HERBERT SCHILDT'S *Java: A Beginner's Guide*, 6TH ED.

Chapter 8 Self Test Answers

1. **Using the code from Try This 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*.**

   The following code satisfies the requirements outlined above. It should be noted that *FixedQueue.java*, *iCharQ.java*, *DynQueue.java* and *CircularQueue.java* are all found in a folder (directory) called *qpack*. *IQDemo.java* is in the same directory as the *qpack* directory.

   IQDemo.java:

```
import qpack.*;

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 character into fixed queue
    for(i=0; i < 10; i++){
      iQ.put((char) ('A' + i));
    }

    System.out.println();

    //Show the queue
    System.out.printf("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 in the dynamic queue
    for(i=0; i< 10; i++){
      iQ.put((char) ('Z'-i));
    }

    // Show the queue
    System.out.printf("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 in the circular queue
    for(i=0; i<10; i++){
      iQ.put((char) ('A' + i));
    }

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

```
        System.out.println();

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

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

        System.out.println();

        System.out.printf("Store and consume from circular queue: ");

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

        System.out.println("\n");

      } // end main
    } // end IQDemo
```

iCharQ.java:

```
    package qpack;

    // 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();

    } // end ICharQ
```

FixedQueue.java:

```
    package qpack;

    // A fixed size queue for characters
    public class FixedQueue implements ICharQ{

          private char q[]; // this array holds the queue
          private int putloc, getloc; // then put and get indicies

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

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

            q[putloc++] = ch;
          }

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

            return q[getloc++];
          }
    } // end FixedQueue
```

DynQueue.java:

```
    package qpack;

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

      // Construct and empty queue given its size
      public DynQueue(int size){
        q = new char[size]; // allocate memory for queue
        putloc = getloc = 0;
      }// end constructor

      // Put a character into the queue
      public void put(char ch){
        if(putloc == q.length){
          // 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;
        }

        q[putloc++] = ch;
      }

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

        return q[getloc++];
      }// end get
    }// end DynQueue
```

CircularQueue.java:

```
    package qpack;

    public class CircularQueue implements ICharQ{
      private char q[]; // this array holds the queue
      private int putloc, getloc; // the put and get indicies

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

      // Put a character in 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;
        }
        q[putloc++] = ch;

        if(putloc==q.length){
          putloc = 0; // loop back
        }
      } // end put

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

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

        return ch;

      }// end get

    } // end CircularQueue
```

2. **What is namespace? Why is it important that Java allows you to partition the namespace?**

   In Java, the namespace is a declaraive region where each class name must be unique. Separate namespaces represent separate packages. Packages give Java programmers a way to separate the namespace within their programs and make programming in Java logistically easier. No two classes in one package may have the same name, but since name of the package is attached to the name of the class, it is possible to have classes of the same name in separate packages.

3. **Packages are stored in ________________.**

   Packages are stored in separate folders (directories).

4. **Explain the difference between *protected* and default access.**

   The *protected* access modifier only allows access from classes within the same package. Put simply, if classes share the same package and they declare their variables or methods *protected*, then they have unrestricted access to those variables and methods. If the classes are not in the same package, it is as if the variables and methods were declared *private*, there is no access available. The *protected* access modifier also allows access via subclasses, even if they do not share the same package.

   With *default*, no access modifier is declared. Java treats these variables and methods as public within the package, but as private outside the package. Most Java programmers believe that leaving a variable or method without an explicit access modifier (ie. leaving it as *default*) is poor coding practice and should be avoided.

5. **Explain the two ways that the members of a package can be used by other packages.**

   The first way members of a package can be used by other packages is to fully qualify the name of the member in-line, for example:

   ```
   java.util.Scanner.sampleMethod;
   ```

   Or you can import the package at the top of the class, for example:

   ```
   import java.util.Scanner;
   ```

   Either way is acceptable, but importing at the top of the class will save a significant amount of typing as the objects can simply be referenced without giving the entire name. Further, if there is an error or change to the code, importing packages at the top of the class make make it easier to fix, and may avoid a significant number of tedious changes.

6. **“One interface, multiple methods” is a key tenet of Java. What feature best exemplifies it?**

   The *interface* exemplifies the “One interface, multiple methods” tenet of Java. An *interface* identifies methods that must be implemented when a class implements the interface. However, the interface itself does not present any implementation details for each method. This is convenient to the programmer, because it is clear which methods are being used by any particular class, but the details of implementation are not necessarily needed. This is an aspect of Java that is strongly polymorphic, one of Java’s key tenets.

7. **How many classes can implement an interface? How many interfaces can a class implement?**

   Any number of classes can implement an interface, there is no limit. Also, a class can implement as many interfaces as it likes as well, however in practice this is impractical as each interface that is implemented will require the development of a varying number of methods, meaning the class could end up being incredibly large.

8. **Can interfaces be extended?**

   Yes, interfaces can be extended in the exact same manner as classes inherit from a superclass. For example:

   ```
   interface B extends A{
     // insert code...
   }
   ```

9. **Create an interface for the *Vehicle* class from Chapter 7. Call the interface *IVehicle*.**

   The following code meets the criteria listed above:

   ```
   // A Vehicle interface
   public interface IVehicle{

