Programming
In Java

BOOK 3

Packages, Abstraction and Exceptions

CONTENTS

1. PACKAGES, CLASSPATH AND THE IMPORT STATEMENT

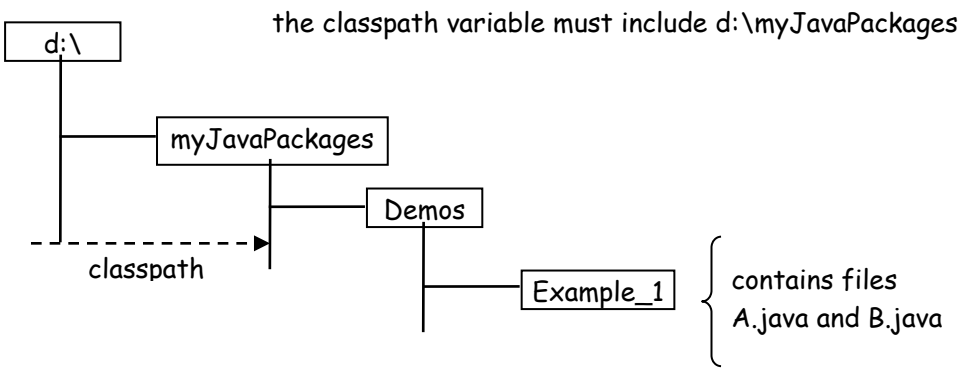
1.1 Packages

A package is a collection of (related) classes. You make a class a member of a package by including the package statement as the first line in the class definition file. If the package statement is omitted from a class definition the class is added to the default package, the package with no name. Packages are organised in a hierarchy structure similar to the directory structure created when the Java system is installed. All the classes in a particular package are stored in the same directory. For example the classes in the java.lang package can be found in the directory ..\java\lang\ and in particular the String class can be found as ..\java\lang\String.java . Furthermore you can refer to a method as java.lang.System.out.println("Hello").

1.2 First Approach - Example 1

In the following explanation we assume the Java SDK is located in c:\jdk1.3.

A Java package is almost the same thing as a directory… it's a group of files; try this to get the feel of packages,



- create the above directory structure
- In the Example_1 directory create the following two Java classes
- the line **package Demos.Example_1;** makes the files in the Example_1 directory, members of the Demos.Example_1 package
- the name of the package is **Demos.Example_1**

```
package Demos.Example_1;
public class A {
  public A() {
    System.out.println("This is class A constructor.");
      this.aMethod("Called from class A constructor.");
  }

  public void aMethod(String aMsg) {
    System.out.println("This is aMethod() of class A... " + aMsg);
  }
}
```

```
package Demos.Example_1;
public class B {
  public B() {
     System.out.println("This is class B constructor.");
      this.bMethod("Called from class B constructor.");
  }

  public void bMethod(String aMsg) {
    System.out.println("This is bMethod() of class B... " + aMsg);
  }
}
```

- Use javac to compile the two classes making sure that the class files finish up in the directory d:\myJavaPackages\Demos\Example_1
- this is now the home directory for the newly created package
- you have now created a package, consisting of just two classes
- create a d:\dump directory and create the following test class in the directory

```
import Demos.Example_1.*;
public class TestDemoPackages {
    public static void main(String args[]) {
        A a = new A();
        B b = new B();
```

```
    }
}
```

- there are now a number of classes involved in the 'project'
  the Demos.Example_1 package classes (A and B)
  the java system classes
  the test class
- when you run your program the JVM (java) needs to be able to find all these classes. This is the purpose of the environment variable **classpath**. This needs to be set so that the search path includes all the directories where these different class files can be found. So (somehow, depending on the way you are working) you should set the classpath variable to include the following directories.
  d:\myJavaPackages;d:\dump;c:\jdk1.3\lib\classes.zip

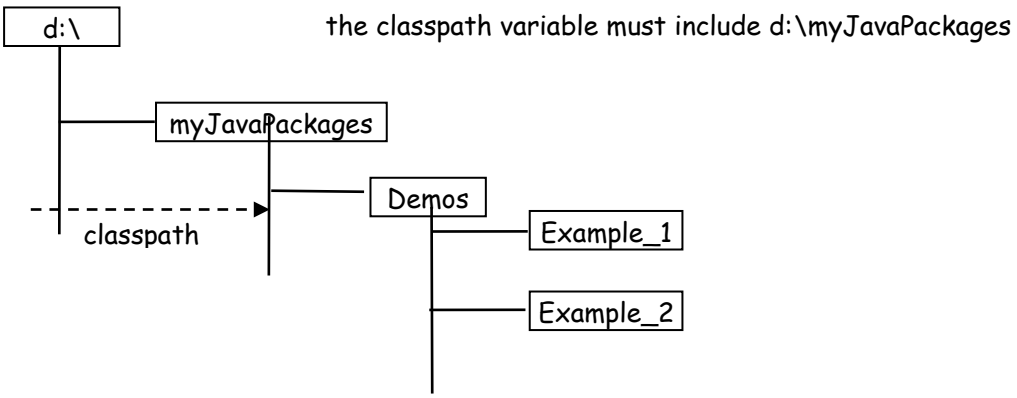
- **notice how the name of the package (Demos.Example_1) is the subdirectory path off the classpath directory**

- use javac/java to compile/run this test class making sure that the class file finishes up in the dump directory.

Did all go well ?

1.3 Package Scope - Example_2

- Create a new directory Example_2 as below,



- then create a new package **Demos.Example_2** with the following two files

```
package Demos.Example_2;
public class A {
  private C c;
  public A() {
     System.out.println("This is class A constructor.");
     c = new C();
     this.aMethod("Called from class A constructor.");
  }

  public void aMethod(String aMsg) {
     System.out.println("This is aMethod() of class A... " + aMsg);
     c.cMethod("called from aMethod() of class A.");
  }
}
```

```
package Demos.Example_2;
public class B {
  public B() {
     System.out.println("This is class B constructor.");
     this.bMethod("Called from class B constructor.");
  }

  public void bMethod(String aMsg) {
     System.out.println("This is bMethod() of class B... " + aMsg);
  }
}

class C {
  public C() {
     System.out.println("This is class C constructor.");
     this.cMethod("Called from class C constructor.");
  }

  public void cMethod(String aMsg) {
     System.out.println("This is cMethod() of class C... " + aMsg);
```

```
    }
}
```

- note that class C is defined within the file B.java and that it's scope is by default 'the package'
- this means that only classes belonging to the package Demos.Example_2 can see class C and hence create and use instances of class C
- class A creates and uses an instance of class C
- compile all classes again making sure that the package classes finish up in the Example_2 directory

```
import Demos.Example_2.*;
public class TestDemoPackages {
    public static void main(String args[]) {
        A a = new A();
        B b = new B();
    }
}
```

- if you run the test class , all should be well.
- Now change the test class so that it attempts to make a reference to a class C instance

```
import Demos.Example_2.*;
public class TestDemoPackages {
    public static void main(String args[]) {
        A a = new A();
        B b = new B();
        C c = new C();
    }
}
```

you will probably get an error message along the following lines,

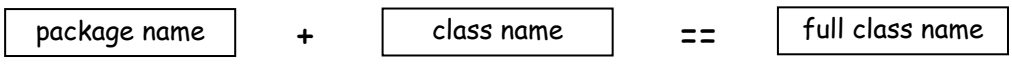
```
TestDemoPackages.java:6: Can't access class Demos.Example_2.C. Only public classes and interfaces in other packages can be accessed.
        C c = new C();
        ^
TestDemoPackages.java:6: Can't access class Demos.Example_2.C. Only public classes and interfaces in other packages can be accessed.
        C c = new C();
                ^
2 errors
```

1.4 A Personal Note

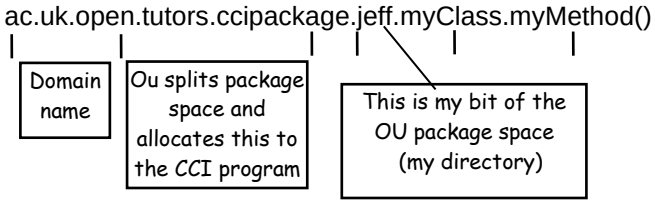
When developing packages they can be developed in 'any' directory but of course must finally be located in the appropriate directory off the classpath so that Javac and Java can find them. However more generally I found this approach to be a little bit messy and sensitive when there are references between the package classes. ie when one class references an object of another package class. For this reason I tend to develop package classes from within their package directory.

1.5 The Fully Qualified/Full Name of Class

You should understand that a java class is completely specified by its fully qualified name as follows,



Sun has extended this 'path' reference to a more formal procedure. With the Internet in mind it is now possible to use Internet domain names as a top level reference to a package, and hence a class, and hence even a method.



1.6 The System Environment variable - classpath

The classpath environment variable tells the JDK programs (eg Javac and Java) where to find Java classes. You must set the classpath variable to identify the location of your classes. You set the classpath variable depending upon your operating system.

1.7 The Import Statement

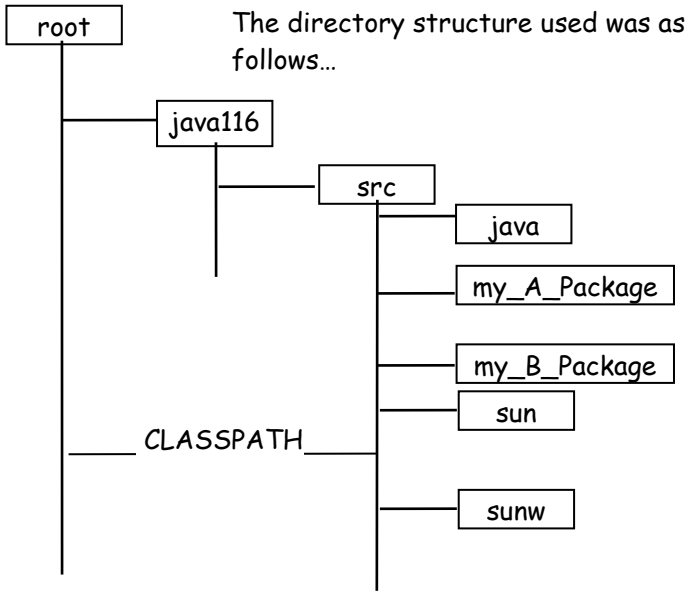
You use this to import classes that your current class needs. This will allow for the possibility of importing all the public classes directly in the package named but not any of its sub-packages. You must name the package (ie the sub-package) directly. The use of something like
import package.*; does not imply a 'wildcard type' use for the *. It is a selective import in that only the classes referred to in your code are actually imported and so the use of package.* is no less efficient that naming the class directly with something like import **package.myClass;**

Remember that you import a class and not a package. Hence trying something like
import java.util;
will not do since java.util is a package. You must do something like,
import java.util.Vector;
to specify the particular class you want to import (ie use). Or you can do something like
import java.util.;*
so that all the classes are imported, or at least those classes that are referenced in the current file since importing is a selective process.

