SUPPLEMENT D Additional Java 1.5 features

There are a number of Java features not touched upon in the text. In particular, JDK 1.5 introduces several extensions to the core language. We briefly examine the most important of these extensions in this supplement.[1]

d.1 Generics

The most significant addition to the language in JDK 1.5 is the inclusion of generic structures. As we have seen in Chapter 12, genericity allows us to define classes, interfaces, and methods with type parameters. There are two additional aspects of generic structures we should mention.

- Generic structures can have more than one type parameter.

  Consider the class *KeyValueTable* defined in Listing d.1. The class is defined with two type parameters, `Key` and `Value`. Instantiating the generic class requires two type arguments. For example,

  ```
  KeyValueTable<String, Integer> grades =
     new KeyValueTable<String, Integer>();
  ```

  The method `grades.add` requires a *String* and an *Integer* as arguments, and the method `grades.lookUp` requires a *String* argument and returns an *Integer*:

  ```
  grades.add("Henry", new Integer(97));
  Integer henrysGrade = grades.lookUp("Henry");
  ```

- Type parameters can be "bounded" by another type.

  Suppose *Keyed* is an interface defined as

  ```
  public interface Keyed {
     public String getKey ();
  }
  ```

1. As this is written, JDK 1.5 is available in beta release only and the specification document has not been published.

Listing d.1 The class *KeyValueTable*

```
/**
 * A simple (key, value) table.
 */
public class KeyValueTable<Key, Value> {

   private List<Key> keys;      // the keys
   private List<Value> values;  // the values

   /**
    * Create a new empty table.
    */
   public KeyValueTable () {
      keys = new DefaultList<Key>();
      values = new DefaultList<Value>();
   }

   /**
    * Add the specified (key, value) to this table. If the
    * key is already in the table, replace the associated
    * value with the one specified.
    */
   public void add (Key key, Value value) {
      int i = keys.indexOf(key);
      if (i = -1) {
         keys.add(key);
         values.add(value);
      } else
         values.set(i,value);
   }

   /**
    * The value associated with the specified key. Returns
    * null if the key is not in the table.
    */
   public Value lookUp (Key key) {
      int i = keys.indexOf(key);
      if (i != -1)
         return values.get(i);
      else
         return null;
   }
}
```

Then the following version of the generic class *KeyValueTable* requires its argument to be a subtype of *Keyed*.

```
public class KeyValueTable<Entry extends Keyed> {

   private List<String> keys;
   private List<Entry> entries;
   ...
   public void add (Entry entry) {
      String key = entry.getKey();
      ...
   }

   public Entry lookUp (String key) {
      ...
   }
}
```

Note that the keyword for bounding a type is **`extends`**, regardless of whether the bounding type is defined by an interface or by a class.

The `add` method takes advantage of the fact that a generic argument will be a subtype of *Keyed*, and thus define the method `getKey`. If *Student* implements *Keyed*,

```
public class Student implements Keyed ...
```

we can instantiate the generic class with *Student* as argument:

```
KeyValueTable<Student> grades =
   new KeyValueTable<Student>();
Student henry = ...;
grades.add(henry);
```

Since `henry` is a *Student*, and *Student* implements *Keyed*, `henry` supports the method `getKey`.

Now suppose *Keyed* is a generic interface:

```
public interface Keyed<Key> {
   public Key getKey ();
}
```

The following version has two type parameters, the second depending on the first.

```
public class KeyValueTable
   <Key, Entry extends Keyed<Key>> {

   private List<Key> keys;
   private List<Entry> entries;
   ...
   public void add (Entry entry) {
      Key key = entry.getKey();
      ...
   }
```

```
    public Entry lookUp (Key key) {
       ...
    }
}
```

If the first generic argument is *String*, for example, the second must be a subtype of *Keyed<String>*. Assuming that *Student* implements *Keyed<String>*, and so defines a method `String getKey()`, we can instantiate the generic class as follows.

```
KeyValueTable<String, Student> grades =
   new KeyValueTable<String, Student>();
Student henry = ...;
grades.add(henry);
```

d.1.1 Wildcard types

Generics and subtyping, revisited

We have seen, in Section 9.2.3, that the fundamental rule of subtyping states that if *A* is a subtype of *B*, then an *A* value can be provided wherever a *B* value is required. For instance, the *Object* method `equals` requires an *Object* argument:

```
public boolean equals (Object obj) ...
```

