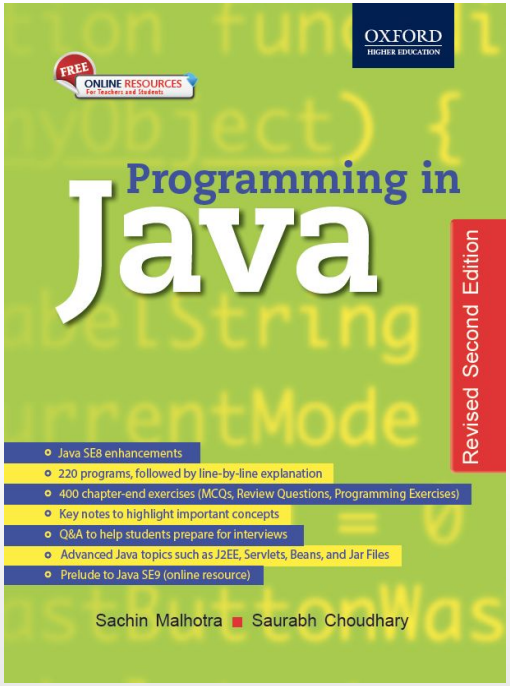
OXFORD
UNIVERSITY PRESS



Programming in Java

Revised 2nd Edition

Sachin Malhotra & Saurabh Choudhary

Chapter 6

Interface, Packages and Enumeration

Objectives

- Understand what interfaces are and how they are different from abstract classes
- Understand the concept behind packages and how they are used
- Know about the java.lang package
- Understand object and wrapper class
- Understand how strings are created, manipulated, and split in Java
- Understand enumerations

Interfaces

- Interface is a collection of methods which are public and abstract by default.
- The implementing objects have to inherit the interface and provide implementation for all these methods.
- *Multiple inheritance* in Java is allowed through interfaces.
- Interfaces are declared with the help of *interface* keyword.

Syntax for Creating Interface

interface interfacename

{

returntype methodname(argumentlist);

…

}

The class can inherit interfaces using implements keyword

- class classname implements interfacename{}

Interface Example

```
interface Calculator {

   int add(int a,int b);

   int subtract(int a,int b);

   int multiply(int a,int b);

   int divide(int a,int b);

}
```

Example (contd.)

```
class Normal_Calculator implements Calculator
{
public int add(int a,int b){
   return a+b;}
public int subtract(int a,int b){
   return a-b;}
public int multiply(int a,int b){
   return a*b;}
public int divide(int a,int b){
   return a/b;}
public static void main(String args[]){
Normal_Calculator c=new Normal_Calculator();
System.out.println("Value after addition = "+c.add(5,2));
System.out.println("Value after Subtraction = "+c.subtract(5,2));
System.out.println("Value after Multiplication = "+c.multiply(5,2));
System.out.println("Value after division = "+c.divide(5,2)); }}
```

The Output

Value after addition = 7

Value after Subtraction = 3

Value after Multiplication= 10

Value after division = 2

Variables in Interface

- They are implicitly *public*, *final*, and *static.*
- As they are *final*, they need to be assigned a value compulsorily.
- Being *static*, they can be accessed directly with the help of an interface name.
- As they are *public* we can access them from anywhere.

Extending Interfaces

- One interface can inherit another interface using the *extends* keyword and not the *implements* keyword.
- For example,

  interface A extends B { }

Interface vs Abstract Class

Interface	Abstract Class
Multiple inheritance possible; a class can inherit any number of interfaces.	Multiple inheritance not possible; a class can inherit only one class.
implements keyword is used to inherit an interface.	**extends** keyword is used to inherit a class.
By default, all methods in an interface are **public** and **abstract**; no need to tag it as **public** and **abstract**.	Methods have to be tagged as **public** or **abstract** or both, if required.
Interfaces have no implementation at all.	Abstract classes can have partial implementation.
All methods of an interface need to be overridden.	Only abstract methods need to be overridden.
All variables declared in an interface are by default **public**, **static**, and **final**.	Variables, if required, have to be declared as **public**, **static**, and **final**.
Interfaces do not have any constructors.	Abstract classes can have constructors
Methods in an interface cannot be static.	Non-abstract methods can be static.

