Module 8

Packages and Interfaces

CRITICAL SKILLS

This module examines two of Java's most innovative features: packages and interfaces. *Packages* are groups of related classes. Packages help organize your code and provide another layer of encapsulation. An *interface* defines a set of methods that will be implemented by a class. An interface does not, itself, implement any method. It is a purely logical construct. Packages and interfaces give you greater control over the organization of your program.

CRITICAL SKILL 8.1

Packages

In programming, it is often helpful to group related pieces of a program together. In Java, this is accomplished by using a package. A package serves two purposes. First, it provides a mechanism by which related pieces of a program can be organized as a unit. Classes defined within a package must be accessed through their package name. Thus, a package provides a way to name a collection of classes. Second, a package participates in Java's access control mechanism. Classes defined within a package can be made private to that package and not accessible by code outside the package. Thus, the package provides a means by which classes can be encapsulated. Let's examine each feature a bit more closely.

In general, when you name a class, you are allocating a name from the *namespace*. A namespace defines a declarative region. In Java, no two classes can use the same name from the same namespace. Thus, within a given namespace, each class name must be unique. The examples shown in the preceding modules have all used the default or global namespace. While this is fine for short sample programs, it becomes a problem as programs grow and the default namespace becomes crowded. In large programs, finding unique names for each class can be difficult. Furthermore, you must avoid name collisions with code created by other programmers working on the same project, and with Java's library. The solution to these problems is the package because it gives you a way to partition the namespace. When a class is defined within a package, the name of that package is attached to each class, thus avoiding name collisions with other classes that have the same name, but are in other packages.

Since a package usually contains related classes, Java defines special access rights to code within a package. In a package, you can define code that is accessible by other code within the same package but not by code outside the package. This enables you to create self-contained groups of related classes that keep their operation private.

Defining a Package

All classes in Java belong to some package. When no **package** statement is specified, the default (or global) package is used. Furthermore, the default package has no name, which makes the default package transparent. This is why you haven't had to worry about packages

before now. While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define one or more packages for your code.

To create a package, put a **package** command at the top of a Java source file. The classes declared within that file will then belong to the specified package. Since a package defines a namespace, the names of the classes that you put into the file become part of that package's namespace.

This is the general form of the **package** statement:

package *pkg*;

Here, *pkg* is the name of the package. For example, the following statement creates a package called **Project1**.

```
package Project1;
```

Java uses the file system to manage packages, with each package stored in its own directory. For example, the **.class** files for any classes you declare to be part of **Project1** must be stored in a directory called **Project1**.

Like the rest of Java, package names are case sensitive. This means that the directory in which a package is stored must be precisely the same as the package name. If you have trouble trying the examples in this module, remember to check your package and directory names carefully.

More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files.

You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

package *pack1.pack2.pack3...packN*;

Of course, you must create directories that support the package hierarchy that you create. For example,

```
package X.Y.Z;
```

must be stored in .../X/Y/Z, where ... specifies the path to the specified directories.

Finding Packages and CLASSPATH

As just explained, packages are mirrored by directories. This raises an important question: How does the Java run-time system know where to look for packages that you create? The answer has two parts. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your class files are in the current directory, or a subdirectory of the current directory, they will be found. Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable.

For example, consider the following package specification.

```
package MyPack;
```

In order for a program to find **MyPack**, one of two things must be true. Either the program is executed from a directory immediately above **MyPack**, or **CLASSPATH** must be set to include the path to **MyPack**. The first alternative is the easiest (and doesn't require a change to **CLASSPATH**), but the second alternative lets your program find **MyPack** no matter what directory the program is in. Ultimately, the choice is yours.

The easiest way to try the examples shown in this book is to simply create the package directories below your current development directory, put the **.class** files into the appropriate directories and then execute the programs from the development directory. This is the approach assumed by the examples.

One last point: To avoid confusion, it is best to keep all **.java** and **.class** files associated with packages in their own package directories.

The precise effect and setting of **CLASSPATH** has changed over time, with each revision of Java. It is best to check Sun's Web site **java.sun.com** for the latest information.

A Short Package Example

Keeping the preceding discussion in mind, try this short package example. It creates a simple book database that is contained within a package called **BookPack**.

```
// A short package demonstration.
package BookPack;
```
← This file is part of the **BookPack** package.

```
class Book {
```
← Thus, **Book** is part of **BookPack**.

```
  private String title;
  private String author;
  private int pubDate;

  Book(String t, String a, int d) {
```

```
    title = t;
    author = a;
    pubDate = d;
  }

  void show() {
    System.out.println(title);
    System.out.println(author);
    System.out.println(pubDate);
    System.out.println();
  }
}

class BookDemo {
  public static void main(String args[]) {
    Book books[] = new Book[5];

    books[0] = new Book("Java: A Beginner's Guide",
                        "Schildt", 2005);
    books[1] = new Book("Java: The Complete Reference",
                        "Schildt", 2005);
    books[2] = new Book("The Art of Java",
                        "Schildt and Holmes", 2003);
    books[3] = new Book("Red Storm Rising",
                        "Clancy", 1986);
    books[4] = new Book("On the Road",
                        "Kerouac", 1955);

    for(int i=0; i < books.length; i++) books[i].show();
  }
}
```

BookDemo is also part of **BookPack**.

