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

Encapsulation

Chapter Overview

- How do I package up implementation details so that a user doesn't have to worry about them?
- How do I make my code easier to read, understand, modify, and maintain?

Good design separates use from implementation. Java provides many mechanisms for accomplishing this. In this chapter, we review a variety of mechanisms that allow this sort of separation.

Procedural abstraction is the idea that each method should have a coherent conceptual description that separates its implementation from its users. You can encapsulate behavior in methods that are internal to an object or methods that are widely usable. Methods should not be too complex or too long. Procedural abstraction makes your code easier to read, understand, modify, and reuse.

Packages allow a large program to be subdivided into groups of related classes and instances. Packages separate the names of classes, so that more than one class in a program may have a given name as long as they occur in different packages. In addition to their role in naming, packages have a role as visibility protectors. Packages provide visibility levels intermediate between public and private. Packages can also be combined with inheritance or with interfaces to provide additional encapsulation and separation of use from implementation.

Inner classes are a mechanism that allows one class to be encapsulated inside another. Perversely, you can also use an inner class to protect its containing class or instance. Inner classes have privileged access to the state of their containers, so an inner class can provide access without exposing the object as a whole.

Objectives of this Chapter

1. To understand how information-hiding benefits both implementor and user.
2. To learn how to use procedural abstraction to break your methods into manageable pieces.
3. To be able to hide information from other classes using visibility modifiers, packages, and types.
4. To recognize inner classes.

Design, Abstraction, and Encapsulation

This chapter is about how information can be hidden inside an entity. There are many different ways that this can be done. Each of these is about keeping some details hidden, so that a user can rely on a commitment, or contract, without having to know how that contract is implemented. There are numerous benefits from such information hiding.

First, it makes it possible to use something without having to know in detail how it works. We do this all the time with everyday objects. Imagine if you had to understand how a transistor works to use your computer, or how a spark plug works to use your car, or how atoms work to use a lever.

Second, information-hiding gives some flexibility to the implementor. If the user is not relying on the details of your implementation, you can modify your implementation without disturbing the user. For example, you can upgrade your implementation if you find a better way to accomplish your task. You can also substitute in different implementations on different occasions, as they may become appropriate.

Finally, hiding information is liberating for the user, who does not expect nor make great commitment to particulars

of the implementation. The name for this idea -- of using more general properties to stand in for detailed implementation -- is **abstraction**. To facilitate abstraction, it is often convenient to package up the implementation details into a single unit. This packaging-up is called **encapsulation**.

Procedural Abstraction

Procedural abstraction is a particular mechanism for separating use from implementation. It is tied to the idea that each particular method performs a well-specified function. In some cases, a method may calculate the answer to a particular question. In others, it may ensure the maintenance of a certain condition or perform a certain service. In all cases, each method should be accompanied by a succinct and intuitive description of what it does.[Footnote: It is not, however, essential that a method have a succinct description of *how it does* what it does. How it accomplishes its task is an implementation detail.] A method whose function is not succinctly describable is probably not a good method. Conversely, almost every succinctly describable function should be a separate method, albeit perhaps a private or final one.

This idea, that each conceptual unit of behavior should be wrapped up in a procedure, is called **procedural abstraction**. In thinking about how to design your object behaviors, you should consider which chunks of behavior -- whether externally visible or for internal use only -- make sense as separate pieces of behavior. You may choose to encapsulate a piece of behavior for any or all of the following reasons:

- It's a big, ugly function and you want to hide the "how it works" details from code that might use it. Giving it a name allows the user to ignore how it's done.
- It's a common thing to do, and you don't want to have to replicate the code in several places. Giving it a name allows multiple users to rely on the same (common) implementation.
- It's conceptually a separate "task", and you want to be able to give it a name.

Note also that the behavior of a method may vary slightly from invocation to invocation, since the parameters can influence what you the code actually does.

The Description Rule of Thumb

Each method in your program should have a well-defined purpose, and each well-defined purpose in your program should have its own method. You should be able to succinctly state what each method in your program does. If you cannot, your methods are either too large (i.e., should be broken into separable conceptual units) or too small (i.e., should be combined so that each performs a "complete" task.

Note that having a succinct description of what a method does is quite different from being to state succinctly how it accomplishes this. It is unfortunately all too common that a method's implementation is obscure. It is important that the user understand when, why, and under what circumstances your method should be used, i.e., what it does. You provide a method precisely so that the user will not have to understand how your method works.