Similarities Between Interface and Abstract Class

- Both cannot be instantiated, i.e. objects cannot be created for both of them.
- Both can have reference variables referring to their implementing class objects.
- Interfaces can be extended, i.e. one interface can inherit another interface, similar to that of abstract classes (using extends keyword).
- **static** / **final** methods can neither be created in an interface nor can they be used with abstract methods.

Packages

- Collection of classes and interfaces.
- Provides a unique namespace for the classes.
- Declaration resides at the top of a Java source file.
- A package can contain the following:
  - ❑ Classes
  - ❑ Interfaces
  - ❑ Enumerated types
  - ❑ Annotations

Example

```
package packexample;

   public class ClassinPackage {

   }
```

- If you want to create sub-packages
  - package rootPackage.subPackage1;

Packages

- Classes that reside inside a package cannot be referred by their own name alone.
- The package name has to precede the name of the class of which it is a part of.
- All classes are part of some or the other package.
- If the “package” keyword is not used in any class for mentioning the name of the package, then it becomes part of the default/unnamed package.

Compiling and Executing Packages

- Compiling the class

  javac –d c:\pack ClassinPackage.java

- Executing the class

  classpath needs to be set

  Set CLASSPATH=%CLASSPATH%;c:\pack;

  java packexample.ClassinPackage

  Considering pack is the parent directory of packexample

Using Packages

- Import keyword is used to import classes from a package.
- Any number of import statements can be given.
- To import a single class
  - import java.awt.Rectangle;
- To import all classes in a package (wild card notation)
  - import java.awt.*;
- To import all the static members of the class
  - import static pkgName.ClassName.*;

Access Protection

Four Access specifier in Java

public(maximum access)	applied to variables,contructors, methods, and classes
protected	applied to variables,contructors, methods, and inner classes (not top- level classes)
default	applied to variables,contructors,methods, and classes
private(minimum access)	can be applied to variables, methods and inner classes (not top-level classes)

Access Specifiers

- Public means accessibility for all
- Private means accessibility from within the class only
- Default (blank) access specifiers are accessible only from within the package
- Protected means accessibility outside the packages but only to subclasses

