1 Collections

The Collections is a set of classes found in the `java.util` package.

Compared with arrays, collection classes:

- Can hold a number of objects as elements (arrays can store both primitives and objects).
- They can have an unlimited number of objects. The size of a collection increases dynamically as more objects are added into it.
- Although primitives cannot be stored directly, wrapper classes can be used to store the value of a primitive in a wrapper object. Such an object is then stored in a collection class.

2 The ArrayList class

An example of a class belonging to Java Collections. An unlimited number of objects can be stored in an `ArrayList`.

Example:

```java
import java.util.*;

public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<String> al = new ArrayList<String>();

        // Add three elements in the list
        al.add("aa");
```

al.add("bb");
al.add("ccc");

for (int i=0; i < al.size(); i++) {
    String s = al.get(i);
    System.out.println(s);
}

// remove second element from the list
al.remove(1);

System.out.println("After remove(), al contains:");
for (String e : al)
    System.out.println(e);
}
}

When the above program is run, it displays:

aa
bb
ccc
After remove(), al contains:
aa
ccc

Note that ArrayLists which can only store a specific type, such as in the above example (al holds string objects), are called generics or parameterised types. This is because prior to Java 1.5, only non-parameterised collections classes were allowed. Non-parameterised collections can store objects of any type (even a mixture of objects belonging to different classes).

3 Wrappers and Storing Numbers in Collections

Primitives cannot be stored directly in a collection class. The Java library contains wrapper classes which correspond to primitives, as their objects are capable to store a primitive value.

The wrapper classes are:

- Byte
- Short
- Integer
- Long
- Float
- Double
• Boolean
• Character

The code below creates an object of `Double` class, which holds a double value.

```java
Double d1 = new Double(3.1);
Double d2 = 3.1; // automatically creates a Double object from 3.1
```

The two above statements are equivalent. The second form can only be used in Java version 1.5 or later. The automatic conversion of a primitive to the corresponding wrapper class (e.g. `double` to a `Double`) is called autoboxing, while the opposite conversion is called unboxing (e.g. automatically converting a `Double` to a `double`).

**Example:**

```java
import java.util.*;

public class WrapperExample {
    public static void main(String[] args) {
        // create an ArrayList object storing Double objects
        ArrayList<Double> a = new ArrayList<Double>();

        Double d1 = new Double(5.4);
        a.add(d1);
        a.add(11.2); // autoboxing occurs (double -> Double conversion)

        //a.add(new Integer(2)); // Error!
        // get the second element from arraylist
        Double d2 = a.get(1);
        // get the 1st element - unboxing occurs (Double -> double conversion)
        double d3 = a.get(0);

        System.out.println("d2=\n" + d2 + " , d3=\n" + d3);
    }
}
```

When the above example is run, it displays:

d2=11.2, d3=5.4

4 Non-parameterised ArrayLists

Although non-recommended, an `ArrayList` object can be declared to store objects of any type. In such cases, explicit casting is required when obtaining an element from the list.

This is illustrated in the example below:
import java.util.*;

public class ArrayListExample2 {
    public static void main(String[] args) {
        ArrayList l1 = new ArrayList();
        l1.add(new Integer(11));
        l1.add(new Integer(3));
        l1.add(new Integer(55));
        for (int i=0; i < l1.size(); i++) {
            Integer k1 = (Integer) l1.get(i); // Cast is required!
            System.out.println(k1);
        }
    }
}

Getting an element from list in l1 requires a cast. If the cast is omitted, the compiler will report an error.

5 Collections vs Arrays

One might wonder why not choose a Collection class (e.g. an ArrayList) as opposed to an array, all of the time. After all, collections provide an unlimited capacity to store elements.

The reason behind choosing arrays instead of collections, is computational efficiency. If the number of elements to be stored in a data structure is constant, then arrays should be preferred.

The computational efficiency of arrays over collections has to do with two main reasons:

1. As collections provide an unlimited capacity, when the initial capacity that a collection class has allocated is reached, additional memory should be allocated to increase the capacity. The allocation of extra capacity from the JVM requires some extra CPU time.

2. Some collections, are implemented in such a way that random access of elements is not possible. This means, that to access e.g. the third element, all elements before the third one must be traversed. A LinkedList is implemented in this way.

