:es.html,v 1.1 2003/06/11 19:39:26 jon Exp $

Introduction to Interactive Programming

By Lynn Andrea Stein

A Rethinking CS101 Project

# Chapter 4.
# Specifying Behavior: Interfaces

## Chapter Overview

- How do programs (and people) know what to expect?
- How do I describe a part or property of an entity to other community members?

This chapter introduces the idea of interfaces as partial program specifications. An interface lets community members know what they can expect of one another and what they can call on each other to do; in other words, interfaces specify "how they interact". In this way, an interface describes a contract between the provider of some behavior and its user. For example, the post office promises to deliver your letter to its intended recipient if you give it to them in the appropriate form. This promise (together with its requirements for a properly addressed and stamped envelope, etc.) constitutes a part of the post office's interface.

In this chapter, you will learn how to read and write Java interfaces. These allow you to use code designed by others -- in the same way that you can drop off an appropriately addressed letter at the post office -- and to tell others how to use the services that you provide. You will also learn about things that an interface *doesn't* tell you. For example, when you drop a letter off at the post office, you don't necessarily know whether it's going by truck or by train to its destination. You may not know when it is going to arrive. This chapter concludes with a discussion of what isn't specified by an interface and how good documentation can make some of these other assumptions explicit.

This chapter is supplemented by a reference chart on the syntax and semantics of Java interfaces. (See Java Charts/Interfaces)

## Objectives of the Chapter

1. To learn how to recognize and read Java method signatures.
2. To understand how an interface specifies a contract between two entities while separating the user from the implementation.
3. To be able to read an interface and know what behavior can be expected of an object that implements it.

## 4.1 Interfaces are Contracts

Programs are communities of interacting entities. How does one entity know what kinds of services another entity provides? How do programmers know what kinds of behavior they can expect from objects and entities that they haven't built? The answer is in the interface that one entity provides to another. In other words, interface is a piece of the answer to the question of how things interact.

In the previous chapter, we saw objects in use. We said that a the object's type specifies what that object can do. In Java, an interface has a very special meaning: some types are interfaces. In this chapter, we will learn how a Java interface specifies the behavior of objects and see how to read an interface definition to understand what the objects of that interface type can do. We will begin by learning about what an interface is in a more general sense.

### 4.1.1 Generalized Interfaces and Java Interfaces

The dictionary defines ***interface*** as "the common region of contact between two independent systems." In Computer Science, we use interface to mean the boundary between two (or more) things. In general, when you are constructing a community of interacting entities, interface refers to the "face" that one of these entities shows another: what services it provides, what information it expects. One entity may, of course, have many interfaces, showing different "faces" to different community members.

For example, in software engineering, the term ***user interface*** refers to the part of a computer program that a person using that program actually interacts with. For example, a ***graphical user interface*** (***GUI***) is one that uses a certain interaction style, e.g., typically contains buttons and menus and windows and icons. [[ Footnote: In fact, graphical user interfaces are sometimes called WIMP interfaces, for Window, Icon, Menu, Pointer. ]] Before GUIs, computer interfaces typically used text, one line at a time, the way that some chat programs work now.

A good interface meets the needs of its customers. For a user interface, this means taking into account the properties of the program and the rather different properties of human users. Some not-so-good user interfaces are not-so-good precisely because they overlook the fact that humans and computers have different skill sets. Like user interfaces, every interface should be designed bearing in mind the needs of the entities on both sides. We will learn more about graphical user interfaces in particular in Parts 3 and 4 of this book.

This more general Computer Science use of the word interface is one sense in which we will use the term in this book. In Java, there is a second, related but much more limited use of the word interface. A Java interface refers to a particular formal specification of objects' behavior: a Java type. The keyword `interface` is used to specify the formal declaration of a particular kind of contract guaranteeing the behavior of this type. (For example, there might be an interface defining clock-like behavior.) The Java language defines the rules for setting out that contract, including what can and can't be specified by it. A particular Java interface is a particular promise.