Access Specifiers Example

Suppose x and y are packages

A is a public class within package x

B is another class within package x

C is subclass of A in package x

D is subclass of A within package y

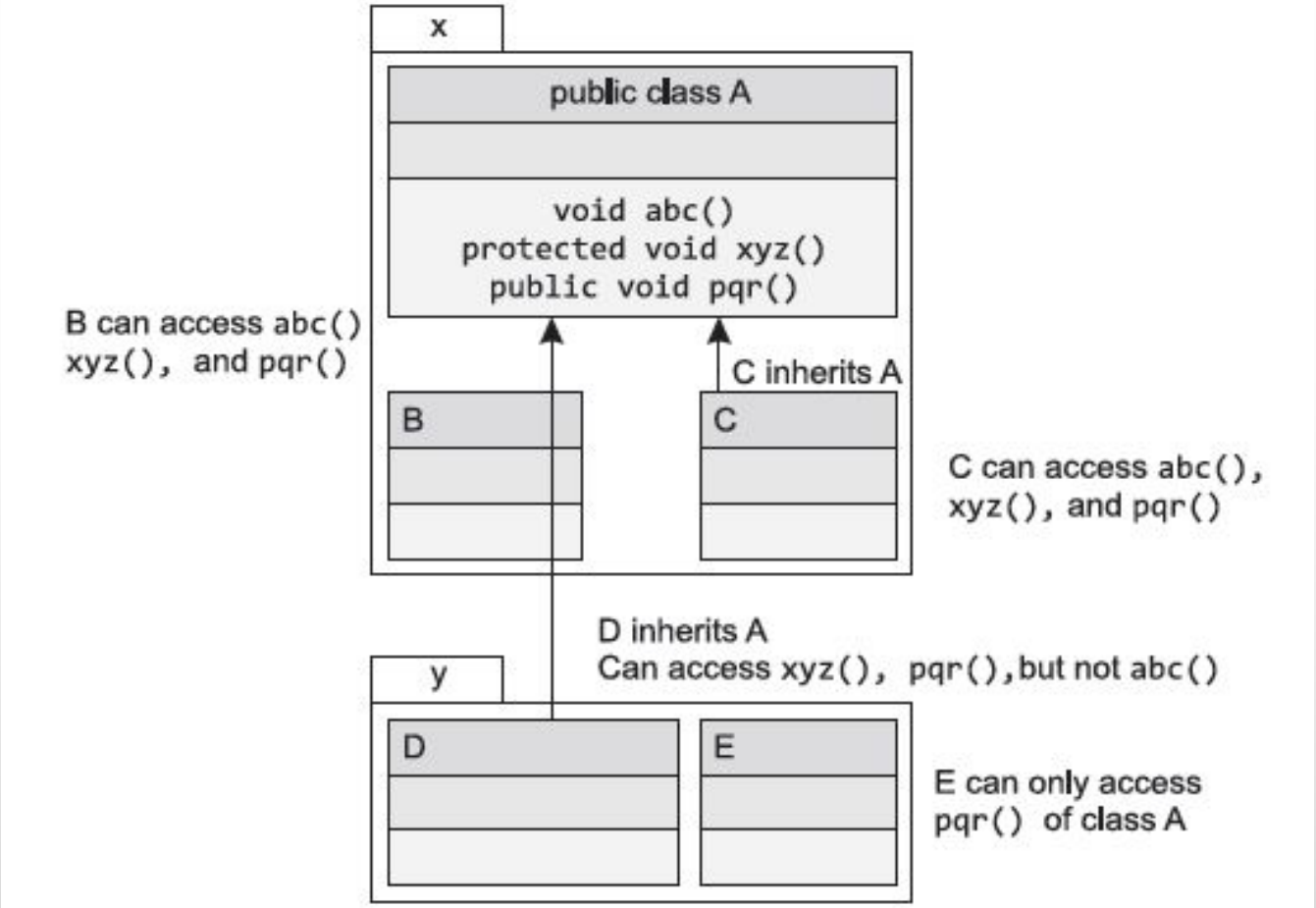
E is class within package y

abc() is a method with default access in class A

xyz() is a method with protected access specifier in class A

pqr() is a method with public access specifier in class A

Access Protection



Access Protection

- Method abc() is accessible from A, B as well as C, but neither from D nor E.
- Protected methods are accessible outside the package also, but only to subclasses outside the package. For example, xyz() method is accessible from classes A, B, C, D, but not from E.
- Public method pqr() is accessible from all classes A, B, C, D and E.

Your Turn

- Differentiate between interface and abstract class?
- What is the significance of classpath in packages?
- What is static import?
- What are various access specifiers in Java?

java.lang package

- java.lang is imported by default in all the classes that we create.
- Remember String and System class.

Java.lang.Object Class

- Parent by default of all the classes (predefined and user-defined) in Java.
- Methods of Object class can be used by all the objects and arrays.
- toString() and equals() are methods of Object class.

toString() Method

```
class Demo {
public String toString() {
   return “My Demo Object created”;
}
public static void main(String args[]) {
System.out.println(new Demo()); }}
```

Output

My Demo Object created

Java Wrapper Classes

- For each primitive type, there is a corresponding *wrapper* class designed.
- Are wrapped around primitive data types.
- Allow for situations where primitives cannot be used but their corresponding objects are required.
- Normally used to convert a numeric value to a string or vice-versa.
- Just like string, wrapper objects are also immutable.
- All the wrapper classes except Character and Float have two constructors—one that takes the primitive value and another that takes the string representation of the value.
- Character has one constructor and float has three.

Wrapper Classes

Primitive	Wrapper
boolean	java.lang.Boolean
byte	java.lang.Byte
char	java.lang.Character
double	java.lang.Double
float	java.lang.Float
int	java.lang.Integer
long	java.lang.Long
short	java.lang.Short
void	java.lang.Void

Wrapper Classes

- Converts primitive to wrapper
  - double a = 4.3;
  - Double wrp = new Double(a);
- Each wrapper provides a method to return the primitive value.
  - double r = wrp.doubleValue();

Converting Primitive Types to Wrapper Objects

- Integer ValueOfInt = new Integer(v)
- Float ValueOfFloat = new Float(x)
- Double ValOfDouble = new Double(y)
- Long ValueOfLong = new Long(z)
- Where v, x, y and z are int, float, double, and long values, respectively.