Call this file **BookDemo.java** and put it in a directory called **BookPack**.

Next, compile the file. Make sure that the resulting **.class** file is also in the **BookPack** directory. Then try executing the class, using the following command line:

```
java BookPack.BookDemo
```

Remember, you will need to be in the directory above **BookPack** when you execute this command or have your **CLASSPATH** environmental variable set appropriately.

As explained, **BookDemo** and **Book** are now part of the package **BookPack**. This means that **BookDemo** cannot be executed by itself. That is, you cannot use this command line:

```
java BookDemo
```

Instead, **BookDemo** must be qualified with its package name.

Progress Check

1. What is a package?
2. Show how to declare a package called **ToolPack**.
3. What is **CLASSPATH**?

CRITICAL SKILL 8.2

Packages and Member Access

The preceding modules have introduced the fundamentals of access control, including the **private** and **public** specifiers, but they have not told the entire story. The reason for this is that packages also participate in Java's access control mechanism, and a complete discussion had to wait until packages were covered.

The visibility of an element is determined by its access specification—**private**, **public**, **protected**, or default—and the package in which it resides. Thus, the visibility of an element is determined by its visibility within a class and its visibility within a package. This multilayered approach to access control supports a rich assortment of access privileges. Table 8-1 summarizes the various access levels. Let's examine each access option individually.

If a member of a class has no explicit access specifier, then it is visible within its package but not outside its package. Therefore, you will use the default access specification for elements that you want to keep private to a package but public within that package.

Members explicitly declared **public** are visible everywhere, including different classes and different packages. There is no restriction on their use or access.

A **private** member is accessible only to the other members of its class. A **private** member is unaffected by its membership in a package.

A member specified as **protected** is accessible within its package and to all subclasses, including subclasses in other packages.

Table 8-1 applies only to members of classes. A class has only two possible access levels: default and public. When a class is declared as **public**, it is accessible by any other code. If a class has default access, it can be accessed only by other code within its same package. Also, a class that is declared **public** must reside in a file by the same name.

---

1. A package is a container for classes. It performs both an organization and an encapsulation role.
2. `package ToolPack;`
3. **CLASSPATH** is the environmental variable that specifies the path to classes.

	Private Member	Default Member	Protected Member	Public Member
Visible within same class	Yes	Yes	Yes	Yes
Visible within same package by subclass	No	Yes	Yes	Yes
Visible within same package by non-subclass	No	Yes	Yes	Yes
Visible within different package by subclass	No	No	Yes	Yes
Visible within different package by non-subclass	No	No	No	Yes

Table 8-1 Class Member Access

Progress Check

1. If a class member has default access inside a package, is that member accessible by other packages?
2. What does **protected** do?
3. A **private** member can be accessed by subclasses within its packages. True or False?

A Package Access Example

In the **package** example shown earlier, both **Book** and **BookDemo** were in the same package, so there was no problem with **BookDemo** using **Book** because the default access privilege grants all members of the same package access. However, if **Book** were in one package and

1. No.
2. It allows a member to be accessible by other code in its package and by all subclasses, no matter what package the subclass is in.
3. False.

BookDemo were in another, the situation would be different. In this case, access to **Book** would be denied. To make **Book** available to other packages, you must make three changes. First, **Book** needs to be declared **public**. This makes **Book** visible outside of **BookPack**. Second, its constructor must be made **public**, and finally its **show( )** method needs to be **public**. This allows them to be visible outside of **BookPack**, too. Thus, to make **Book** usable by other packages, it must be recoded as shown here.

```
// Book recoded for public access.
package BookPack;

public class Book {
  private String title;
  private String author;
  private int pubDate;

  // Now public.
  public Book(String t, String a, int d) {
    title = t;
    author = a;
    pubDate = d;
  }

  // Now public.
  public void show() {
    System.out.println(title);
    System.out.println(author);
    System.out.println(pubDate);
    System.out.println();
  }
}
```

Book and its members must be **public** in order to be used by other packages.