In this book, when we use the term "Java interface" or the code keyword `interface`, we are referring to this formal declaration. When we use the terms "generalized interface" or "user interface", we are referring to more general computer science notions of interfaces. A Java interface is one way to (partially) specify a generalized interface. There may also be things that are part of the general promise -- such as how long a particular request might take to answer or the actual appearance of a component on the computer screen -- that can't be specified in a Java interface.

This chapter deals specifically with Java interfaces. The ideas of generalized interfaces permeate all parts of this book; the generalized notion of an interface is central to interactive program design. We will explicitly revisit this issue -- generalized interface design -- in the chapters on Protocols and Communication in Part 4 of the book.

### 4.1.2 Interfaces Encode Agreements

Why do we want interfaces? What work do they do for us? Interfaces -- in both their general and theirs Java-specific senses -- are very useful to us. They allow us to use things without knowing precisely how they work. And they allow us to build things without knowing precisely how they will be used. Let us now look at a particular every-day kind of interface to see how this works.

An excellent example of a standardized interface is an electrical outlet. In the United States, there is a particular standard for the shape, size, and electrical properties of wall outlets. This means that you can take almost any US appliance and plug it in to almost any US wall outlet and rest assured that your appliance will run. The power company doesn't need to know what you're plugging in -- there are no special toaster outlets, distinct from food processor outlets, for example -- and you don't need to know whether the power company produced this electricity through a hydroelectric plant or a wind farm. The outlet provides a standard interface, with a particular contract, and as long as you live within the parameters of that contract, the two sides of the interface can remain relatively independent.

This is the power of an interface: An interface is a contract that one object or entity makes with another. Interfaces represent agreements between the ***implementor*** (or builder) of an object and its ***users***. In many ways, these are like legal contracts: they specify some required behavior, but not necessarily how that behavior will be carried out. They also leave open what other things the parties to the contract may be doing. As a result, an interface separates what the user needs to know from what the implementor needs to know.

In some cases, there may be multiple different interfaces that provide similar services. For example, US appliances don't generally work in European outlets. There are several standard electrical outlet interfaces throughout the world. It isn't clear that one of them is particularly better than another, but it is unquestionably true that you can't use one side of the US outlet interface (e.g., a US appliance) with the other side of the European interface (a 220V outlet). In the same way, software will only work if the user and the implementor are relying on the same interface. If you want to mix and match disparate electrical interfaces, you will need a special adapter component. The same is true for software.

There are also, even in the US, certain appliances that can't use standard wall outlets because they don't meet the conditions of the interface contract. For example, an electric oven draws too much current, and so needs a special kind of wall outlet. The physical connector -- the plug -- is different on this appliance, to indicate that it fits a different interface. You can't plug an electric oven in to a standard US wall outlet. This is because its needs don't meet the (sometimes implicit) constraints of standard (15 or 25 amp) US circuits. Sometimes this happens in software, too -- you need a different interface because the standard one doesn't provide precisely the functionality that you need.

## 4.1.3 A Java Interface Example

Consider, for example, a counter such as appears on the bottom of many web pages, recording the number of visitors. Most such counting objects have a very simple interface. If you have a counting object, you expect to be able to increment it -- add one to the number that the counting object keeps track of -- and to be able to read -- or get -- its current value. This is true pretty much no matter how the counting object actually works or what other behavior it might provide. In fact, by this description, a stopwatch might be a special kind of counting object that automatically increments itself. So we might say that increment and getValue form a useful interface contract specifying what a (minimal sort of a) counting object might be. In Java, we write this as:

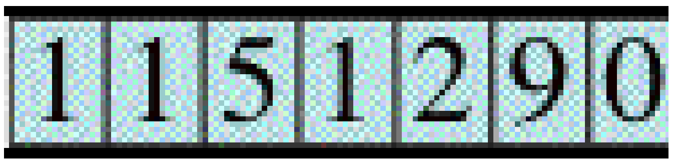

***Figure 4.1. A counter like this one appears on the bottom of many web pages. It could also be used as an automobile odometer.***

```
interface Counting        // gives the name of the interface
{
    void increment();     // describes the increment contract
    int getValue();       // describes the get value contract
}
```

By the end of this chapter, you will know how to read this interface declaration.