For example, it is common to test complex conditions using a single predicate. One such instance might be the Calculator's `isDigitButton()` method, which determines whether a particular Calculator button represents the digits 0 through 9 (or instead is, e.g., an arithmetic operator). The logic behind `isDigitButton()` might be somewhat obscure. However, it is easy to succinctly state what the method determines and, therefore, when and why you might use it. This use of predicates as abstractions make code for easier to read, decompose, and understand.

The importance of succinct summarizability does not mean that there is exactly one method per description. For example, one succinctly summarizable method may in turn rely on many other succinctly summarizable methods. This is the "packaging up substeps" idea from Chapter 1: making a sandwich may be described in terms of spreading the peanut butter, spreading the jelly, closing and cutting the sandwich. Each substep may itself be a method. When the substeps are not likely to be useful for anything except the larger method of which they are a part, these methods should be private to their defining class.

It may also be the case that multiple methods each implement the same well-defined purpose. For example, multiple similar methods may operate on different kinds of arguments. A method that draws a rectangle may be able to take a java.awt.Rectangle, two java.awt.Points, or four ints as arguments. Each of these methods will have a different signature. They may, however, rely on a common (shared) method to actually perform much of the work, sharing as much code as possible. (See the repetition rule of thumb, below.)

Or it may be the case that multiple distinct object types each have similar methods performing similarly

summarized functions. In this case, it may make sense to have a common interface implemented by each of these classes, documenting their common purpose. Occasionally it even makes sense to split off the method into its own class, turning instances of the new class into components of the old. (See the discussion of using contained objects in the chapter on Object Oriented Design.)

When a single method does too many things, it can be difficult to decide whether you want to invoke it. It can be awkward to figure out what it is really doing. And the interdependencies among subtasks can make your code hard to maintain, especially if the assumptions that caused you to bundle these pieces together no longer hold.

Succinct summarizability makes your code immensely easier to read. By choosing descriptive names, you can often make your code read like the English description of what it does. This makes it easier to read, understand, modify, and maintain your code.

The Length Rule of Thumb

A single method should ideally fit on a single page (or screen). Often a method will only be a few lines long. If you find yourself writing longer methods, you should work on figuring out how to break them up into separable substeps. The description rule of thumb is handy here.

When a method's implementation takes up too much space, it is difficult to read, understand, or modify. It can be hard to hold the whole method in your head. It can be overwhelming to try to figure out what it is actually doing.

Appropriate method length is a matter of some individual judgment. Some people don't like to write methods longer than a half-page. Others regularly write much longer methods. As you become a more skilled programmer, you will become accustomed to keeping track of larger and more complex programs. But more complex programs do *not* mean longer methods. It will always be the case that brevity of individual units -- such as methods -- makes the overall flow easier to understand. Mnemonic names (describing what the method accomplishes) and programs that read like English descriptions of their behavior (through the use of well-chosen names) make your code more comprehensible to subsequent readers.

How do you know when to break code into pieces? If you discover that you have written a method that does not fit on a single page, you should write an outline for how the code works. Each of the major steps of this outline should be turned into a method. The original code should be rewritten in terms of these methods. The major steps should now be shorter methods. If these are still too long, repeat this process until each piece of code has a succinct description and occupies no more than two pages of code.

Note: Do not worry about inefficiency created by having too many small methods. First, intelligible code is so much easier to read and maintain, and code carefully optimized for efficiency so much more difficult to work with, that it rarely pays to do this sort of optimization until you are a skilled programmer. Further, a good compiler should be able to optimize. For example, if you make a method private or final, the compiler can in-line it.

The Repetition Rule of Thumb

Any time that the same code appears in two different places, you should consider capturing this common patterns of usage in a single method. When this happens, it is often because there is an idea expressed by this code. It is useful to give this idea a name, and to encapsulate or abstract it for reuse. Even if there are minor differences in the code as it appears, you may be able to abstract to a common method by supplying the distinguished information as arguments to the method. Each of the original pieces of code should be rewritten to use the common method.

Methods created by abstracting two or more pieces of code within the same class are often declared private. This is appropriate whenever the common behavior is local to the particular object and not something you want to make generally available. At other times, though, the common code is a useful and nameable function on its own. Though you may discover the commonality by replicating code, the existence of a separate method to replace this redundancy can be turned into an opportunity to export this functionality if it should make sense to do so.

Combining redundant code is also important in the case of constructors. Constructors can share code by having one invoke another -- using the special `this()` construct -- or by using a call to one or more (private) helper methods. A common programming mistake is to modify only one constructor when in reality the same change must be made to every constructor. Having the bulk of the work of the constructor done by a common method (or shared by using `this()`-constructors) eliminates this error.

