# UNIT - I

## CLASS FEATURES

# Garbage Collection

- Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation.
- In C++, dynamically allocated objects must be manually released by use of a delete operator.
- Java handles deallocation for you automatically.
- The technique that accomplishes this is called garbage collection.
- **It works like this:** when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- There is no need to explicitly destroy objects.

# Garbage Collection contd…

**Nulling a reference:**

```
Patient p1=new Patient();
  p1=null;
```

**Assigning a reference to another:**

```
Patient p1 = new Patient();
Patient p2 = new Patient();
p1=p2;
```

**Anonymous object:**

```
new Patient();
```

# Pgm1

```
public class GarbageCollection1{
 public void show()
   {
     System.out.println("object is garbage collected");
   }
 public static void main(String args[]){
  GarbageCollection1 s1=new GarbageCollection1();
  GarbageCollection1 s2=new GarbageCollection1();
     s1=null; // calling garbage collector
     s2=null; // calling garbage collector
    System.gc();
 }
}
```

# the finalize () Method

- finalize() method is used to perform clean-up processing just before an object is garbage collected.
- It is used with objects.
- finalize method performs the cleaning with respect to the object before its destruction.
- finalize method is executed just before the object is destroyed.

# When to Use finalize() Method in Java?

- Garbage collection is done automatically in Java which is handled by JVM, and it uses finalize method in Java for releasing the resources of the object, that has to be destroyed.
- finalize() method gets called only once by GC, if an exception is thrown by finalizing method or the object revives itself from finalize(), the garbage collector will not call the finalize() method again.
- There are other ways to release the used resources in Java like the close() method in case of file handling or destroy() method. But, the issue with these methods is they don't work automatically, we have to call them manually every time.
- We can use finalize() method in final block. finally block will execute finalize() method even though the user has used close() method manually.
- We can use finalize() method to release all the resources used by the object, in finally block by overriding finalize() method.

# Introducing Access Control

- Encapsulation links data with the code that manipulates it.
- In encapsulation provides another **important attribute: access control.**
- Through encapsulation, we can control **what parts of a program can access the members of a class.**
- By controlling access, you can prevent misuse.

# Public

- Java Compiler going to call the program. But compiler is separate software program.
- Compiler wants to access program which is in outside the package.
-

```
// MedicalRecord.java
   class MedicalRecord {
    // Public fields
    public String patientName;
    public int patientAge;
    public String diagnosis;


    // Public method
    public void displayMedicalRecord() {
        System.out.println("Patient Name: " + patientName);
        System.out.println("Patient Age: " + patientAge);
        System.out.println("Diagnosis: " + diagnosis);
    }
    public void setValues(String pname, int page, String pdiagnosis)
    {
     patientName = pname;
     patientAge = page;
     diagnosis = pdiagnosis;
    }
}
// Main.java
public class AccessModifier1 {
    public static void main(String[] args) {
        // Create a MedicalRecord object
        MedicalRecord patientRecord = new MedicalRecord();
        patientRecord.setValues("Aryan", 12, "Completed");

        patientRecord.displayMedicalRecord();
    }
}
```

# Private

- Within class variable and method can be access, but outside class or project is not able to access.

# Private - Variable is accessed because its within class.
# Method not possible baecasue its accessing from outside class

```
// MedicalRecord.java
   class MedicalRecord {
    // Public fields
    private String patientName;
    public int patientAge;
    public String diagnosis;


    // Public method
    Private void displayMedicalRecord() {
        System.out.println("Patient Name: " + patientName);
        System.out.println("Patient Age: " + patientAge);
        System.out.println("Diagnosis: " + diagnosis);
    }
    public void setValues(String pname, int page, String pdiagnosis)
    {
     patientName = pname;
     patientAge = page;
     diagnosis = pdiagnosis;
    }
}
// Main.java
public class AccessModifier2 {
    public static void main(String[] args) {
        // Create a MedicalRecord object
        MedicalRecord patientRecord = new MedicalRecord();
        patientRecord.setValues("Aryan", 12, "Completed");

        patientRecord.displayMedicalRecord();
    }
}
```