Once you and I agree on an interface for a counting object, I can build such an object -- and you can use it -- without your needing to know all of the details of how I built it. You can rely on the fact that you will be able to ask my counting object for its current value using `getValue()`. Your code, which uses my counting object, doesn't need to know whether increment adds one point (for a soccer goal) or six (for a touchdown in American football). It doesn't need to know whether I represent the current value internally in decimal or binary or number of touchdowns, field goals, etc.

If I one day exchange my original counting object for a more sophisticated one that can be reset before each game (or each time I rewrite my web page), your code should continue to work. This is because your code depends only on being able to increment and read the value of my counting object -- the properties specified by the `Counting` interface -- and my new object still satisfies that contract. Similarly, I can go off and build a counting object using whichever internal representations I wish to provide, so long as I meet the contract's commitments (`increment()` and `getValue()`).

Of course, you may want to know more about my counting object than what the `increment/getValue` interface tells you. Some of this information may be contained in the documentation for `Counting`. (This counting object's value will always be non-negative.) Other information may be contained in the documentation for my particular implementation. (My `BasicCounter` Counting object implementation is guaranteed to increase; its value cannot decrease.) If you want to know whether my object provides additional services, though, you may need to use an interface that specifies this additional behavior (e.g., a `Resetable` interface). We will explore the kinds of information conveyed by an interface, and that which should be included in interface documentation, towards the end of this chapter.

## 4.2 Method Signatures

In the previous chapter, we saw how to ask an object to perform a service using the . () notation. For example, to find the author of the book *Moby Dick*, we could ask it:

```
        mobyDick.getAuthor()
```

This asks the object named `mobyDick` to perform its `getAuthor` behavior. In an object-oriented programming language such as Java, the formal name for a behavior (or service) provided by an object is a ***method***: a method is a thing that an object knows how to do. In an interface, we focus on the specifications for these methods (or services) and not on the instructions for how to achieve them. That is, an interface is a collection of service specifications. Any object that implements that interface must satisfy those specifications, though there are virtually no limits on how it might do that.

The formal name for this kind of service specification is a ***method signature***. For example, the `Counting` interface specifies two services -- `increment` and `getValue` -- that every counting object must provide. The body of the interface declaration is these two method signatures, or service specifications. A method signature describes what things that method expects (or needs to know about) and what the method will return. It also needs a name, so that you can refer to and invoke the method (of course). In the chapter on Exceptions, we will see that there is one other kind of thing that can be a part of a method specification.

A method signature specifies a particular behavior that an object provides. It does *not* say anything about *how* the object will perform that behavior. A method signature does not, for example, contain a recipe that an object could follow to produce that behavior. As we saw in the first chapter of this book, a program needs to have a recipe for every piece of behavior that it produces, and in the next few chapters we will see how such recipes can be constructed. But producing behavior is not the job of an interface; an interface only specifies *what* behavior needs to be produced.

The goal of an interface is to provide an agreement between the provider of some behavior and its users. If it is done right, it allows both the provider and the user to work independently. In Java, an interface is a collection of method signatures, each of which describes a particular behavior that is part of an object's contract. Each method signature consists of three parts:

1. the name, or what the method is called;
2. the parameter specifications, or what things the method needs to do its job; and
3. the return type, or what the method will provide when it is done. [[ Footnote: There is actually one other part of some method signatures, the `throws` clause. Every method signature must have a name, parameter list, and return type, but some methods do not have a `throws` clause. The `throws` clause will be introduced in the chapter on Exceptions. In addition, certain modifiers -- such as `abstract`, explained below -- may be included in a method signature. ]]

The next three subsections describe each of these three pieces of a method signature. For each part, we will look especially at what that part tells the user of the interface and at what it requires of someone providing the behavior behind an interface. Once we have understood how a method signature works and is used, we will return to see how method signatures can be put together to form a complete interface.