To use **Book** from another package, either you must use the **import** statement described in the next section, or you must fully qualify its name to include its full package specification. For example, here is a class called **UseBook**, which is contained in the **BookPackB** package. It fully qualifies **Book** in order to use it.

```
// This class is in package BookPackB.
package BookPackB;

// Use the Book Class from BookPack.
class UseBook {
  public static void main(String args[]) {
    BookPack.Book books[] = new BookPack.Book[5];
```

Qualify **Book** with its package name: **BookPack**.

```
    books[0] = new BookPack.Book("Java: A Beginner's Guide",
                       "Schildt", 2005);
    books[1] = new BookPack.Book("Java: The Complete Reference",
                       "Schildt", 2005);
    books[2] = new BookPack.Book("The Art of Java",
                       "Schildt and Holmes", 2003);
    books[3] = new BookPack.Book("Red Storm Rising",
                       "Clancy", 1986);
    books[4] = new BookPack.Book("On the Road",
                       "Kerouac", 1955);

    for(int i=0; i < books.length; i++) books[i].show();
  }
}
```

Notice how every use of **Book** is preceded with the **BookPack** qualifier. Without this specification, **Book** would not be found when you tried to compile **UseBook**.

CRITICAL SKILL 8.3

Understanding Protected Members

Newcomers to Java are sometimes confused by the meaning and use of **protected**. As explained, the **protected** specifier creates a member that is accessible within its package and to subclasses in other packages. Thus, a **protected** member is available for all subclasses to use but is still protected from arbitrary access by code outside its package.

To better understand the effects of **protected**, let's work through an example. First, change the **Book** class so that its instance variables are **protected**, as shown here.

```
// Make the instance variables in Book protected.
package BookPack;

public class Book {
  // these are now protected
  protected String title;
  protected String author;
  protected int pubDate;

  public Book(String t, String a, int d) {
    title = t;
    author = a;
    pubDate = d;
  }
```

These are now **protected**.

```
   public void show() {
    System.out.println(title);
    System.out.println(author);
    System.out.println(pubDate);
    System.out.println();
  }
}
```

Next, create a subclass of **Book**, called **ExtBook**, and a class called **ProtectDemo** that uses **ExtBook**. **ExtBook** adds a field that stores the name of the publisher and several accessor methods. Both of these classes will be in their own package called **BookPackB**. They are shown here.

```
// Demonstrate Protected.
package BookPackB;

class ExtBook extends BookPack.Book {
  private String publisher;

  public ExtBook(String t, String a, int d, String p) {
    super(t, a, d);
    publisher = p;
  }

  public void show() {
    super.show();
    System.out.println(publisher);
    System.out.println();
  }

  public String getPublisher() { return publisher; }
  public void setPublisher(String p) { publisher = p; }

  /* These are OK because subclass can access
     a protected member. */
  public String getTitle() { return title; }
  public void setTitle(String t) { title = t; }
  public String getAuthor() { return author; }
  public void setAuthor(String a) { author = a; }
  public int getPubDate() { return pubDate; }
  public void setPubDate(int d) { pubDate = d; }
}
```

Access to **Book**'s members is allowed for subclasses.

```
class ProtectDemo {
  public static void main(String args[]) {
    ExtBook books[] = new ExtBook[5];

    books[0] = new ExtBook("Java: A Beginner's Guide",
                       "Schildt", 2005, "Osborne/McGraw-Hill");
    books[1] = new ExtBook("Java: The Complete Reference",
                       "Schildt", 2005, "Osborne/McGraw-Hill");
    books[2] = new ExtBook("The Art of Java",
                       "Schildt and Holmes", 2003,
                       "Osborne/McGraw-Hill");
    books[3] = new ExtBook("Red Storm Rising",
                       "Clancy", 1986, "Putnam");
    books[4] = new ExtBook("On the Road",
                       "Kerouac", 1955, "Viking");

    for(int i=0; i < books.length; i++) books[i].show();

    // Find books by author
    System.out.println("Showing all books by Schildt.");
    for(int i=0; i < books.length; i++)
      if(books[i].getAuthor() == "Schildt")
        System.out.println(books[i].getTitle());

//    books[0].title = "test title"; // Error - not accessible
  }
}
```

Access to **protected** field not allowed by non-subclass.