Sharing redundant code shortens your program, making it easier to read, understand, modify, and maintain. It also

helps to isolate a single point where each piece of behavior is performed. This single point can be understood, modified, and debugged once rather than each time it (redundantly) appears.

Example

In the example immediately below, we will modify code based on redundancy, i.e., the repetition rule of thumb. The result will also make our code more succinct and easier to read. The newly created method will be succinctly summarizable and a legitimately separable subtask.

Consider a bank account, which might have a method that allows the account's owner to obtain balance information:

```
int getBalance( Signatory who ) throws InvalidAccessException
{
        if ( ! who == this.owner  )
        {
                throw new InvalidAccessException( who, this )
        }
        // else
        return this.balance;
}
```

It might also have a withdraw method that allows the owner to remove amount from the account, returning that amount as cash:

```
public Instrument withdraw( int amount, Signatory who ) throws InvalidAccessException
{
        if ( ! who == this.owner )
        {
                throw new InvalidAccessException( who, this )
        }
        // else
        this.balance = this.balance - amount;
        return new Cash( amount );
}
```

We could abstract the common pattern here, which is the verification of a signatory's right to access this account:

```
private void verifyAccess( Signatory who ) throws InvalidAccessException
{
          if ( ! who == this.owner )
          {
                  throw new InvalidAccessException( who, this )
          }
}
```

Now, we can rewrite getBalance and withdraw:

```
int getBalance( Signatory who ) throws InvalidAccessException
{
        this.verifyAccess( who );
        return this.balance;
}

public Instrument withdraw( int amount, Signatory who ) throws InvalidAccessException
{
        this.verifyAccess( who );
        this.balance = this.balance - amount;
        return new Cash( amount );
}
```

Much simpler, much more succinct, and in addition if we later need to modify the access verification routine, there is only a single place -- `verifyAccess()` -- where changes will need to be made.

Style Sidebar

Procedural Abstraction

- Use procedural abstraction when a method call would make your code (at least one of)
  - shorter, or
  - easier to understand.
- Your method should be concisely describable as "single function", though the function may itself have many pieces.
- Use parameters to account for variation from one invocation to the next.
- Return a value when the target of an assignment varies; leave the actual assignment out of the method body.
- Share code where possible. This is especially true among constructors, where one constructor can call another using this().
- Make internal helper procedures private. Make generally useful common functionality public (or protected).

Benefits of Abstraction

Abstracting procedures -- creating short, succinctly describable, non-redundant methods -- has many benefits. Even in the simple example of the preceding section, we can see many of these.

Procedural abstraction makes it easier to read your code, especially if methods have names corresponding to their succinct descriptions and the flow of code reads like the logic of the English description. Compare the before-and-after withdrawal methods of the bank account in the previous section.

Greater readability makes it easier to understand and figure out how to modify and maintain code. Separating functionality into bite-sized pieces also creates many opportunities to modify individual methods. Sharing these methods also centralizes the locations needing modification. For example, we could add a digital signature check to the verification procedure of the bank account by modifying only verifyAccess, not the bodies of getBalance or withdraw.

In contrast, long methods with complicated logic can be particularly hard to modify, either because their interconnected logic can be so difficult to understand or because it can be hard to find the right place to make the change.

As the needs of your code change, you will also find it easier to rearrange and reconfigure what your code does if the logical pieces of the code are separated. For example, we might add a wireTransfer method to the bank account. In doing so, we can reuse the verifyAccess method.

Of course, smaller methods make for bite-sized debugging tasks. It is much easier to see how to debug access verification in the newer bank account than in the version where each account interaction has its own verification code and where verification is intimately intertwined with each transaction. And if we need to modify the verification procedure -- to give diagnostic information, to step through the method, or to fix it -- there is a central place to make these changes.

Procedural abstraction also makes it easier to change behavior by substituting a new version of a single method. If a method is not private, it can be overridden by a subclass, specializing or modifying the way in which it is carried out without changing its succinct specification. We could, for example, have a more secure kind of bank account using the digital signature verification method alluded to above.

Many of the advantages of procedural abstraction are also provided by good object design. A method signature is a reasonable abstraction of the behavior of an individual method. An interface plays a similar role for an entire object, packaging up (encapsulating) the behavioral contract of an object so that its particular implementation may vary. Interfaces also make it easier to see how a single abstraction can have many coexisting implementations.