1.8 Creating The Package

This is one possible strategy for creating a package, there are of course others… but by following through the way things have been done should give you enough clues to allow you to do things differently if you want to.

- First of all you need to create a directory to hold the classes that will form the package. This directory must be the same name as the package. I used my_A_Package and my_B_package as the package and directory names.



- You then need to create the source files that form your package. This demonstration uses 2 files for my_A_Package. Here they are. They were located in the my_A_Package directory.

```
package my_A_Package;

class A {
    private String theMessage;

    public A() {
        theMessage = "An ok... message from class A";
    }

    public void say() {
        System.out.println(theMessage);
    }
}
```

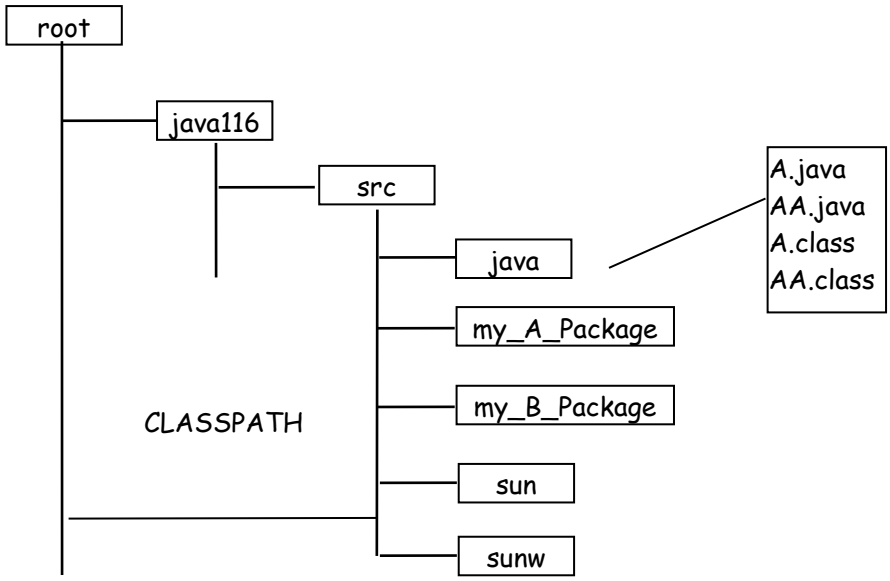
```
package my_A_Package;

public class AA {
    private A myA = new A();

    public void doMethodFromAA() {
        myA.say();
    }
}
```

Notice that the class A is a 'package' class, ie it's access modifier is 'missing' and hence defaults to a package scope. This means that class A can only be referenced by other classes in the same package. In this simple case only class AA. The class AA is made public so that it can be accessed from the test class PackageTest.java.

- Compile the source code,
  to compile A.java use… javac -classpath c:\java116\lib\classes.zip A.java
  and to compile AA.java use… javac -classpath c:\java116\lib\classes.zip AA.java

making sure the resulting class files are directed to the same directory (ie my_A_Package).

- the package is now completed, so you should have the following situation,



1.9 Testing The Package

create a test class something like the following,

```
import my_A_Package.*;
public class PackageTest {
    public static void main(String args[]) {
        AA anAAObject = new AA();
        anAAObject.doMethodFromAA();
    }
}
```

- compile TestPackage with
  javac -classpath c:\java116\lib\classes.zip TestPackage.java

The test class PackageTest.java was located in the directory c:\dump

- and finally run the test program using
  java -classpath .;c:\java116\lib\classes.zip TestPackage

Note how the classpath has been extended to include the current directory, so that TestPackage.class can be found as well as the package classes.

1.10 Scope Across Packages

Recall that the (default) scope of class A is it's package and that the scope of class AA is public. To verify the comments made in the section on scope we will create another package, my_B_Package and look at the scoping of the classes and methods within the packages.

```
package my_B_Package;
import my_A_Package.*;
class B {
    public static void main(String args[]) {
        AA myAA = new AA();
        //A myA = new A();
        myAA.doMethodFromAA();
```

```
        }
    }
```

If you try to include the line //A myA = new A(); you will probably get an error message from the compiler something like,

B.java:7: Can't access class my_A_Package.A. Only public classes and interfaces in other packages can be accessed.

```
A myA = new A();
```

B can't reference the (package) class A directly, but can reference the public class AA.

A crucial point here is the use of the fully qualified class name for class B, ie my_B_package.B *in the line passed to the Java interpreter.*

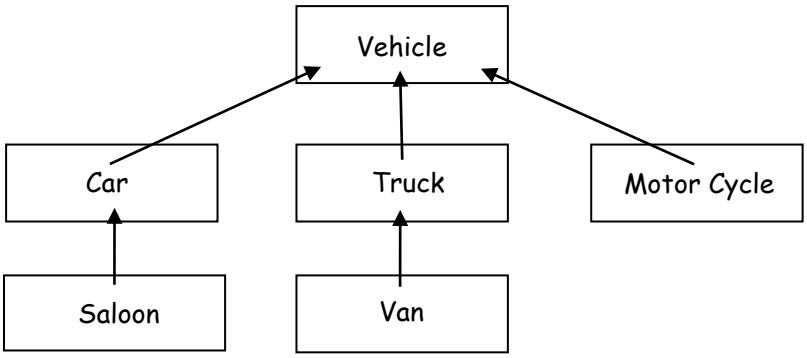
```
java -classname .;c:\java116\lib\classes.zip my_B_package.B
```

2. ABSTRACTION MECHANISMS

2.1 Abstract Classes

- An abstract class is used to specify a common message protocol for all its subclasses.
- Instances of abstract classes cannot be created.
- Subclasses may implement the common protocol in their own manner by overriding the inherited methods where necessary.
- Subclasses may also extend the inherited message protocol.
- A subclass of an abstract class may itself be an abstract class.
- Abstract classes specify behaviour for its subclass objects by providing a protocol but no instances.

In describing a collection of cars, trucks, motor cycles etc we note that some components of each are common. All have wheels, engines, colour, speed etc. All have engines that need to be started, all can be accelerated, stopped etc. Each is an example of a more general (abstract) class which we can call a vehicle class. While each of the cars, trucks, etc share these common attributes and actions they individually implement them in different ways. The vehicle class is a generic class from which all the other classes are derived. Adopting this standpoint in an object oriented programming application we might describe *the vehicle class as an abstract class.* As the classes 'move down' in the hierarchy they become more specialised. A Van is a special kind of Truck which is a special kind of vehicle.



An abstract class type is a class specifically designed to provide inheritable characteristics for it's descendent class types. The purpose of an abstract class is to have descendants and not instances. This means that we will never actually create, that is use, an instance of the abstract class. We are only interested in it's properties so that we can re-use them as we create new classes from it.

It is the case that the base class objects are never created directly or in fact are used directly. In object systems the base class is often created just to provide features that their derived classes can inherit.
Abstract classes correspond to general concepts, which are not easily translated into specific objects, but are useful for providing a description of all common features of objects that individually might be very different from one another. For this reason, even if abstract classes define their functionality's, they seldom implement them. The implementation is left to the individual objects.
In every subclass we define only those features that have been added to the class from which they are derived, without having to repeat the description of the common features. The elements present in all classes will be described only once in the generic class (the vehicle class). This class will provide a common interface to all the vehicles subclasses. This common interface, which often consists of mostly virtual methods, allows us to make a uniform reference to all subclasses in the hierarchy, which are always free to add new attributes and redefine the various methods as appropriate.

2.2 Abstract Methods

The designer of the Vehicle class knows that all vehicle types will want to understand the startEngine() message and is able to provide a 'part' implementation of the method which involves all functions common to the starting of all vehicle engines. However part of the start up process involves some engine management calculations which are special to the particular vehicle being started. Likewise the console display differs for different types of vehicle. The vehicle class designer can anticipate these differences and allow for them within the design of the startEngine() method by simply referencing them and declaring them as abstract methods to be implemented by the designer of the subclasses Car and Truck. Making this run-time link between an object and an abstract method is known as late-binding.

The actual calculation to be performed on the startEngine() parameters depends on the type of vehicle receiving the startEngine() message. Since this is not known at this stage the start engine methods doCalc() and display() cannot be defined and are made abstract. It is up to the various Vehicle subclasses to provide appropriate implementations

```
public abstract class Vehicle {
    // Possible constructor method and other methods
    public void startEngine(int a, int b, String vehicleType) {
        System.out.println("Starting " + vehicleType + " engine...");
        this.doCalc(a,b);
        this.display();
    }
    public abstract void doCalc(int x, int y);
    public abstract void display();
}
```

Each of the subclasses provides implementations of the abstract methods doCalc() and display()

```
public class Car extends Vehicle {
    public void doCalc(int x, int y) {
        System.out.println("This is the doCalc() of the Car class...");
    }
    public void display() {
        System.out.println("This is the display() of the Car class...");
    }
}

public class Truck extends Vehicle {
    public void doCalc(int x, int y) {
        System.out.println("This is the doCalc() of the Truck class...");
    }
    public void display() {
        System.out.println("This is the display() of the Truck class...");
```

```
    }
}
```

```
public class TestClass {
    public static void main(String args[]) {
        Car myCar = new Car();
        Truck myTruck = new Truck();
        myCar.startEngine(2,3,"Car");
        myTruck.startEngine(7,8,"Truck");
    }
}
```

```
Output from running TestClass is
Starting Car engine...
This is the doCalc() of the Car class...
This is the display() of the Car class...
Starting Truck engine...
This is the doCalc() of the Truck class...
This is the display() of the Truck class...
```

Abstract methods are a kind of replaceable method... they are replaced by the appropriate version belonging to the object that receives the message. An abstract message is contextual... it depends on the context in which it is used. Other languages (Pascal, C++) use the term 'virtual methods' to describe abstract methods. The previous example is a bit 'canned', but hopefully it explains late binding and convinces you of the possibilities of such abstraction mechanisms, especially from the class designers standpoint. A class designer can foresee the need for a particular kind of behaviour and include it in his class definition, but he has no need to be aware of the actual details of the behaviour. These can be left to the user of the class to complete at a 'later' stage... when the user is designing more specific classes by inheritance from the super classes.

Abstract classes cannot be instantiated. If a subclass does not implement an inherited abstract method it too remains an abstract class.