Since *Student* is a subtype of *Object*, we can invoke the method with a *Student* as argument:

```
if (someObject.equals(henry)) ...
```

In Section 12.2.2 we learned that *A* a subtype of *B* does not imply that *List<A>* is a subtype of *List<B>*. If it were, the fundamental rule of subtyping would be violated. For instance, we could write a method that adds a *String* to a *List<Object>*,

```
public void addString (List<Object> list) {
   list.add("end"); // OK: String a subtype of Object
}
```

and then invoke the method with a *List<Integer>* as argument:

```
List<Integer> numbers = new DefaultList<Integer>();
addString(numbers);
 // OK if List<Integer> is a subtype of List<Object>!
```

The rule is true in general for generic types: if *T<E>* is a generic type (with parameter *E*), then *A* a subtype of *B* does not imply that *T<A>* is a subtype of *T<B>*.

An example

Suppose we have an interface *ClosedFigure* that specifies a method for computing area:

```
public interface ClosedFigure
```
A regular closed two-dimensional geometric figure.

```
   public double area ()
```
The area of this figure.

We can write a method that takes a *List<ClosedFigure>* and produces the total area of the elements of the list.

```
public double totalArea (List<ClosedFigure> list) {
   double sum = 0.0;
   for (int i = 0; i < list.size(); i = i+1)
      sum = sum + list.get(i).area();
   return sum;
}
```

If *Circle* is a *ClosedFigure*, we should be able to invoke the method with a *List<Circle>* as argument. All the method does is query each list element for its area, and certainly a *Circle* can be queried for its area. But if `hoops` is a *List<Circle>*, the invocation

```
totalArea(hoops)
```

fails to compile because *List<Circle>* is not a subtype of *List<ClosedFigure>*.

Wildcards

Wildcards are an extension to the type system intended to improve the flexibility of generic structures. Syntactically, a wildcard is an expression of the form `?`, `? extends` *T*, or `? super` *T*, where *T* is a type. Wildcards denote types, and can be read as follows:

`?`	–	"some type"
`? extends` *T*	–	"*T* or some subtype of *T*"
`? super` *T*	–	"*T* or some super type of *T*"

The first form is called an "unbounded wildcard" and is essentially equivalent to `? extends Object`.

Wildcards can only be used as type arguments in generic instantiations. For example, we can write variable declarations like these

```
List<?> list;
List<? extends Exception> exceptionList;
```

but not like these:

```
? something;
? extends Exception someException;
```

Consider the simple generic class shown in Listing d.2. The expression `Item<?>` denotes "*Item<some type>*," while `Item<? extends Exception>` denotes "*Item<some type of Exception>*." For example, the parameter of the following method

Listing d.2 The class *Item*

```
public class Item<Element> {

   private Element value;

   public Item (Element value) {
      this.value = value;
   }

   public Element value () {
      return value;
   }

   public void setValue (Element value) {
      this.value = value;
   }
}
```

(I)

```
public Object getItemValue (Item<?> item) {
   return item.value();
}
```

specifies that the argument must be "an *Item* of some type." The method can be invoked with any kind of *Item* as argument, for example

```
Item<String> i1 = new Item<String>("hello");
Item<Integer> i2 = new Item<Integer>(new Integer(2));
Object o1 = getItemValue(i1);
Object o2 = getItemValue(i2);
```

Since the argument of `getItemValue` can be any type of *Item*, the only thing we can conclude about the value returned by `item.value()` is that it is an *Object*. Thus the method `getItemValue` is specified as returning an *Object*.

Now consider the method

```
public String getString (
   Item<? extends Exception> item) {
   return item.value().getMessage();
}
```

We can be sure that the argument supplied to this method will be an *Item*<*T*>, where *T* is some type of *Exception*. Thus the value returned by `item.value()` is an *Exception* and has a method `getMessage` that returns a *String*.

Finally, consider the method

```
public boolean sameValue (Item<?> one, Item<?> two) {
   return one.value().equals(two.value());
}
```

The first argument must be an *Item* of some type and the second argument must be an *Item* of some type. But there is no requirement that the types of the *Items* be the same. That is, we cannot assert that `one.value()` and `two.value()` have the same type. The method can be invoked, for instance, with an *Item<String>* first argument and an *Item<Integer>* second argument.