Protecting Internal Structure

Procedural abstraction is an important way to separate use from implementation and a significant part of good program design. Procedural abstraction is not the only kind of abstraction that you need in a program, though. Often, other techniques are used, either alone or with procedural abstraction, to hide implementation details. For example, if you use procedural abstraction to create local helper methods, you generally will not want these helper methods to be available for other objects to use.

In this section, we will look at several ways to protect internal structure -- such as helper methods -- from use by

others. These techniques protect implementation by making parts of the inner structure of an object inaccessible from outside that object or that group of interrelated objects. This packaging of internal structure is another kind of encapsulation. This section discusses some Java-specific ways to encapsulate functionality. Many programming languages offer similar mechanisms.

private

One of the most straightforward ways to protect internal structure -- such as fields or helper methods -- is to declare them private. We have seen in the section above how private methods can be used for procedural abstraction -- to break up a long procedure, to capture common patterns, etc. -- without exposing these functions to other objects. A method (or other member) declared private can only be called from within the class.

Beware: This is not the same thing as saying that only an object can call its own private methods. An object can call the private methods of any other instance of the same class.

Private is extremely effective at protecting methods and other members from being used by other objects. However, a member declared private cannot be accessed from code within a subclass. This means that if you modify code in a subclass that relies on a private helper method in the superclass, you will have to recreate that private helper method.

Packages

An alternative to the absolute protection of private is the use of packages. A package is a collection of associated classes and interfaces. You can define your own packages. Libraries -- such as the Java source code or the cs101 distribution -- generally define packages of their own. The association among classes and interfaces in a package can be as loose or as tight as you wish to make it.

Sometimes the association among objects is merely by convenience: many kinds of objects deal with the same kind of thing. Most of the cs101 packages are of this sort. Often, it makes sense to define a set of interrelated classes and interfaces in a single package and to provide only a few entry points into the package, i.e., a few things that are usable from outside the package. These packages represent associations by shared interconnectedness. Most of the interlude code is of this sort. Java defines a large number of packages, some of each kind.

In the bank account, we might well choose to define the interface `Instrument` (representing cash and checks, among other things) and classes `BankAccount`, `CheckingAccount`, `Cash`, etc. in a single package, say `finance`.

Packages play two roles in Java. The first concerns names and nicknames. Packages determine the proper names of Java classes and interfaces. The second role of packages is as a visibility modifier somewhere between private and public.

Packages and Names

A class or interface is declared to be in particular package *packageName* if the first non-blank non-comment line in the file says

```
package packageName;
```

packageName may be any series of Java identifiers separated by periods, such as java.awt .event and cs101.util. By convention, package names are written entirely in lower case. A file that is not declared to be in a specific package is said to be in the default package, which has no name.

Every Java class or interface actually has a long name that includes its package name before its type name. So, for example, String is actually java.lang.String, because the first line of the file String.java says

```
package java.lang;
```

and Console is cs101.util.Console, because it is declared in a file that begins

```
package cs101.util;
```

Any (visible) class or interface can always be accessed by prefacing its name by its package name, as in java.awt.Graphics or cs101.util.Console. If we declare the package `finance` as described above, the interface

`finance.Instrument` would actually have a distinct name from the interface `music.Instrument`.

In some cases, you can also access the class more succinctly. If you include the statement

```
import packageName.ClassName;
```

after the (optional) package statement in a file, you may refer to *`ClassName`* using just that name, not the long (package-prefaced) name. So, for example, after

```
import cs101.util.Console;
```

the shorter name `Console` may be used to refer to the `cs101.util.Console` class. Similarly,

```
import packageName.*;
```

means that any class or interface name in *`packageName`* may be referred to using only its short name, unprefaced by *`packageName`*.

Note, however, that this naming role for packages is only one of convenience and does not provide any sort of actual encapsulation. The use of a shorter name does not give you access to anything additional. In particular, it does not change the visibility of anything. Anything that can be referred to using a short name after an import statement could have been referred to using the longer version of its name in the absence of an import statement.

There are three exceptions to the need to use an import statement, i.e., three cases in which the shorter name is acceptable even without an explicit import.

1. Names in the default package can always be referred to using their short names.
2. Names in the current package (i.e., the package of which the file is a part) can always be referred to using their short names.
3. Names in the special package `java.lang` can always be referred to using their short names.

You are not allowed to have an import statement that would allow conflicts. So, for example, you could not have both statements

```
import finance.*;
import music.*;
```

if both packages contain a type named `Instrument`. You could, however,

```
import finance.BankAccount;
import music.*;
```

since the first of these import statements doesn't shorten the name of the interface `finance.Instrument`. If you do import `finance.BankAccount` and `music.*`, you can still refer to the thing returned by `BankAccount`'s `withdraw` method as a `finance.Instrument`.

