Unit 1(Part2)

JAVA Programming

Primitive vs reference types

	Primitive types	**Reference types**
1)	The primitive types are boolean, byte, char, short, int, long, float and double.	All other types are reference types, so classes, which specify the types of objects, are reference types.
2)	A primitive-type variable can store exactly one value of its declared type at a time.	Programs use variables of reference types (called references) to store the location of an object in the computer's memory.
3)	Primitive-type instance variables are initialized by default. Variables of types byte, char, short, int, long, float and double are initialized to 0 and boolean are initialized to false.	Reference-type fields are initialized by default to the value null.
4)	A primitive-type variable does not refer to an object and therefore cannot be used to invoke a method.	A reference to an object is required to invoke an object's instance methods.

Wrapper Classes

- Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.
- Sometimes you must use wrapper classes, for example when working with Collection objects, such as ArrayList, where primitive types cannot be used.

Primitive Data Type	**Wrapper Class**
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Autoboxing and Unboxing

- *Autoboxing* is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes.
- For example, converting an int to an Integer, a double to a Double, and so on.
- If the conversion goes the other way, this is called *unboxing*.

Here is the simplest example of autoboxing:

- Character ch = 'a';

Interfaces

- An interface is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.
- An interface is not a class. Writing an interface is similar to writing a class, but they are two different concepts. A class describes the attributes and behaviors of an object. An interface contains behaviors that a class implements.
- Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

- **An interface is similar to a class in the following ways:**
  - An interface can contain any number of methods.
  - An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
  - The bytecode of an interface appears in a **.class** file.
  - Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

- **However, an interface is different from a class in several ways, including:**
  - You cannot instantiate an interface.
  - An interface does not contain any constructors.
  - All of the methods in an interface are abstract.
  - An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
  - An interface is not extended by a class; it is implemented by a class.
  - An interface can extend multiple interfaces.

```
/* File name : NameOfInterface.java */
import java.lang.*;
//Any number of import statements

public interface NameOfInterface
{ //Any number of final, static fields
//Any number of abstract method
declarations\
}
```

- Interfaces have the following properties:
- An interface is implicitly abstract. You do not need to use the **abstract** keyword when declaring an interface.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

-
  ```
  interface Bicycle {
          // wheel revolutions per minute
          void changeCadence(int newValue);
          void changeGear(int newValue);
           void speedUp(int increment);
           void applyBrakes(int decrement);      }

  class ACMEBicycle implements Bicycle {
           // remainder of this class
           // implemented as before
    }
  ```

Inheritance

- Inheritance can be defined as the process where one object acquires the properties of another.
- When we talk about inheritance the most commonly used keyword would be **extends** and **implements.**

  Example:

  ```
  public class Animal{ }
  public class Mammal extends Animal{ }
  public class Reptile extends Animal{ }
  public class Dog extends Mammal{ }
  ```

The super keyword

The **super** keyword is similar to **this** keyword. Following are the scenarios where the super keyword is used.

- It is used to **differentiate the members** of superclass from the members of subclass, if they have same names.
- It is used to **invoke the superclass** constructor from subclass.

Polymorphism

- Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
- It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared the type of a reference variable cannot be changed.

Overriding

- If a class inherits a method from its superclass, then there is a chance to override the method provided that it is not marked final.
- The benefit of overriding is: ability to define a behavior that's specific to the subclass type, which means a subclass can implement a parent class method based on its requirement.

Arrays

- **ARRAYS**

  An Array is a group of contiguous or related data items that share a common name. For instance, we can define an array name salary to represent a set of group of employees. A particular value is indicated by writing number called *index* number or subscript in brackets after the array name. For example

  Salary [10];

  represents the salary of the 10th employee.

STRINGS

String manipulation is the most common part of many Java programs. Strings represent a sequence of characters. The easiest way to represent a sequence of characters in Java is using a character array.

Example,

```
char        charArray [ ] = new char [4];
charArray [0] = ‘J’;
charArray [1] = ‘a’;
charArray [2] = ‘v’;
charArray [3] = ‘a’;
```

Although character arrays have the advantage of being able to query their length, they themselves are not good enough to support the range of operations we may like to perform on string.

