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

Designing With Objects

Chapter Overview

- How do I design using objects and entities?

In the preceding chapters, we have seen how interfaces specify contracts and how classes implement them. We have used expressions and statements to create instructions that describe the processes of performing actions, making up meth

od and constructor bodies. And we have used names to retain an object's state even while none of the object's methods is executing. In this chapter, we turn to the question of how we design systems using these various tools.

The first part of this chapter looks at one simple example to illustrate how the fields and methods of an object can be identified and implemented. Although the example is small, the principles described here are general and will be used in the design of any object-oriented program. This example also provides an opportunity to look briefly at the question of privacy, or how an object separates internal information from information that it makes available to other objects.

The next section of this chapter turns to look at three important kinds of objects that appear in many systems. These kinds of objects -- data repositories, resource libraries, and traditional objects -- each play distinctly different roles in any system, and their designs reflect these roles. A fourth distinct kind of object -- animate objects -- is the topic of the next chapter.

The chapter concludes with a discussion of the ways in which different objects and types are interrelated.

Objectives of this Chapter

- To become familiar with the identification of objects, methods, fields, interfaces, and classes from a problem description.
- To recognize common kinds of objects and the roles that they play.
- To learn to identify opportunities to use these patterns in designing systems.

Object Oriented Design

So far, you've seen a lot of Java how-to: how to declare, define, assign, and invoke variables of primitive and object types, classes, object instances, methods, and control flow. Now that you have some fluency with the basic building blocks of Java, it is time to start looking at why each of these constructs is used and

how they are combined to build powerful programs. In this chapter, we'll look at objects and classes; in the next, we'll continue this discussion by focusing on instruction-followers and self-animating objects.

Objects are Nouns

When you are constructing a computational system, you need to build pieces of code to play various roles in the system you're constructing. To a first approximation, you can do this by writing down a description in English of the system and the interactions you want to have with it (and that you want its parts to have with one another), then mapping these things onto elements of Java. When you do this, you will find that Java objects correspond roughly to the nouns of your description.

To be a bit more precise, Java objects are things in your computational world, but not all of the things are Java objects. Some of the things will have primitive types -- numbers, for example, will probably be `double`s or ints -- but most of the things that are important enough to represent and complex enough that Java doesn't have a built-in type for them will be objects in your world. This means that you will have to define a Java class which describes what this type of object is (more below).

For example:

- A counter has a number associated with it. When it starts out, the number is 0. You can increment the counter, and each time you do so, the number goes up by one. At any time, the counter can also be asked to provide the current value of its associated number.

The nouns in this paragraph are counter and number. (And you, but we'll assume that you either refers to the user, which we don't need to implement, or to some other component outside of the current system.) The counter will be a Java object; we can use an `int` for the number since it isn't asked to do anything, just to be there.

Methods are Verbs

When you write down your description, you will also find that there are lots of things that these objects do to/with/for one another, or that you want to do to/with/for them. These things correspond to the verbs in your English description, and they are the methods of your Java objects. Every verb has a noun associated with it -- its subject -- and ***every Java method belongs to some object***.

In our basic counter example, the verbs are increment (and its alternate form, goes up by one) and provide (as in "provide the current value"). Increment is something you need to be able to do to the counter object. We could handle provide in either of two ways: we could give the counter someone or something to provide the value to, or we could ask it. We will adopt the second of these options, though we will return to the first option in the chapter on Communication Patterns. This means that the counter object is going to have to have (at least) these methods.

Interfaces are Adjectives

Interfaces and classes are both types. How do you know when to use which one? As a general rule of thumb, names (including parameters, fields, and local variables) should generally be declared using an interface type whenever possible. Constructor expressions, of course, require a class type.

Interfaces are good at capturing commonality. It is almost always useful to define an interface corresponding to the set of features of your objects that would hold for any implementation of them. For example, the need for any counter to have an increment method and a getValue method makes these good properties to encapsulate in an interface. No matter how we implement the counter, these method properties will hold. In contrast, the fact that most counters will keep track of their value using a field (perhaps even an int field is an implementation-specific detail that cannot be expressed in an interface. An interface talks about what an object can do, not about how it accomplishes these tasks.

