9.1 Developing Reusable Solutions

Consider the DataSet class of Chapter 6. We used that class to compute the average and maximum of a set of input values. However, the class was suitable only for computing the average of a set of numbers. If we wanted to process bank accounts to find the bank account with the highest balance, we would have to modify the class, like this:

```java
public class DataSet // modified for BankAccount objects
{
    public void add(BankAccount x)
    {
        sum = sum + x.getBalance();
        if (count == 0
            || maximum.getBalance() < x.getBalance())
            maximum = x;
        count++;
    }

    public BankAccount getMaximum()
    {
        return maximum;
    }

    private double sum;
    private BankAccount maximum;
    private int count;
}
```

Or suppose we wanted to find the coin with the highest value among a set of coins. We would need to modify the DataSet class again.

```java
public class DataSet // modified for Coin objects
{
    public void add(Coin x)
    {
        sum = sum + x.getValue();
        if (count == 0
            || maximum.getValue() < x.getValue())
            maximum = x;
        count++;
    }

    public Coin getMaximum()
    {
        return maximum;
    }

    private double sum;
```
private Coin maximum;
private int count;
}

Clearly, the fundamental mechanics of analyzing the data is the same in all cases, but the
details of measurement differ.

Suppose that the various classes could agree on a single method getMeasure that
obtains the measure to be used in the data analysis, such as the balance for bank
accounts, value for coins, and so on. Then we could implement a single reusable
Dataset class. The add method would look like this:

```java
    sum = sum + x.getMeasure();
    if (count == 0 || maximum.getMeasure() < x.getMeasure())
        maximum = x;
    count++;
```

What is the type of the variable x? Ideally, x should refer to any class that has a getMeasure
method. In Java, an interface type expresses that concept. Here is the interface type declaration
for a Measurable type.

```java
public interface Measurable
{
    double getMeasure();
}
```

A Java interface declares a set of methods and their signatures. Unlike a
class, it provides no implementation.

The interface declaration lists all methods that the interface requires.
This interface requires a single method, but in general, an interface
can require multiple methods.

An interface is similar to a class, but there are several important
differences:

- All methods in an interface are abstract; that is, they have a
  name, parameters, and a return type, but they don't have an implementation.
- All methods in an interface are automatically public.
- An interface does not have instance variables.

Now we can use the type Measurable to declare the variables x and maximum.

```java
public class DataSet
{
    public void add(Measurable x)
    {
        sum = sum + x.getMeasure();
        if (count == 0
            || maximum.getMeasure() < x.getMeasure())
            maximum = x;
        count++;
    }

    public Measurable getMaximum()
```
This `DataSet` class is usable for analyzing objects of any class that realizes the `Measurable` interface. A class realizes an interface if it declares the interface in an `implements` clause, and if it implements the method or methods that the interface requires.

```java
class Class_Name implements Measurable {
    public double getMeasure() {
        // implementation
    }

    // additional methods and fields
}
```

A class can realize more than one interface. Of course, the class must then define all the methods that are required by all the interfaces it realizes.

Let us modify the `BankAccount` class to realize the `Measurable` interface.

```java
public class BankAccount implements Measurable {
    public double getMeasure() {
        return balance;
    }
    ...
}
```

Note that the class must declare the method as `public`, whereas the interface does not—all methods in an interface are `public`.

Similarly, it is an easy matter to modify the `Coin` class to realize the `Measurable` interface.

```java
public class Coin implements Measurable {
    public double getMeasure() {
        return value;
    }
    ...
}
```
Now `DataSet` objects can be used to analyze collections of bank accounts or coins. Here is a test program that illustrates the fact.

```java
/**
 * This program tests the DataSet class.
 */

public class DataSetTest {
    public static void main(String[] args) {
        DataSet bankData = new DataSet();
        bankData.add(new BankAccount(0));
        bankData.add(new BankAccount(10000));
        bankData.add(new BankAccount(2000));
        System.out.println("Average balance = " + bankData.getAverage());
        Measurable max = bankData.getMaximum();
        System.out.println("Highest balance = " + max.getMeasure());

        DataSet coinData = new DataSet();
        coinData.add(new Coin(0.25, "quarter"));
        coinData.add(new Coin(0.1, "dime"));
        coinData.add(new Coin(0.05, "nickel"));
        System.out.println("Average coin value = " + coinData.getAverage());
        max = coinData.getMaximum();
        System.out.println("Highest coin value = " + max.getMeasure());
    }
}
```

Figure 1 shows the relationships between the classes and interfaces. In the UML notation, interfaces are tagged with a "stereotype" indicator <<interface>>. A dotted arrow with a triangular tip denotes the realization relationship between a class and an interface. You have to look carefully at the arrow tips—a dotted line with an open v-shaped arrow tip denotes dependency.
This diagram shows that the `DataSet` class depends only on the `Measurable` interface. It is decoupled from the `BankAccount` and `Coin` classes. This decoupling makes the `DataSet` class reusable. Any class that is willing to implement the `Measurable` interface can be used with the `DataSet` class.

**Syntax 9.1: Defining an Interface**

```
public interface InterfaceName
{
   method signatures
}
```

**Example:**

```
public interface Measurable
{
   double getMeasure();
}
```

**Purpose:**
To define an interface and its method signatures. The methods are automatically public.
### Syntax 9.2: Implementing an Interface

```java
public class ClassName
    implements InterfaceName, InterfaceName, ...
{
    methods
    instance variables
}
```

**Example:**

```java
public class BankAccount
    implements Measurable
{
    // other BankAccount methods
    public double getMeasure()
    {
        // method implementation
    }
}
```

**Purpose:**

To define a new class that implements the methods of an interface

---

### Common Error

**Forgetting to Define Implementing Methods as Public**

The methods in an interface are not declared as public, because they are public by default. However, the methods in a class are not public by default—their default access level is "package" access, which we discuss in Chapter 11. It is a common error to forget the public keyword when defining a method from an interface:

```java
public class BankAccount implements Measurable
{
    double getMeasure() // oops—should be public
    {
        return balance;
    }
}
```

Then the compiler complains that the method has a weaker access level, namely package access instead of public access. The remedy is to declare the method as public.