# Protected

# This program will throw error because protected variable not possible to access from outside class and package

```
// MedicalRecord.java
   class MedicalRecord {
    // Public fields
    protected String patientName;
    public int patientAge;
    public String diagnosis;


    // Public method
    public void displayMedicalRecord() {
        System.out.println("Patient Name: " + patientName);
        System.out.println("Patient Age: " + patientAge);
        System.out.println("Diagnosis: " + diagnosis);
    }
    public void setValues(String pname, int page, String pdiagnosis)
    {
     patientName = pname;
     patientAge = page;
     diagnosis = pdiagnosis;
    }
}
// Main.java
public class AccessModifier3 {
    public static void main(String[] args) {
        // Create a MedicalRecord object
        MedicalRecord patientRecord = new MedicalRecord();
        patientRecord.setValues("Aryan", 12, "Completed");
        patientRecord.displayMedicalRecord();
        System.out.println("Patient Name: " + patientName);
    }
}
```

```
// MedicalRecord.java
   class MedicalRecord {
    // Public fields
    protected String patientName;
    public int patientAge;
    public String diagnosis;


    // Public method
    public void displayMedicalRecord() {
        System.out.println("Patient Name: " + patientName);
        System.out.println("Patient Age: " + patientAge);
        System.out.println("Diagnosis: " + diagnosis);
    }
    public void setValues(String pname, int page, String pdiagnosis)
    {
     patientName = pname;
     patientAge = page;
     diagnosis = pdiagnosis;
    }
    public String getvalue()
       {
         return patientName;
       }

}
// Main.java
public class AccessModifier3 {
    public static void main(String[] args) {
        // Create a MedicalRecord object
        MedicalRecord patientRecord = new MedicalRecord();
        patientRecord.setValues("Aryan", 12, "Completed");
        patientRecord.displayMedicalRecord();

        String pa2 = patientRecord.getvalue();
        System.out.println("Patient Name2: " + pa2);
    }
}
```

# Access Modifiers in Java

| Access Modifier | within class | within package | outside package by subclass only | outside package |
|---|---|---|---|---|
| **Private** | Y | N | N | N |
| **Default** | Y | Y | N | N |
| **Protected** | Y | Y | Y | N |
| **Public** | Y | Y | Y | Y |

# About final keyword

- **Final variables:** When a variable is declared as final, **its value cannot be changed once it has been initialized.** This is useful for declaring constants or other values that should not be modified.

- **Final methods:** When a method is declared as final, **it cannot be overridden by a subclass.** This is useful for methods that are part of a class's public API and should not be modified by subclasses.

- **Final classes:** When a class is declared as final, **it cannot be extended by a subclass.** This is useful for classes that are intended to be used as is and should not be modified or extended.

# When to use a final variable?

- The primary distinction between a **normal variable and a final variable lies in their mutability.**
- While a normal variable allows for the reassignment of its value, a final variable, once assigned a value, **remains constant throughout the execution of the program;** it cannot be altered or reassigned.

# Difference of these two programs

```
class Static3 {

        final int a=5;
    public static void main(String[] args)
    {

          a= 6;
      System.out.println(a);

    }
}
```

```
class Static3 {

        final int a=5;
    public static void main(String[] args)
    {

          int a= 6;
      System.out.println(a);

    }
}
```

# sample program for reference final variable

```
class Static2 {

        public static void main(String[] args)
    {

         final StringBuilder obj1 = new StringBuilder("Java");

         System.out.println(obj1 );

        obj1 .append("Program");

        System.out.println(obj1 );
    }
}
```

# Final classes

- One is definitely to prevent inheritance, as final classes cannot be extended.

```
final class A
{
    // methods and fields
}
// The following class is illegal
class B extends A
{
    // COMPILE-ERROR! Can't subclass A
}
```

# Final Methods

- When a method is declared with final keyword, it is called a final method.
- A final method cannot be overridden.

```
class A
{
    final void m1()
    {
        System.out.println("This is a final method.");
    }
}

class B extends A
{
    void m1()
    {
        // Compile-error! We can not override
        System.out.println("Illegal!");
    }
}
```

# About static

**Methods declared as static have several restrictions:**

- They can only directly call other static methods of their class.
- They can only directly access static variables of their class.
- They cannot refer to this or super in any way.