2.2.1 Abstract Method vs Overriding

If the abstract method is replaced by a method with an empty body and then overridden by the implementor of a subclass then effectively the same result is achieved. So what's the difference… the difference is that by declaring a method as abstract, this action forces subclass implementors to override the method otherwise no objects can be declared of the subclass. A subclass remains an abstract class until the abstract method is overridden. By using an empty body approach no enforcement is placed on the subclass implementor to override the method in question. This of course is a potential source of trouble.

2.3 Some Points

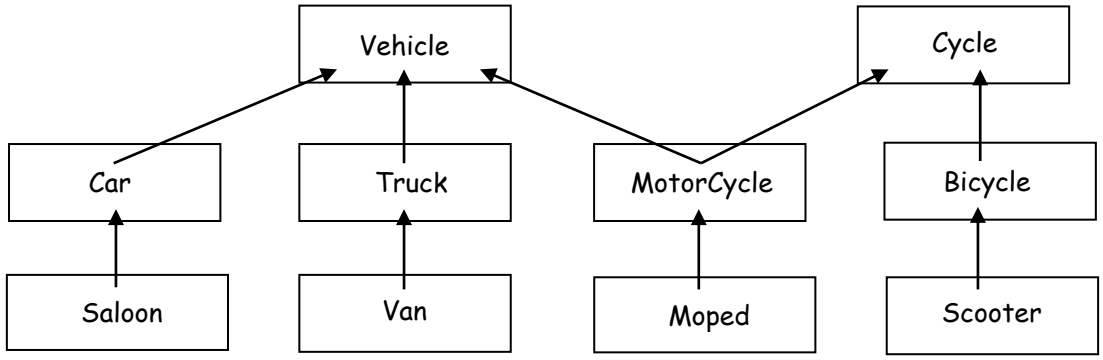
- Any class with an abstract method is automatically abstract itself and must be declared as such.
- An abstract class cannot be instantiated.
- A subclass of an abstract class can be instantiated only if it overrides each of the abstract methods of its superclass and provides an implementation (i.e., a method body) for all of them. Such a class is often called a concrete subclass, to emphasize the fact that it is not abstract.
- If a subclass of an abstract class does not implement all the abstract methods it inherits, that subclass is itself abstract.
- static, private, and final methods cannot be abstract, since these types of methods cannot be overridden by a subclass. Similarly, a final class cannot contain any abstract methods.
- A class can be declared abstract even if it does not actually have any abstract methods. Declaring such a class abstract indicates that the implementation is somehow incomplete and is meant to serve as a superclass for one or more subclasses that will complete the implementation. Such a class cannot be instantiated.

2.4 Interfaces

An interface is like an abstract class but one where **all** the methods in the interface are abstract methods.

2.4.1 Providing a Form of Multiple Inheritance

Java implements simple inheritance through it's class hierarchy structure with all classes subclassed from the Object class. Any class has only one immediate superclass. This restriction can in some situations prove too restrictive and so Java offers a partial solution and allows a kind of multiple inheritance. This situation is shown in the following diagram where the MotorCycle class inherits characteristics from both the Vehicle class and the Cycle class.



The above hierarchy can be achieved by using an interface.

Any variables defined in an interface are implicitly public, final and static. Being final means they are effectively constants to all the subclasses of the interface. (Hence the term variables seems contradictory…). Being static means they act as class variables. In other words there can be no such thing as an instance variable defined in an interface.

The methods defined in an interface are implicitly public and abstract.
So any of the following forms will do. Perhaps the middle one is best.

```
public interface Cycle {
  public static final int p = 10;
  public void cycleMethod1();
  public void cycleMethod2();
}
```

```
public interface Cycle {
  public static final int p = 10;
  public abstract void cycleMethod1();
  public abstarct void cycleMethod2();
}
```

```
public interface Cycle {
  int p = 10;
  abstract void cycleMethod1();
  abstract void cycleMethod2();
}
```

```
public class MotorCycle extends Vehicle implements Cycle {
    public void doCalc(int x, int y) {
        System.out.println("This is the doCalc() of the MotorCycle class...");
    }
    public void display() {
        System.out.println("This is the display() of the MotorCycle class...");
    }
    public void cycleMethod1() {
        System.out.println("Interface variable = " + p);
    }
    public void cycleMethod2() {}
}
```

2.4.2 Abstract Classes vs Interfaces

Note that with an interface all the methods are abstract. With an abstract class this need not be the case and in practice is unlikely to be the case. Abstract classes often provide a basic set of common behaviours but leave specialised behaviour for subclasses to implement.

2.4.3 Interfaces Provide Software Quality Control

Interfaces can act as a kind of software quality mechanism. Consider for example if we were in a software project and we had asked programmers to implement a number of sub-classes and that these subclasses

had to contain certain methods. We would have to carry out extensive quality assurance procedures, such as technical reviews, which checked that they had included these methods. When we use the keyword implements we tell the programmer what is expected: that methods from the interface are to be coded and, moreover, if the programmer does not carry out this instruction a Java error is created by the compiler. In this context the compiler enforces a quality check which might otherwise have been expensive to check.

2.4.4 Interfaces Can Provide System Constants

The variables declared in an interface body are available as constants to classes that implement the interface. This enables pools of related constants to be defined and made available to classes that require these constants. Variables declared in an interface are implicitly public, static and final and must be initialised at their declaration.

```
public interface Constants {
    String bsRef   = "BS8006";
    int  zType = 100;
}
```

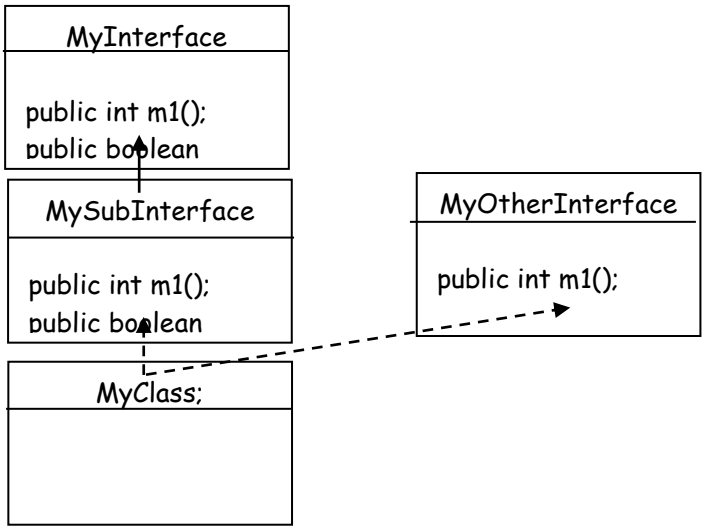
```
public class Truck implements Constants {
    public void doCalc(int x) {
        // zType = zType * 2; not allowed of course... zType is a final variable
        System.out.println("Z factor = " + x * zType);
    }

    public void display() {
        System.out.println("BS ref = " + bsRef);
    }
}
```

2.4.5 Inheritance and Interfaces Compared

A comparison can be made by noting that inheritance should be used to derive the new class when there is a structural relationship between the proposed new class and the existing class. The new class will exhibit similar structure to the existing class when it needs to have the same data fields and methods as the existing class but will further have need for some additional data fields and methods of it's own. An interface exists to provide common behaviour to those classes that implement the interface. Where a number of classes agree to implement an existing interface they are effectively agreeing to share similar behaviour, although each class will implement it own version of that behaviour. These classes are likely to have little in the way of common structure.

2.4.6 Inherited Interfaces ?

The following interface and class structure was created...




```
public interface MyInterface {
    public int m1();
  public boolean m2(int t);
}

public interface MySubInterface extends MyInterface{
    public String m3();
}
```

```
public class MyClass implements MySubInterface, MyOtherInterface
{
    public int m1() {
      return 0;
    }

    public boolean m2(int t) {
      return true;
    }
```

```
public interface MyOtherInterface {
    public int m1();
}
```

```
    public String m3() {
      return "all is ok...";
    }
}
```

The class MyClass compiles ok. Thus we can extend an Interface.
Although interesting it appears the redeclaration of m1() in MyOtherInterface is just a waste of time. The class MyClass has contracted to implement a method with the signature public int m1() and it has done so, whether this is by contract with MyInterface or with MyOtherInterface... it does not care !

2.5 Enumerations

A special kind of interface provided by the Java system is the Enumeration interface which provides the abstraction of two methods for stepping/scanning through an indexed set of objects. We say abstraction because as always with an interface you have to provide the implementation details of the abstract methods. This section will demonstrate another use of interface objects in the form of enumeration objects which are generally associated with scanning operations over a list of objects.

2.5.1 Example

We introduce the ideas of an Enumeration object by assumimg we are the designers of the following simple class that provides the means of storing and retrieving names.

```
public class NameList {
    private String[] theList;
    private int namePtr;

    public NameList(int size) {
        theList = new String[size];
        namePtr = 0;
    }

    public void add(String aName) {
        theList[namePtr++] = aName;
    }

    public int getNumNames() {
        return namePtr;
    }
}
```

One of the most likely common operations that a user of our class will want to perform is to scan/search the list. Clearly there are a number of ways this facility could be provided. We could leave the work up to the user. We could provide a means of retrieving a name with a method such as getElement(int pos) but note that this method would rely heavily on the fact that the storage structure for the names is an array. The user would need to know this and then provide the position of the name to be retrieved. If we, as designers, decided to change the way in which we held the names, ie change the data structure to perhaps a linked list, then this action may well make the users code redundant.

Java's approach is to provide a kind of standard way in which such a scanning feature can be implemented and in particular to remove the need for the user to know the way the data was structured within our class. This is achieved by use of the Enumeration interface. Some of the Java API collection classes use this approach. eg the Vector and Hashtable classes.

The Enumeration interface, defined in the API util package, provides the abstractions of two methods…

public abstract boolean hasMoreElements()
Tests if this enumeration contains more elements.
Returns true if this enumeration contains more elements,
false otherwise.
public abstract Object nextElement()
Returns the next element of this enumeration.

The normal approach is in two steps,
1. Define a new class as implementing the Enumeration interface and make the class file of this new class available to the user. This new class will provide the implementation details of the scanning methods, hasMoreElements() and nextElement(), from the Enumeration interface.
2. Add a method to the target class, in this case our NameList class, to allow the user to create one of these Enumeration objects.

This means you have to provide a new class something like the following,

2.5.2 Defining the Enumeration Sub Class

```
import java.util.*;
public class NameListEnumeration implements Enumeration {
    private int ptr;
    private int numElements;
    private String[] theList;

    public NameListEnumeration(String[] aList, int n) {
        ptr = 0;
        theList = aList;
        numElements = n;
    }

    public boolean hasMoreElements() {
        return ptr < numElements;
    }

    public Object nextElement() {
        return theList[ptr++];
    }
}
```

and then include an additional method in the target class,
public Enumeration elements() {
which the user will use to create a NameListEnumeration object.