### 4.2.1 Name

The purpose of a method name is to make it possible to talk about the method. The name is how you ask an object to perform that behavior using the . () syntax. The method name should, of course, make it easy for both behavior providers and behavior users to figure out what the method is supposed to do.

When you are using an interface, the name of the method is whatever name the interface says it is. You can determine this by reading the interface specification. Hopefully, the name was chosen well so that it is easy to remember and to figure out what that method does. The interface should, of course, also have documentation to help you understand how it works.

If you are building an interface, you can give a particular method any name that you want. It is a good idea to give it a name that will help you (and the users of your code) remember what it does. Recall that Java names are allowed to include alphanumeric and a few symbolic characters. The syntax of Java names sidebar in Chapter 3 lists the precise rules for legal Java names. By convention, the name of a method should start with a lower case letter.

### 4.2.2 Parameters and Parameter Types

***Figure 4.2. If you want an object to do something for you, you may have to supply it with some of the necessary things, called arguments. The object will think of these things as parameters.***

Parameters are the things that a method needs in order to work. For example, to check a book back into a library, the `CirculationDesk`'s `checkIn` method will need the `BookID` of that book. When you ask an object to perform one of its methods, you need to provide these things between the parentheses of the . () syntax:

```
circDesk.checkIn( mobyDickID )
```

The information that you supply to the method -- in this case, `mobyDickID`, the `BookID` corresponding to *Moby Dick* -- is called an ***argument***. The user of the method needs to have, and to provide the method with, all of the necessary arguments. Once the user hands the arguments to the method, the method will need a way to keep track of them. The way that the method does this is with a special kind of name called a parameter. A ***parameter*** is a temporary name associated with an argument supplied to a method. [[ Footnote: In chapter Classes and Objects, we will see how a parameter can be used by the provider of the method behavior to refer to the arguments supplied when the method is invoked. ]] Because a method signature specifies what the method does, it needs to describe what kinds of arguments the method requires.

For example, the method signature for the `CirculationDesk's checkIn` method looks like this:

```
boolean checkIn( BookID whichBook )
```

Just as arguments are supplied to a method request between parentheses, method signatures indicate what arguments are required in the same place. This specification of required arguments is called the method signature's ***parameter list***.

When you are designing an interface, you will need to specify a type and a name for each parameter. (The *type-of-thing name-of-thing* rule (from the Chapter on Things, Types, and Names) strikes again.) The type can be any legal Java type (including both primitive and object types); the name can be any Java-legal name that you choose to give the parameter. It is advisable that you give your parameters names that make it easy for the users and implementors of your method to figure out what role the particular parameter plays in the method. Our convention is to use names that begin with a lower-case letter for parameters.

The list of parameters is separated by commas: *type-of-thing name-of-thing, type-of-thing name-of-thing,* and so on until the last *type-of-thing name-of-thing* which doesn't have a comma after it. The whole list is enclosed in parentheses. You can list your parameters in any order. Of course, some orders will naturally make more sense than others, and although the choice is arbitrary, once chosen the order is fixed. This means that users and implementors of the method will need to follow the order declared in the interface.