By now, you probably wonder what we have bought with all this new syntax. Why could we not just write

(II)

```
public Object getItemValue (Item<Object> item) {
   return item.value();
}

public String getString (Item<Exception> item) { …
   return item.value().getMessage();
}
```

and so on. To see the difference consider the invocation of the `getItemValue` shown above:

```
Item<String> i1 = new Item<String>("hello");
Object o1 = getItemValue(i1);
```

If `getItemValue` is defined as *(II)*, this invocation will not compile because *Item<String>* is not a subtype of *Item<Object>*. If it were, we could write a method

```
public void setNumber (Item<Object> item) {
   Integer integer = new Integer(1);
   item.setValue(integer);
}
```

and invoke it with

```
Item<String> i1 = new Item<String>("hello");
setString(i1);
```

But we have seen that if `getItemValue` is defined as *(I)*, the invocation

```
Object o1 = getItemValue(i1);
```

succeeds, where `i1` is an *Item<String>*. So *Item<String>* must be a subtype of *Item<?>*, even though it is not a subtype of *Item<Object>*.

What if we write

```
public void setNumber (Item<?> item) {
   item.setValue(new Integer(1));
}
```

This method will not compile. Since `item` is of type *Item<?>*, all we know is that `item.setValue` requires “some type” of argument. We cannot conclude that *Integer* is an appropriate type. That is, inside `setNumber`, the signature if `item.setValue` is essentially **`void`** `setValue(?)`. There are no (proper) subtypes of *?*.

On the other hand, suppose we define

```
public void setRTE (Item<? super Exception> item) {
   item.setValue(new RuntimeException());
}
```

Now we can be sure that the argument supplied to `setRTE` is of type *Item<T>* where *T* is *Exception* or a supertype of *Exception*. Thus `item.setValue` will require an argument of type *T*, where *T* is *Exception* or an *Exception* supertype. Now *RuntimeException* is a subtype of *Exception*, and so of any *Exception* supertype. Hence *RuntimeException* is a subtype of whatever type `item.setValue` expects, and the invocation is legal. Inside `setNumber`, the signature of `item.setValue` is `void setValue(? super Exception)`. *Exception* and its subtypes are subtypes of *? super Exception*.

Figure 4.1 illustrates the subtype relationship between wildcard types. In the figure, *T* is a type, *SubT* is a subtype of *T*, and *SuperT* is a supertype of *T*. Remember that wildcard type expressions, such as `?` and `? extends T`, can only be written as type arguments for generic types.

Returning to the *ClosedFigure* example, the solution is to use a wildcard type in the definition of `totalArea`:

```
public double totalArea (
   List<? extends ClosedFigure> list) {
   double sum = 0.0;
   for (int i = 0; i < list.size(); i = i+1)
      sum = sum + list.get(i).area();
   return sum;
}
```

Since *Circle* is a subtype of *ClosedFigure*, *List<Circle>* is a subtype of *List<? extends ClosedFigure>*. The invocation `totalArea(hoops)`, where hoops is a *List<Circle>*, is legal.

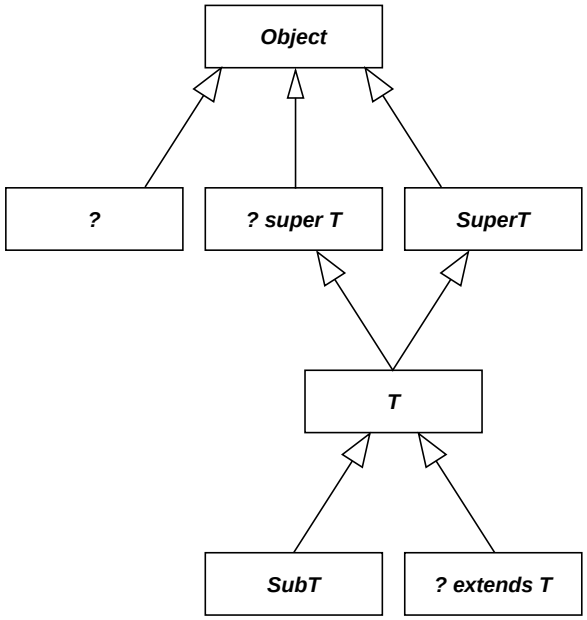
Finally, we should mention that a wildcard type can be the type of a variable. For example, suppose we wanted to keep a list of all the *Item* instances ever created. (Don't ask why.) We can write the following

```
public class Item<Element> {

   private Element value;
   public static List<Item<?>> items =
      new DefaultList<Item<?>>();

   public Item (Element value) {
      this.value = value;
      items.add(this);
   }
   …
```

The static variable `items` is of type *List<Item<?>>*. This means that an *Item* of any type can be added to the list.