A `Counting` interface might say:

```
interface Counting
{

     void increment();
     int getValue();

}
```

Why would we use this? By referring to any actual counters by their interface type -- `Counting` -- rather than their implementation types, we make it possible for the implementor to modify details of the implementation -- or, even, to change which underlying implementation we're using -- without changing the code that uses it. We also avoid committing to any specific aspects of the implementation -- such as the representation of the current value through a long or a double or even a String -- that really shouldn't matter to the user of the class.

The name of this interface is only moderately adjectival, but most interfaces are named using adjectives. For example, we have seen `Resettable` and will soon see `Animatable`, `Runnable`, and `Cloneable`. We could almost call `Counting` `Incrementable` instead.

Classes are Object Factories

So if the nouns are objects, the verbs are methods, and the interfaces are adjectives, what is left for the classes? Java classes are *kinds* of objects. They correspond, roughly, to machines (or factories) that tell you (or Java) how to make new objects, not (necessarily) to anything explicitly in your English description.

For example, the `class BasicCounter` is something that tells Java how to make a `new` BasicCounter`()`. It doesn't appear explicitly in the English description, but parts of the description are about it and other parts imply things about what it must say. The phrase "When it starts out, the number is 0" talks about **initial conditions** for `BasicCounter` objects; the class is the thing responsible for establishing these (since it is the factory where `Counter`s are made).

For that matter, the class is responsible for establishing what the parts of an object are. "Parts" here refers to **methods** and **fields**. What are the pieces of a `BasicCounter` object? In this case, its number (and maybe an associated display). What are the things a `BasicCounter` can do (or that we can do with/to a BasicCounter)? increment and provide its value, at least. So the `class BasicCounter` will most likely include a number field (which is going to be of type `int`), as well as methods corresponding to incrementing and value-providing. It will also initialize the number field to 0.

Q. Is this a static or dynamic initialization? Where does it take place?

[Pic of counters]

Style Sidebar

Class and Member Documentation

This list summarizes many of the main features that good documentation will capture about classes and their members. For more detail, see the specific documentation sidebars in the previous chapter.

- methods
  - parameters: type and role
  - return value: type and role
  - function: why you'd do it
  - "side effects": what else it does (esp. values changed)
- fields
  - type and role
  - how it changes & which methods use/change it
  - constraints and interdependencies
- constructors
  - parameters: type and role
  - relation of parameters to the particular instance produced
  - "side effects": what else it does (esp. values changed)
- class
  - its interface, especially key methods and fields & how they interact

Some Counter Code

Here is a very basic implementation of the counter class:

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

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

     int getValue()
     {
```

```
            return this.currentValue;
        }

    }
```

Some notes on this code:

- The class is a factory for making `BasicCounter`s. Its body talks about what each individual `BasicCounter` looks like, not about the factory itself.
- Each individual `BasicCounter` has its ***own*** `currentValue` field. Each one starts out with the value `0`, but they can change independently: each `currentValue` field belongs to a specific `BasicCounter`.
- We haven't included a constructor because, in this case, Java's default constructor does what we want. This is in general true when there is no dynamic initialization (each instance starts out in the same state).
- The `increment` and `getValue` methods are methods that belong to each `BasicCounter` instance. In each case, they refer to the `currentValue` field of ***that*** `BasicCounter` instance. We note this by using the java keyword `this`.

Someone wanting to use a `BasicCounter` could now do so by invoking an instance creation expression with this `BasicCounter` factory:

```
new BasicCounter()
```

This expression is probably more useful if we embed it inside another expression or statement, e.g.,

```
Counting myCounter = new BasicCounter();
```

Note the use of the interface type when declaring the name, but the class type within the construction expression.

Now we can ask `myCounter` to increment itself or to give us its value:

```
myCounter.increment();
Console.println( myCounter.getValue() );
   // prints 1
   myCounter.increment();
myCounter.increment();
myCounter.increment();
Console.println( myCounter.getValue() );
   // prints 4
```

Final

A name in Java may be declared with the modifier `final`. This means that the value of that name, once assigned, cannot be changed. Such a name is, in effect, constant.

The most common use of this feature is in declaring final fields. These are object properties that represent constant values. Often, these field are static as well as final, i.e., they belong to the type object rather than to its instances. Making a constant static as well as final makes it easy for other objects to refer to this value. It is appropriate for static final fields to be declared public and to be accessed directly by other objects. Static final fields are the only fields allowed in interfaces.

In addition to final fields, Java parameters and even local variables can be declared final. A final parameter is one whose value may not be changed during the execution of the method. A final variable is one whose value is unchanged during its scope, i.e., until the end of the enclosing block.[Footnote: Final fields and parameters are unnecessary unless you plan to use inner classes. They may, however, allow additional efficiencies for the compiler, and in any case they cannot be detrimental.]

Java methods may also be declared final. In this case, the method cannot be overridden in a subclass. Such methods can be inlined by the compiler, i.e., the compiler can make these methods execute more efficiently than other non-final methods. A static method is implicitly final. An abstract method may not be declared final.

Java classes declared final cannot be extended (or subclassed).

Public and Private

When we defined the `BasicCounter` class, we intended that the rest of the world would interact with its instances (things produced by the `class BasicCounter` factory) only through `increment()` and `getValue()`. But there is nothing about the code we've written that prevents someone from defining a BasicCounter name and then changing the value of that `BasicCounter` instance's `currentValue` field. For example, it would be perfectly possible for another object to say

```
BasicCounter anotherCounter = new BasicCounter();
anotherCounter.currentValue = anotherCounter.currentValue + 1;
```

instead of

```
anotherCounter.increment();
```

This would be rather rude of it (and very bad style), but it is technically possible and unfortunately done all of the time. Using the interface type -- `Counting` -- rather than the class type -- `BasicCounter` -- is one way to avoid this, and this is yet another reason why it is generally better to use the interface type. But as the implementor of `BasicCounter`, we can't require that it always be treated as a `Counting` instead of as a `BasicCounter`. Further, coercion (such as `(BasicCounter) myCounter`) will get you around the interface-associated name.[Footnote: Specifically, it would be legal, if longwinded, to say

```
((BasicCounter) myCounter).currentValue
```

```
= ((BasicCounter) myCounter).currentValue + 1;
```

] Class designers don't always get to choose how users of the class will interact with it or as what type they'll choose to treat it.

We can take a stronger position on the matter of direct field access, though. We can, in fact, prevent direct field access by **protecting** the `currentValue` field of each `BasicCounter` instance. We do this by changing the declaration of the field in `class BasicCounter`:

```
class BasicCounter {

     private int currentValue = 0;

     void increment ...