Converting Wrapper Objects to Primitives

- int v = ValueOfInt.intValue();
- float x = ValueOfFloat.floatValue();
- long y = ValueOfLong.longValue();
- double z = ValueOfDouble.doubleValue();

Converting Primitives to String Object

- String xyz = Integer.toString(v)
- String xyz = Float.toString(x)
- String xyz = Double.toString(y)
- String xyz = Long.toString(z)

Converting back from String Object to Primitives

- int v = Integer.parseInt(xyz)
- long y = Long.parseLong(xyz)
- public float parseFloat(String x)
- public double parseDouble(String x)
- public double parseByte(String x)
- public double parseShort(String x)
- May throw *NumberFormatException* if the value of the string does not represent a proper number.

Converting Primitives Represented by String to Wrapper

- Double ValueOfDouble = Double.valueOf(xyz);
- Float ValueOfFloat = Float.valueOf(xyz);
- Integer ValueOfInteger = Integer.valueOf(xyz);
- Long ValueOfLong = Long.valueOf(xyz);
- Double ValueOfDouble = Double.valueOf(xyz);
- Float ValueOfFloat = Float.valueOf(xyz);
- Integer ValueOfInteger = Integer.valueOf(xyz);
- Long ValueOfLong = Long.valueOf(xyz);
- String argument method generates a NumberFormatException in case the value in a string does not contain a number

Autoboxing and Unboxing

- Introduced in Java 5
- Conversion from primitives to wrappers is known as *boxing*, while the reverse is known as *unboxing*.
- Integer wrap_int = 5;
  - primitive 5 autoboxed into an integer
- int prim_int = wrap_int;
  - automatic unboxing integer into int

String Class

- Are basically immutable objects in Java.
- Immutable means once created, strings cannot be changed.
- Whenever we create strings, it is this class that is instantiated.
- In Java strings can be instantiated in two ways:
  - String x= “String Literal Object”;
  - String y=new String (“String object is created here”);