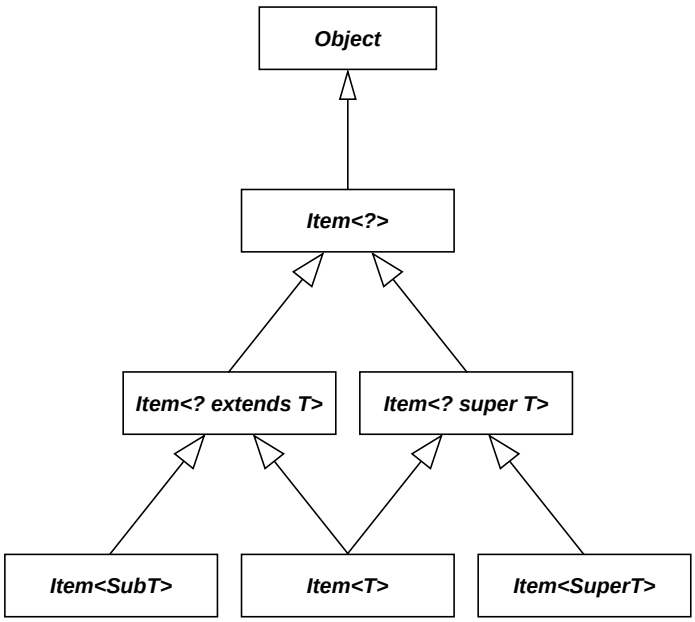


Figure 4.1 **Subtype relationship between types and wildcard types.**

Wildcards and generic methods

Some of the methods written above with wildcards could have been written as generic methods. For example,

```
public <Type> Object getItemValue (Item<Type> item) {
   return item.value();
}

public <Type extends Exception> String getString (
   Item<Type> item) {
   return item.value().getMessage();
}

public <Type1, Type2> boolean sameValue (
   Item<Type1> one, Item<Type2> two) {
   return one.value().equals(two.value());
}

public <Type extends ClosedFigure> double totalArea (
   List<Type> list) {
   double sum = 0.0;
   for (int i = 0; i < list.size(); i = i+1)
      sum = sum + list.get(i).area();
   return sum;
}
```

(We cannot write `setRTE` as a generic method.)

When should we use wildcard types and when should we write generic methods? Wildcards are considered clearer and easier to understand than generic methods. The general guideline is to use wildcards if we want to express polymorphism. That is, we want to express the fact that the method can be invoked with different argument types. If we want to express dependencies between arguments, or between arguments and return type, we write a generic method. For example, we can write a tighter version of `getItemValue` as a generic method:

```
public <Type> Type getItemValue (Item<Type> item) {
   return item.value();
}
```

Here we have expressed a dependency between the argument type and the return type. With the argument type expressed as a wildcard, the most we can say about the returned value is that it is an *Object*.

Similarly, we can use a generic method to require that the arguments of `sameValue` be of the same type. If we write

```
public <Type> boolean sameValue (
   Item<Type> one, Item<Type> two) {
   return one.value().equals(two.value());
}
```

then `sameValue` cannot be invoked, for instance, with an I*tem*<*String*> first argument and an *Item*<*Integer*> second argument.

Opening wildcards

Occasionally we want to use a wildcard parameter for purposes of expression, but need a name for the type in the implementation. For example, suppose we are writing a method that swaps two items of a list. It is natural to express the method with a wildcard list type:

```
public void swap (List<?> list, int i, int j)
```
Swap the elements with indexes i and j.