Look first at the code inside **ExtBook**. Because **ExtBook** extends **Book**, it has access to the **protected** members of **Book** even though **ExtBook** is in a different package. Thus, it can access **title**, **author**, and **pubDate** directly, as it does in the accessor methods it creates for those variables. However, in **ProtectDemo**, access to these variables is denied because **ProtectDemo** is not a subclass of **Book**. For example, if you remove the comment symbol from the following line, the program will not compile.

```
//    books[0].title = "test title"; // Error - not accessible
```

CRITICAL SKILL 8.4

Importing Packages

When you use a class from another package, you can fully qualify the name of the class with the name of its package, as the preceding examples have done. However, such an approach could easily become tiresome and awkward, especially if the classes you are qualifying are

Ask the Expert

Q: **I know that C++ also includes an access specifier called** protected**. Is it similar to Java's?**

A: Similar, but not the same. In C++, **protected** creates a member that can be accessed by subclasses but is otherwise private. In Java, **protected** creates a member that can be accessed by any code within its package but only by subclasses outside of its package. You need to be careful of this difference when porting code between C++ and Java.

deeply nested in a package hierarchy. Since Java was invented by programmers for programmers—and programmers don't like tedious constructs—it should come as no surprise that a more convenient method exists for using the contents of packages: the **import** statement. Using **import** you can bring one or more members of a package into view. This allows you to use those members directly, without explicit package qualification.

Here is the general form of the **import** statement:

import *pkg.classname*;

Here, *pkg* is the name of the package, which can include its full path, and *classname* is the name of the class being imported. If you want to import the entire contents of a package, use an asterisk (*) for the class name. Here are examples of both forms:

```
import MyPack.MyClass
import MyPack.*;
```

In the first case, the **MyClass** class is imported from **MyPack**. In the second, all of the classes in **MyPack** are imported. In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions.

You can use **import** to bring the **BookPack** package into view so that the **Book** class can be used without qualification. To do so, simply add this **import** statement to the top of any file that uses **Book**.

```
import BookPack.*;
```

For example, here is the **UseBook** class recoded to use **import**.

```
// Demonstrate import.
package BookPackB;
import BookPack.*; <-- Import BookPack.

// Use the Book Class from BookPack.
class UseBook {
  public static void main(String args[]) {
    Book books[] = new Book[5]; <-- Now, you can refer to Book directly, without qualification.

    books[0] = new Book("Java: A Beginner's Guide",
                        "Schildt", 2005);
    books[1] = new Book("Java: The Complete Reference",
                        "Schildt", 2005);
    books[2] = new Book("The Art of Java",
                        "Schildt and Holmes", 2003);
    books[3] = new Book("Red Storm Rising",
                        "Clancy", 1986);
    books[4] = new Book("On the Road",
                        "Kerouac", 1955);

    for(int i=0; i < books.length; i++) books[i].show();
  }
}
```

Notice that you no longer need to qualify **Book** with its package name.

Ask the Expert

Q: **Does importing a package have an impact on the performance of my program?**

A: Yes and no! Importing a package can create a small amount of overhead during compilation, but it has no impact on performance at run time.

CRITICAL SKILL 8.5

Java's Class Library Is Contained in Packages

As explained earlier in this book, Java defines a large number of standard classes that are available to all programs. This class library is often referred to as the Java API (Application Programming Interface). The Java API is stored in packages. At the top of the package hierarchy is **java**. Descending from **java** are several subpackages, including these:

Subpackage	Description
java.lang	Contains a large number of general-purpose classes
java.io	Contains the I/O classes
java.net	Contains those classes that support networking
java.applet	Contains classes for creating applets
java.awt	Contains classes that support the Abstract Window Toolkit

Since the beginning of this book, you have been using **java.lang**. It contains, among several others, the **System** class, which you have been using when performing output using **println( )**. The **java.lang** package is unique because it is imported automatically into every Java program. This is why you did not have to import **java.lang** in the preceding sample programs. However, you must explicitly import the other packages. We will be examining several packages in subsequent modules.

Progress Check

1. How do you include another package in a source file?
2. Show how to include all of the classes in a package called **ToolPack**.
3. Do you need to include **java.lang** explicitly?

1. Use the **import** statement.
2. `import ToolPack.*;`
3. No.

CRITICAL SKILL 8.6

Interfaces

In object-oriented programming, it is sometimes helpful to define what a class must do but not how it will do it. You have already seen an example of this: the abstract method. An abstract method defines the signature for a method but provides no implementation. A subclass must provide its own implementation of each abstract method defined by its superclass. Thus, an abstract method specifies the *interface* to the method but not the *implementation*. While abstract classes and methods are useful, it is possible to take this concept a step further. In Java, you can fully separate a class's interface from its implementation by using the keyword **interface**.

Interfaces are syntactically similar to abstract classes. However, in an interface, no method can include a body. That is, an interface provides no implementation whatsoever. It specifies what must be done, but not how. Once an interface is defined, any number of classes can implement it. Also, one class can implement any number of interfaces.