The `getValue` and `increment` methods of `Counting` don't have any parameters, i.e., they don't need any information to begin operation. Their parameter lists are empty: `()` as in `getValue()` and `increment()`. `CirculationDesk`'s `checkIn` method has one parameter, a `BookID`. We can call that `BookID` anything we want to; the name given a parameter in a method signature turns out not to matter at all. Of course, calling it something like `whichBook` makes it easier for the eventual user of the method signature to figure out what information to supply.[[ Footnote: It is also conventional to give a parameter a name that starts with a lower case letter. ]]

A more complex `AlarmedCounting` interface might be mostly like our `Counting` interface but in addition have a `setAlarm` method that takes two parameters, one an `int` indicating the value at which the alarm should go off and the other a `String` that should be printed out when the alarm is supposed to be sounded.

```
setAlarm( int whatValue, String alarmMessage )
```

When you are using a method, you need to pass the method a set of arguments that match the order and types of the parameter list. That is, between the parentheses after the name of the method you're invoking, you need to have an expression whose type matches the type of the first parameter, followed by a comma, followed by an expression whose type matches the type of the second parameter, and so on, until you run out of parameters: `increment()`, `transform( "a string to transform" )`, or `setAlarm( 1000, "capacity exceeded" )` You can tell what arguments you need to provide to a method by reading the parameter list in its signature.

### 4.2.3 Return Type

A parameter list specifies the information that a method needs in order to do its work. A method signature also needs to specify what -- if anything -- its users can expect to get back. In many cases, a method request to an object returns a value. The `Counting` method called `getValue` is, not surprisingly, an example of such a method. So is a `CardCatalog`'s `lookup` method -- which returns a `BookID` -- or `Console`'s `readln` method. Returning this value is part of the method's specification, so the method signature needs to include this information.

Specifically, the signature of a method indicates the method's ***return type***: the type of the value returned. So, for example, `getValue`'s signature indicates that it returns an `int`, while `lookup`'s signature indicates that it returns a `BookID`:

```
int getValue();
BookID lookup( String whatToLookUp );
```

In some cases, the method does not return a value. (`increment` is an example of such a method: it changes the value stored inside the counting object, but doesn't give anything back to the entity that invoked it.) The return type of such a method is a special Java keyword: ***`void`***. The only purpose for `void` is as the return type of methods that don't return a value. The `Counting` interface's `increment` method doesn't return anything, so its return type `is void`.

When you use a method, you may or may not want to do something with the value returned. The return type of the method signature tells you what type of thing you can expect to get back, e.g., so that you can declare an appropriate name to store the result:

```
int counterValue = myCounting.getValue()
```

where `myCounting` is something that implements the `Counting` interface, i.e., satisfies the `Counting` contract (and therefore has an `int`-returning `getValue` method). After this statement, `counterValue` is a name that refers to whatever `int myCounting`'s `getValue` method returned.

### 4.2.4 Putting It All Together: Abstract Method Declaration Syntax

Now you know about all of the components of a method signature. All you need to know is how to put them together. The *type-of-thing name-of-thing* rule comes into play here as well. The type of a method is its return type, so a method specification is:

```
returnType methodName ( paramType1 paramName1, ...  paramTypeN paramNameN );
```

For example,

```
int getValue();
```

or

```
void increment();
```

or

```
void setAlarm( int whatValue, String alarmMessage );
```

Note that these declarations end with a semi-colon (`;`). This means that the method signature is being used here as a specification -- a contract. It doesn't say anything about how the method -- say `increment` -- ought to work. That is, it doesn't even have a space for how to perform this method, just the method specification.

This form -- method signature followed by a semi-colon -- is called an ***abstract method***. There is even a Java keyword -- abstract -- to describe such methods. It is OK, if sometimes redundant, to say

```
abstract void increment();
```

instead of the form given above. Method signatures are used as abstract methods in creating interfaces. But method signatures are also used in other parts of a Java program as part of actually creating behavior. We will see how to use method signatures in that way in the chapter on Classes and Objects.

Since interfaces always specify only method signatures, interface method declarations are always `abstract`. If you don't say so explicitly, Java will still act like the word `abstract` is there. However, if your method definition does not end with a semi-colon, your Java interface will not compile.

## 4.2.5 What a Signature Doesn't Say

The properties of a method that are documented by its signature are its name, its parameters, and its return type.[[ Footnote: In addition, method signatures may include visibility and other modifiers and any exceptions that the method may throw. ]] That leaves a whole lot open.

