Chapter 7
Collections and Control Structures

Pedagogic Motivations

- Introduce the notion of a collection of objects.
- Distinguish between a composite from a collection.
- Demonstrate how to perform an operation on every member of a collection
- Demonstrate how to traverse a collection
- Motivate the decoupling of structure and traversal logic in the Iterator pattern

7.1 Introduction

Up to this point we have been dealing with individual objects. It is, however, quite common to have to deal with many objects at once. We saw, in one of our projects, that having a composite object was quite useful. A composite object allows us to group together several objects *as if they were one object*. But sometimes we want to group together objects for a different purpose: we want to treat the group as a collection of objects, not an individual of the same type, and we want to be able to access each member of the collection as an individual object.

In this chapter we introduce the notion of a collection, and see how Java's collection classes work. We do not worry about how these collections are implemented, only about what they can do for us.

7.2 Collections and Mappings

It turns out that it is very common in the software realm to deal with not just one object at a time, but many objects at a time. In fact, much work in computer science is focused on discovering efficient ways to store and process large amounts of data.

A collection is an object whose responsibility it is to store an arbitrary number of object references. A collection is, in this sense, a container for object references, and behaves a little bit like a shopping cart at the grocery store. You can add things into the cart, as well as remove them. With this image of a collection as a shopping cart it is easy to see the difference between a collection and a composite: if a shopping cart were a composite of grocery items, then it would also *be* a grocery item. But a shopping cart is not a grocery item. Instead, it is a *container* for grocery items.

Java provides two main ways to group objects: collections and mappings. Collections are simple containers, as we just discussed. Mappings are containers which provide a correspondence between something called a *key* and a corresponding *value*. This is not as confusing as it sounds. A phone book provides a mapping between names a telephone numbers. The keys in a telephone book are the names, and the values associated with names are the corresponding telephone numbers. Similarly, a dictionary provides a mapping between words and their definitions. A dictionary which was simply a collection of definitions would not be very useful, would it?

In this chapter we will discuss mostly collections, in the Java sense, but we will touch on one mapping in particular towards the end of the chapter.

7.3 Generics

A collection holds objects of some type. Thus, a collection of groceries holds grocery items, while a collection of books holds books.

In Java, the type of the elements of a collection is specified using a special bit of syntax. Let us model a shopping cart as an object which has, as its contents, a collection of `Grocery` items, we write:

```
public class ShoppingCart {
    private java.util.Collection<Grocery> _contents;

    public ShoppingCart() {
        _contents = ...
    }
}
```

The type given inside the angled brackets is the type of the elements of the collection.[1] We do not yet initialize the `_contents` variable in the constructor because we do not yet know about any particular collection implementations.

7.4 java.util.Collection

The main interface for collections in the Java libraries is `java.util.Collection`. As mentioned above, it is a parameterized (generic) interface. We refer to it generically as `java.util.Collection<E>`, where `E` is the element type.

There are quite a few methods that are available on a `java.util.Collection<E>` object. We will start by considering just one, a method to add an item to a collection. This method's name is add, and it has a single parameter, the item to be added. The method header for this method is therefore specified as follows:

```
public boolean add(E item)
```

[1] Any class can be defined to be parameterized, in which case we call it *generic*. In this text we will not show you how to define generic classes, only how to use them.

There are two things we need to explain in detail about this method header. The first is that the element type is specified generically as `E`. For a specific collection, `E` will be replaced by whatever the element type really is. Thus, if we have a collection of `Grocery` items, the effective method header for the add method will be:

```
public boolean add(Grocery item)
```

Of course, you will not see the code written this way, but the `add` method of any collection instance whose elements are of type `Grocery` will behave as if it were defined this way: it will only accept `Grocery` items as an actual parameter.

The second thing we need to explain is the return type of this method. The return type is written as `boolean`. But what is `boolean`? Is it a type? Is it a class? Is it an interface?

It turns out that `boolean` is indeed a type, but it is neither a class nor an interface. Instead, it is what is called a *primitive* type. Unlike class and interface types, the set of primitive types in Java is fixed: you as a programmer cannot define new primitive types. Moreover, the set of elements of each primitive type is fixed too: you cannot instantiate primitive types to create new elements. Thus, the only `boolean` values are `true` and `false`.

Why are primitive types called primitive? They are called primitive because their values are “simple”, or “atomic”, or “lacking internal structure”. An object can have both properties and capabilities. An primitive value can have neither – it just *is*.

So why does the `add` method on a `Collection` object return a value of type `boolean`? It turns out that some collections allow duplicates and some do not. If the `add` method returns `true`, that means that the new element was added to the collection. If the `add` method return `false`, that means that the new element was not added to the collection because it was already there.