# Static Sample Pgm

```
class Static1
{
  static int PaId=10;
  static int PaAge=25;

        static void method1(String a)
         {

         System.out.println("Patient Name: "+ a);
         System.out.println("Patient ID"+ PaId);
         System.out.println("Patient Age"+ PaAge);

         }

              static {
                   System.out.println("static Block");
                   PaAge = PaAge + 1;
              }

                   public static void main (String args[])
                            {
                              method1("Sheeba");
                            }
}
```

# Nested Classes

The Java programming language allows you to define a class within another class. Such a class is called a nested

class and is illustrated here:

```
class OuterClass {
    …
    class NestedClass {
        …
    }

}
```

# Static Nested Classes

Terminology: Nested classes are divided into two categories: non-static and static. Non-static nested classes are called inner classes. Nested classes that are declared static are called static nested classes.

```
class OuterClass {
                    ...

    class InnerClass {
                        …
                }
static class StaticNestedClass {
                            ...
                        }
            }
```

# Introducing nested and inner classes

## The Program illustrates how to define and use an inner class

```
class Outer
{
int outer_x=100;
int y =20;

void test()
        {
        InnerClass  i1 = new InnerClass ();
        i1.display();
        i1.show();
        }
        class InnerClass
                {
                int y =10;
                void display()
                        {System.out.println("Display: outer_x=" +outer_x);}
                                void show()
                                        {System.out.println(y);}
                        }
}

class InnerClassDemo
{
public static void main(String args[])
        {
        Outer outer = new Outer();
        outer.display();


        }
}
```

# The Program illustrates how to define and use an inner class

```
class Outer
{
int outer_x=100;
int y =20;

void test()
        {
        InnerClass  i1 = new InnerClass ();
        i1.display();
        i1.show();
        }
        class InnerClass
                {
                int y =10;
                void display()
                        {System.out.println("Display: outer_x=" +outer_x);}
                                void show()
                                        {System.out.println(y);}
                        }
}

class InnerClassDemo
{
public static void main(String args[])
        {
        Outer outer = new Outer();
        //outer.display();
        outer.test();

        }
}
```

# String class

```
class StringDemo
{
    public static void main(String args[])
    {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    String strOb3 = strOb1 + " and " + strOb2;

    System.out.println(strOb1);
    System.out.println(strOb2);
    System.out.println(strOb3);
    }

}
```

```
class StringDemo2
{
    public static void main(String args[])
    {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    String strOb3 = strOb1 + " and " + strOb2;

    System.out.println("Length of strOb1:" +strOb1.length());
    System.out.println("Char at index 3 in: " +strOb1.charAt(3));
    System.out.println(strOb3);
    if(strOb1.equals(strOb2))
    System.out.println("strOb1 == strOb2");
    else
    System.out.println("strOb1 != strOb2");

    }


}
```

# Types of String Methods

- charAt() Returns the character at the specified index (position) char
- codePointAt() Returns the Unicode of the character at the specified index int
- codePointBefore() Returns the Unicode of the character before the specified index int
- codePointCount() Returns the number of Unicode values found in a string. int
- compareTo() Compares two strings lexicographically int
- compareToIgnoreCase() Compares two strings lexicographically, ignoring case differences int
- concat() Appends a string to the end of another string String
- contains() Checks whether a string contains a sequence of characters boolean
- contentEquals() Checks whether a string contains the exact same sequence of characters of the specified CharSequence or StringBuffer boolean