For example, for each parameter:

- What is that parameter intended to represent?
- What relationships, if any, are expected to exist among the parameters?
- Are there any restrictions on the legal values for a particular parameter?
- Will the object represented by a particular parameter be modified during the execution of the method?

For the return type:

- What is the relationship of the returned object to the parameters (or to anything else)?
- What may you do with the object returned? What may you not do?

Other questions not included in the method signature:

- What preconditions must be satisfied before you invoke this method?
- What expectations should you have after the method returns?
- How long can the method be expected to take?
- What other timing properties might be important?
- What else can or cannot happen while this method is executing?

Not all of these questions are relevant to every method. For example, the precise amount of time taken by the counting object's `getValue` method is probably not important; it is important that it return reasonably quickly, so that the value returned will reflect the state at the time that the request was made. However, it is important to recognize that these and other questions are not answered by your method signatures alone, so you must be careful to document your assumptions using Java comments.

## 4.3 Interface Declaration

Now that we know all about Java method signatures, it is very easy to declare a Java interface. A Java interface is simply a collection of method signatures.

### 4.3.1 Syntax

A Java interface is typically declared in its very own file. The file and the interfaces generally have the same name, except that the file name ends with `.java`. (For example, the `Counting` interface would be declared in a file called `Counting.java`.)

Like most other declarations, an interface follows the *type-of-thing name-of-thing* rule. The *type-of-thing* is, in this case, `interface`. The name is whatever name you're giving the interface, if you're declaring it:

```
interface Counting
```

Now comes an ***interface body***: an open-brace followed by a set of method signatures followed by a close-brace. Note that it doesn't matter in which order the two methods are declared; the two possible orders are equivalent. The whole thing (including the `interface Counting` part) looks like this:

```
interface Counting
{
    abstract void increment();
    abstract int getValue();
}
```

That's all there is to it.

**Q.** In this definition of `Counting`, the word `abstract` appears twice. In the previous definition, above, it doesn't appear at all. Explain.

In fact, that was so easy, let's try another interface. This one is `Resetable`, and it is a very simple interface. (Good interfaces often are.) `Resetable` has a single method:

**Style Notes**

### Method Documentation

Documentation for a method should always include the following items:

- Why would you want to use this method? What does i do? When is it appropriate (or not appropriate) to use this method? Are there other methods that should be used instead (or in addition)? Are there any other "hidden assumptions" made by this method?
- What does each parameter represent? Is it informatior supplied by the caller to the method? Is it modified during the execution of the method? What additional assumptions does the method make about these parameters?
- What does the return value of the method represent? How is it related to the method's arguments or other Thing in the environment? What additional assumptions may be made about this return value?
- What else might be affected by the execution of this method? Is something printed out? Is another (non-parameter) value modified when it is run? These non-parameter non-return effects are called ***side effects***.

In addition, if there are other assumptions made by the method -- such as how long it can take to run or what else can (or cannot) happen at the same time -- these should be included in the method's documentation.

Java provides additional support for some of these items in its javadoc utilities. See the appendix on javadoc for details.

```
interface Resetable
{
    abstract void reset();
}
```

This interface is fine, but it could do with a little bit of documentation. After all, there are many things that an interface *doesn't* specify.

Q. Can you identify some things that should be included in `Resetable`'s documentation?

For the precise specification of what may be included in an interface definition, in what order, and under what circumstances, see the Java Chart on Interfaces.

## 4.3.2 Method Footprints and Overloading

It might seem that each method in an interface would have a unique name. However, it turns out that this isn't the case -- at least, not exactly. Instead of a unique name, each method in an interface (or class) definition must have a unique ***footprint***. The method's footprint consists of its name *plus* its ordered list of parameter types. Only the ordered list of parameter types counts; the return type of the method, and the names given to the parameters, are not relevant to its footprint.