To implement an interface, a class must provide bodies (implementations) for the methods described by the interface. Each class is free to determine the details of its own implementation. Thus, two classes might implement the same interface in different ways, but each class still supports the same set of methods. Thus, code that has knowledge of the interface can use objects of either class since the interface to those objects is the same. By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

Here is the general form of an interface:

```
access interface name {
  ret-type method-name1(param-list);
  ret-type method-name2(param-list);
  type var1 = value;
  type var2 = value;
  // ...
  ret-type method-nameN(param-list);
  type varN = value;
}
```

Here, *access* is either **public** or not used. When no access specifier is included, then default access results, and the interface is available only to other members of its package. When it is declared as **public**, the interface can be used by any other code. (When an **interface** is declared **public**, it must be in a file of the same name.) *name* is the name of the interface and can be any valid identifier.

Methods are declared using only their return type and signature. They are, essentially, abstract methods. As explained, in an **interface**, no method can have an implementation.

Thus, each class that includes an **interface** must implement all of the methods. In an interface, methods are implicitly **public**.

Variables declared in an **interface** are not instance variables. Instead, they are implicitly **public**, **final**, and **static** and must be initialized. Thus, they are essentially constants.

Here is an example of an **interface** definition. It specifies the interface to a class that generates a series of numbers.

```
public interface Series {
  int getNext(); // return next number in series
  void reset(); // restart
  void setStart(int x); // set starting value
}
```

This interface is declared **public** so that it can be implemented by code in any package.

CRITICAL SKILL 8.7

Implementing Interfaces

Once an **interface** has been defined, one or more classes can implement that interface. To implement an interface, include the **implements** clause in a class definition and then create the methods defined by the interface. The general form of a class that includes the **implements** clause looks like this:

access class *classname* extends *superclass* implements *interface* {
  // class-body
}

Here, *access* is either **public** or not used. The **extends** clause is, of course, optional. To implement more than one interface, the interfaces are separated with a comma.

The methods that implement an interface must be declared **public**. Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

Here is an example that implements the **Series** interface shown earlier. It creates a class called **ByTwos**, which generates a series of numbers, each two greater than the previous one.

```
// Implement Series.
class ByTwos implements Series {
  int start;
  int val;

  ByTwos() {
    start = 0;
```

Implement the **Series** interface.

```
    val = 0;
  }

  public int getNext() {
    val += 2;
    return val;
  }

  public void reset() {
    start = 0;
    val = 0;
  }

  public void setStart(int x) {
    start = x;
    val = x;
  }
}
```

Notice that the methods **getNext( )**, **reset( )**, and **setStart( )** are declared using the **public** access specifier. This is necessary. Whenever you implement a method defined by an interface, it must be implemented as **public** because all members of an interface are implicitly **public**.

Here is a class that demonstrates **ByTwos**.

```
class SeriesDemo {
  public static void main(String args[]) {
    ByTwos ob = new ByTwos();

    for(int i=0; i < 5; i++)
      System.out.println("Next value is " +
                         ob.getNext());

    System.out.println("\nResetting");
    ob.reset();
    for(int i=0; i < 5; i++)
      System.out.println("Next value is " +
                         ob.getNext());

    System.out.println("\nStarting at 100");
    ob.setStart(100);
    for(int i=0; i < 5; i++)
      System.out.println("Next value is " +
                         ob.getNext());
  }
}
```

The output from this program is shown here.

```
Next value is 2
Next value is 4
Next value is 6
Next value is 8
Next value is 10

Resetting
Next value is 2
Next value is 4
Next value is 6
Next value is 8
Next value is 10

Starting at 100
Next value is 102
Next value is 104
Next value is 106
Next value is 108
Next value is 110
```

It is both permissible and common for classes that implement interfaces to define additional members of their own. For example, the following version of **ByTwos** adds the method **getPrevious( )**, which returns the previous value.

```
// Implement Series and add getPrevious().
class ByTwos implements Series {
  int start;
  int val;
  int prev;

  ByTwos() {
    start = 0;
    val = 0;
    prev = -2;
  }

  public int getNext() {
    prev = val;
    val += 2;
    return val;
  }

  public void reset() {
    start = 0;
```

```
    val = 0;
    prev = -2;
  }

  public void setStart(int x) {
    start = x;
    val = x;
    prev = x - 2;
  }

  int getPrevious() {  <-- Add a method not defined by Series.
    return prev;
  }
}
```

Notice that the addition of **getPrevious( )** required a change to the implementations of the methods defined by **Series**. However, since the interface to those methods stays the same, the change is seamless and does not break preexisting code. This is one of the advantages of interfaces.

As explained, any number of classes can implement an **interface**. For example, here is a class called **ByThrees** that generates a series that consists of multiples of three.

```
// Implement Series.
class ByThrees implements Series {  <-- Implement Series a different way.
  int start;
  int val;

  ByThrees() {
    start = 0;
    val = 0;
  }

  public int getNext() {
    val += 3;
    return val;
  }

  public void reset() {
    start = 0;
    val = 0;
  }

  public void setStart(int x) {
    start = x;
    val = x;
  }
}
```

One more point: If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as **abstract**. No objects of such a class can be created, but it can be used as an abstract superclass, allowing subclasses to provide the complete implementation.