- copyValueOf() Returns a String that represents the characters of the character array String
- endsWith() Checks whether a string ends with the specified character(s) boolean
- equals() Compares two strings. Returns true if the strings are equal, and false if not boolean
- equalsIgnoreCase() Compares two strings, ignoring case considerations boolean
- format() Returns a formatted string using the specified locale, format string, and arguments String
- getBytes() Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array byte[]
- getChars() Copies characters from a string to an array of chars void
- hashCode() Returns the hash code of a string int

# String Buffer Class

- StringBuffer is a peer class of String that provides much of the functionality of strings.
- The string represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences.
- StringBuffer may have characters and substrings inserted in the middle or appended to the end.
- It will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.
- **StringBuffer class is used to create mutable (modifiable) string**.
- The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

# append() method - concatenates the given argument with this string.

```
import java.io.* ;

class SB1 {
    public static void main(String args[])
    {
        StringBuffer sb = new StringBuffer("Apollo Hospital ");
        sb.append("BTM Layout Branch");
        System.out.println(sb);
    }
}
```

# insert() method - inserts the given string with this string at the given position.

```
import java.io.* ;

class SB2 {
    public static void main(String args[])
    {
        StringBuffer sb = new StringBuffer("Apollo ");
        sb.insert(7, "BTM Layout");
        System.out.println(sb);
    }
}
```

# replace() method - replaces the given string from the specified beginIndex and endIndex

```
import java.io.* ;

class SB3{
    public static void main(String args[]){
        StringBuffer sb=new StringBuffer("Apollo Hospital");
        sb.replace(0,6, "Manipal");
        System.out.println(sb);
    }
}
```

# delete() method - deletes the string from the specified beginIndex

```
import java.io.* ;

class SB4{
    public static void main(String args[]){
        StringBuffer sb=new StringBuffer("Apollo Hospital");
        sb.delete(0,2);
        System.out.println(sb);
    }
}
```

# reverse() method - method of StringBuilder class reverses the current string

```
import java.io.* ;

class SB5{
    public static void main(String args[]){
        StringBuffer sb=new StringBuffer("Apollo");
        sb.reverse();
        System.out.println(sb);
    }
}
```

# Methods of StringBuffer class

| Methods | Action Performed |
|---|---|
| append() | Used to add text at the end of the existing text. |
| length() | The length of a StringBuffer can be found by the length( ) method |
| capacity() | the total allocated capacity can be found by the capacity( ) method |
| charAt() | This method returns the char value in this sequence at the specified index. |
| delete() | Deletes a sequence of characters from the invoking object |
| deleteCharAt() | Deletes the character at the index specified by *loc* |

# Methods of StringBuffer class

| Methods | Action Performed |
|---|---|
| ensureCapacity() | Ensures capacity is at least equals to the given minimum. |
| insert() | Inserts text at the specified index position |
| length() | Returns length of the string |
| reverse() | Reverse the characters within a StringBuffer object |
| replace() | Replace one set of characters with another set inside a StringBuffer object |
| | |

# Command Line Arguments

- Sometimes we want to pass information into a program during time of run the program.
- This is accomplished by passing command-line arguments to main( ).
- A command-line argument is the information that directly follows the program's name on the command line when it is executed.

# Sample Program

```
class CommandLine1
{
public static void main(String args[])
    {

for(int i=0;i<args.length;i++)
System.out.println("Your first argument is: "+args[i]);

    }
}
```

```
C:\Java Programs>javac CommandLine1.java

C:\Java Programs>java CommandLine1

C:\Java Programs>java CommandLine1 Java program
Your first argument is: Java
Your first argument is: program

C:\Java Programs>
```

# Sample program 2

```
class CommandLine2 {

    public static void main(String[] args)
    {

        if (args.length > 0) {

            // Print statements
            System.out.println("The command line"
                               + " arguments are:");

            // Iterating the args array
            // using for each loop
            for (String val : args)

                // Printing command line arguments
                System.out.println(val);
        }

        else

            // Print statements
            System.out.println("No command line "
                               + "arguments found.");
    }
}
```