2.5.3 Adding a Method to the Target Class

```
import java.util.*;
public class NameList {
    private String[] theList;
    private int namePtr;

    public NameList(int size) {
        theList = new String[size];
        namePtr = 0;
    }

    public void add(String aName) {
        theList[namePtr++] = aName;
    }

    public int getNumNames() {
```

a bit of explanation is required here... the elements() method is shown as returning an Enumeration object in its signature but actually creates a NameListEnumeration object. Because of substitution, a subclass can stand in for a super class... so rather than return the object as a NameListEnumeration object we return it as an Enumeration object. We do this so that as Java programmers we follow an adopted standard of using Enumeration objects as scanning objects. This way any code involved in scanning will have a standard look/feel about it, ie always something like,

```
Enumeration e = anObject.elements();
while (e.hasMoreElements()) {
    ---- e.nextElement();
}
```

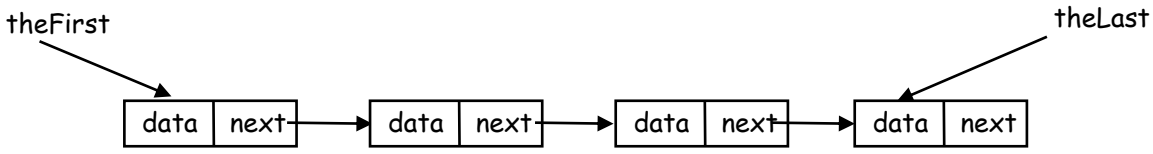
```
        return namePtr;
    }

    public Enumeration elements() {
        return new NameListEnumeration(theList,this.getNumnames());
    }
}
```

The new class and added method will allow us to use the Enumeration object as follows,

```
public class NameListTest {
    public static void main(String arg[]) {
        NameList theList = new NameList(10);
        theList.add("Jeff"); theList.add("Pat");
        theList.add("John"); theList.add("Debbie");
        Enumeration e = theList.elements();
        while (e.hasMoreElements()) {
            System.out.println(e.nextElement());
        }
        // add another name...
        theList.add("Roger");
        // The list has now changed, how do we now get another scan of the list
        // to show all 5 names ?
    }
}
```

2.5.4 A Linked List Example

The following classes implement a linked list of nodes. The data items stored at the nodes are strings. It is a familiar data structure and is often represented as follows,



2.5.5 Implementation of a Linked List

```
public class LinkedList {
    private Node theFirst, theLast;
    private int nodeCount;
    public LinkedList() {
        theFirst = null; theLast = null;
        nodeCount = 0;
    }
    public void push(String s) {
        Node temp = new Node(s);
        if (this.isEmpty()) theFirst = temp;
        else theLast.setNext(temp);
        theLast = temp;
        nodeCount++;
    }
    public String pop() {
        if (this.isEmpty()) return null;
        else {
            String temp = theFirst.getData();
            theFirst = theFirst.getNext();
            nodeCount--;
            return temp;
        }
    }
    public Node getFirst() {return theFirst;}
    public Node getLast() {return theLast;}
    public int getNodeCount() {return nodeCount;}
    public boolean isEmpty() { return nodeCount == 0;}
    public Enumeration elements() {
        return new ListEnumerator(this.getFirst());
    }
}
```

```
public class Node {
    private String data;
    private Node next;

    public Node(String s) {
        data = new String(s);
        next = null;
    }

    public void setData(String s) {
        data = new String(s);
    }

    public void setNext(Node p) {
        next = p;
    }

    public String getData() {
        return data;
    }

    public Node getNext() {
        return next;
    }
}
```

We now define an Enumeration object to scan the list. This requires the definition of a new ListEnumerator class which implements the Enumeration interface.

```
import java.util.*;
public class ListEnumerator implements Enumeration {
    private Node currentNode;

    public ListEnumerator(Node p) {
        currentNode = p;
    }

    public boolean hasMoreElements() {
        return currentNode != null;
    }

    public Object nextElement() {
        String temp = currentNode.getData();
        currentNode = currentNode.getNext();
        return temp;
    }
}
```

When providing a class with an Enumeration object the standard approach is to include a method in the target class that creates the Enumeration object. In our case we add a method called **elements()** to our LinkedList class. The user can then use this to create an Enumeration object on their linked list.

```
public class ListTest {
    public static void main(String args[]) {
        LinkedList theList = new LinkedList();
        theList.push("Java"); theList.push("Pascal");
        theList.push("C++");  theList.push("Basic");
        theList.push("Smalltalk");
        ListEnumerator e = theList.elements();
        while (e.hasMoreElements()) {
            System.out.println(e.nextElement().toString());
        }
        theList.pop();
        e = theList.elements();  // note the need to create another ListEnumerator object
        while (e.hasMoreElements()) {
            System.out.println(e.nextElement().toString());
        }
    }
}
```

The enumeration is consumed by use; its values may only be counted once, hence note the need to create another ListEnumerator object for the second scan of the list.

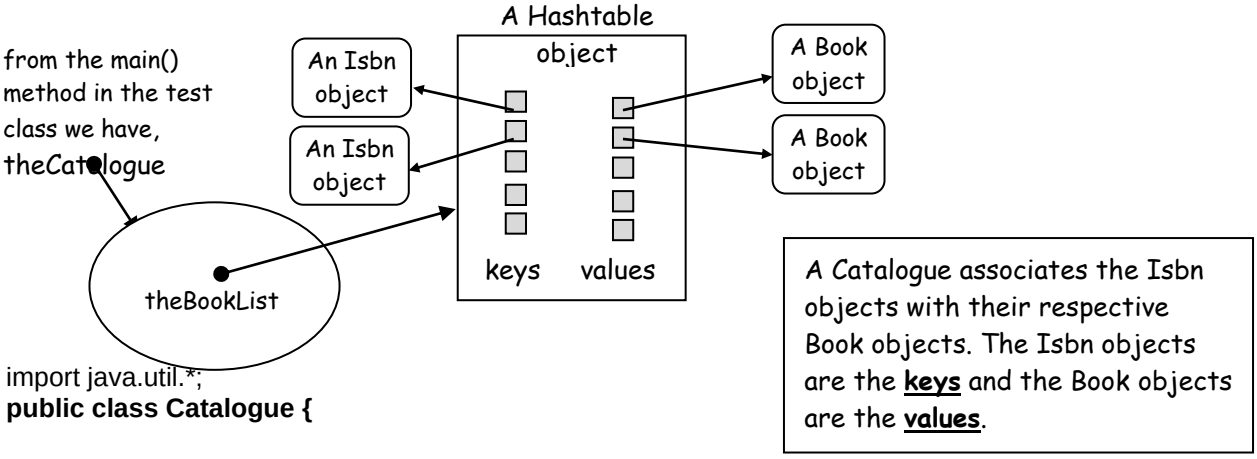
One feature to note is that although we have provided the user of our LinkeList class with the ability to scan the elements of the list, there are no clues as to how our list is implemented. Our user simply needs to understand the purpose of the scanning methods, hasMoreElements() and nextElement().

3. THE HASHTABLE CLASS

The Hashtable class implements a hash table data structure. A hash table indexes and stores objects in a dictionary using hash codes as the objects' keys. A Hash codes is an integer value that identify the objects position in the hash table. They are computed in such a manner that different objects are very likely to have different hash values and hence different positions in the hash table. The mathematical algorithms used to compute hash codes can generate clashes in which case an overflow strategy is invoked. That is what to do when a slot is already occupied.

The Java Hashtable class is very similiar to the Dictionary class from which it is derived. Objects are added to the hashtable as key-value pairs. The object used as the key is hashed, using its hasCode() method and the hash code is used as the actual key for the value object. When an object is added to or retrieved from the hashtable using a key, the key's hash code is computed and used to find a slot for the object or to find the object.

3.1 An Introductory Example




```
import java.util.*;
public class Catalogue {
```

```
    private Hashtable theBookList;

    public Catalogue() {
        theBookList = new Hashtable();
    }
    public void addBook(Book aBook) {
        theBookList.put(aBook.getISBN(),aBook);
    }
    public boolean includes(Book aBook) {
        return theBookList.containsKey(aBook.getISBN());
    }
    public String toString() {
        Book aBook;
        String tempStr = "";
        Enumeration e = theBookList.keys();
        while (e.hasMoreElements()) {
            aBook = (Book)theBookList.get(e.nextElement());
            tempStr += aBook + "\n";
        }
        return tempStr;
    }
}
```

this method creats a key in the hastable and inserts the Book object as the associated object

this method checks if a Book object with the Isbn value is in the catalogue

```
public class Book {
    private String author;
    private String isbn;
    private String title;

    public Book(String anAuthor, String anISBN, String aTitle) {
        this.setAuthor(anAuthor);
        this.setISBN(anISBN);
        this.setTitle(aTitle);
    }

    public void setAuthor(String anAuthor) { author = anAuthor;}
    public void setISBN(String anISBN) { isbn = anISBN; }
    public void setTitle(String aTitle) { title = aTitle; }
    public String getAuthor() { return author;}
    public String getISBN() { return isbn;}
    public String getTitle() { return title;}
    public String toString() {
        return "Author : " + author + " ISBN : " + isbn + " Title : " + title;
    }
}

public class BookTest {
    public static void main(String args[]) {
        Catalogue theCatalogue = new Catalogue();
        Book aBook = new Book("Tolkein","09-08-07","The Hobbit");
        theCatalogue.addBook(aBook);
        aBook = new Book("Gray","11-22-33","The Return");
        theCatalogue.addBook(aBook);
        aBook = new Book("Jameson","00-01-02","The New Land");
        theCatalogue.addBook(aBook);
        System.out.println(theCatalogue);
        aBook = new Book("Gray","11-22-33","The Return");
        if (theCatalogue.includes(aBook)) System.out.println("Book found\n" + aBook);
        else System.out.println("Book not found\n" + aBook);
    }
}
```

Output from the test class was...
theCatalogue
Author : Jameson ISBN : 00-01-02 Title : The New Land Price : 2000
Author : Tolkein ISBN : 09-08-07 Title : The Hobbit Price : 1234
Author : Gray ISBN : 11-22-33 Title : The Return Price : 1000
Book found
Author : Gray ISBN : 11-22-33 Title : The Return Price : 1000

3.2 The equals() and hashCode() methods

The books are stored in the catalogue (ie the Hashtable) by using the isbn (a String object) as the **key** and the book object as the **value**. The includes() method of the Catalogue class searches for a book by using the containsKey() method of the Hashtable Class. The containsKey() method will use the hashCode() method to determine a hash value for the parameter and then compare the parameter with the object stored at that hash location. If it finds a match at this location (the isbn's in this case) it returns true.
An extract from the Java API in the Hashtable class…
"To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode() method and the equals() method."
(See comments at the end of the chapter on the hashCode() method)

Thus we must make sure that the equals() method of the Class of the key type is capable of making such a comparison and that consistent hash codes are generated to identify locations. In the above example the keys are of the String Class and hence the equals() and hashCode() methods used by the containsKey() method will be valid since the equals() and hashCode() methods are defined in the String Class.