But when we implement the method, we find that we need to name the list element type:

```
public void swap (List<?> list, int i, int j) {
✘     ? temp = list.get(i);   // whoops! Can't do this
   list.set(i, list.get(j));
   list.set(j, temp);
}
```

One approach is to define the public method with a wildcard, and have it call a private generic version. The generic version, which must have a different name, "captures" the list element type with a name.

```
public void swap (List<?> list, int i, int j) {
   swapImp(list, i, j);
}

private <Type> void swapImp (
   List<Type> list, int i, int j) {
   Type temp = list.get(i);
   list.set(i, list.get(j));
   list.set(j, temp);
}
```

d.2 Autoboxing and unboxing

Recall that there is a wrapper class defined in `java.lang` for each primitive type. For instance, the wrapper class for the primitive type **`int`** is the class *Integer*. An *Integer* instance wraps an **`int`** value in an immutable object. An *Integer* can be created by providing the **`int`** value as a constructor argument,

```
public Integer (int value)
```
Create an *Integer* that represents the specified int value.

and the **`int`** value can be retrieved with the *Integer* method `intValue`,

```
public int intValue ()
```
The value of this *Integer* as an **int**

Thus, after

```
Integer obj = new Integer(3);
int i = obj.intValue();
```

`i` will contain the **int** value 3.

Boxing is simply wrapping a primitive value in an object,

```
new Integer(3)
```

an *unboxing* is retrieving the wrapped value,

```
obj.intValue()
```

With JDK 1.5, the compiler will automatically box a primitive value that appears in a context requiring an object, and will automatically unbox an object that appears in a context requiring a primitive value.

For example, suppose `grades` is defined as a *List<Integer>*:

```
List<Integer> grades = new DefaultList<Integer>();
```

The *List<Integer>* method `add` requires an *Integer* as argument. If we write

```
grades.add(100);
```

the compiler will automatically box the value 100 in an *Integer*. That is, the method invocation will be effectively translated into

```
grades.add(new Integer(100));
```

Conversely, an *Integer* will be unboxed if an **int** is required. For example, we can write

```
int sum = grades.get(0) + grades.get(1);
```

even though the method `grades.get` returns an *Integer*. The code is effectively translated into

```
int sum = grades.get(0).intValue() +
   grades.get(1).intValue();
```

d.3 Enumeration types

The *enumeration type* mechanism (the *enum facility*) provides a convenient way for defining a class that has a small fixed number of instances. Such classes are sometimes useful in situations where we have previously used named **int** constants.

For example, recall the class *PlayingCard*, defined in Section 2.6. A *PlayingCard* has two attributes, suit and rank. Suits were defined by four named constants,

```
public static final int CLUB = 1;
```

```
public static final int HEART = 2;
public static final int DIAMOND = 3;
public static final int SPADE = 4;
```

The *PlayingCard* constructor required the argument specifying suit to be one of these four values,

```
/**
 * Create a new PlayingCard with the specified suit and
 * rank.
 * @require    suit == PlayingCard.CLUB ||
 *             suit == PlayingCard.DIAMOND ||
 *             suit == PlayingCard.HEART ||
 *             suit == PlayingCard.SPADE
 *             ...
 */
public PlayingCard (int suit, int rank) { ...
```

and the query `suit` promised to return one of these values,

```
/**
 * The suit of this PlayingCard.
 * @ensure     this.suit() == PlayingCard.CLUB ||
 *             this.suit() == PlayingCard.DIAMOND ||
 *             this.suit() == PlayingCard.HEART ||
 *             this.suit() == PlayingCard.SPADE
 */
public int suit () { ...
```

The problem is that there is no way for the compiler to verify that a client will provide appropriate arguments when the constructor is invoked. The best we can do is to include a run-time check in the constructor:

```
public PlayingCard (int suit, int rank) {
   assert suit == CLUB || suit == DIAMOND ||
      suit == HEART || suit == SPADE;
   ...
```

Furthermore, the value returned by `suit` is just an **`int`**, and not particularly helpful in testing or debugging. For instance, given

```
PlayingCard c = new PlayingCard(PlayingCard.CLUB,2);
```

the statement

```
System.out.println(c.suit());
```

displays 1.

With the enumeration facility, we can easily define a class that contains only four objects modeling the suits. We write in the class *PlayingCard*

```
public enum Suit {clubs, diamonds, hearts, spades}
```

This defines a public, static, *PlayingCard* member class named *Suit*. *Suit* has four instances, referenced by named constants `clubs`, `diamonds`, `hearts`, and `spades`. It is roughly equivalent to the following:

```
public static class Suit {

   private final String name;

   public static final Suit clubs =
      new Suit("clubs");
   public static final Suit diamonds =
      new Suit("diamonds");
   public static final Suit hearts =
      new Suit("hearts");
   public static final Suit spades =
      new Suit("spades");

   private Suit (String name) {
      this.name = name;
   }

   public String toString () {
      return this.name;
   }
}
```

Suit is a public static class defined in *PlayingCard*. Thus *PlayingCard.Suit* is a class. `clubs` is a named constant defined in the class and referencing one of the four instances of the class *Suit*. Thus `PlayingCard.Suit.clubs` references an instance of *PlayingCard.Suit*. Since the constructor for *PlayingCard.Suit* is private, a client cannot create new *Suit* instances.