CRITICAL SKILL 8.8

Using Interface References

You might be somewhat surprised to learn that you can declare a reference variable of an interface type. In other words, you can create an interface reference variable. Such a variable can refer to any object that implements its interface. When you call a method on an object through an interface reference, it is the version of the method implemented by the object that is executed. This process is similar to using a superclass reference to access a subclass object, as described in Module 7.

The following example illustrates this process. It uses the same interface reference variable to call methods on objects of both **ByTwos** and **ByThrees**.

```
// Demonstrate interface references.

class ByTwos implements Series {
  int start;
  int val;

  ByTwos() {
    start = 0;
    val = 0;
  }

  public int getNext() {
    val += 2;
    return val;
  }

  public void reset() {
    start = 0;
    val = 0;
  }

  public void setStart(int x) {
    start = x;
    val = x;
  }
}
```

```
class ByThrees implements Series {
  int start;
  int val;

  ByThrees() {
    start = 0;
    val = 0;
  }

  public int getNext() {
    val += 3;
    return val;
  }

  public void reset() {
    start = 0;
    val = 0;
  }

  public void setStart(int x) {
    start = x;
    val = x;
  }
}

class SeriesDemo2 {
  public static void main(String args[]) {
    ByTwos twoOb = new ByTwos();
    ByThrees threeOb = new ByThrees();
    Series ob;

    for(int i=0; i < 5; i++) {
      ob = twoOb;
      System.out.println("Next ByTwos value is " +
                         ob.getNext());
      ob = threeOb;
      System.out.println("Next ByThrees value is " +
                         ob.getNext());
    }
  }
}
```

Access an object via an interface reference.

In **main( )**, **ob** is declared to be a reference to a **Series** interface. This means that it can be used to store references to any object that implements **Series**. In this case, it is used to refer to **twoOb** and **threeOb**, which are objects of type **ByTwos** and **ByThrees**, respectively,

which both implement **Series**. An interface reference variable has knowledge only of the methods declared by its **interface** declaration. Thus, **ob** could not be used to access any other variables or methods that might be supported by the object.

Progress Check

1. What is an interface? What keyword is used to define one?
2. What is **implements** for?
3. Can an interface reference variable refer to an object that implements that interface?

Project 8-1 Creating a Queue Interface

`ICharQ.java`
`IQDemo.java`

To see the power of interfaces in action, we will look at a practical example. In earlier modules, you developed a class called **Queue** that implemented a simple fixed-size queue for characters. However, there are many ways to implement a queue. For example, the queue can be of a fixed size or it can be "growable." The queue can be linear, in which case it can be used up, or it can be circular, in which case elements can be put in as long as elements are being taken off. The queue can also be held in an array, a linked list, a binary tree, and so on. No matter how the queue is implemented, the interface to the queue remains the same, and the methods **put( )** and **get( )** define the interface to the queue independently of the details of the implementation. Because the interface to a queue is separate from its implementation, it is easy to define a queue interface, leaving it to each implementation to define the specifics.

In this project, you will create an interface for a character queue and three implementations. All three implementations will use an array to store the characters. One queue will be the fixed-size, linear queue developed earlier. Another will be a circular queue. In a circular queue, when the end of the underlying array is encountered, the get and put indices automatically loop back to the start. Thus, any number of items can be stored in a circular queue as long as items are also being taken out. The final implementation creates a dynamic queue, which grows as necessary when its size is exceeded.

---

1. An interface defines the methods that a class must implement but defines no implementation of its own. It is defined by the keyword **interface**.
2. To implement an interface, include that interface in a class by using the **implements** keyword.
3. Yes.

Step by Step

1. Create a file called **ICharQ.java** and put into that file the following interface definition.

```
// A character queue interface.
public interface ICharQ {
  // Put a character into the queue.
  void put(char ch);

  // Get a character from the queue.
  char get();
}
```

   As you can see, this interface is very simple, consisting of only two methods. Each class that implements **ICharQ** will need to implement these methods.