Package Naming Summary

A class or interface with name *`TypeName`* that is declared in package *`packageName`* may always be accessed using the name *`packageName.TypeName`*, provided that it is visible. (See the visibility summary sidebar.)

The class or interface may be also accessed by its abbreviated name, *`TypeName`*, without the package name, if one of the following holds:

- The class or interface is declared in the default (unnamed) package.
- The class or interface is declared in the current package, i.e., *`packageName`* is also the package where the accessing code appears.
- The class or interface is declared in the special package `java.lang`, i.e., *`packageName`* is `java.lang`.
- The file containing the accessing code also contains one of the following import statements:
  - `import packageName.TypeName;`
  - `import packageName.*;`

Packages and Visibility

The second use of packages is for visibility and protection. This use does accomplish a certain kind of encapsulation. We have already seen `private` and `public`, visibility modifiers that prevent the marked member from being seen or used or make it accessible everywhere. These two modifiers are absolute. Packages allow intermediate levels of visibility.

Between `private` and `public` are **two** other visibility levels. One uses the keyword `package`. The other is the level of visibility that happens if you do not specify any of the other visibility levels. This is sometimes called "package" visibility, although it differs from friendly visibility in other languages and, additionally, there is no corresponding keyword for it.

A member marked `protected` visible may be used by any class in the same package. In addition, it may be referenced by any subclass. It is illegal -- and causes a compiler error -- if something outside the package, not a subclass, tries to reference a member marked `protected` visible.

A member, class, or interface not marked with a visibility modifier is visible only within the package. It may not be accessed even by code within subclasses of the defining class or interface, unless they are within the package.

This means that classes and interfaces may be declared without the modifier `public`, in which case they can only be used as types within the package. Members may be declared without a modifier, in which case they can be used only within the package, or they may be declared `protected`, in which case they can be used only within the package or within a subclass. A non-public class or interface need not be declared in its own separate Java file.

Note, however that although a subclass may increase the visibility of a member, it may not further restrict visibility. So a subclass overriding a `protected` method may declare that method `public`, but not unmodified (package) or `private`.

There is no hierarchy in package names. This means that the package java.awt.event is completely unrelated to the package java.awt; their names just look similar.

Visibility Summary

A member, class, or interface marked `public` may be accessed anywhere.

A member marked `protected` may be accessed anywhere within the containing package or anywhere within a subclass (or implementing class).

A member, class, or interface not marked has "package" visibility and may be accessed anywhere and only within the containing package.

A member marked `private` may only be accessed within the containing class or interface.

We can use this approach to encapsulate certain aspects of our BankAccount example without making all of the relevant members private. After all, we want to protect these members from misuse by things outside the financial system (and therefore presumably outside the package `finance`), not from legitimate use by other things within the banking system.

So we might declare:

```
public class BankAccount
{
        ...
}
```

and

```
public interface Instrument
{
        public abstract int getAmount();
        public abstract void nullify();
}
```

but

```
class Cash implements Instrument
{
        private int amount;
        private boolean valid;

        protected Cash( int amount )
        {
                this.amount = amount;
                this.valid = true;
        }

        public int getAmount()
        {
                return this.amount;
        }

        protected void nullify()
        {
                this.valid = false;
        }
}
```

This absence of the keyword `public` on the class definition means that the class `Cash` is accessible only to things inside the `finance` package. The `Cash` constructor is declared `protected`, so `Cash` may be created only from within this package. But the two methods that `Cash` implements for its interface, `Instrument`, must be `public` because you cannot reduce the visibility level declared for a method and the interface's methods are declared public.[Footnote: The methods of a public interface must be public, but an interface not declared public may have methods without a visibility modifier.]

Unfortunately, the guarantees of packaging are not absolute. There is nothing to prevent someone else from defining a class to reside in an arbitrary package. For example, I could declare a class `Thief` in package `financial`, allowing `Thief` instances full access to the `Cash` constructor.

Inheritance

Inheritance can be used as a way of hiding behavior. Specifically, you can create hidden behavior by extending a class and implementing the additional behavior in the subclass. Conversely, labelling an object with a name of a superclass type has the property that it makes certain members of that object invisible.