String Example

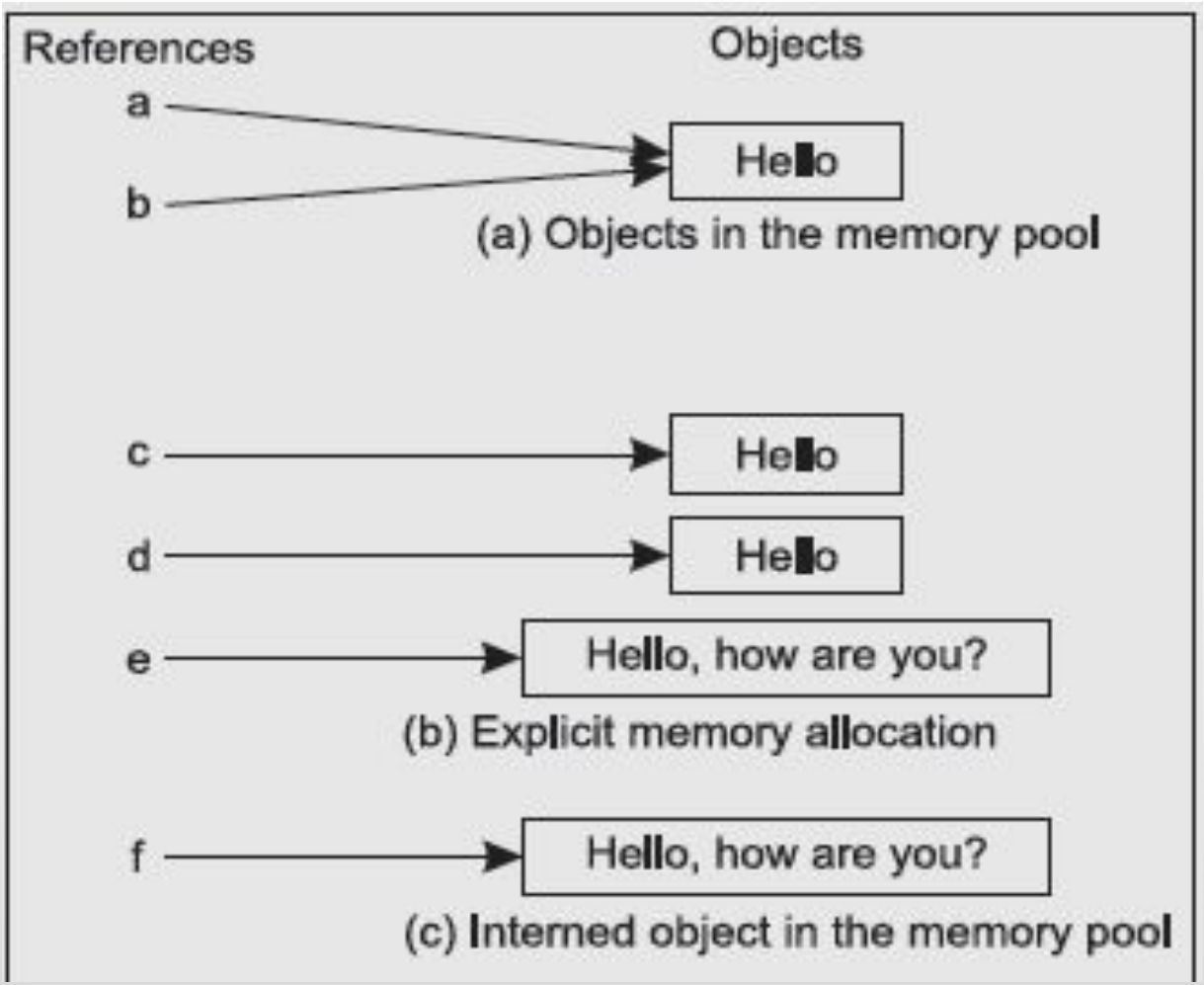
```
String a="Hello";
String b="Hello";
String c=new String("Hello");
String d=new String("Hello");
String e=new String("Hello, how are you?");
   if(a==b)
      System.out.println("object is same and is being shared by a & b");
   else
      System.out.println("Different objects");
   if(a==c)
      System.out.println("object is same and is being shared by a & c");
   else
      System.out.println("Different objects");
```

Example (contd.)

```
 if(c==d)
    System.out.println("same object");
else
    System.out.println("Different objects");
String f=e.intern();
if(f==a)
System.out.println("Interned object f refers to the already created object a in the pool");
else
System.out.println("Interned object does not refer to the already created objects, as literal was not present in the pool. It is a new object which has been created in the pool");
```

- We have omitted the class and the main method declaration from this example. You need to add it to run the example (refer Book for complete program).

String Example



The Output

object is same and is being shared by a & b

Different objects

Different objects

Interned object does not refer to the already created objects, as literal was not present in the pool. It is a new object which has been created in the pool.

String Manipulation

- Strings in Java are immutable (read only) in nature. Once defined cannot be changed.
- Let us take an example:
  - String x = “Hello”; // ok
  - String x = x + “World”; // ok, but how?
- The ‘+’ operator concatenates if at least one of the operands is a string.
- The second statement gets converted to the following statement automatically:
  - String x=new StringBuffer(). append(x). append(“World”). toString();

Common Methods of the String Class

Method Name with Signature	Method Details
int length()	to find length of the string (Line 1, Example 6.8)
boolean equals(String str)	Used to check equality of String objects. In contrast to == operator, the check is performed character by character. If all the characters in both the Strings are same, true is returned else false. (Line 2, Example 6.8)
int comparetTo(String s)	Used to find whether the invoking String (Figure 6.2) is Greater than, less than or equal to the String argument. It returns an integer value. If the integer value is a) < than zero – invoking string is less than String Argument b) > than Zero – invoking String is greater than String Argument c) = to Zero – invoking String and String argument are Equal (Line 3 – 9, Example 6.8)
boolean regionMatches(int startingIndx, String str,int strStartingIndx,int numChars)	Matches a specific region of String with specific region of the invoking String. The argument details : **startingIndx** – specifies the region from the invoking String to be matched. **str** – is the second string to be matched **strStartingIndx** – specifes the region from str to be matched with invoking String. **numChars** – specifies the number of character to be matched in both strings from their respective starting indexes. (Line 10, Example 6.8)