The problem of the equals() method becomes more real if we consider searching the catalogue by the values rather than by the keys. The Hashtable Class provides a method for doing this. The contains() method is similar to the containsKey() method but instead of looking for a match on the keys it looks for a match on the values of the Hashtable. So this time we shall have to be able to compare two book objects for equality.

We now revise the previous classes so that the search is done with 'Book values' rather than isbn value. In the first instance we modify the includes() method of the Catalogue class so that the search is done on the Hashtable values (not the keys, ie the Book objects and not the ISBN strings).

```
public boolean includes(Book aBook) {
    return theBookList.contains(aBook);
}
```

When we run the same test program as before output is as follows,

theCatalogue
Author : Jameson ISBN : 00-01-02 Title : The New Land Price : 2000
Author : Tolkein ISBN : 09-08-07 Title : The Hobbit Price : 1234
Author : Gray ISBN : 11-22-33 Title : The Return Price : 1000

Book not found
Author : Gray ISBN : 11-22-33 Title : The Return Price : 1000

The required book cannot now be found... so what has happened ?

The contains() method used in the revised includes() method has used an equals() method to do the comparison between the Book objects. However since the Book Class does not contain an equals() method the one used is the one inherited from the superclass Object. The version of equals() inherited from the Object class does not compare the states of two objects, in this case two Book objects, but simply compares if they are the same object. It works like the == operator in that it simply tests if the variables referencing the two objects are equal. That is, do the two variables point to the same object. In this case they don't. The second 'Gray' book is a different object to the first 'Gray' book. Generally in a real application we might well

want the results of the search to say that the book was already in the catalogue in the sense that it has the same attributes as one already in the catalogue. If this is the case then we will have to override the inherited equals() by adding a suitable version to the Book Class. We can do this with something like,

```
public boolean equals(Object aBookArg) {
    Book aBook = (Book)aBookArg;
    return    (   this.getAuthor().equals(aBook.getAuthor()) &&
                  this.getISBN().equals(aBook.getISBN()) &&
                  this.getTitle().equals(aBook.getTitle()));
}
```

Note the following,

- since we are overriding the Object version of equals() the new equals() method of the Book Class has to have exactly the same interface
- hence in the first line of the method we cast the incoming object to a Book object

Now when we run the book test… the “Book found” message appears

(For more discussion on this equality topic see Chapter 9 in Book 1).

3.2.1 Overriding vs Overloading Again

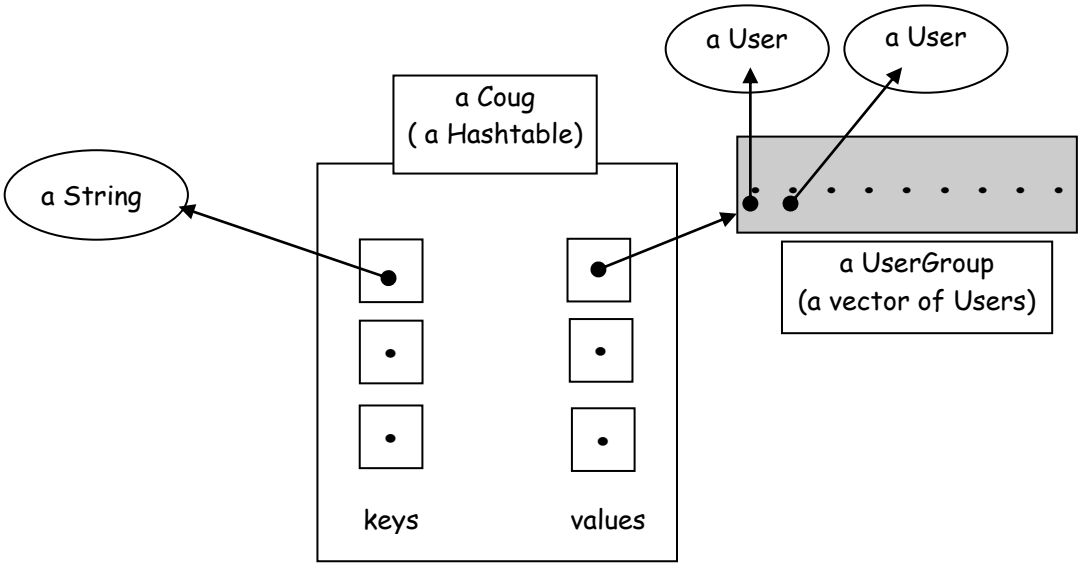
In developing an equals method for the Book class we may have been tempted to write something like

```
public boolean equals(Book aBook) {
    return    (   this.getAuthor().equals(aBook.getAuthor()) &&
                  this.getISBN().equals(aBook.getISBN()) &&
                  this.getTitle().equals(aBook.getTitle()));
}
```

This however will not work in the context that we want it to. Certainly it will return true or false depending whether the two books are the 'same' or not. We have not overidden the equals() method from the Object class but simply given the Book class a new method calls equals. The failure is due to the fact that the contains() method will search the Book class for an equals() method that takes an Object as an argument, it will not find one and therefore will search the superclass (Object) where it will find the matching equals() method and hence we are back to the original problem. So the moral is... we must override the equals() method not just overload it .

3.3 A Second Example

This class implements a hashtable, which maps keys to values. Any non-null object can be used as a key or as a value.



```
import java.util.*;
public class Coug {       // a Collection of User Groups
    private Hashtable theCoug;

    public Coug(int size) {
        theCoug = new Hashtable(size);
    }

    public int getNumGroups() {
        return theCoug.size();
    }

    public void addGroup(UserGroup aUserGroup) {
        theCoug.put(aUserGroup.getGroupId(),aUserGroup);
    }

    public UserGroup getGroup(String idStr) {  // no validation checks...
        return (UserGroup)theCoug.get(idStr);      // Note the cast...
    }
}
```

A test class could be something like,

```
public class CougTest {
    public static void main(String args[]) {
        Coug theCoug = new Coug(5); // create the collection of user groups
        char ch = 'A';
        String chStr;

        for (int i = 0; i < 4; ++i) {    // there will be 4 user groups...
            UserGroup tempGroup = new UserGroup("G" + (i+1),10);
            for (int j = 0; j < 6; ++j) {    // there will be 6 users in each group
                chStr = (new Character(ch)).toString();// convert ch to a String
                User aUser = new User("User_" + (j+1),chStr);   // +1 sets first user = 1, not 0.
                tempGroup.addUser(aUser);
            } // end for j
            theCoug.addGroup(tempGroup);
            ch++;
        } // end for i

        for (int i = 0; i < theCoug.getNumGroups(); ++i) {
            UserGroup aGroup = theCoug.getGroup("G" + (i+1));
            CougTest.show("\nSome details of the group " + aGroup.getGroupId());
            for (int j = 0; j < aGroup.getNumUsers(); ++j) {
                User aUser = aGroup.getUser(j);
                CougTest.show(aUser.getName() + " : " + aUser.getUserId());
            }
        }
    }

    private static void show(String str) {
        System.out.println(str);
    }
}
```

3.4 The hashCode() Method

The class associated with the objects that are to be the keys in the hashtable should have a method hashCode(). All is not lost if you fail to define one, since the system will (as always) search for one up the inheritance hierarchy, and will if necessary use the default one in Object. By its nature, this method cannot be well tuned for every class, so when performance is an issue you should certainly define one for the class in question. See NOTE below for an explanation of 'well-tuned'.

The class Hashtable provides an implementation of a kind of data structure called a hash table! The basic idea is that we would like something like an array, which allows us to perform instantaneous lookup (rather than needing a linear or binary search). This means that, given a 'key', we would like to calculate the index position in the array of the corresponding 'value'. A naive solution is to look for a way of generating a unique (non-negative) integer from each key value, and use this as the index position. This is not satisfactory, because we would probably need an enormous array with most locations unused. Instead, we set aside a smaller array, and make do with a 'hash algorithm' which generates 'almost unique' integers. That is, uniqueness is not guaranteed, but the integer values generated are spread uniformly over the range of index values associated with the array. When inserting a new value, we try first to insert it at the 'home position' given by the hashed value. If this is vacant, fine. Othrwise, some 'overflow technique' is used. Provided clashes are not too common, the overhead in dealing with overflow will not be too great. A hash table is 'well tuned' (in the sense I used above) if it is big enough to avoid too many clashes for the given hash function.

4. THE STRINGTOKENIZER OBJECT

The StringTokenizer class allows you to create StringTokenizer objects around strings and tokens. The class provides a number of useful methods for parsing the strings. It parses the string according to a set of delimiter characters. It implements the Enumeration interface in order to provide access to the tokens contained within the string.

```
import java.util.*;
public class StrTokenizerDemo {
    public static void main(String args[]) {
        String theMessage = new String("This.is.a.short.message");
        System.out.println(theMessage);
        StringTokenizer st_1 = new StringTokenizer(theMessage,".");
        String nextWord;
        System.out.println("Number of words = " + st_1.countTokens());
        while (st_1.hasMoreTokens()) {
            nextWord = st_1.nextToken();
            System.out.println(nextWord);
        }
        System.out.println();

        // You can even mix the delimiters...
        theMessage = new String("This.is*a.short/message.");
        System.out.println(theMessage);
        StringTokenizer st_2 = new StringTokenizer(theMessage,"./*");
        System.out.println("Number of words = " + st_2.countTokens());
        while (st_2.hasMoreTokens()) {
            nextWord = st_2.nextToken();
            System.out.println(nextWord);
        }
        System.out.println();

        // Notice the space (whitespace) is not included as a delimiter and so
        // there are now only 4 words...
        theMessage = new String("This.is.a short/message");
        System.out.println(theMessage) ;
        StringTokenizer st_3 = new StringTokenizer(theMessage,"./");
        System.out.println("Number of words = " + st_3.countTokens());
```

```
This.is.a.short.message
Number of words = 5
This
is
a
short
message
tokens = 5
```

```
This.is.a.short.message
Number of words = 5
This
is
a
short
message
```

```
This.is.a short/message
Number of words = 4
This
is
a short
message
```

```
        while (st_3.hasMoreTokens()) {
            nextWord = st_3.nextToken();
            System.out.println(nextWord);
        }
        System.out.println();

        // Note this time that 2 delimiters are next to each other and
        // one of the delimiters is repeated (ie the whitespace)
        theMessage = new String("This./is    a.short/message.");
        System.out.println(theMessage);
        StringTokenizer st_4 = new StringTokenizer(theMessage,"./ ");
        System.out.println("Number of words = " + st_4.countTokens());
        while (st_4.hasMoreTokens()) {
            nextWord = st_4.nextToken();
            System.out.println(nextWord);
        }
        System.out.println();
    }
}
```

```
This./is    a.short/message.
Number of words = 5
This
is
a
short
message
```

5. EXCEPTIONS

Java code can detect errors and indicate to the run time system what those errors were. When an error occurs, the Java system can create an exception object using the keyword throws. The type of exception object created depends on the type of error. Normally a thrown exception will cause the execution of the code to terminate and an error message will be printed. If you want to handle the exception yourself you can include a try/catch statement to trap the exception.