You cannot invoke a subclass method on an object labelled with a superclass type that does not define that method, even though the object manifestly has the method. You can take advantage of this in combination with the visibility modifiers, for example creating a package-only subclass of a public class. Outside the package, instances of this subclass will be regarded as instances of the superclass, but because the subclass type is not available (since it is not visible outside the package), its additional features cannot be used.

For example, a specialized package-internal type of `BankAccount` might allow checks to be written:

```
class CheckingAccount extends BankAccount
{
     ...

        protected Instrument writeCheck( String payee,
                                         int amount,
                                         Signatory who )
        {
            try
            {
                return new Check( payee, amount, who );
            }
            catch ( BadCheckException e )
            {
                return null;
            }
        }
}
```

Now, if I have a `CheckingAccount` but choose to label it with a name of type `BankAccount`, I cannot write a check from that account:

```
BankAccount rainyDayFund = new CheckingAccount(...);

rainyDayFund.withdraw( 10000 );
```

works fine, but not

```
rainyDayFund.writeCheck( "Tiffany's", 10000, diamondJim );
```

That is, the only methods available on an expression whose type is `BankAccount` are the `BankAccount` methods. The fact that this is really a `CheckingAccount` is not relevant.

The idea of using superclass types as ways of abstracting the distinctions between a `CheckingAccount` and a `MoneyMarketFund` is an important one. Sometimes subclasses provide extra (or different versions of) functionality. These distinctions are not necessarily relevant to the user of the class, who should be able to treat all `BankAccounts` uniformly.

Note, however, that the true type of an object is evident at the time of its construction; it must be constructed using the class name in a `new` expression. Also, if the type is visible, an explicit cast expression can be used to access subclass properties.[Footnote: For example, `(CheckingAccount) rainyDayFund;`]

Finally, recall the discussion in chapter 10 on the inappropriateness of inheritance unless you are legitimately extending behavior. Inheritance should not be used, for example, when you need to "cancel" superclass properties.

Clever Use of Interfaces

The discussion above of inheritance and encapsulation applies doubly for interfaces. Interfaces are a good way of achieving the subtype properties of inheritance without the requirements of strict extension. Further, an interface type cannot contain implementation, only static final fields and non-static method signatures. This means that an interface cannot divulge any properties of the implementation that might vary from one class to another or that a subclass might override. If it's in the interface, it's in every instance of every class that implements that interface.

The example in the preceding section of a `CheckingAccount` protected by subclassing are even cleaner in the case of the `Cash` and `Check` classes, which are package-local but implement the public interface `Instrument`. This means that things outside the package may hold `Cash` or `Check` objects, but will not know any more than that they hold an `Instrument`. Any methods defined by `Cash` or `Check` but not by `Instrument` are inaccessible except inside the package `finance`.

Like a superclass, the protections of an interface can be circumvented if the implementing class type is visible to the invoking code. And, as always, the true type of an object is known when you invoke its constructor.

These issues are covered further in chapters 4 and 8, on Interfaces and Designing With Objects.

Inner Classes

The final topic in this chapter is inner classes. Inner classes allow a variety of different kinds of encapsulation. At base, an inner class is a remarkably simple idea: An inner class is a class defined inside another. There are several varieties of inner classes, and some of their behavior may seem odd.

Because an inner class is defined inside another class, it may be protected by making it invisible from the outside, for example by making it private. This makes inner classes particularly good places to hide implementation. The actual types of private inner classes are invisible outside of their containing objects, making the inheritance and interface tricks of the previous section more powerful.

Conversely, inner classes can also be used to protect their containing objects. An inner class lives inside another object and has privileged access to the state of this "outer" object. For this reason, inner classes can be used to provide access to their containing objects without revealing these outer objects in their entirety. That is, an inner class's instance(s) can (perversely) be used to limit access to its containing class.

Beware: Although an inner class is defined inside the text of another class, there is no particular subtype

relationship established between the inner and outer classes. For example, an inner class normally does not extend its containing (outer) class.

Static Classes

A static inner class is declared at top level inside another class. It is also declared with the keyword `static`. Static inner classes are largely a convenience for defining multiple classes in one place. A static class declaration is a static member of the class in which it is declared, i.e., it is similar to a static field or static method declaration.

Understanding static inner classes is quite straightforward. There are only a few real differences between a static inner class and a regular class. First, the static inner class does not need to be declared in its own text file, even if it is public. In contrast, an ordinary public class must be declared in a file whose name matches the name of the class. Second, the static inner class has access to the static members of its containing class. This includes any private static methods or private static fields that the class may have.

The proper name of a static inner class is *`OuterClassName.InnerClassName`*.