In Java strings are class objects and implemented using two classes, namely, **String** and **StringBuffer.** A Java string is an instantiated object of String class. Java string is not a character array and is not NULL terminated. Strings may be declared and created as follows:

String stringname;

stringname = new String ("Hello");

<u>String Arrays</u>

We can also create and use arrays that contain strings. The statement

String itemArray [ ] = new String [3];

will create an itemArray of size 3 to hold three string constants.

String Methods

Method Call	Task performed
s2 = s1.toLowerCase( )	Converts the string s1 to all lowercase
s2 = s1.toUpperCase( )	Converts the string s1 to all uppercase
s2 = s1.replace('x','y')	Replace all appearances of x to y
s2 = s1.trim( )	Removes white spaces at the beginning and end of the string s1
s2.equals(s1)	Returns 'true' if s1 is equal to s2
s2.equalsIgnoreCase(s1)	Returns 'true' if s1=s2, ignoring the case of characters
s1.length ( )	Gives the length of s1
s1.charAt (n)	Returns the character at the specified index n.
s1.compareTo (s2)	Returns negative if s1 < s2, positive if s1 > s2 and zero if s1 equal to s2
s1.concat(s2)	Concatenates s1 and s2
s1.substring(n)	Gives substring starting from nth character
s1.substring(n,m)	Gives substring starting from nth character up to mth (not including mth)
String.valueOf(p)	Creates a string object of the parameter p
p.toString ()	Creates a string representation of the object p.
s1.indexOf('x')	Gives the position of the first occurrence of 'x' in the string s1.
s1.indexOf('x', n)	Gives the position of 'x' that occurs after nth position in the string s1
String.valueOf(variable)	Converts the parameter value to string representation

StringBuffer Class

StringBuffer class is peer class String. While String creates strings of fixed_length, StringBuffer creates strings of flexible length that can be modified in terms of both length and content. We can insert characters and substrings in the middle of a string, or append another string to end.

Methods of StringBuffer class

Method Call	Task performed
s1.setCharAt (n, ‘x’)	Modifies the nth character to x
s1.append (s2)	Appends the string s2 to s1 at the end
s1.insert (n , s2)	Inserts the string s2 at the position n of the string s1
s1.setLength (n)	Sets the length of string s1 to n. If n < s1.length() s1 is truncated. If n > s1.length() then zeros are added to s1.

Use of final, finally and finalize

- **Final:-** It is used in the following three cases:If the final keyword is attached to a variable then the variable becomes constant i.e. its value cannot be changed in the program.

  If a method is marked as final then the method cannot be overridden by any other method.

  If a class is marked as final then this class cannot be inherited by any other class.

  example

With a variable final int c=1;

With a method

```
class A
{
    final void func()
    {       System.out.println("Inside class A"); }
}
 class B extends A
{         void func() // Error, cannot override
          { ...
```

- With a class
  final class A { … }
  class B extends class A{ //Error, cannot inherit }

- **Finally:-** If an exception is thrown in try block then the control directly passes to the catch block without executing the lines of code written in the remainder section of the try block. In case of an exception we may need to clean up some objects that we created. If we do the clean-up in try block, they may not be executed in case of an exception. Thus finally block is used which contains the code for clean-up and is always executed after the try ...catch block.

```
class call
{
      static void callA()
                  {
                  try
                  {

                  } catch( Exception e) {
                              System.out.println("Exception caught");
                                          }
                  finally {
                  System.out.println("finally of callA"); ...//clean-up
code written here which is executed }
                  }
       }
```

- **Finalize:-** It is a method present in a class which is called before any of its object is reclaimed by the garbage collector. Finalize() method is used for performing code clean-up before the object is reclaimed by the garbage collector.

```
public class MyClass
{
        protected void finalize()
       { ... //code written here for clean-up
       }
}
```

Generics

- Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.
- Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.
- Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

- Generic Methods- You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.