}
```

By making `currentValue` **private** to `class BasicCounter`, only the instance of `BasicCounter` itself can access the `currentValue` field. Now, this rudeness on the part of the calling object would simply be impossible. (The compiler would complain that the calling object could not access BasicCounter's private field currentValue.)

In general, it's a good idea to define fields as `private` when you don't want them to be accessed directly by other objects. You can also define `private` methods, which are generally things an object uses for its internal computations but not intended to be used from outside the object. Private things are a part of the class's or its instances' own internal representations and machinations; they are not to be shared.

Any member, not just a field or a method, can be `private`. You can even define `private` constructors. Although this may seem like an odd thing to do, it actually isn't all that strange. It means that the class object (along with any instances it creates) maintains complete control over whether and when new instances can be created. The class can refuse to create any instances, or it can create just one instance and return this any time someone asks for a new one (using a special method the class defines for this purpose, such as getInstance(), not the (private) constructor), or it can ask for the secret password before creating an instance if it (or its designer) wants to.

The opposite of `private` is **public**. You should declare things `public` when you want them to be accessible from any part of anyone's code. You can also declare classes and interfaces to be `public`, in which case they must be defined in a file whose name is the same as the name of the class or interface, plus `.java`.

If you don't declare something `private` or `public`, it is in an intermediate state. There are actually two intermediate states, `protected` and the default state. These two are in fact equivalent to one another and to `public` unless you use packages, a Java feature that we will explore in the chapter on Abstraction. Until then -- until you are building complex enough code that you need to subdivide it at finer levels than all-or-none -- you should use `public` and `private` all of the time, i.e., everything in your code should be one or the other.

Kinds of Objects

Objects are the nouns of programming: the people, places, and things. Nouns do a lot of different things in the world and, similarly, objects do a lot of different things in programs. In this section, we take a closer look at several kinds of objects, their typical construction, and why you might use them. The objects discussed here are all relatively passive; they do nothing until asked. In the next chapter, we go on to look at active objects, objects that have their own instruction followers.

Data Repositories

A **data repository** is a very simple object that exists solely to hold a set of interrelated data. The data repository object simply glues these things together, providing a convenient way to deal with the grouped data as a single unit.

One example of a data object might be a postal address. This might consist of a street address, a city or town, a state or province, a postal code, and a country. There isn't really much that you would do with an address, other than pull out the individual pieces or maybe modify one or more of the pieces. (For example, the postal service just changed my postal code, so although my address object stayed the same, its postal code field needed to change.) The whole address is useful and meaningful in a way that the pieces individually are not, so it is often convenient to be able to package these pieces together and to pass the address object around as a single unit.

[Pic of address object]

Here is some code for a very simple address object. Note that this code has some aesthetic problems, which we will address shortly.

```
public class OversimplifiedAddress {
    public String streetAddress,
                  city,
                  state,
                  postalCode;
}
   // Problems with this class:
// Non-final fields ought not to be public.
// Fields ought to be initialized by (missing) constructor or default.
```

Like instances of this OversimplifiedAddress class, data repository objects exist to hold a collection of pieces together. Typically, each of these pieces is represented by a field of the object. The simplest form of data repository object is one -- like an instance of the Oversimplified Address class -- that has a set of public fields and nothing else. However, this form is not recommended.

One object should never access another object's fields directly. [Footnote: Actually, this should read "One object should never access another object's *non-final* fields directly." Final fields are in effect constants; the reasons for objecting to field access do not apply to read-only accesses to a constant.] Instead, an object should provide methods for accessing its fields. [Teacher's note: Where getter methods are simply long-winded ways of doing field access, a good compiler should be able to inline this code. In Java, this can be done when the getter method is declared final.]

In our simple address object, we violated this rule. To fix that class definition, we should instead make each of these fields internal to the object. So that other objects can access these fields, we need to provide getter and setter methods to access them. A **getter** method is a method that returns the value of a field. A **setter** method is one that has a single parameter, the new (desired) value of the field; evaluating this method modifies the state of the object to reflect this new value. Getter methods are sometimes called **selectors** and setter methods are sometimes called **mutators**. It is common to use the name of the field prefixed with get as the name of the getter method and the name of the field prefixed with set as the name of the setter method.

Note that getter and setter methods need not correspond one to one with fields. Instead, a setter method may change the value of more than one field; a getter value may return an object that encapsulates more than one field value. Alternately, a getter or setter may make reference to an apparent field that doesn't actually exist *per se*.

We can improve the address class by modifying it to use getter and setter methods. Only one pair of these methods is shown here, although the complete class definition would presumably contain four pairs of getter and setter methods.