Beware: This naming convention looks like package syntax (or field access syntax), but it is not.

The constructor for a static class is accessed using the class name, i.e.,

```
new OuterClassName.InnerClassName()
```

perhaps with arguments as with any constructor.

Member Classes

A member class is defined at top level inside another class, but without the keyword static. A member class declaration is a non-static member of the class in which it is declared, i.e., it is similar to a non-static field or method declaration. This means that there is exactly one inner class (type) corresponding to each instance of the outer class. If there are no instances of the outer class, there are in effect no inner class types. When an outer instance is created, a corresponding inner class (i.e., factory) is created and may be instantiated. Note that this does not necessarily make any inner class instances; it just creates the factory object. The inner class and all of its instances have privileged access to the state of the corresponding outer class instance. That is, they can access members, including private members.

An example may make this clearer. Suppose that we want to have a Check class corresponding to each CheckingAccount. The Check class that corresponds to my CheckingAccount is similar to the Check class that corresponds to your CheckingAccount, but with a few differences. Specifically, my Check class (and any Check instances I create) should have privileged access to my CheckingAccount, while your Check class should have privileged access to *your* CheckingAccount. So, in effect, the Check class corresponding to my CheckingAccount is different from the Check class corresponding to your CheckingAccount. It differs precisely in the details of the particular CheckingAccount to which it has privileged access. Creating a third CheckingAccount -- say, Bill Gates's CheckingAccount -- should cause a new kind of Check, Bill Gates's Checks, to come into existence. These Checks differ from yours and mine. Note that creating Bill Gates's CheckingAccount also creates Bill Gates's Check type, but doesn't necessarily create any of Bill Gates's Check instances. Bill still has to write those....

```
class CheckingAccount extends BankAccount
{
     ...

        protected class Check implements Instrument
        {
                private BankAccount originator = CheckingAccount.this;

                private String payee;
                private int amount;

                private boolean valid;

          ....
```

```
                    protected Check( String payee, int amount, Signatory who )
                    {

                            if ( ! who.equals( CheckingAccount.this.owner ) )
                            {
                                   throw new BadCheckException( this );
                            }
                       this.validate( Signatory );

                            this.payee = payee;
                            this.amount = amount;
                            this.valid = true;
                    }

                    Instrument cash() throws BadCheckException
                    {
                            if ( ! this.valid )
                            {
                                    throw new BadCheckException( this );
                            }
                            Instrument out = this.originator.withdraw( this.amount );
                            this.nullify();
                            return out;
                    }
          }

     }
```

In this case, there is in effect one Check class for each CheckingAccount. This is precisely what you'd want: each CheckingAccount has a slightly different kind of Check, varying by who is allowed to sign it, etc.

The proper name of a member class is *`instanceName`*`.`*`InnerClassName`*, where *`instanceName`* is any expression referring to the containing instance. So a way to name Bill's check type is `gatesAccount.Check` (assuming `gatesAccount` is Bill's CheckingAccount), and he can write a new Check using

```
new gatesAccount.Check( worthyCharity, 1000000, billSignature )
```

Note that he can't just say `new Check(...)`, because that leaves ambiguous whether he's writing a check from his account or from mine.

There is a special syntax that may be used inside the inner class to refer to the containing (outer class) instance: *`OuterClassName`*`.this`. For example, in the Check constructor code above, a particular Check's Signatory is compared against the owner of the containing CheckingAccount by comparing it with the owner of the containing CheckingAccount instance. This ensures that I can't sign a Bill Gates Check, nor he one of mine. It is accomplished by looking at `CheckingAccount.this`'s owner field. Note the use of the `CheckingAccount.this` syntax to get at the particular CheckingAccount whose Check class is being defined.

The Check serves as a safely limited access point into the CheckingAccount. For example, each Check knows its CheckingAccount's owner. When a new Check is being created, the Check's Signatory is compared against the account owner (CheckingAccount.this.owner, a field access expression) to make sure that this person is an authorized signer. The identity of the allowable Signatory of the check is hidden, but it is fully encapsulated inside the Check itself. Anyone can get hold of the Check without being able to get hold of the Signatory (or BankAccount balance) inside.

Local Classes and Anonymous Classes

There are two additional kinds of inner classes, local classes and anonymous classes. They are briefly explained here but their intricacies are beyond the scope of this chapter.

A local class declaration is a statement, not a member. A local class may be defined inside any block, e.g., in a method or constructor. There is in effect exactly one local class for each execution of the block. For example, if a local class is defined at the beginning of a method body, there is one local class type corresponding to each invocation of the method, i.e., the class depends on the invocation state of the method itself.