For example, a `reset()` method with no parameters (an empty parameter list, `()` ) has a different footprint from a `reset( int newValue )` method (with the parameter list `(int)` ), and both are different from `reset( String resetMessage )` (parameter list `(String)` ). Only the parameter type matters, though, not the parameter names: `reset( String resetMessage )` is the same as `reset( String whatToSay )`.

As long as two methods have different footprints, they can share the same name. This is very common and even has its own name: ***overloading***. Overloading allows an object to have two (or more) similar methods that do slightly different things. For example, there are two very similar mathematical rounding methods. One has the signature

```
int round( float f );
```

while the other has the signature

```
long round( double d );
```

Java's `Math` object has both of these methods. If you ask `Math` to `round` a `float`, it will give you an `int`. If you ask `Math` to `round` a `double`, it will give you a `long`. This is very convenient: in both cases, a floating point number is converted to an integer, but in either case the more appropriate size is used. Parameter type pairing like this -- `float`s with `int`s, `double`s with `long`s -- is one reason for method overloading.

Another kind of method overloading involves optional parameters. For example, our `AlarmedCounting` interface has a

```
void setAlarm( int whatValue, String alarmMessage )
```

method. It might also be useful to give it a second method that allows you to specify the alarm message, without changing the value at which it is set:

```
void setAlarm( String alarmMessage )
```

If both of these method signatures are part of the `AlarmedCounting` specification, the request

```
yourAlarm.setAlarm( 1000, "Capacity reached" )
```

sets the alarm message to trigger at 1000, printing the message "Capacity reached", while

```
yourAlarm.setAlarm( "Oops, all full" )
```

simply changes the warning to be issued when the `AlarmedCounting` reaches capacity.

Overloading method names is the choice of the interface builder. The interface user simply makes use of the interface as it is given.

## 4.3.3 Interfaces are Types: Behavior Promises

Now that we have these interfaces, what good do they do? Interfaces are *kinds of Things:* they are Java types.

In Java, every interface name is automatically a type name. That is, when you are declaring a (label) name, you can declare it suitable for labeling things that implement a specific interface. In the chapter on Classes and Objects, we will see how to declare Java classes and how to indicate what interface(s) the class implements.

So, for example, the declared type of `myCounting`, above, was `Counting`:

```
Counting myCounting;
```

In this example, `myCounting` is declared to be of type `Counting`, i.e., something that satisfies the `Counting` contract (interface) that we declared in the preceding sections. For example, we might have an interface called `Game` that includes a `getScoreCounter()` method that returns a `Counting`:

```
interface Game
{
    abstract Counting getScoreCounter();
    // maybe some other method signatures....
}
```

If `theWorldCupFinal` is a `Game`, then we might say

```
Counting myCounting = theWorldCupFinal.getScoreCounter();
```

In this case, we don't know anything more about the type of `myCounting` ; we just know that it is a `Counting`. *Often, as users of other people's code, interfaces are the only types we need to know about.*

### 4.3.4 Interfaces are Not Implementations

We have seen that an interface can be used as the type of an object. You can use names associated with that type to label the object. You can pass objects satisfying that interface to methods whose parameter types are that interface type, and you can return objects satisfying that interface from a method whose return type is that interface. The `Counting` in the previous paragraph was an example of the power of interfaces.

However, there are certain things that you cannot do with an interface.

Of course, when we're manipulating that `Counting` object, we don't know anything about how it works inside. We don't know, for example, whether it has a touchdown part and a field goal part, or is represented in decimal or in binary, or is likely to keep going up while we're thinking about it (since players might keep scoring). To figure this out, we'd need to know more than just the interface -- the contract -- that it satisfies; we'd need to know how it is implemented.

Interfaces are about contracts, promises. They don't, for example, tell you how to create objects that satisfy those promises. In the next several chapters, we'll learn about building implementations that satisfy these promises and about creating brand new objects that meet these specifications. To do that will require additional machinery beyond the contract/promise of an interface.