```
public class BetterAddress {
    private String streetAddress,
                   city,
                   state,
                   postalCode;
....
    public void setPostalCode( String code )
    {
        this.postalCode = code;
    }

    public String getPostalCode()
    {
        return this.postalCode;
    }
}
   // Remaining problems with this class:
// Fields should be initialized by (absent) constructor or default.
```

Why shouldn't one object access the fields of another directly? (Why should you use getter and setter methods?)

1. Methods separate use from actual (internal) representation. The user of a class shouldn't need to know (or care) how information is actually represented inside the class. For example, US postal codes are commonly written as five-digit numbers. A different implementation of addresses intended

for use only in the US might actually represent the postalCode field using an int instead of a String. The getter and setter methods of this USAddress object could do the conversion for the user:

```
public String getPostalCode()
{
    return new String( this.postalCode );
}
```

We might have an interface (say, GeneralizedAddress) containing (an abstract version of) this method. Both USAddress and BetterAddress classes could implement the GeneralizedAddress interface, even though they use different internal representations.

Another variant of separating use from actual representation involves getter and/or setter methods for fields that don't actually exist. For example, it might be useful for these address objects to have a getAddressLabel field, which would return the multiline String containing the complete address suitable for printing on an envelope. This getter method would automatically calculate the appropriate value from the individual fields of the address object; there is no actual field corresponding to the information that this getter field provides.