The syntax of a local class method is much like member class declaration, but the name of a local class may only be used within its containing block. A local class's name has the same visibility rules as any local name, i.e., its scope

persists from its declaration until the end of the enclosing block. You may only invoke a local class's constructor with a `new` expression within this scope. You may return these instances from the method or otherwise use these instances elsewhere, but their correct type will not be visible elsewhere. Instead, you must refer to them using a superclass or interface type.

A local class has privileged access to the state of its containing block as well as to the state of its containing object (class or instance). The local class may access the parameters of its containing method, as well as any local variables in whose scope it appears, provided that they are declared final. If a local class is defined in a nonstatic member (method or constructor), the local class's code may access its containing instance using the *`OuterClassName`*`.this` syntax. If a local class is defined in a static member (e.g., in a static method), the local class has only a containing class, not a containing instance.

An anonymous class declaration is always a part of an anonymous class instantiation expression. Anonymous classes may be defined and instantiated anywhere where an instantiation expression might occur. They have a special, very strange syntax. An anonymous class is only good for making a single instance as an anonymous class declaration cannot be separated from its instantiation. Anonymous classes are a nice match for the event handling approaches of the Event Delegation chapter.

The syntax for an anonymous class declaration-and-instantiation expression is

```
new TypeName () { memberDeclarations }
```

where *`TypeName`* is any visible class or interface name and *memberDeclarations* are non-static field and method declarations (but not constructors).[Footnote: If there is necessary instance-specific initialization of an anonymous class, this may be accomplished with an instance initializer expression. Such an expression is a block that appears at top level within the class and is executed at instance construction time.] If *`TypeName`* is a class, the anonymous class extends it; if *`TypeName`* is an interface, the anonymous class implements it. In either case, *memberDeclarations* must include any method declarations required to make an instantiable (sub-)class. The evaluation rules for this expression create a single instance of this new -- and strictly nameless -- class type. Like a local class, the anonymous class's code may access any final parameters or local variables within whose scope it appears, and may use *`OuterClassName`*`.this` to refer to its containing instance if its declaration/construction expression appears within a non-static member.

Inner Classes

	Static Inner	**Member**	**Local**	**Anonymous**
Type Name	*`OuterClass`*`.`*`InnerClass`*	*`outerInstance`*`.`*`InnerClass`*	*`InnerClass`*, but name is accessible only within containing block.	none
Type Name Accessibility	like static member (public, protected, private, etc.)	like member (public, protected, private, etc.)	like local variable name, i.e., only within block	invisible
Class is contained within	(outer) class	instance of (outer) class	block	expression
Access to static members of containing class?	yes	yes	yes	yes
Access to containing instance (including its fields and methods)?	no	yes, using `OuterClass.this`	yes, using `OuterClass.this`	yes, using `OuterClass.this`
Access to parameters and local variables of containing block?	no	no	yes, if they are declared `final`	yes, if they are declared `final`
	`visibility` `static class` *`ClassName`* `{`	*`visibility`* `class` *`ClassName`* `{`	`class` *`ClassName`* `{`	only possible in

Declaration syntax	`members }`	`members }`	`members }`	instantiation (see below).
Where declared?	at top level in *OuterClass*	at top level in *OuterClass*	as statement inside a block (including method, constructor)	in anonymous class instantiation expression
Instantiation syntax	`new OuterClass . InnerClass (...)`	`new outerInstanceExpr . InnerClass (...)`	`new InnerClass (...)`	`new SuperTypeName() { members }`

Chapter Summary

- An abstraction relies only on general properties, leaving implementation details to vary.
- Encapsulation packages up and hides those details.
- Procedural abstraction uses methods to accomplish abstraction and encapsulation.
- A method should be short, have a succinctly summarizable function, and not contain code that is redundant with other methods.
- Abstraction and encapsulation enhance the readability, comprehensibility, modifiability, and maintainability of code.
- Packages provide grouping among interrelated classes.
- The full name of a class or interface is prefaced by its package name.
  - Import statements allow you to circumvent this longer name.
  - Some other short names are automatically available, even without an import statement.
- Visibility modifiers limit access to class members, including inner classes. Together with the use of superclass or interface type names, they provide a way to limit access to an object.
- Inner classes are a mechanism for defining one class inside another.
  - This can be used to hide the inner class.
  - This can also be used to limit access to the outer class by distributing the inner class instead.

Exercises

© 2003 Lynn Andrea Stein

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>