7.5 An implementation of java.util.Collection

The `java.util.Collection` interface tells us what capabilities every collection class in Java must, at a minimum, provide. There are many concrete implementations of this interface in the Java library. Let us pick one implementation, the `java.util.LinkedList`. How can initialize our variable named `_contents`, of type `java.util.Collection<Grocery>`, so that it refers to an instance of `java.util.LinkedList<Grocery>`?

```
_contents = new java.util.LinkedList<Grocery>();
```

Suppose now that `Milk`, `Bread`, `Butter`, `Jam`, `Egg`, and `Coffee` are classes, all subtypes of `Grocery`. How do we add one instance of each to the `_shoppingCart`? We need to define a method on the `ShoppingCart` that adds an item to the cart's `_contents`:

```
      public boolean addItemToCart(Grocery grocery) {
            return _contents.add(grocery);
      }
```

With this method in place, we can add items to the shopping cart as follows:

```
_shoppingCart.add(new Milk());
_shoppingCart.add(new Bread());
_shoppingCart.add(new Butter());
_shoppingCart.add(new Jam());
_shoppingCart.add(new Egg());
_shoppingCart.add(new Coffee());
```

The collection `_contents` now holds a reference to each of these objects. Exactly how this works (i.e. how the collection `_contents` holds these references) we will not explore here – that is a topic for a later course.

7.6 Iterators

Now that we have a collection of objects, we might well wonder how we go about doing things with those objects. One way to gain access to the elements of a collection is to iterate through the collection. To iterate through a collection means to visit each member of the collection, one element at a time, until each member has been visited exactly once.

Java's collection framework supports iteration through collections by way of iterators. An iterator is an object which can iterate through a collection. Where does an iterator come from? From the collection itself. The `java.util.Collection` interface requires that every implementation provide a method named `iterator` which, when called, will return a new iterator object which can iterate over the collection.

An iterator must implement the `java.util.Iterator` interface. Of the methods in this interface, we are interested in two: `hasNext()` and `next()`. The `hasNext()` method returns `true` if there are items in the collection which have not yet been visited, and `false` otherwise (i.e. if all members of the collection have been visited). The `next()` method returns a reference to the next unvisited object from the collection, if there is one.[2] If the iterator is iterating over a collection of elements of type `E`, then `next()` returns a reference to an object of type `E`.

7.7 A conditional statement

We just said that a safe way to work with an iterator is to check the value of the `hasNext()` method, and perform different actions depending on whether the method returned `true` or `false`. But how can we do this? The only decision-making mechanism

[2] More precisely, if calling `hasNext()` would return `true`, then calling `next()` returns a reference to the next object in the iteration of this iterator. If calling `hasNext()` would return `false`, then calling `next()` does not return a value; instead, a `NoSuchElementException` is thrown. These notes have not discussed exceptions, so for our purposes it is best to always check the return value of `hasNext()`, and only call `next()` if `hasNext()` does in fact return `true`.

we have seen to this point is polymorphism, which allows us to select a particular method implementation to execute based on the type of object which the method is invoked on. Since `true` and `false` are not objects, we cannot attach methods to them and we cannot use polymorphism to select. Because Java has primitive types which fall outside its object model, Java has some special bits of syntax to allow for decision-making based on values (rather than types).

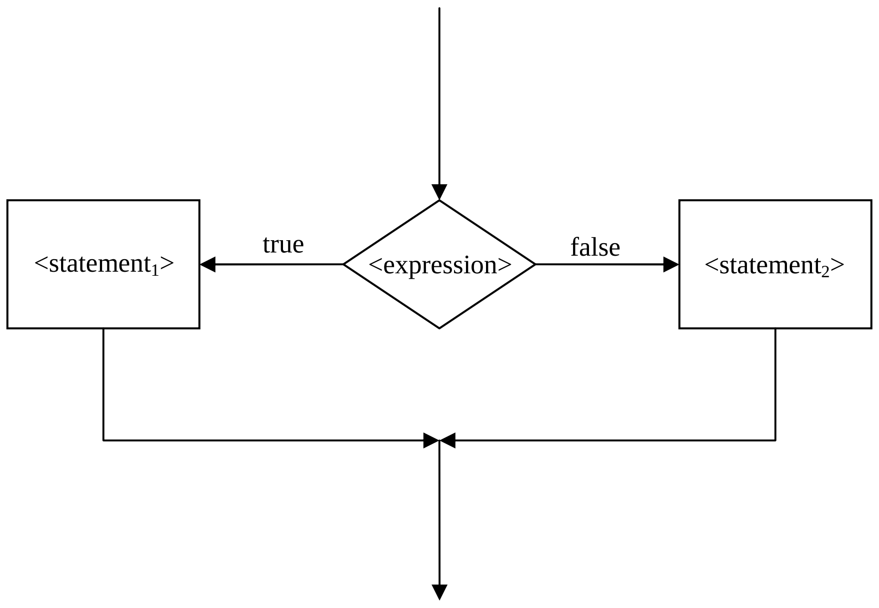
The basic conditional statement is the if-else statement. Syntactically the if-else statement has the following form:

```
if ( <expression> ) <statement1> else <statement2>
```

Let us first deal with the syntax of the if-else statement. ‘`if`’ is a reserved word, as is ‘`else`’. The parentheses around `<expression>` are required. The first statement, `<statement`$_1$`>`, is called the then-clause, while the second statement, `<statement`$_2$`>`, is called the else-clause.