```
public String getAddressLabel()
{
    return new String( this.streetAddress + "\n"
                       + this.city + ", "
                       + this.state + "  "
                       + this.postalCode + "\n"
                       + this.country );
}
```

Getter and/or setter methods like this one, which do not correspond to any actual field of the object, are sometimes called **virtual fields**. To the user of the object, it looks as though there's a field there. Whether that field actually exists or just looks like it is nobody's business but the implementing object's.

2. Methods can provide additional behavior, including access control and error checking. For example, BetterAddress could be augmented with an internal list of the states or provinces within each country. If the setter method were given an argument that didn't match one of the appropriate values, it could report an error. The most extreme case of this is a read-only field, one in which no non-private setter method is supplied. This prevents a user of the object from ever modifying the value of that field.[Footnote: Note that a read only field is different from a constant (final) field. A read-only field can be changed by its owning object, but not by anyone else. A final field's value, once set, cannot be changed. This is enforced by the Java compiler.]

   Another example of augmenting the behavior of a setter might involve automatically filling in the city and state whenever a postal code is entered. The postal code's setter method could look up the appropriate city and state information based on the postal code supplied and propagate this information to these other fields as well, saving the user the work of providing this information separately. (Some mail order companies do this now: you give them your postal code, and they tell you what city and state you live in!)

There are other reasons why methods, rather than fields, are a good idea. Some of these involve issues that

class exists so that its instances can hold both (horizontal and vertical) coordinates, e.g., of a window size. This allows them to be simultaneously returned from a method such as Window's getSize() method. If getSize() weren't able to return a data repository type such as Dimension, you'd first have to invoke a method that returned the Window's horizontal dimension, then one that returned its vertical dimension. If the Window's size changed in between these two method invocations, your two individual dimension components would combine to produce a nonsensical value!

Pure data repository objects are actually quite rare in good object-oriented design. This is because most objects do more than hold some state. The extensions we've described above, including propagation of changes, virtual fields, and access control already begin to expand the data repository idea. In the next subsection, we look at objects that exist to provide behavior without state. In the following subsection, we will return to objects that contain both data and more interesting behavior.

Resource Libraries

We have seen objects that hold together an interrelated set of data. Sometimes, an object exists to hold together an interrelated set of methods. If these methods are not tied to any particular state of the world, they may usefully be grouped together within a (generally non-instantiable) class that exists solely for this purpose. Consider, for example, the square root function. It is a useful function, and it is often convenient to have it lying around. But, in Java, any function must be a method belonging to a particular object. Java has a square root method; but whose method is it?

The answer to this question is that sqrt() belongs to a special class called Math. Math is a class that exists precisely so that you can use its methods, like sqrt(). Math is a canonical function library; it has no use beyond being the place to find its member functions. It exists to provide the answer to the question, "Whose method is sqrt()?"

Because Math is a place to find these functions, it is not a class of which you would want to make instances. Instead, Math has only static methods and static fields. This means that you can use its methods and data members through the class object (Math) itself.For example, a typical method is Math.sqrt(double d), which takes a double and returns a double that is the square root of its argument. Without the Math class to collect it and other mathematical functions, it is hard to imagine to whom this sqrt function could belong. Math exists so that there is a place to collect sqrt and a number of other abstract mathematical functions.

The Math class has static methods for the trigonometric functions, logarithms and exponentiation, various flavors of rounding, and very simple randomization. Math also has two (static final, i.e., constant) fields: E and PI, doubles representing the corresponding mathematical constants. See the sidebar on Math for details.

Q. Since it's not instantiable, why couldn't Math be an interface?

Math -- the class, with its static methods and fields -- is a very useful class. However, it wouldn't make sense to create any instances of it. In fact, Math has no publicly available constructor. This is a common way to prevent a class from being instantiated: give it only a private constructor. In general, a resource collection is the kind of object of which wouldn't have any use for multiple copies.

Another resource collection class is cs101.util.Console. Console -- documented in a sidebar in the chapter

[illegible] java.lang.System, cs101.util.MoreMath, and cs101.util.Coerce.

class Math

The built-in Java class Math may be the canonical resource library. It contains two (static) fields, Math.E and Math.PI, both doubles, corresponding to the mathematical constants e and pi, respectively.

Math also contains a host of useful mathematical functions, again all static. Each of the following methods takes a double as an argument and returns a double:

cos	cosine of its argument	acos	arc cosine of its argument
sin	sine of its argument	asin	arc sine of its argument
tan	tangent of its argument	atan	arc tangent of its argument
exp	Math.E raised to the power of its argument	log	Logarithm base Math.E of its argument
sqrt	square root of its argument	ceil	smallest double corresponding to an integer value that is larger than its argument
floor	largest double corresponding to an integer value that is smaller than its argument	rint	double corresponding to the integer value nearest its argument

`Math.abs` takes a double, a float, a long, or an int, and produces a value of the same type as its argument that is guaranteed to be non-negative.

`Math.max` and `Math.min` each take two arguments of the same type (both double, float, long, or int). max returns the larger of its arguments; min the smaller.

`Math.round` takes a double and returns the long closest in value to its argument.

Math.pow takes two doubles and yields the value of the first raised to the power of the second. (`Math.pow( base, expt )` = $base^{expt}$.)

Math.random takes no arguments and returns a double equal to or larger than 0.0 and strictly smaller than 1.0.

There are a few other Math methods not included here. In addition, there are extra mathematical functions (including more flexible and powerful randomization) available in the package java.math. For these additional methods, see the Java API documentation on the Javasoft web site.

Traditional Objects

Some objects, like data repositories, exist primarily to bundle together certain pieces of data. Other objects exist primarily to hold stateless, general-purpose functional behavior. Most objects fall into neither of these categories. Instead, most objects represent things with both state -- what happens to be true of them Right

Now -- and behavior -- how that object can change over time. Some of these objects, like Windows, Buttons, and Menus, have visual manifestations. Other objects, like the ones that represent Strings or URLs, are more obviously internal to programs. Many of the objects that you create will be of this kind.

A String is an object that keeps track of the sequence of characters of which it is composed, so somewhere inside the String object must be data that corresponds to those characters. But a String is not simply a data repository; it has a diverse set of methods. What kinds of things might you want to do with a String? Certainly look at some of the characters, which you can do using the String's `charAt(int index)` method. Java's String class provides additional methods, though, which allow you to do more than simply look at parts of the String. For example, there is `toUpperCase()`, which returns a String just like the one whose method you invoke, but with all letters in upper case. (For example, `"Hi there".toUpperCase()` returns a String that would print out as `"HI THERE"`.) String's `toUpperCase()` method is neither a selector nor a mutator. More complete descriptions of the String class and its methods are included in the sidebar on the String class in the first Interlude.

Another kind of traditional object that we've seen is a counter. This object has internal state (whatever the current count is set to) and methods providing access to this state (e.g., `increment()` and `getValue()`). The methods can't work without the state; the state isn't directly accessible, but provides the basis for method behavior. This is an extremely typical kind of object.

Here is some code implementing a slightly more sophisticated Counter class than the one described at the beginning of this chapter. In addition to the functionality provided by that BasicCounter class, this class implements the `Resettable` interface, i.e., provides a `reset()` method.