Methods of String Class

int indexOf(char c)	to find the index of a character in the invoking String object. (Line 11, Example 6.8)
int indexOf(String s)	Overloaded method to find the starting index of a String argument in the invoking String object. (Line 12, Example 6.8)
int lastIndexOf(char c)	to find the last occurrence of character in the invoking String. (Line 13, Example 6.8)
int lastIndexOf(String s)	Overloaded method to find the last occurrence of String argument in the invoking String object. (Line 14, Example 6.8)
String substring(int sIndex)	to extract the String from the invoking String Object starting with sIndex till the End of the String. (Line 15, Example 6.8)
String substring(int startingIndex, int endingIndex)	Overloaded method to extract the String starting with startingIndex till the endingIndex from the invoking String Object String. (Line 16, Example 6.8)
int charAt(int pos)	to find the character at a particular position(pos). (Line 17, Example 6.8)
String toUpperCase()	to change the case of entire String to Capital letters. (Line 18)
String toLowerCase()	to change the case of entire String to small letters. (Line 19)
boolean startsWith(String ss)	to find whether invoking String starts with String argument (Line 20)
boolean endsWith(String es)	to find whether invoking String ends with String argument (Line 21)
Static String valueOf(int is)	Converts primitive type int value to String. (Line 22)
Static String valueOf(float f)	Overloaded static method to Convert Primitive type float value to String.
Static String valueOf(long l)	Overloaded static method to Convert Primitive type long value to String.
Static String valueOf(double d)	Overloaded static method to Convert Primitive type double value to String.

StringBuffer Class

- StringBuffer class is used for representing changing strings.
- StringBuffer offers more performance enhancement whenever we change strings, because it is this class that is actually used behind the curtain.
- Just like any other buffer, StringBuffer also has a capacity and if the capacity is exceeded, then it is automatically made larger.
- The initial capacity of StringBuffer can be known by using capacity() method.

Methods of StringBuffer Class

Method Name with Signature	Method Details
int capacity()	Returns the current capacity of the storage available for character in the Buffer. (Line 2). When the capacity is approached the capacity is automatically increased. (Line 6)
StringBuffer append(String str)	appends String argument to the Buffer. (Line 3)
StringBuffer replace(int sindx,int eIndx,String str)	The characters from start to end are removed and str is inserted at that position (Line 4)
StringBuffer reverse()	Reverses the buffer character by character (Line 5)
Char charAt(int index)	Returns the character at specified index (Line 7)
Void setCharAt(int indx,char c)	Sets the specified character at specified index (Line 8)

StringBuilder Class

- Introduced in Java 5.
- A substitute of StringBuffer class.
- This class is faster than StringBuffer class, as it is not synchronized.
- append(), insert(), delete(), deleteCharAt(), replace(), and reverse() return StringBuilder objects rather than StringBuffer objects.
- The following line creates a StringBuilder object.
  - StringBuilder s=new StringBuilder();
  -construct a StringBuilder object with an initial capacity of 16 characters. Similar to that of StringBuffer.

Splitting Strings

- StringTokenizer is a utility class provided by the **java.util** package.
- Now a legacy code.
- This class used to be of utmost importance when we wanted to divide the entire string into parts (tokens) on the basis of delimiters.
- The delimiters can be any of the whitespace, tab space, semicolon, comma, etc.
- J2SE 1.4 added split( ) method to the **String** class for simplifying the task of splitting a string and also added **Pattern** and **Matcher** classes in java.util.regex package.

