware: Since Java will automatically invoke the no-args version of super() unless you explicitly invoke a superclass constructor, either (1) the superclass must *have* a no-args constructor or (2) you must explicitly invoke the superclass constructor yourself, supplying the requisite arguments. If you create a class without a no-args constructor, you can get into trouble extending it.

Style Sidebar

Explicit use of this. and super()

Although it is not strictly speaking necessary, it is good style to Use `this.` wherever it is appropriate, i.e., to denote calls to an object's own fields or methods. While it makes your code somewhat more verbose, it also makes it easier to read and to understand what's going on. No method call should ever be made without reference to its target (i.e., whose method is being called). Field accessor expressions should always include a reference to the field's owner, distinguishing them from other name accesses (including parameter and local variable references).

A class declaration that does not contain an explicit `extends` clause still `extends Object`. Stating this explicitly may make it easier to read your code.

A constructor that does not call another (`this()`) constructor explicitly calls the superclass constructor. If the superclass constructor is not invoked explicitly, Java will insert a(n implicit) call to `super()`, the superclass's no-args constructor. You can make this implicit call explicit by including `super();` as the first line of any constructor that doesn't explicitly invoke another self- or superclass constructor. This helps to remind you that it is being called anyway.

Interface Inheritance

A class cannot inherit from an interface; it implements the interface, providing behavior to match the interface's specification. But one interface can extend another. Interface inheritance is much simpler than class inheritance. In interface inheritance, the methods and fields of the inherited (super) interface are simply combined into the methods and fields of the inheriting (sub) interface. The syntax for interface inheritance is identical to the syntax for class inheritance, but since there can be no overriding of method specifications, and since all fields are public and static therefore cannot be overridden, there is really no complexity to interface inheritance.

As with class inheritance, if one interface extends another, all instances implementing the subinterface are instances belonging to both types.

Relationships Between Types

There are three different type-to-type relationships that will be important in creating systems. These three relationships correspond to three distinct mechanisms: implementation, extension, and coupling.

Implementation is a relationship in which one type provides a specification and a second type provides a specific way of implementing that specification. In this case, the first type is called an interface and the second type is called a class. For example, an Alarm is one way of implementing the Resetable specification; an Animation is another.

Extension is a relationship in which one type adds functionality to another. There are actually two variants of extension. In one, both types are specifications (i.e., interfaces) and the extending specification adds commitments to the extended specification. StartableAndResetable is an extension of Startable. In the other, both types are implementations (i.e., classes) and the extending implementation adds functionality to the extended implementation. A CheckingAccount adds check-writing functionality to a BankAccount. Extension is implemented using inheritance, the primary subject of this chapter.

Coupling is a way of giving one object the ability to ask another to help it. For example, a MicrowaveOven may have a Clock, but a MicrowaveOven *isn't* a Clock. MicrowaveOven doesn't implement Clock behavior or extend it. Each MicrowaveOven *has* a corresponding Clock, and when the MicrowaveOven needs to know what time it is, it checks with its own Clock. In this case, the relationship is one-to-one (one MicrowaveOven per Clock, one Clock per MicrowaveOven). There are other cases in which the relationship may be many-to-one (many Chickens, one Coop) or one-to-many. [IM: Unlike extension and implementation, coupling is really a relationship between instances; however, like implementation and extension, it is generally defined within the class.]

It is important to know which of these three relationships ought to hold as you design your code.

It is always advisable to factor out common commitments and to separate the users of these contracts from their implementors. Wherever possible, an object should be known by an interface type rather than a class type to make it possible for alternate implementations to be used. This is true for both name declarations and method return types. The only time when an interface cannot be used routinely is in a construction expression.[Footnote: But see, e.g., the Factory pattern [GHJV] for an approach to this problem.]

Interface implementation, the result of introducing these interfaces, is generally easy to recognize. An interface, after all, provides the contract without the actual implementation.

It is generally more difficult, especially for the novice programmer, to determine whether it is appropriate to use inheritance or merely containment. Inheritance is actually relatively rare (among classes) and should be used only when the new class really reuses the complete behavior of the existing class. This is because inheritance makes the implementation of the new class tremendously dependent on the details of the implementation of the existing class. Coupling is a much more general mechanism. In this case, the new kind of object simply relies on a previously existing kind of object to provide behavior, forwarding messages on to the instance of the pre-existing class. If the coupling relies on an interface type rather than on a class type, a different implementation can easily be substituted.

If you are constructing a class and want to make use of behavior implemented by another class, you must determine whether you are better off using inheritance (i.e., extension) or coupling. Here are some questions that you should ask:

- Does this new class present to its users the full range of behavior provided by the existing class (inheritance) or just some of that behavior (coupling)?
- Does this new class add behavior to the existing class (inheritance) or override it (coupling or a common subclass)?
- Can instances of this new class legitimately be treated as instances of the existing class (inheritance) or would this be inappropriate (coupling or common interface)?
- Does an instance of this new class have a different lifetimes from the associated instance of the existing class (coupling)?

It is only when the superclass will be wholly reused, and when the subclass really is an extension of the implementation provided by the superclass, that inheritance should be used. Occasionally, this justifies the use of an abstract class to encapsulate common behavior that is extended differently by different classes.

Abstract Classes

A class *can* have a method that is just a signature -- an `abstract` method. In a class, however, the abstract method must be explicitly declared `abstract`. (Recall that methods in an interface are assumed to be `abstract`, even if they are not explicitly so declared.)

If a class has one or more `abstract` methods, it isn't a complete implementation. (It doesn't specify how to do the un-implemented method!) In this case you cannot directly make an instance of this class. (This is like a partial recipe -- you can't cook anything edible with it, but it may be useful in building more complete recipes. We will see how to use one recipe to build another in the chapter on Inheritance.)

A class with one or more `abstract` methods is called an `abstract` class. You cannot construct an instance of an `abstract` class.[Footnote: Technically, a class can be `abstract` even if it has no `abstract` methods. However, *every* class with at least one `abstract` method *must* be declared `abstract`.]

Abstract classes can be useful when you want to specify a partial implementation. You should not use an abstract class when you only want to specify a contract; that is the function of an interface.

We will see examples of abstract classes in later chapters.

Chapter Summary

- Inheritance is a mechanism that allows one class to reuse the implementation provided by another.
- Inheritance should be used only when instances of the subclass can also reasonably be considered instances of the superclass.
- A class always extends exactly one superclass. If a class does not explicitly extend another, it implicitly extends the class `Object`.
- Method lookup always begins with an object's actual (most specific sub)class, even when the method is invoked by a `this.` expression in superclass code.
- A superclass method or (non-private) field can be accessed using a `super.` expression.
- If a constructor does not explicitly invoke another (`this()` or `super()`) constructor, it implicitly invokes the superclass's no-args constructor.

Exercises

1. In the first interlude, we wrote "UpperCaser extends StringTransformer". Explain.

2. Extend the Counter to count by 2.

3. Complete the definition of `ChangeDetectingCountingMonitor` from above.

4. In this exercise, you will re-implement AnimateTimer in two different ways and then compare them.

   a. Re-implement Timer by extending Counter.
   2. Extend the class in the previous exercise by making it Animate.
   3. Now re-implement AnimateTimer by extending AnimateObject directly.
   4. What if any type relations would exist between an instance of the class produced in (b) and the class produced in (c)?

This chapter is excerpted from a draft of *Introduction to Interactive Programming In Java*, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>


WHILE (TRUE) {
ECHO
}