2. Create a file called **IQDemo.java**.

3. Begin creating **IQDemo.java** by adding the **FixedQueue** class shown here:

```
// A fixed-size queue class for characters.
class FixedQueue implements ICharQ {
  private char q[]; // this array holds the queue
  private int putloc, getloc; // the put and get indices

  // Construct an empty queue given its size.
  public FixedQueue(int size) {
    q = new char[size+1]; // allocate memory for queue
    putloc = getloc = 0;
  }

  // Put a character into the queue.
  public void put(char ch) {
    if(putloc==q.length-1) {
      System.out.println(" - Queue is full.");
      return;
    }

    putloc++;
    q[putloc] = ch;
  }

  // Get a character from the queue.
  public char get() {
    if(getloc == putloc) {
      System.out.println(" - Queue is empty.");
      return (char) 0;
```

(continued)

```
    }

    getloc++;
    return q[getloc];
  }
}
```

This implementation of **ICharQ** is adapted from the **Queue** class shown in Module 5 and should already be familiar to you.

4. To **IQDemo.java** add the **CircularQueue** class shown here. It implements a circular queue for characters.

```
// A circular queue.
class CircularQueue implements ICharQ {
  private char q[]; // this array holds the queue
  private int putloc, getloc; // the put and get indices

  // Construct an empty queue given its size.
  public CircularQueue(int size) {
    q = new char[size+1]; // allocate memory for queue
    putloc = getloc = 0;
  }

  // Put a character into the queue.
  public void put(char ch) {
    /* Queue is full if either putloc is one less than
       getloc, or if putloc is at the end of the array
       and getloc is at the beginning. */
    if(putloc+1==getloc |
       ((putloc==q.length-1) & (getloc==0))) {
      System.out.println(" - Queue is full.");
      return;
    }

    putloc++;
    if(putloc==q.length) putloc = 0; // loop back
    q[putloc] = ch;
  }

  // Get a character from the queue.
  public char get() {
    if(getloc == putloc) {
      System.out.println(" - Queue is empty.");
      return (char) 0;
    }

    getloc++;
    if(getloc==q.length) getloc = 0; // loop back
```

```
    return q[getloc];
  }
}
```

The circular queue works by reusing space in the array that is freed when elements are retrieved. Thus, it can store an unlimited number of elements as long as elements are also being removed. While conceptually simple—just reset the appropriate index to zero when the end of the array is reached—the boundary conditions are a bit confusing at first. In a circular queue, the queue is full not when the end of the underlying array is reached, but rather when storing an item would cause an unretrieved item to be overwritten. Thus, **put( )** must check several conditions in order to determine if the queue is full. As the comments suggest, the queue is full when either **putloc** is one less than **getloc**, or if **putloc** is at the end of the array and **getloc** is at the beginning. As before, the queue is empty when **getloc** and **putloc** are equal.

5. Put into **IQDemo.java** the **DynQueue** class shown next. It implements a "growable" queue that expands its size when space is exhausted.

```
// A dynamic queue.
class DynQueue implements ICharQ {
  private char q[]; // this array holds the queue
  private int putloc, getloc; // the put and get indices

  // Construct an empty queue given its size.
  public DynQueue(int size) {
    q = new char[size+1]; // allocate memory for queue
    putloc = getloc = 0;
  }

  // Put a character into the queue.
  public void put(char ch) {
    if(putloc==q.length-1) {
      // increase queue size
      char t[] = new char[q.length * 2];

      // copy elements into new queue
      for(int i=0; i < q.length; i++)
        t[i] = q[i];

      q = t;
    }

    putloc++;
    q[putloc] = ch;
  }

  // Get a character from the queue.
```

(continued)

```
  public char get() {
    if(getloc == putloc) {
      System.out.println(" - Queue is empty.");
      return (char) 0;
    }

    getloc++;
    return q[getloc];
  }
}
```

In this queue implementation, when the queue is full, an attempt to store another element causes a new underlying array to be allocated that is twice as large as the original, the current contents of the queue are copied into this array, and a reference to the new array is stored in **q**.

6. To demonstrate the three **ICharQ** implementations, enter the following class into **IQDemo.java**. It uses an **ICharQ** reference to access all three queues.

```
// Demonstrate the ICharQ interface.
class IQDemo {
  public static void main(String args[]) {
    FixedQueue q1 = new FixedQueue(10);
    DynQueue q2 = new DynQueue(5);
    CircularQueue q3 = new CircularQueue(10);

    ICharQ iQ;

    char ch;
    int i;

    iQ = q1;
    // Put some characters into fixed queue.
    for(i=0; i < 10; i++)
      iQ.put((char) ('A' + i));

    // Show the queue.
    System.out.print("Contents of fixed queue: ");
    for(i=0; i < 10; i++) {
      ch = iQ.get();
      System.out.print(ch);
    }
    System.out.println();

    iQ = q2;
    // Put some characters into dynamic queue.
    for(i=0; i < 10; i++)
      iQ.put((char) ('Z' - i));
```

```
    // Show the queue.
    System.out.print("Contents of dynamic queue: ");
    for(i=0; i < 10; i++) {
      ch = iQ.get();
      System.out.print(ch);
    }

    System.out.println();

    iQ = q3;
    // Put some characters into circular queue.
    for(i=0; i < 10; i++)
      iQ.put((char) ('A' + i));

    // Show the queue.
    System.out.print("Contents of circular queue: ");
    for(i=0; i < 10; i++) {
      ch = iQ.get();
      System.out.print(ch);
    }

    System.out.println();

    // Put more characters into circular queue.
    for(i=10; i < 20; i++)
      iQ.put((char) ('A' + i));

    // Show the queue.
    System.out.print("Contents of circular queue: ");
    for(i=0; i < 10; i++) {
      ch = iQ.get();
      System.out.print(ch);
    }

    System.out.println("\nStore and consume from" +
                       " circular queue.");

    // Use and consume from circular queue.
    for(i=0; i < 20; i++) {
      iQ.put((char) ('A' + i));
      ch = iQ.get();
      System.out.print(ch);
    }

  }
}
```