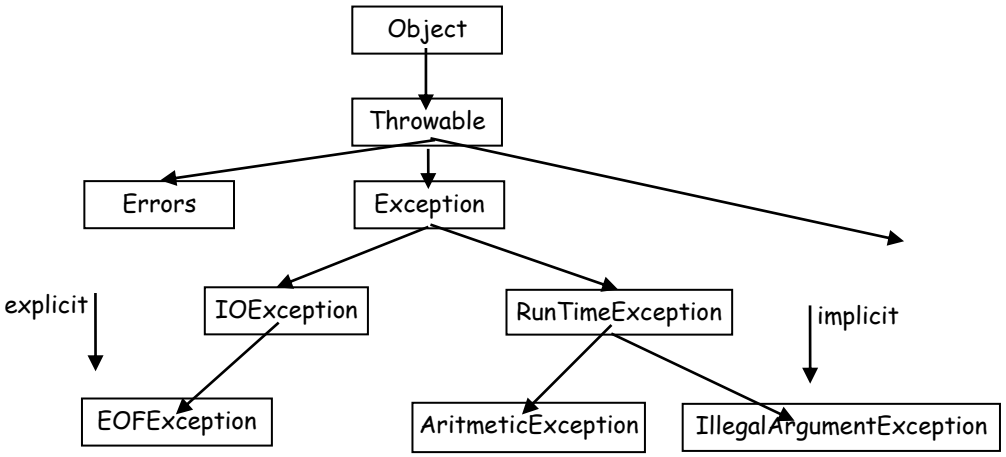
5.1 Exceptions verses Return Values

Errors in methods can be flagged either by throwing an exception or by a special return value such as true/false, 0/1. The difference between the two methods is that exceptions provide an active indication that an error has occurred where as return values provides a passive. Exceptions cannot be ignored. If a function needs to send an error message to the caller of that function, it throws an object detailing the error out of that function. If the caller doesn't catch the error and handle it, it goes to the next enclosing scope, and so on until someone catches the error. Using return values the errors can be ignored. Exceptions are also a good mechanism to collect information about the error condition, which can be used to display detailed feedback to the user of the class. Error codes do not readily cope with upgrade modification as the user methods may need to be amended if the error handler has to be up-issued. Exceptions only need to have additional methods added to provide more info. If the interface between the error creating method and the exception handler stays constant then the exception class can be modified without affecting the user class.

Exceptions are the only way to signal problems that happen in the constructors, since the constructors do not return any values. Implementing an exception class is an OO approach to solving the error-handling problem. This OO approach provides us with the fact that the error handling is now taken care of by a separate object.

Overall both approaches have their uses. Returned values are suitable for conditions best described as unusual rather than errors, such as end-of-file, or value does not exist on a look up table. The use of exception objects lends itself much better to a standard error handling approach within an application, with exceptions handled in a uniform way by generic code. Most applications will see both approaches used, with informational and warning conditions often handled by returned values, and anything more severe handled by exception objects being thrown.

The Java Exception classes can be grouped into a class hierarchy. Here's a section,



The basic ideas of exceptions are illustrated in the following examples. We take a classical division by zero error and show how Java exceptions might be used to handle such an error,

There are two sorts of exceptions: implicit (or unchecked), and explicit (or checked). The difference is that the compiler checks that you have thrown and caught exceptions in the 'explicit' classes, but not those in the implicit classes. Since we want the compiler to help us as much as possible, it seems more sensible to

extend Exception (giving a new explicit class) than RuntimeException (giving a new 'implicit' class). Using checked exceptions also introduces some controlled error handing during compilation since if a checked exception is thrown as a source code statement but not caught the compiler will indicate this.

5.2 Example 1

This simple class defines a mod method.

```
public class ExceptionDemo_1 {
    private int x;
    public ExceptionDemo_1(int anInt) {
        x = anInt;
    }

    public int mod(int y) {
        int ans;
        ans = x % y;
        return ans;
    }
}

public class TestExceptionDemo_1 {
    public static void main(String args[]) {
        int ans;
        ExceptionDemo_1 myInt = new ExceptionDemo_1(6);
        ans = myInt.mod(0);
        System.out.println("Back in main()... possibly further code...");
    }
}
```

When a division by zero error occurs the Java system acts just like most languages… it will crash on this kind of error. However just before terminating it creates an exception object appropriate to the kind of error that has occurred. In this example it creates (throws) an ArithmeticExeception and the execution stops with error message something like…

```
java.lang.ArithmeticException: / by zero
    at ExceptionDemo_1.mod(ExceptionDemo_1.java:14)
    at TestExceptionDemo_1.main(TestExceptionDemo_1.java:21)
```

Note that none of the code makes a direct (explicit) reference to the exception. The ArithmeticException is an example of a Java **implicit exception**. The exception was not required to be published (ie declared) anywhere. It just happens ! It is implied by the use context (a division operation).

If an exception is thrown it must be caught (somewhere) to prevent the program from terminating. This requires the use of a try/catch statement block

5.3 Where to Catch the Exceptions

There are two possible approaches to catching the thrown exception. Either we catch the exception at ‘source’ ie within the method where the exception occurred or we catch the exception at the point where the 'offending' method is invoked... generally this second approach may be the preferred approach since the way is which the error is handled is probably best left to the user of the method where the error occurs

5.3.1 Example 2 - Catching The Exception at Source

```
public class ExceptionDemo_2 {
    private int x;

    public ExceptionDemo_2(int anInt) {
        x = anInt;
    }

    public int mod(int y) {
        int ans = 0;
        try {
            ans = x % y;
        }
        catch (ArithmeticException e) {
            System.out.println("Dealing with the ArithmeticException in the ‘source’ method");
        }
        return ans;
```

```
    }
}

public class TestExceptionDemo_2 {
    public static void main(String args[]) {
        int ans;
        ExceptionDemo_2 myInt = new ExceptionDemo_2(6);
        ans = myInt.mod(0);
        System.out.println("Back in main()... possibly further code...");
    }
}
```

Catching and handling the exception at source, as we have done above, raises the difficult issue as to just what value we return from the mod() method when the exception occurs. In the simple example above we have chosen to return 0, but of course this may unsatisfactory from the users point of view and hence reinforces the argument that exceptions should be trapped and the handling/action to be taken left up to the method user. This is addressed in the next example where the try/catch statement is transferred to the calling code.

5.3.2 Example 3 - Catching the Exception in the Calling Method

```
public class ExceptionDemo_3 {
    private int x;
    public ExceptionDemo_3(int anInt) {
        x = anInt;
    }
    public int mod(int y) {
        int ans;
        ans = x % y;
        return ans;
    }
}
public class TestExceptionDemo_3 {
    public static void main(String args[]) {
        int ans;
        ExceptionDemo_3 myInt = new ExceptionDemo_3(6);
        try {
            ans = myInt.mod(0);
        }
        catch (ArithmeticException e) {
            System.out.println("User now dealing with the ArithmeticException in the calling method");
        }
        System.out.println("Back in main()... possibly further code...");
    }
}
```

5.4 Creating Your Own Exceptions

You could decide to handle the division by zero error by creating your own exception class but where should you join the Exception hierarchy ? Perhaps this extract is a guide,

“The Exception class is the superclass of classes used to represent exceptional conditions. There are two types of exceptions, implicit (unchecked) and explicit (checked). Implicit exceptions are characterised by the RunTimeException group. All other subclasses of Exception are explicit exceptions.

Any method that throws an instance of Exception that is not a RuntimeException must declare the exception(s) in its throws clause as part of the method’s declaration. This is a Java language requirement. Any method that calls this method must either catch the exception(s) by using try/catch statements or declare the exception(s) in its own throws clause. The Java compiler will generate a compilation error for any code that does not follow these rules.

Many people will argue that you should subclass Exception directly so that the compiler will require the use of throws clauses and try/catch statements more fully. The programmer is then required to publish the existence of the throwing an exception in the throws clause. When using a method that throws an exception you will be required to try to catch it. These will lead to better, safer programming."

5.4.1 Creating an Exception Subclass To Handle the Divsion By Zero Exception

If instead of using the system exception class ArithmeticException we decide to create our own specialised exception class then following from the above we shall create it as a subclass of Exception. Because we are subclassing from the Exception class we are creating an **explicit (or checked) exception** class. This means we have to publish/announce/declare the throwing of the exception and expicitly catch the exception somewhere. This is done in the mod() method and the test class.

5.4.2 Example 4

```
public class DivisionByZeroException extends Exception {
    public DivisionByZeroException() {
        super("Division by zero error...");
    }
}
public class ExceptionDemo_4 {
    private int x;
    public ExceptionDemo_4(int anInt) {
        x = anInt;
    }

    public int mod(int y) throws DivisionByZeroException {
        int ans;
        if (y==0) throw new DivisionByZeroException();
        ans = x % y;
        return ans;
    }
}

public class TestExceptionDemo_4 {
    public static void main(String args[]) {
        int ans = 0;
        ExceptionDemo_4 myInt = new ExceptionDemo_4(6);
        try {
            ans = myInt.mod(0);
        }
        catch (DivisionByZeroException e) {
            System.out.println(e.getMessage());
        }
        System.out.println("More code lines...");
        System.out.println("By now we have trapped and dealt with the exception...");
        System.out.println("and our program is still running...");
    }
}
```

To reinforce the ideas outlined in the previous comments regarding extending from the Exception class rather than from the RunTimeException branch you should try changing the declaration of the DivisionByZeroException class to

```
    public class DivisionByZeroException extends IllegalArgumentException {
```

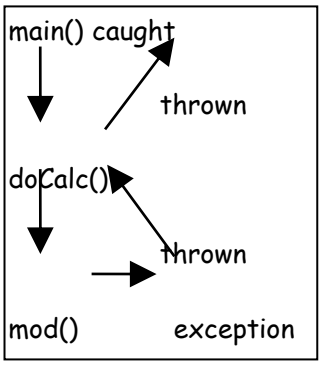
and remove the try/catch statements from the test class (you should of course leave ans = myInt.mod(0);).