     // calculate the range of the vehicle
     int range();
     // calculate fuel needed
     double fuelNeeded(int miles);

   } // end IVehicle

   // Vehicle Superclass
   class Vehicle implments IVehicle{
     private int passengers; // number of passengers
     private int fuelCap; // fuel capacity in gallons
     private int mpg; // fuel consumption in miles per gallon

     // Constructor
     Vehicle (int p, int f, int m){
       passengers = p;
       fuelCap = f;
       mpg = m;
     }// end Constructor

     // Return the fuel range
     int range(){
       return mpg * fuelCap;
     }

     // Compute fuel needed for a given distance
     double fuelNeeded(int miles){
       return (double) miles / mpg;
     }

     // Accessor methods for instance variables
     int getPassengers(){
       return passengers;
     }

     void setPassengers(int p){
       passengers = p;
     }

     int getFuelCap(){
       return fuelCap;
     }

     void setFuelCap(int f){
       fuelCap = f;
     }

     int getMpg(){
       return mpg;
     }

     void setMpg(int m){
   ```

```
    mpg = m;
  }

}// end Vehicle

class Truck extends Vehicle {
  private int cargoCap; // cargo capacity in pounds

  // Constructor
  Truck(int p, int f, int m, int c){
    super(p, f, m);
    cargoCap = c;
  }

  // Accessor methods for cargoCap
  int getCargoCap(){
    return cargoCap;
  }

  void setCargoCap(int c){
    cargoCap = c;
  }

}// end Truck

class TruckDemo{
  public static void main(String args[]){
    Truck semi = new Truck(2, 200, 7, 44000);
    Truck pickup = new Truck(3, 28, 15, 2000);

    double gallons;
    int dist = 252;

    gallons = semi.fuelNeeded(dist);

    System.out.println("\nA semi-truck can carry "
                    + semi.getCargoCap() + " pounds.");
    System.out.println("To go " + dist + " miles, a semi-truck needs "
                      + gallons + " gallons of fuel.\n");

    gallons = pickup.fuelNeeded(dist);

    System.out.println("A pickup truck can carry "
                      + pickup.getCargoCap() + " pounds.");
    System.out.println("To go " + dist + " miles, a pickup truck needs "
                      + gallons + " gallons of fuel.\n");
  }// end main
}// end TruckDemo
```

10. **Variables declared in an interface are implicitly *static* and *final*. Can they be shared with other parts of a program?**

    Variables that are declared in interfaces can easily be shared with other parts of the program - the interface simply must be implemented by any class that would like access to those variables. This technique is controversial - some programmers feel it is poor coding practice to declare *static* and *final* variables within interfaces.

11. **A package is, in essence, a container for classes. True or False?**

    True. Packages are how code is organized in directories (folders) in Java.

12. **What standard Java package is automatically imported into a program?**

    In every single Java program, even the infamous “HelloWorld.java” program, *java.lang* is automatically imported.

13. **What keyword is used to declare a default *interface* method?**

    In order to declare a default interface method, the method simply needs to be preceded by the keyword *default*. This technique is very useful when modifying interfaces after they've already been implemented. Because default interface methods have a default return (usually something like *-1*), it does not need to be implemented by every class using the interface. This means that it won't break existing code, so working on a program that is already in use won't cause catastrophe!

14. **Beginning with JDK 8, is it possible to define a *static* method in an *interface*?**

    Beginning in JDK 8, it is possible to define a *static* method in an *interface.*

15. **Assume that the** *ICharQ* **interface shown in Try This 8-1 has been in widespread use for several years. Now, you want to add a method to it called** *reset()***, which will be used to reset the queue to its empty, starting condition. Assuming JDK 8 or later, how can this be accomplished without breaking preexisting code?**

    This can be done by adding a default interface method to the interface. Default interface methods have a default return, and do not *have* to be implemented by classes which implement the interface (unlike normal interface methods). This means that it is possible to add the reset method, defined as a default interface method, and simply implement it in the classes which will need a reset, and ignore it in the classes that do not.

16. **How is a** *static* **method in an interface called?**

    Static methods in an interface are called by specifying the interface name, a period, and then the method name. For example:

    ```
    int counter = SampleInterface.getCounter();
    ```