Next we must talk about the semantics of the if-else statement: how is it executed? The whole point of the if-else statement is to selectively execute exactly one of the statements, either the then-clause or the else-clause, depending on whether the value of `<expression>` is `true` or `false`. If `true` the then-clause is executed, whereas if it is `false` the else-clause is executed.

Graphically, we can use a flow chart to convey how control flows through an if-else statement:



The lines with arrowheads in the diagram indicate the flow of control of the program through the flow chart. The diamond-shaped box denotes a decision – in the if-else statement this is the (boolean) expression. If the value of the expression is true, control flows through the left branch, whereas if its value is false control flows through the right branch. In either case, the flow of control continues with the next statement in the program.

How can we use this to help control our iterator? We can use it to "protect" a call to `next()` as follows:[3]

[3] This code example shows that the `<statement>` in both the then-clause and the else-clause can be a compound statement, surrounded by '`{`' and '`}`'. It is always a good idea to use a compound statement in an if-else statement, even if one of the clauses of the statement only contains one statement: it makes the code clearer, because each clause is clearly delimited by the curly braces, and it makes it less error-prone to add more statement to one of the clauses, because the statements will automagically be grouped together correctly inside the curly braces.

```
java.util.Iterator<Grocery> iterator = _myShoppingCart.iterator();
if ( iterator.hasNext() ) {

    // Here we know that there are more unvisited items, so call to
    // iterator.next() is safe:

    Grocery grocery = iterator.next();

    // Do something with grocery.

}
else {

    // No more unvisited items in the grocery cart.
    // Do something different.

}
```

7.8 A repetition statement

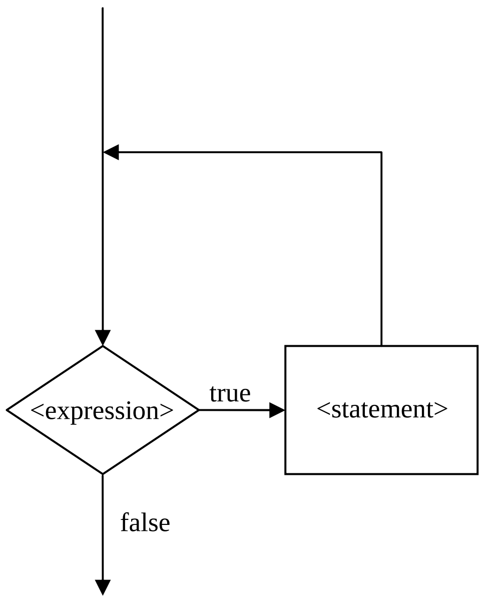
We will discuss one last statement type – a repetition statement which repeats a statement until some (boolean) condition is satisfied.

while loop

Let us start with a while-loop. A while-loop has the following syntactic form:

```
while ( <expression> ) <statement>
```

The corresponding flow-chart is:



What are the semantics of the while-loop? At first `<expression>` is evaluated. If `<expression>` evaluates to `false`, then execution continues on with the next statement after the for-statement. If `<expression>` evaluates to `true`, then `<statement>` is executed, after which `<expression>` is evaluted again, and the cycle either repeats (if `<expression>` evaluates to true) or not (if `<expression>` evaluates to false).

What can we do with the while-loop? One thing we can do is iterate through a collection. Why might we want to do that? Consider the problem of calculating the total cost of all the items in our shopping cart. To compute the total cost, we need to add up the cost of each item in the cart. Here's one way to do that (assuming that a `Grocery` item has a `getPrice()` method that returns a `Money` object):

```
public Money totalCost() {
      Money cost = new Money(0,0);
      java.util.Iterator<Grocery> iter = _contents.iterator();
      while ( iter.hasNext() ) {
            cost = cost.add(iter.next().getPrice());
      }
      return cost;
}
```

Notice that prior to the while loop the `_contents` collection is asked for an iterator. The status of the iterator is checked as a condition for continuing the while-loop: as long as the iterator has more items to visit (i.e. `iter.hasNext()` returns `true`), the body of the loop is executed. Once `iter.hasNext()` returns `false`, indicating that there are no more unvisited items in the collection, the while loop exits, and the next statement to be executed in the code above is:

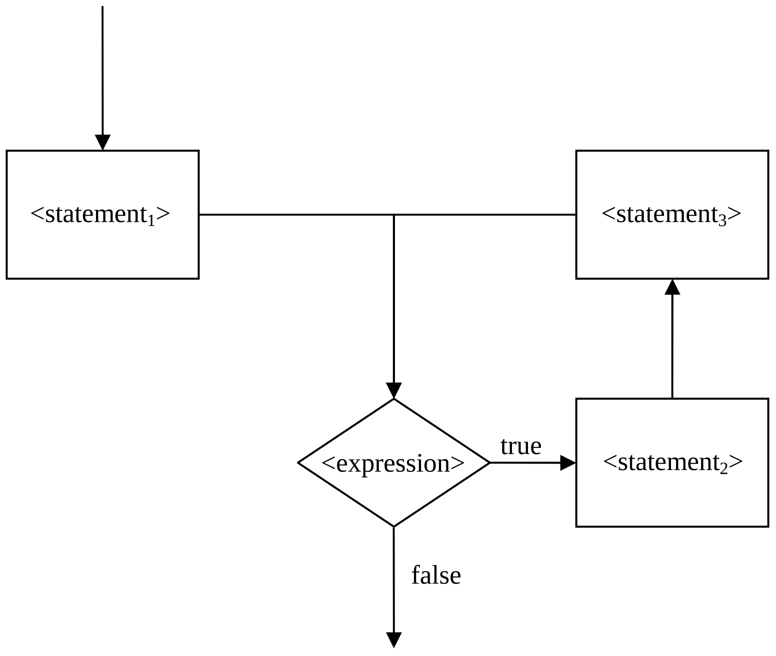
```
        return cost;
```

for loop

Another common sort of loop is the for-loop. A for-loop has the following syntactic form:

```
for ( <statement1> ; <expression> ; <statement2> ) <statement3>
```

This is syntactically a little bit more complex than the while-loop syntax, so we will explain each part in turn. To help explain how the for-loop works, it helps to have the corresponding flow-chart handy:



What are the semantics of the for-loop? At first `<statement`$_1$`>` is executed. `<statement`$_1$`>` is referred to as an initialization statement, because it is typically used to set up the initial conditions for the loop.

After `<statement`$_1$`>` is executed, `<expression>` is evaluated. If `<expression>` evaluates to `false`, then the for-loop exits and execution continues on with the next statement after the for-statement. If `<expression>` evaluates to `true`, then `<statement`$_2$`>` is executed, followed by `<statement`$_3$`>`. After this “loop”, `<expression>` is evaluted again, and the cycle either repeats (if `<expression>` evaluates to true) or not (if `<expression>` evaluates to false).

`<statement`$_2$`>` is the main body of the loop, while `<statement`$_3$`>` is an “update” statement. This sort of loop is often used to loop a fixed the number of times, in which case the “update” consists of incrementing a counter variable.

Here's a silly example of how to use the for-loop, to add 30 loaves of Bread to the shopping cart:

```
for (int i=0; i<30; i = i+1) {
    _cart.addItemToCart(new Bread());
}
```

What is `int`? Like `boolean` it is a primitive type, whose values are integers like 0, 1, -3, and 5732894. The set of `int` values is finite. It is not important for us right now to know what the limits are (you can check in a Java reference book if you're curious). What sorts of things can we do with integer values? We can create new integer values by adding them together (i.e. `i+1` is the integer value that is one more than `i`), and we can compare them (i.e. `i<30` is true if and only if the value of `i` is less than 30).

foreach loop

There is one more loop construct that is convenient to know about, the so-called foreach-loop. The foreach-loop implicitly uses an iterator to loop through a collection; the syntax is a little nicer than the while-loop, because it is not necessary to manage the iterator explicitly yourself. For example, the `totalCost` method, which used a while-loop, could be written using the foreach-loop syntax as follows:

```
      public Money totalCost() {
            Money cost = new Money(0,0);
            for ( Grocery item : _contents ) {
                  cost = cost.add(item.getPrice());
            }
            return cost;
      }
```

The variable `item` is declared to be of the same type as is contained in the `_contents` collection. It is assigned to each object in `_contents` in turn. Remember that there is in fact an iterator implicitly in the code – Java implicitly does the following:

```
java.util.Iterator<Grocery> iter = _contents.iterator();
while ( iter.hasNext() ) {
      Grocery item = iter.next();
      cost = cost.add(item.getPrice());
}
```

7.9 Mappings

This section will be a brief discussions of mappings in general (e.g. what they are and how they are used, *not* how they are implemented), and Java's `java.util.HashMap` in particular. Stay tuned!

Chapter Wrap-Up