```
public class Counter implements Counting, Resettable
{
     private int currentValue;

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

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

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

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

}
```

The two methods -- `increment()` and `reset()` -- rely on the current state (count) of the individual instance whose methods they are. Two different counters can have two different states (e.g., one can have count 4 and the other count 27). Incrementing the first will have a different effect (producing 5, etc.) from incrementing the second (which produces 28). Resetting one will not reset the other. `Increment()` and `reset()` make no sense without reference to the particular counter instance they're incrementing or resetting. This relationship between state (data members) and methods is typical of "traditional" objects.

[Picture of multiple counters.]

Traditional objects are exemplified by the following properties:

- Each instance has its own state.
- This state is not directly accessible. Instead, it provides the basis for method behavior.
- Method behavior is dependent on the internal state of a particular instance.
- State plus behavior, packaged together, provide a single logical unit.

Types and Objects

Declared Type and Actual Type

What happens when we take an object of one type and treat it as though it had another type? One common example of this that we've seen is using an interface-type name to hold an object. The object is an instance of some class. The name says that it's in instance of some interface. The interface provides a much more limited view of the object than the actual implementation. Does this change the object? What happens when we ask whether the object is an instanceof its class, for example.

The answer is that the object is the same object no matter what its declared type (e.g. the declared type of the name that may be holding it, or of the method that may return it, or wherever else its type may be declared). It can do all of the same things regardless of its declared type. And it responds the same way when asked whether it is an instanceof its class, regardless of whether its declared type is some more specialized interface.

For example, if we take an instance of the Counter class defined above, with its reset(), increment() and getValue() methods, and assign it to a name of type Counting (an interface with only increment() and getValue() methods), we haven't actually changed the Counter instance:

```
Counting count = new Counter();
```

If we ask whether

```
count instanceof Counter
```

this is true. Of course

```
count instanceof Counting
```

is also true. But

```
count instanceof BasicCounter
```

is false, given the definitions earlier in this chapter.

Using a Counting name instead of a Counter name does have some effect, though. First, we may not know about the Counter type. In this case, we are limited to treating count as though it were a Counting, not a Counter. For example, we couldn't call its reset() method, because Countings don't have reset() methods. Even if we did know about Counters, we'd have to explicitly cast count to be a Counter before we could use its Counter-specific properties:

```
( (Counter) count ).reset();
```

So an interface provides a limited view without limiting the actual object.

Use Interface Types

When declaring names and otherwise using objects, you should generally use interface types rather than class types. This allows the implementation of objects to vary independently of their use. It also allows different versions of the object to be used without dependence on unnecessary or possibly mutable properties. An interface allows common behavior to be abstracted and relied on. An interface can also be used to allow for future abstraction and variation, such as the Counting interface that allowed for the creation of a Timer.

For example, suppose that we are building a video game. The outer window of the video game is likely to be the same whether the game is Pong or Battleship or SpaceInvaders. It has controls such as start, stop, reset, and pause. What exactly happens when these controls are invoked depends on the particular game that is displayed in this window. But we want to build a generic DefaultGameFrame window that doesn't have to rely on the particular type of game that it will hold. We can accomplish this using an interface.