Example

```
import java.util.*; import java.util.regex.*;
class StringTokenizerDemo {
public static void main(String args[]) {
int i=1;
String str=”Never look down on anybody unless you’re helping him up”;
System.out.println(“Splitting String Using StringTokenizer class”);
StringTokenizer tr=new StringTokenizer(str,” “);
while (tr.hasMoreTokens()) {
    System.out.print(“Token “+i+” :”);
    System.out.println(tr.nextToken());
            ++i; }
System.out.println(“Splitting String Using split() method”);
String[] tk=str.split(“ “);
for(String tokens: tk)
    System.out.println(tokens);
Pattern p=Pattern.compile(“ “);
tk= p.split(str,3);
for(String tokens: tk)
    System.out.println(tokens); }}
```

Enum Type

- Is a kind of class definition.
- Subclass of Object class and inherits the comparable interface.
- Defines the type along with the possible set of enum values which are listed in the curly braces, separated by commas.
- All enum types are subclasses of the java.lang.Enum class.
- Each value in an enum is an identifier.
- The following statement declares a type, named *Games*, with the values CRICKET, FOOTBALL, TENNIS, and BASKETBALL.
  - enum Games { CRICKET, FOOTBALL, TENNIS, BASKETBALL };
- By convention, all names must be in upper case.

Enum Type

- Once a type is defined, you can declare a variable of that type:
  - Games G;
  - G can hold one of the values defined in the enumerated type *Games* or null, but nothing else.
  - An attempt to assign a value other than the enumerated values or null will result in a compilation error.
  - The enumerated values can be accessed using the syntax.
    - enumeratedTypeName.valueName
    - G = Games.TENNIS;

Enum Example

```
public class EnumDemo {
static enum Games {CRICKET, FOOTBALL, CHESS, BASKETBALL,
TENNIS,BADMINTON};
public static void main (String[] args) {
Games G1 = Games.CHESS;
Games G2 = Games.TENNIS;
System.out.println(“First game is “ +G1.name());
System.out.println(“Second game is “ +G2.name());
System.out.println(“First game’s ordinal is “ +G1. ordinal());
System.out.println(“Second game’s ordinal is “ +G2.ordinal());
System.out.println(“G1.equals(G2) returns “ +G1. equals(G2));
System.out.println(“G1.toString() returns “ +G1. toString());
System.out.println(“G1.compareTo(G2) returns “+G1.compareTo(G2)); }}
```

The Output

First game is CHESS

Second game is TENNIS

First game's ordinal is 2

second game's ordinal is 4

G1.equals(G2) returns false

G1.toString() returns Chess

G1.compareTo(G2) returns -2

for Loop with Enumeration

- Each enumerated type has a static method *values( )* associated with them that returns all enumerated values for the type in an array.
- For example,
  - Games[] G = Games.values();
- You can use **for loop** to process all the values in the array.
  - for (int i = 0; i < G.length; i++)
  - System.out.println(G[i]);

Conditional Statements with Enumeration

- Can use if or switch-case with enumerations to test the value in the variable, as shown below.

If statement:

```
if (G1.equals(Games.CRICKET)) {
      // action to be performed
} else if (G1.equals(Games.FOOTBALL)) {
      // action to be performed
} else ....
```

Conditional Statements with Enumeration

Switch Statement:

```
switch (Games) {
case CRICKET:     // case CRICKET and not                    //
Games.CRICKET
   // action to be performed;
case FOOTBALL:
   // action to be performed;
...
}
```

Attributes and Methods within Enumeration

```
public enum Desc {
   CRICKET (“Sachin Tendulkar”), CHESS (“Vishwanathan     Anand”),
         TENNIS (“Sania Mirza”);
   private String description;
   private Desc(String description) {
   this.description = description;    }
public String getDesc() {
System.out.print(“Indian Delight: ”);
return description;     } }
public class UseDesc {
public static void main(String[] args) {
Desc player = Desc.TENNIS; System.out.println(player.getDesc());
}}
```

Summary

- Java does not support multiple inheritance among classes.
- Multiple inheritance can be achieved using interfaces.
- Classes can be grouped together to form a Package.
- Package is a collection of Java files similar to a directory.
- A fundamental predefined package is discussed in this chapter: *java.lang* package.
- For all the primitive data types, wrapper classes are defined which encapsulate the functionality of the primitive data types.
- A few other classes like Object, String, StringBuffer and StringBuilder are frequently used in programming.
- All classes, whether predefined or user-defined, inherit ultimately from the Object class implicitly.
- Enum type is a kind of class and is useful when we know the type along with the possible set of values in that type.