Now the *PlayingCard* constructor can require an argument of type *PlayingCard.Suit*, and the method `suit` can return a value of this type:

```
/**
 * Create a new PlayingCard with the specified suit and
 * rank.
 * ...
 */
public PlayingCard (PlayingCard.Suit suit, int rank)...
/**
 * The suit of this PlayingCard.
 */
public PlayingCard.Suit suit () ...
```

The type of the argument in a constructor invocation can be verified by the compiler. The client must write something like this:

```
PlayingCard c =
   new PlayingCard(PlayingCard.Suit.clubs,2);
```

Of course, the same approach can be taken with the *PlayingCard* rank: The class is shown, with comments omitted, in Listing d.3.

As suggested above, the method `toString` returns the name of the constant. For instance,

```
PlayingCard.Suit.clubs.toString() ⇒ "clubs"
```

Other methods defined for an enum class include

`public int compareTo (`*`EnumClass`*` obj)`
: Compare this enum constant with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object. Enum constants are comparable only to other enum constants of the same enum class. The natural order implemented by this method is the order in which the constants are declared.

`public final int ordinal ()`
: The ordinal of this enumeration constant (its position in its enum declaration, where the initial constant is assigned an ordinal of zero).

Listing d.3 The class *PlayingCard*

```
public class PlayingCard {

   public enum Suit {clubs, diamonds, hearts, spades}
   public enum Rank {two, three, four, five, six, seven,
      eight, nine, ten, jack, queen, king, ace}

   private Suit suit;
   private Rank rank;

   public PlayingCard (Suit suit, Rank rank) {
      this.suit = suit;
      this.rank = rank;
   }

   public Suit suit () {
      return suit;
   }

   public Rank rank () {
      return rank;
   }

   public String toString () {
      return rank + " of " + suit;
   }
}
```

public static final *EnumClass*[] values ()
An array containing the elements of the enum type in the order in which they were declared.

For example,

```
Suit.clubs.compareTo(Suit.hearts)⇒ a negative value
Suit.clubs.compareTo(Suit.clubs) ⇒ 0
Suit.hearts.compareTo(Suit.clubs)⇒ a positive values
Suit.clubs.ordinal()             ⇒ 0
Suit.spades.ordinal()            ⇒ 3
Suit.values[0]                   ⇒ clubs
Suit.values[3]                   ⇒ spades
```

The array returned by `values` can be used to iterate through the elements of an enum class. (But see Section d.4 for a cleaner approach.) For example,

```
List<PlayingCard> deck =
   new DefaultList<PlayingCard>();
for (int i = 0; i < Suit.values().length; i = i+1)
   for (int j = 0; j < Rank.values().length; j = j+1)
      deck.add(new PlayingCard(Suit.values()[i],
         Rank.values()[j]));
```

Adding features to an enum type

An enum declaration defines a class with a set of predefined features. However, it is possible to define additional features for the class. For example, suppose we want a class that model coins. We might define an enum class as

```
public enum Coin {penny, nickel, dime, quarter, half}
```

But suppose we want the *Coin* objects to know their monetary value. We can create an enum class with additional features:

```
public enum Coin {
   penny(1),
   nickle(5),
   dime(10),
   quarter(25),
   half(50);

   private int monetaryValue;
   private Coin (int monetaryValue) {
      this.monetaryValue = monetaryValue;
   }

   public int monetaryValue () {
      return monetaryValue;
   }
}
```

A *Coin* now has a private instance variable `monetaryValue`, and a public query with the same name. We have also explicitly defined a constructor requiring an **int** argument. The numbers in the definition of the enum constants, 10 in `dime(10)` for instance, are constructor arguments. For example,

```
Coin.dime.monetaryValue() ⇒ 10
```

Modifying the behavior of enum instances

Let's take a look at the class *TrafficSignal*, specified in Listing 2.3. Recall that this class defines three named constants,

```
public static final int GREEN = 0;
public static final int YELLOW = 1;
public static final int RED = 2;
```

It includes a query for the current light and a command to change to the next light.

```
public int light ()
```
The light currently on.

ensure:
```
this.light() == TrafficSignal.GREEN ||
this.light() == TrafficSignal.YELLOW ||
this.light() == TrafficSignal.RED.
```

```
public void change ()
```
Change to the next light.