(continued)

7. The output from this program is shown here.

```
Contents of fixed queue: ABCDEFGHIJ
Contents of dynamic queue: ZYXWVUTSRQ
Contents of circular queue: ABCDEFGHIJ
Contents of circular queue: KLMNOPQRST
Store and consume from circular queue.
ABCDEFGHIJKLMNOPQRST
```

8. Here are some things to try on your own. Create a circular version of **DynQueue**. Add a **reset( )** method to **ICharQ** which resets the queue. Create a **static** method that copies the contents of one type of queue into another.

CRITICAL SKILL 8.9

Variables in Interfaces

As mentioned, variables can be declared in an interface, but they are implicitly **public**, **static**, and **final**. At first glance, you might think that there would be very limited use for such variables, but the opposite is true. Large programs typically make use of several constant values that describe such things as array size, various limits, special values, and the like. Since a large program is typically held in a number of separate source files, there needs to be a convenient way to make these constants available to each file. In Java, interface variables offer a solution.

To define a set of shared constants, simply create an **interface** that contains only these constants, without any methods. Each file that needs access to the constants simply "implements" the interface. This brings the constants into view. Here is a simple example.

```
// An interface that contains constants.
interface IConst {
  int MIN = 0;
  int MAX = 10;
  String ERRORMSG = "Boundary Error";
}

class IConstD implements IConst {
  public static void main(String args[]) {
    int nums[] = new int[MAX];

    for(int i=MIN; i < 11; i++) {
      if(i >= MAX) System.out.println(ERRORMSG);
      else {
        nums[i] = i;
        System.out.print(nums[i] + " ");
      }
    }
  }
}
```

These are constants.

Ask the Expert

Q: **When I convert a C++ program to Java, how do I handle** #define **statements in a C++-style header file?**

A: Java's answer to the header files and **#define**s found in C++ is the interface and interface variables. To port a header file, simply perform a one-to-one translation.

CRITICAL SKILL 8.10

Interfaces Can Be Extended

One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain. Following is an example:

```
// One interface can extend another.
interface A {
  void meth1();
  void meth2();
}

// B now includes meth1() and meth2() - it adds meth3().
interface B extends A {
  void meth3();
}

// This class must implement all of A and B
class MyClass implements B {
  public void meth1() {
    System.out.println("Implement meth1().");
  }

  public void meth2() {
    System.out.println("Implement meth2().");
  }

  public void meth3() {
    System.out.println("Implement meth3().");
```

B inherits A.

```
  }
}

class IFExtend {
  public static void main(String arg[]) {
    MyClass ob = new MyClass();

    ob.meth1();
    ob.meth2();
    ob.meth3();
  }
}
```

As an experiment, you might try removing the implementation for **meth1( )** in **MyClass**. This will cause a compile-time error. As stated earlier, any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.

Although the examples we've included in this book do not make frequent use of packages or interfaces, both of these tools are an important part of the Java programming environment. Virtually all real programs and applets that you write in Java will be contained within packages. A number will probably implement interfaces as well. It is important, therefore, that you be comfortable with their usage.

Module 8 Mastery Check

1. Using the code from Project 8-1, put the **ICharQ** interface and its three implementations into a package called **QPack**. Keeping the queue demonstration class **IQDemo** in the default package, show how to import and use the classes in **QPack**.
2. What is a namespace? Why is it important that Java allows you to partition the namespace?
3. Packages are stored in _____.
4. Explain the difference between **protected** and default access.
5. Explain the two ways that the members of a package can be used by other packages.
6. "One interface, multiple methods" is a key tenet of Java. What feature best exemplifies it?
7. How many classes can implement an interface? How many interfaces can a class implement?
8. Can interfaces be extended?