When you compile the test class no reference will be made to the exception. The exception will of course still occur at run time (and stop the program). By choosing to extend from IllegalArgumentException we have ignored the existence of the exception at compile time and perhaps ignored handling the exception.
If you now change the declaration of the DivisionBy ZeroException back to it's original form, ie extending from Exception and again try to compile the test class, the compiler will now complain that the exception thrown by the DivisionByZeroException has not been caught. Is this better coding/safer coding ?

5.4.3 Example 5

We said before that the thrown exception must be caught *somewhere*. The following example shows how a chain of methods can work, providing the exception is caught *somewhere*. The method doCalc() does not have to do the catching, it passes it on…

```
public class TestExceptionDemo_5 {
    static ExceptionDemo_4 myInt;
    private int ans;

    public void doCalc() throws DivisionByZeroException {
        ans = myInt.mod(0);
    }

    public static void main(String args[]) {
        myInt = new ExceptionDemo_4(6);
        TestExceptionDemo_5 p = new TestExceptionDemo_5();
        try {
            p.doCalc();
        }
        catch (DivisionByZeroException e) {
            System.out.println(e.getMessage());
        }
    }
}
```



5.5 Another Example

```
public class Tester {
    public static void main(String args[]) {
        ExceptionDemo ed = new ExceptionDemo();
        ed.addName("jeff");
        ed.addName("pat");
        ed.addName("jeff");
    }
}
```

```
public class DuplicateNameException extends Exception {
    public DuplicateNameException(String aName) {
        System.out.println(aName + " is duplicated");
    }
}
```

Our first attempt is…

```
import java.util.*;
public class ExceptionDemo {
    private Vector theList;

    public ExceptionDemo() {
        theList = new Vector(10);
    }
public void addName(String aName) {
        if (theList.indexOf(aName) != -1)
            throw new DuplicateNameException(aName);
        else theList.addElement(aName);
    }
}
```

But… the Java compiler will not allow this.. it gives the message,
DuplicateNameException must be caught, or it must be declared in the throws clause of this method.
This means we can
either
    catch the exception in the method itself
or
    announce to the world that an exception may be thrown and it must be caught somewhere

If we choose to catch the exception in the method itself our revised method becomes,

```
import java.util.*;
public class ExceptionDemo {
```

```
    private Vector theList;

    public ExceptionDemo() {
        theList = new Vector(10);
    }

    public void addName(String aName) {
        try {
            if (theList.indexOf(aName) != -1)
                throw new DuplicateNameException(aName);
            else theList.addElement(aName);
        }
        catch (DuplicateNameException e) {
            // exception handled... program continues...
        }
    }
}
```

The alternative (better?) approach is to have the method announce that it may throw the exception and leave the catching to be done somewhere else…

```
import java.util.*;
public class ExceptionDemo {
    private Vector theList;

    public ExceptionDemo() {
        theList = new Vector(10);
    }

    public void addName(String aName)
    throws DuplicateNameException {
        if (theList.indexOf(aName) != -1)
                throw new DuplicateNameException(aName);
        theList.addElement(aName);
    }
}
```

annouce that the method may throw the exception

it means that we now have to catch the exception somewhere else otherwise the program will crash if the exception is thrown and not caught anywhere. We catch it in the Tester class

```
public class Tester {
    public static void main(String args[]) {
        ExceptionDemo ed = new ExceptionDemo();
        try {
            ed.addName("jeff");
            ed.addName("pat");
            ed.addName("jeff");
        }
        catch (DuplicateNameException e) {}
        System.out.println("End of main()...");
    }
}
```

5.6 A Final Example

```
public class Account {
    private int accNumber;
    private int balance;
    private char type;

    public Account(int accN,int initBal,char aType)
    throws BadAccountNumberException
    {
        if ((accN < 1000) || (accN > 9999))
            throw new BadAccountNumberException(accN);
        else {
            accNumber = accN;
            balance = initBal;
            type = aType;
        }
    }

    // other methods...

public class BadAccountNumberException extends Exception {
    public BadAccountNumberException(int n) {
        super("Bad Account Number : " + n);
    }
}

public class TestException {
    public static void main(String args[]) {
        try {
            Account a = new Account(5000,100,'S');
            Account b = new Account(10000,0,'S');
        }
        catch (BadAccountNumberException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Although this simple example does not show it, other methods can of course be included in the exception class if there are reasons for this. For example there may be requirements for additional actions to be taken if the exception occurs.

getMessage() is an inherited method from the Exception class

5.7 Class (Static) Methods Can Throw Exceptions

Class methods can of course throw exceptions... as this rather contrived example shows...

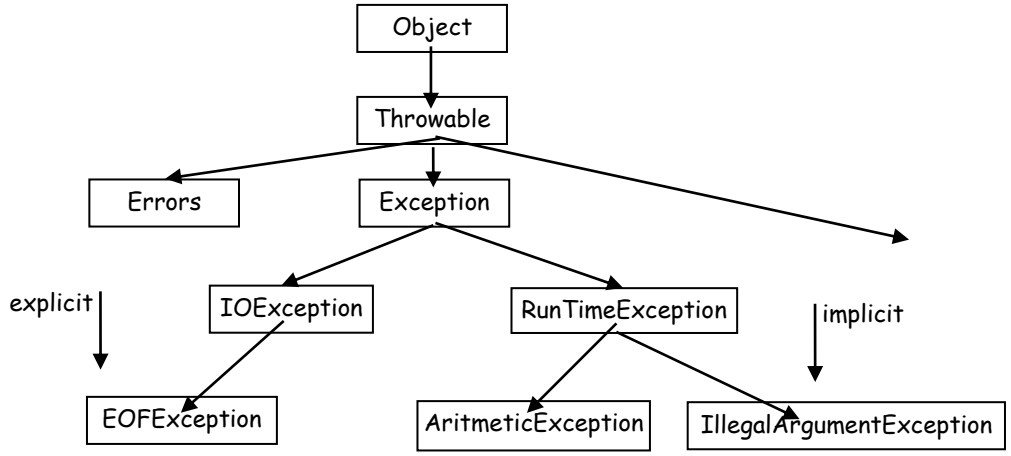
```
public class DemoClass {
    private static final int MAX_OBJECTS = 3;
    private static int objectCount = 0;
    private int objectNumber;

    public DemoClass() throws TooManyObjectsException {
        this.createObject();
        objectNumber = objectCount;
    }

    public static void createObject() throws TooManyObjectsException {
        if (objectCount == MAX_OBJECTS) throw new TooManyObjectsException();
        else ++objectCount;
    }

    public static int getObjectCount() { return objectCount; }
    public int getNumber() { return objectNumber; }
    public void whoAmI() {
        System.out.println("You are object number " + this.getNumber());
    }
}

public class TestDemo {
    public static void main(String args[]) {
        int count;
        DemoClass obj1, obj2, obj3, obj4;
        try {
            obj1 = new DemoClass();
            obj2 = new DemoClass();
            count = DemoClass.getObjectCount();
            System.out.println("Current number of objects = " + count);
            obj3 = new DemoClass();
            count = DemoClass.getObjectCount();
            System.out.println("Current number of objects = " + count);
            obj2.whoAmI();
            obj4 = new DemoClass();
        }
        catch (TooManyObjectsException e) {
            System.out.println(e.getMessage());
        }
    }
}

public class TooManyObjectsException extends Exception {
    public TooManyObjectsException() {
        super("Too Many Objects...");
    }
}
```

6. NEW STUFF

The Java programming language offers strong exception handling tools. This section will show how properly constructed exception handling can help in the construction of a clean, robust program. An exception is generally an error condition or other event that interrupts normal execution in an application. When an exception is raised, it causes control to be transferred from the current point of execution to an exception handler. Java implements these exceptions through a class structure for the following reasons:

- Exceptions can be grouped into hierarchies using inheritance.
- An exception object can be assigned information at the point it was thrown which can then be retrieved at the point it is handled.

The Java Exception classes are grouped into a class hierarchy. Here's a section,



There are two sorts of exceptions: implicit (or unchecked), and explicit (or checked). The difference is that the compiler checks that you have thrown and caught exceptions in the 'explicit' classes, but not those in the implicit classes. Since we want the compiler to help us as much as possible, it seems more sensible to extend Exception (giving a new explicit class) than RuntimeException (giving a new 'implicit' class). Using checked exceptions also introduces some controlled error handing during compilation since if a checked exception is thrown as a source code statement but not caught the compiler will indicate this.

Declaring Exception Classes

Since an exception in Java is simply a class, the declaration of an exception is the same as the declaration of an ordinary class. It is general practice to derive all exceptions from the class Exception which is defined in the SysUtils unit.

Some examples of exception declarations include:

```
type
  EIntError = class(Exception);
  EDivByZero = class(EIntError);
  ERangeError = class(EIntError);
  EIntOverflow = class(EIntError);
```

Three of the above declarations define a family of related Integer error exceptions, whose parent or base exception class is EIntError. Using the concept of inheritance, it is possible to handle an entire family of exceptions under one name. So the exception handler for EIntError will also handle EDivByZero, ERangeError and EIntOverflow exceptions, and any other exceptions that are directly or indirectly derived from EIntError. Cunning.

Exception classes may also define additional fields, methods and properties which may be used to provide more information about the exception.

Using Built-in Exception Handling

Exception generation and handling for the Java Pascal run-time library is implemented in the SysUtils unit. Since Java's VCL fully supports exception handling, any application that uses the VCL automatically uses the SysUtils unit, thereby enabling exception handling across the board.

The use of the SysUtils unit by an application results in the automatic conversion of all run-time errors into exceptions, which means that the error conditions which would otherwise terminate an application can be trapped and handled.

The TApplication class that encapsulates your application has an associated event, called OnException and a method, called ShowException. When activated, this method displays a dialog box indicating that an exception has occurred, as well as showing any related exception messages.

Raising Exceptions

The raise statement, promised in Chapter 2, is used to generate an exception. Since exceptions are classes, the argument to a raise statement must be an object.

Once an exception is raised, exception handling logic takes over the exception object. Once the exception has been handled, the exception object is automatically destroyed through a call to the object's Destroy destructor.

An application should never attempt to manually destroy a raised exception object, since it is automatically destroyed once it has been handled.

A raise statement that omits the exception object argument will re-raise the current exception. This is allowed only in an exception block - we'll come back to this in a moment.

The following example, inspired by the user help example, creates a user-defined exception to handle invalid dates:

```
type
    EDateError     = class(Exception);
    EInvalidMonth = class(EDateError) ;
    EInvalidDay     = class(EDateError) ;
```

A strDateToStr function converts a date to a string and raises exceptions if either the month is invalid or the day value is invalid for a given month:

```
function strDateToStr(nYear, nMonth, nDay : Integer) : string;
begin
  result := '';
  if (nYear < 0 ) or  (nMonth < 0) or ( nDay < 0 ) then
    raise EDateError.Create('Error In Date Term(s)');
  if (nMonth < 1) or  (nMonth > 12) then
    raise EInvalidMonth.CreateFmt('%d is an invalid numeric
       Month',[nMonth]);
  if NOT bValidDayForMonth(nYear,nMonth,nDay) then
    raise EInvalidDay.CreateFmt('%d is an invalid numeric Day for the
       numeric month %d',[,nDay,nMonth]);

  result := IntToStr(nDay) + '/' + IntToStr(nMonth);
end;
```

Program control doesn't return from a raise statement. Instead, the try..except block is abandoned, and control is passed to the calling procedure, or the program's response is curtailed and the application gets on

with handling the next event. By abandoned, we mean that the application attempts to find a way of handling the exception before the control moves on from this block.

It is normal practice to create the exception object when raising the exception, as we have done in the previous example where the exception's object constructor CreateFmt was employed. The exception object actually has more than one constructor, and it is up to the developer to select the most appropriate one. The constructors allow for messages to be retrieved from .RES resource files, which is quite useful where internationalization is required. Alternatively, the exception object can be constructed to be associated with a help context:

```
  constructor Create(const Msg: string);
  constructor CreateFmt(const Msg: string; const Args: array of const);
  constructor CreateRes(Ident: Word);
  constructor CreateResFmt(Ident: Word; const Args: array of const);
  constructor CreateHelp(const Msg: string; AHelpContext: Longint);
  constructor CreateFmtHelp(const Msg: string; const Args: array of const;
    AHelpContext: Longint);
  constructor CreateResHelp(Ident: Word; AHelpContext: Longint);

  constructor CreateResFmtHelp(Ident: Word; const Args: array of const;
    AHelpContext: Longint);
```

The try...except Statement

The try...except statement provides the primary method for handling exceptions. The statements that are listed in the try statement block are executed in their listed order. If the statements execute without any exceptions being raised, the except statement block is bypassed and the program executes the statement following the statement's end keyword.

The exception handlers for exceptions that are raised as a result of executing a statement of a try statement list are defined in the associated except...end section. An exception handler can be invoked only by a raise statement in the try block, or as a result of a function or procedure executed by one of the statements of the try block. For example:

```
{ code shows unhandled exception }
procedure TForm1.Button1Click(Sender: TObject);
var
  nDummy,
  nDivisor  : Integer ;
begin
  nDivisor := 0 ;
  nDummy := 32 div nDivisor ;
end;

{ code to handle the divide by zero error }
procedure TForm1.Button1Click(Sender: TObject);
var
  nDummy,
  nDivisor  : Integer ;
begin
  nDivisor := 0 ;
  try
     nDummy := 32 div nDivisor ;
  except
    MessageDlg('Severe Error Attempting to Divide by Zero',
               mtError,[mbOK],0) ;
  end ;
end;
```

When an exception is raised, program control is immediately transferred to the exception handler that can handle exceptions of the raised exception's class. The search for an exception handler begins with the current innermost try statement block. If the associated except block can't or doesn't handle the exception, the next try..except statement block out is examined.

This 'bubbling up' of the exception continues until either an appropriate handler is found or there are no more active try...except statements. In the latter case, a run-time error occurs and the application displays a dialog box indicating the exception.

Using an else section at the end of the try..except statement will effectively trap all errors within the first (and innermost) error-handling block.

On finding a suitable exception handler, all previous functions or procedures referenced to the procedure or function handling the exception which are present on the stack are discarded, thus cleaning it.

The except block may contain a statement, but as the previous example has just shown, if a statement, or indeed a list of statements, is placed within the except block, they will be executed, regardless of the type of exception generated by the associated try block. Java Pascal allows you to define the appropriate set of responses to the exception that is generated by including the on..do keywords in the except block.

Consider the following code snippet where a function is being used to convert three integer inputs into a date string. Assuming that the function strDateToStr is the same function that was used to illustrate the raise statement, then the except block as shown makes use of exception handlers to provide the desired response to each kind of exception:

```
try
   strNewDate :=  strDateToString(nYear,nMonth,nDay) ;
except
   on EInvalidDay do HandleInvalidDay ;
   on EInvalidMonth do HandleInvalidMonth ;
   on EDateError do HandleDateError;
else
   MessageDlg('Unknown Exception Generated' , mtInformation,[mbOK],0) ;
end;
```

The on..do handlers are processed in the order in which they are listed. In the above example, as EInvalidDay and EInvalidMonth are derived from EDateError, the EDateError handler will also handle the derived exceptions. Thus, to ensure that the derived exception handlers are called if the relevant exception is raised, these exceptions have to be listed before the handler for the base class.

If the EDateError exception handler were listed first, the derived exception handlers would never be activated. HandleInvalidDay and its sisters are some user-defined methods for the error objects.

As exceptions are also objects, it is also possible to interrogate them for more information. The on..do exception handler can come in quite useful for this. An identifier is declared to represent the exception. This is done by separating the identifier from the desired exception class with a colon. The previous example can be rewritten as follows:

```
try
   :
  strNewdate :=  DateToString(nMonth,nDay) ;
except
  on ErrMsg:EInvalidMonth do
       MessageDlg(ErrMsg.Message , mtInformation,[mbOK],0) ;
  on ErrMsg:EInvalidDay do
       MessageDlg(ErrMsg.Message , mtInformation,[mbOK],0) ;
end;
```

In the above example, the dialog box invoked by the exception would hold the text extracted from each of the exception objects that was generated when they were created.

Re-raising Exceptions

Consider a call chain of procedures or functions (procedure 1 calling function 2 calling procedure 3 and so on) where each of the procedures or functions is affected by a given exception condition. If a normal exception handler is invoked, the exception will be handled at that level, and the upper levels in the call chain, which may need to perform some clean up code (perhaps for an orderly application shutdown) would remain unaware that an exception has occurred.

Typical examples include opening a file or processing a special type of database record where an exception could occur.

To allow all the procedures and or functions in the call chain to process the exception, the except..end sub-block of try..except would employ the keyword raise with no arguments. The exception will then be propagated up to the calling procedure or function:

```
try
   strNewdate :=  DateToString(nMonth,nDay) ;
except
   on EInvalidDay do HandleInvalidDay ;
   on EInvalidMonth do HandleInvalidMonth ;
   on EDateError do HandleDateError;
else
   MessageDlg('Unknown Exception Generated' , mtInformation,[mbOK],0) ;
   raise ;
end;
```

Nested Exceptions

It is possible to raise and handle exceptions within an exception handler. Provided that exceptions raised in an exception handler are also handled within that exception handler, they don't affect the original exception. The following example illustrates this:

```
type
  EParamError = Class(EIntError) ;

function nComputeTerm(nTerm1,nTerm2,nDivisor : Integer ) : Integer;
begin
  try
    Result := ( nTerm1 * nTerm2 )  div nDivisor;
  except
    on EIntError do
    begin
      try
        raise EParamError.Create('Invalid Term to Function') ;
      except
        on ErrMsg:EParamError do
          MessageDlg('Nested Handler -'+ ErrMsg.Message ,mtInformation,[mbOK],0) ;
      end ;
      raise ;

    end ;
  end ;
end;
```

This is a fairly simplistic example, but it does illustrate the salient points. Should any Integer exception be triggered, such as a divide by zero, the EIntError handler is activated. This handler raises another exception EParamError which is handled locally. The original EIntError can still be re-activated by the raise and is then treated as if it had not been handled, so is free to propagate up the procedure and function call chain as discussed earlier.

If, however, no exception handler is provided for any EParmError type exception, the original EIntError exception will be lost, should raise EParamError.Create fail. This is the case in our following example, where EParamError replaces EIntError as the exception to be handled:

```
type
     EParamError = Class(EIntError) ;

function nComputeTerm(nTerm1,nTerm2,nDivisor : Integer ) : Integer;
begin
  try
     Result := ( nTerm1 * nTerm2 )  div nDivisor;
  except
     on EIntError do
        raise EParamError.Create('Invalid Term to Function') ;
  end ;
end;
```

The try...finally Statement

When a section of code acquires a resource, it is often necessary to ensure that the resource be released again, regardless of whether the code completes as normal or whether it is interrupted by an exception. For example, a section of code that creates temporary objects or requests GDI resources must release these resources back to the pool, otherwise droughts will occur. Thus the try part consists of the requests for resources, while the finally part contains the releases for them.

Should an exception occur, program execution will immediately pass to the first statement of the finally block. If no exception occurs, the finally block is still executed regardless. An example of the use of a try..finally block is as follows:

```
procedure TForm1.Button1Click(Sender: TObject);
var
   MyString TStrings;
begin
   MyString.Create;
   try
    { Do whatever with your Tstring }
   finally
    MyString.Free;
   end;
end;
```

The code in the finally section effectively does a clean-up operation. The developer should ensure that Java routines employed to deallocate memory and remove objects are routines that cope with partially constructed objects e.g the free method for normal objects and the release method for forms.

A call to either of the Exit, Break, or Continue standard procedures during the execution of a try block will result in the immediate execution of the finally block. Likewise, should any one of these procedures be called in an exception handler, the exception handler will terminate and the exception object disposed of.

6.1 Exceptions verses Return Values

Errors in methods can be flagged either by throwing an exception or by a special return value such as true/false, 0/1. The difference between the two methods is that exceptions provide an active indication that an error has occurred where as return values provides a passive. Exceptions cannot be ignored. If a function needs to send an error message to the caller of that function, it throws an object detailing the error out of that function. If the caller doesn't catch the error and handle it, it goes to the next enclosing scope, and so on until someone catches the error. Using return values the errors can be ignored. Exceptions are also a good mechanism to collect information about the error condition, which can be used to display detailed feedback to the user of the class. Error codes do not readily cope with upgrade modification as the user methods may need to be amended if the error handler has to be up-issued. Exceptions only need to have additional methods added to provide more info. If the interface between the error creating method and the exception handler stays constant then the exception class can be modified without affecting the user class.

Exceptions are the only way to signal problems that happen in the constructors, since the constructors do not return any values. Implementing an exception class is an OO approach to solving the error-handling problem. This OO approach provides us with the fact that the error handling is now taken care of by a separate object.

Overall both approaches have their uses. Returned values are suitable for conditions best described as unusual rather than errors, such as end-of-file, or value does not exist on a look up table. The use of exception objects lends itself much better to a standard error handling approach within an application, with exceptions handled in a uniform way by generic code. Most applications will see both approaches used, with informational and warning conditions often handled by returned values, and anything more severe handled by exception objects being thrown.