Clearly we can use an enum type rather than **int** constants to define the lights:

```
public enum Light {green, yellow, red}
...
/**
 * The Light currently on.
 */
public Light light () ...
```

Rather than implementing the method `change` as a cascade of if statements (see Section 4.3), we can produce a cleaner solution if we let each *Light* instance know which *Light* follows it. We add this functionality to the class *Light*:

```
public enum Light {
   green, yellow, red;

   private Light next () {
      return values()[this.ordinal()+1];
   }
}
```

When queried for `next`, a *Light* returns the next *Light* in the enumeration. Thus

```
Light.green.next()  ⇒ Light.yellow
Light.yellow.next() ⇒ Light.red
```

But if we query `red` for `next`, we generate an *ArrayIndexOutOfBoundsException*, since `Light.red.ordinal()` is 2, and `Light.values()` contains only three elements, with indexes 0, 1, and 2. There is no enum value with index 3.

We want the next method for `red` to return the first enum value, `green`. We accomplish this by making `red` an instance of an anonymous *Light* subclass that overrides the implementation of `next`. The enum syntax makes this easy:

```
public enum Light {
   green,
   yellow,
   red {
      protected Light next () {
         return values()[0];
      }
   };

   protected Light next () {
      return values()[this.ordinal()+1];
   }
}
```

The method `next` cannot now be private, since it is to be overridden in the anonymous subclass of `red`. The complete implementation of *TrafficSignal* is shown in Listing d.4.

Listing d.4 The class *TrafficSignal*

```
/**
 * A simple green-yellow-red traffic signal.
 */
public class TrafficSignal {

   private Light current; // The Light currently on.

   /**
    * The signal lights.
    */
   public enum Light {
      green,
      yellow,
      red {
         protected Light next () {
            return values()[0];
         }
      };
```

continued

Listing d.4 The class *TrafficSignal* (cont'd)

```
      /**
       * The light that comes on after this one.
       */
      protected Light next () {
         return values()[this.ordinal()+1];
      }
   }

   /**
    * Create a new TrafficSignal, initially green.
    * @ensure     this.light() == Light.green
    */
   public TrafficSignal () {
      current = Light.green;
   }

   /**
    * The light currently on.
    */
   public Light light () {
      return current;
   }

   /**
    * Change to the next light.
    */
   public void change () {
      current = current.next();
   }
}
```

d.4 Enhanced *for* statement

The for statement has been enhanced in JDK 1.5 to make iteration over a container easier. The format of the enhanced for statement is:

```
for (type identifier : expression)
   bodyStatement
```

Expression denotes the container to be iterated over. Its type must implement or extend the new interface, *java.lang.Iterable*, or it must be an array. (Every container type,

such as *List*, should implement or extend the interface.) *Type* denotes the type of the elements in the container. *Identifier* is used in the body to refer to a container element.

For example, the method to compute the average final exam grade for a nonempty list of *Students* (Section 12.5.1) can be written as follows:

```
public double average (List<Student> students) {
   int sum = 0;
   for (Student s : students)
      sum = sum + s.finalExam();
   return (double)sum / (double)students.size();
}
```

In each iteration of the body of the for loop, `s` denotes a different element of the list `students`. The code is essentially equivalent to

```
public double average (List<Student> students) {
   int sum = 0;
   for (int i = 0; i < student.size(); i = i +1) {
      Student s = students.get(i);
      sum = sum + s.finalExam();
   }
   return (double)sum / (double)students.size();
}
```

Note that we can iterate over an enum type by using the `values` method. For example,

```
for (Suit suit : Suit.values())
   for (Rank rank : Rank.values())
      deck.add(new PlayingCard(suit,rank));
```

d.5 Importing static methods and named constants

The import statement has been enhanced so that it is now possible to import static methods and constants into a compilation unit. The formats are:

```
import static type.identifier ;
import static type.* ;
```

Identifier must be a static member of the class or interface named by *type*. The first format imports the name into the compilation unit. The second format imports the names of all static members of the class or interface.