   This is not true for ArrayLists whose internal implementation use an array to store the elements. Therefore there is no time overhead associated with the random access of an element in ArrayLists.

6 Methods with variable number of arguments

Methods can be declared with a variable number of arguments (i.e. a non constant number of arguments). This is only possible in version 1.5 (J2SE 5.0) or later.

The ellipsis symbols ... is used to indicate that a method can receive any number of arguments.
In the example below, method `foo1` accepts any number of doubles. Method `foo2` accepts one `String` followed by one `int`, followed by zero or more `doubles`.

```java
public class VariableArgumentsExample {
    void foo1(double... x) { }

    void foo2(String s, int i, double... y) {
        // print all arguments
        System.out.println(s);
        System.out.println(i);
        for (double e : y)
            System.out.println(e);
    }

    public static void main(String[] args) {
        VariableArgumentsExample v = new VariableArgumentsExample();

        v.foo1(4.2);  // OK
        v.foo1();    // OK
        v.foo1(5.5, 7.7); // OK
        v.foo2("Ar", 5); // OK
        v.foo2("Ba", 7, 5.1); // OK
        v.foo2("Cd", 9, 1.5); // OK
    }
}
```

The above example produces the output:

```
Ar
5
Ba
7
5.1
Cd
9
1.5
```

Note that only one ellipsis `...` is permitted per method. The ellipsis can only be present in the last argument of a method. Thus, the following methods are not valid, and the compiler will produce an error:

```java
void foo3(String... a, double... d) { } // Error! Only one ellipsis allowed

void foo4(String... a, double d) { } // Error! Ellipsis can only be in last argument
```

An argument accepting an arbitrary number of elements is represented internally as an array. In the above example, the formal parameters `x` and `y` of methods `foo1` and `foo2` respectively are arrays whose size is decided when the methods are called.
7 Static Methods

A method declared as static can be called on a class, without the need to create an object of a class. Static methods are sometimes called class methods.

```java
class A {
    static int bar() {
        System.out.println("bar() called!");
        return 0;
    }
}

class Book {
    static int numberOfBooks;
    int numberOfPages;

    Book(int pages) {
        ++numberOfBooks;
        numberOfPages = pages;
    }
}
```

As the example indicates, a static method can also be called on an object of the class.

Methods should be static when the function they perform is not associated with an individual object, but rather with the class itself. It is common to have static methods in utility classes. For example, consider a class Calculator providing methods `add(int, int)`, `subtract(int, int)`, etc. Such methods do not depend on objects of the class, but they are used to perform arithmetic operations between their arguments. Therefore, they should be declared as static.

8 Static Data

Class fields which are declared static, are shared among all objects of the class. This means, that a single instance of the field will be created, independent of the number of objects of the class. This is illustrated in the example below, and shown in Figure 1.

```java
class Book {
    static int numberOfBooks;
    int numberOfPages;

    Book(int pages) {
        ++numberOfBooks;
        numberOfPages = pages;
    }
}
```

As the example indicates, a static method can also be called on an object of the class.
public static void main(String[] args) {
    Book b1 = new Book(10);
    Book b2 = new Book(20);
    Book b3 = new Book(5);

    System.out.println("b1.numberOfBooks: " + b1.numberOfBooks);
    System.out.println("b2.numberOfBooks: " + b2.numberOfBooks);
    System.out.println("b3.numberOfBooks: " + b3.numberOfBooks);
    System.out.println("Book.numberOfBooks: " + Book.numberOfBooks);

    System.out.println("b1.numberOfPages :" + b1.numberOfPages);
    System.out.println("b2.numberOfPages :" + b2.numberOfPages);
    System.out.println("b3.numberOfPages :" + b3.numberOfPages);
    // System.out.println(Book.numberOfPages); // Error!
}

Figure 1: Static fields are shared among all objects of a class. Non-static fields (instance fields) are unique for each object of the class.

The output of the StaticFieldsExample is:

b1.numberOfBooks: 3
b2.numberOfBooks: 3
b3.numberOfBooks: 3
Book.numberOfBooks: 3
b1.numberOfPages : 10
b2.numberOfPages : 20
b3.numberOfPages : 5