- Generic Classes-A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section. As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas.

```java
public class Box<T> {
   private T t;

   public void add(T t) {
      this.t = t;
   }

   public T get() {
      return t;
   }

   public static void main(String[] args) {
      Box<Integer> integerBox = new Box<Integer>();
      Box<String> stringBox = new Box<String>();

      integerBox.add(new Integer(10));
      stringBox.add(new String("Hello World"));

      System.out.printf("Integer Value :%d\n\n", integerBox.get());
      System.out.printf("String Value :%s\n", stringBox.get());
   }
}
```

Java Enums

- The **Enum in Java** is a data type which contains a fixed set of constants.
- The Java enum constants are static and final implicitly.
- Enums are used to create our own data type like classes. The **enum** data type is used to define an enum in Java.

```
class EnumExample1{
  public enum Season { WINTER, SPRING, SUMM
  ER, FALL }
  public static void main(String[] args) {
        for (Season s : Season.values())
                System.out.println(s);
  }}
```

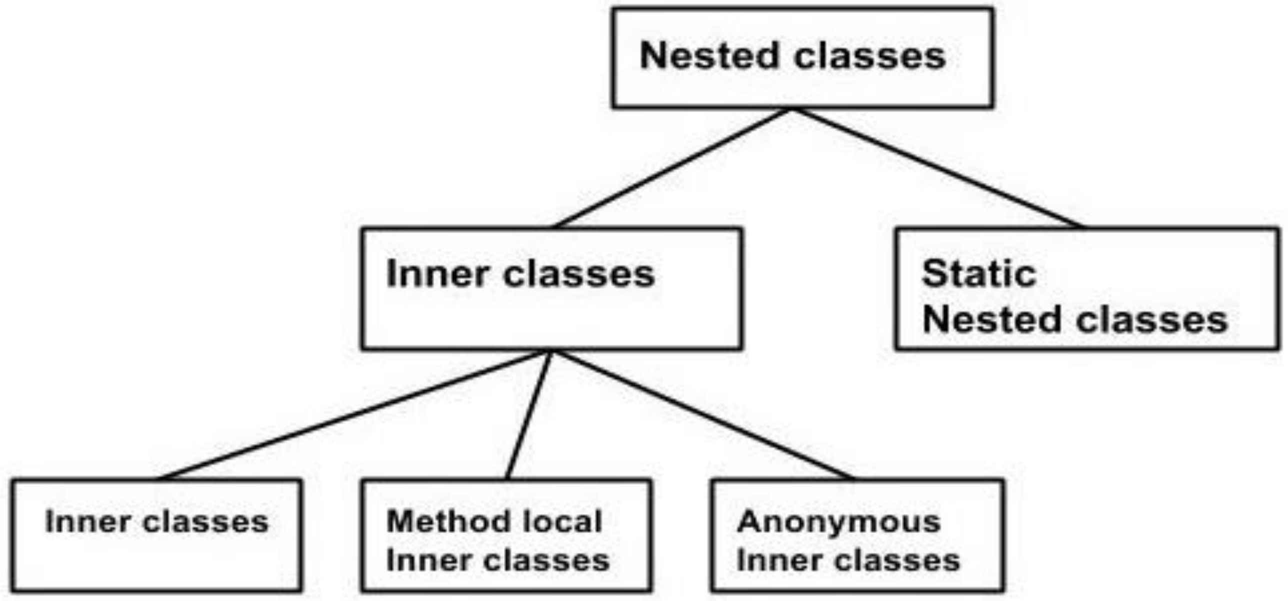
Inner classes

- Nested Classes- In Java, just like methods, variables of a class too can have another class as its member. Writing a class within another is allowed in Java. The class written within is called the **nested class**, and the class that holds the inner class is called the **outer class**.

```
class Outer_Demo {
        class Inner_Demo { }
}
```

Nested classes are divided into two types –

- **Non-static nested classes** – These are the non-static members of a class.
- **Static nested classes** – These are the static members of a class.



Inner Classes (Non-static Nested Classes)

- Inner classes are a security mechanism in Java. We know a class cannot be associated with the access modifier **private**, but if we have the class as a member of other class, then the inner class can be made private. And this is also used to access the private members of a class.
- Inner classes are of three types depending on how and where you define them. They are –
  - Inner Class
  - Method-local Inner Class
  - Anonymous Inner Class

- Inner Class: Creating an inner class is quite simple. You just need to write a class within a class. Unlike a class, an inner class can be private and once you declare an inner class private, it cannot be accessed from an object outside the class.

- Method-local Inner Class: In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method. A method-local inner class can be instantiated only within the method where the inner class is defined.

- Anonymous Inner Class: An inner class declared without a class name is known as an **anonymous inner class**. In case of anonymous inner classes, we declare and instantiate them at the same time. Generally, they are used whenever you need to override the method of a class or an interface. Example
  AnonymousInner an_inner = new AnonymousInner()