For example,

```
import static java.lang.Math.*;
```

imports into a compilation unit all the static functions defined in the class *Math*. For example, we can then write `sqrt(2)` rather than `Math.sqrt(2)`.

If the class *cardGame.PlayingCard* defines enumeration types

```
public enum Suit {clubs, diamonds, hearts, spades}
public enum Rank {two, three, four, five, six, seven,
   eight, nine, ten, jack, queen, king, ace}
```

we can import the enumeration constants into a compilation unit by including

```
import static cardGame.PlayingCard.Suit.*;
import static cardGame.PlayingCard.Rank.*;
```

We can then invoke the constructor by writing, for instance

```
new cardGame.PlayingCard(spades,ace)
```

d.6 The class *java.util.Scanner*

The class *java.util.Scanner* provides a means for reading character input, similar to our *BasicFileReader*. The class *Scanner* offers more flexibility, and so is much more complex, than our class. We consider only the most elemental features here.

A *Scanner* is created with a static factory method named `create`. There are eight overloaded versions of the method. The most basic require a single argument, the source of the input:

```
public static Scanner create (InputSource source)
```
Create a *Scanner* to read from `source`.

InputSource possibilities include, among others, *java.io.File*, *java.io.InputStream*, and *java.io.Reader*.

A *Scanner* views its input stream as a sequence of *tokens*, separated by *delimiters*. By default, a token is a sequence of nonwhite characters, and delimiters are whitespace. (Recall that whitespace includes characters such as space, line feed, horizontal tab, *etc*.) For example, if the input stream consisted of

(I)
```
•••+12345a••↲•bac••xyz↲••↲•••12.3e+2••↲zzz↲
```

where "•" represents a space and "↲" represents the line termination character(s), a *Scanner* would see five tokens: `+12345a`, `bac`, `xyz`, `12.3e+2`, and `zzz`.

The basic *Scanner* method `next` reads and returns the next token from its input source:

```
public String next ()
```
Find and return the next complete token from the input source. A complete token is preceded and followed by delimiters. This method may block while waiting for input.

Throws *java.util.NoSuchElementException* if no more tokens are available. Throws *java.lang.IllegalStateException* if this *Scanner* is closed

Note that this method is neither a proper command nor a proper query. It both returns a value and changes the state of the *Scanner.*

To determine whether or not there are tokens remaining in the input, *Scanner* provides the query

public boolean `hasNext ()`

This scanner has another token in its input. This method may block while waiting for input.

Throws *java.lang.IllegalStateException* if this *Scanner* is closed.

Assume that `scanner` is a *Scanner* with input as shown in *(I)* above. Then the following iteration

```
while (scanner.hasNext()) {
   String token = scanner.next();
   System.out.println(token);
}
```

produces five lines of output:

```
+12345a
bac
xyz
12.3e+2
zzz
```

Note that `this.hasNext()` is a precondition for the method `next`.

Scanner also has methods for recognizing whether or not the next token can be interpreted as an **`int`**, **`float`**, **`double`**, **`boolean`**, *etc*. For example,

public boolean `hasNextBoolean ()`

The next token in this *Scanner*'s input can be interpreted as a **`boolean`**. That is, it is the string `"true"` or `"false"` ignoring case.

public boolean `hasNextInt ()`

The next token in this *Scanner's* input can be interpreted as an **`int`** value.

public boolean `hasNextDouble ()`

The next token in this *Scanner's* input can be interpreted as a **`double`** value.

(All these methods throw a *java.lang.IllegalStateException* if the *Scanner* is closed.)

If the method `hasNextInt` is invoked with the input shown above in *(I)*, it will return false because the next token, `+12345a`, does not have the format of an **`int`**.

There are corresponding "next" methods that read the next token, and return the primitive value denoted by the token. For example,

public boolean `nextBoolean ()`

Read the next token from the input source and return the **`boolean`** denoted by the token.

require:
`this.hasNextBoolean()`

`public int nextInt ()`
Read the next token from the input source and return the **`int`** denoted by the token.

require:
`this.hasNextInt()`

`public double nextDouble ()`
Read the next token from the input source and return the **`double`** denoted by the token.

require:
`this.hasNextDouble()`

Finally, the method `close` closes the input:

`public void close()`
Closes this *Scanner* and its associated input stream.