**Style Notes**

**Interface Documentation**

An interface should be properly documented, typically usin a multi-line or javadoc comment immediately preceding its declaration.

Documentation for an interface should include the followi information:

- What kind of thing does this interface represent? Why would you want to use an object of this kind? What could i do for you? What could you do with it?
- What kinds of assumptions or conditions does this ki of object need to do its job? Are there any special objects that it might need to have around or to work with?
- What services does this kind of object provide, and how do you use them? These questions are typically answered by the individual methods, but a brief overview c what methods the interface provides is always useful. It is may also be useful for the interface to document which method(s) to use when, especially when multiple similar methods exist.
- Each method signature in an interface should also be individually documented. See the sidebar on Method Documentation for further discussion.

The interface's documentation should make it easy for a potential user to find the method(s) s/he wants. It should also mak it possible for someone seeking to implement this interface to determine whether s/he has met the intent as well as the formal specification of the interface. If I am building a stopwatch, do I wa to subscribe to the `Counting` interface?

Remember that an interface declaration is largely about *wha* not *how*. It specifies contracts and promises, not mechanism.

Java provides additional support for some of these items in its javadoc utilities. See the appendix on javadoc for details.

## Chapter Summary

- An interface is a contract that a particular kind of object promises to keep.
- Java interfaces are Java types.
- Every (public) interface must be declared in a Java file with the same name as the interface.

- Java interfaces contain method signatures.
- A method signature is also called an `abstract` method.
- A method signature specifies a method's name, parameter types, and return type. It does not say anything about how the method actually works.
- Arguments are things supplied to a method; parameters are what the method calls them.
- One interface may have multiple methods with the same name, as long as they have different ordered lists of parameter types. Method name plus ordered parameter type list is called the method's footprint. Having two methods with the same name but different footprints is called overloading the method name.
- An interface does not contain enough information to create a new object, though it can be used as a type for an existing object (that implements the interface's promises).
- Many important properties of a method specification or interface are not specified by the method or interface declaration. Good documentation describes these additional assumptions.

## Exercises

1. StringTransformer has a transform method. Declare an interface, Transformer, that contains this single method specification, so that StringTransformer might be an implementation of this interface.

2. A Clock is an object that needs a method to read the time (say, `getTime`) and one to set the time (say `setTime`). Assuming that you have a type Time already, write the interface for a Clock.

3. Extend the interface of Clock (from the previous exercise) to include a setAlarm method (that should specify the Time at which the alarm should go off.

4. Extend the Clock interface further so that there is a second setAlarm method that takes a Time and a boolean specifying whether the alarm should be turned on.

5. Write the interface AlarmedCounting.

6. Consider the following interface:

```
interface Game
{
   /* returns the Counting that keeps track of the team's score */
       abstract Counting getScoreCounter( Team team );

   /* returns the Counting that keeps track of how many fouls */
   /* each player has committed */
       abstract Counting getFoulCounter( Team team, int playerNumber );

   /* returns the counting that keeps track of how much time */
   /* has passed in the period so far */
       abstract AlarmedCounting getTimeCounter();

   /* returns the length of a period */
       abstract int getPeriodLength();
}
```

Assume that `theWorldCup` is a particular `Game`, according to this interface.

a. Write a type declaration for the name `theWorldCup`. Don't worry about where its value comes from.

b. Write a type declaration suitable for holding the result of `theWorldCup.getTimeCounter()`.

c. Write an expression that returns the object that counts the fouls of player 5 on Team `manchesterUnited`.

d. Write an expression that returns the current score of Team `juventus` in `theWorldCup`

e. Write a method invocation that sets up theWorldCup (and its internal representation) so that it will print "Period over!" when the elapsed time reaches the length of the period.

7. Write the interfaces for `CirculationDesk` and `CardCatalog` (from Chapter 2). Can you write the interface for `BookShelf`?

AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>