```
public interface GameControllable
{
    public void start();
    public void stop();
    public void reset();
    public void pause();
    public void unpause();
}
```

Now, the DefaultGameFrame can refer to the game using the type GameControllable. As long as Pong or Battleship or SpaceInvaders implements GameControllable, any of these games can be used inside the DefaultGameFrame. When the DefaultGameFrame's reset control is invoked, DefaultGameFrame simply calls its GameControllable's reset() method. If the GameControllable happens to be Pong, it will bring the paddles back to rest and set the scores to 0. If the GameControllable is space invaders, the player will begin again with a full set of ammunition and plenty of aliens to shoot.

Use Contained Objects to Implement Behavior

One object can use another to provide behavior on the first object's behalf. For example, we might have a Clock object that provides a getTime() method and a setTime() method. We might also have a VCR object

that includes among its functionality getTime() and setTime(). Should the VCR implement its own getTime() and setTime() methods? This seems awfully inefficient. Or should the VCR reuse the Clock's getTime() and setTime() methods directly? (We will see a mechanism by which this can be accomplished in the chapter on Inheritance.) The problem with this solution is that the VCR isn't really a Clock (or a kind of Clock). Instead, the VCR can provide these methods by having a Clock inside it.

For example, the code for the VCR might say (in part):

```
public class VCR
{

     private Clock clock;

     public Time getTime()
     {
         return this.clock.getTime();
     }

     public void setTime( Time t )
     {
         this.clock.setTime( t );
     }

     // etc.

}
```

In this way, the VCR provides access to the Clock's methods indirectly. This reuse of behavior by inclusion is a very powerful mechanism. In this case, the VCR might be providing access to the full set o Clock's methods. In another case, the including class might only provide a subset of the included class's methods, or it might provide a superset by combining those methods in different ways. The including class and the included class can even implement a common interface (such as TimeStorer) so that code that uses one or the other can't really tell the difference *so long as it only uses the interface's methods.*

The DefaultGameFrame and GameControllable described above are similar. When the DefaultGameFrame is asked to perform a reset (or a start or a stop or...), it passes this request along to the GameControllable. In that case, the use of an interface type -- GameControllable -- for the included object increases the flexibility and usability of the including class.

The Power of Interfaces

Why are interfaces so good at providing this flexibility? Because and interface is all about the contract an object makes and not about implementation. By relying on an interface, you defer any dependence on implementation details that might not be true of another implementation. This independence from implementation-specific details is enforced by the compiler, which will not let you rely on properties of an object specified by its interface type beyond those explicitly declared in the interface.

An object can also implement many different interfaces. In this case, it can be "seen" by other objects through each of these different interface types. Each interface type provides a different view of the object. By controlling these interfaces, a programmer controls the view that the object's users have of that object.

Reliance on interface types doesn't work perfectly, though. For example, a resource library such as Console or Math doesn't have an interface type. This is because resource libraries are typically non-instantiable classes. Only instances can have interface types.

Chapter Summary

- In an informal description of the program, nouns generally correspond to objects or to fields, methods to verbs, and interfaces to adjectives.
- Classes are the factories from which objects are created.
- Interface types provide a valuable layer of abstraction, allowing the implementation to vary without affecting the use.
- Members, classes, and instance marked public are accessible from anywhere within a program. Members marked private are only accessible within their defining class or instance.
- A data repository object exists to glue together a set of interdependent data. It has fields corresponding to this data and methods that allow you to read and modify this data.
- A resource library exists to hold a collection of methods or system-wide resources. Generally, a resource library supplies these methods and resources statically, i.e., it is not a class that is ever instantiated.
- Traditional objects mix both data and methods. These objects provide the kind of integrated state-dependent behavior that we expect of real world objects.

Exercises

1. Design and implement a class called Time that keeps track of the hour and minute together. Give it a nextMinute method that returns another Time, a minute later. How do you access the fields of Time objects?

2. Design and implement a class that provides IntegerArithmetic functions add(int, int), sub(int, int), mul(int, int), and div(int, int). You can give it any other methods you think might be useful. What does its constructor do? Why do you think that Java doesn't have such a class?

3. Design and implement a 2DVector class representing vectors in the plane. Include sum, difference, and product methods.

Questions or comments:
<webmaster@cs101.org>


WHILE (TRUE) {
ECHO
}