Java Basics:
Part 3

(Object Passing, Static, Nested Class, I/O, Wrapper)

UD

July 2023

Recap of Class, Object & Constructor

- Discussed with examples in previous classes
- Quick review (identify classes, objects & constructors):

```
class MyClass {
      int x;
      MyClass() {
            x = 5;
      }
      MyClass( int a ) {
            x = a;
      }
}

class ConsDemo {
      public static void main( String[] args ) {
            MyClass t1 = new MyClass();
            MyClass t2 = new MyClass( 10 );
            System.out.println( t1.x + " " + t2.x );
      }
}
```

Note: **Constructors doesn’t have any return type**

Creation of Class, Object & Constructor

- What will happen when the constructors are removed?

```
class MyClass {
      int x;
      /* MyClass() {
            x = 5;
      }
      MyClass( int a ) {
            x = a;
      } */
}

class ConsDemo {
      public static void main( String[] args ) {
            MyClass t1 = new MyClass(); t1.x = 5;
            MyClass t2 = new MyClass( /* 10 */ ); t2.x = 10;
            System.out.println( t1.x + " " + t2.x );
      }
}
```

Method Overloading

- Multiple **methods with same name but different parameters**
  - Different signatures achieved with difference in number / types of parameters
  - Difference in return type is insufficient for overload resolution
- C used abs( ), labs( ) or fabs( ) for essentially the same thing. Why?
  - Java deals that with one method name thru method overloading (**within the same class**)
- Java uses as one of the ways for implementing Polymorphism, e.g. static polymorphism

```
class SampleOvld {
  void f(int x){ System.out.println("f/int/" + x); }
  void f(double x){ System.out.println("f/dbl/" + x); }
  void f(int x, int y){ System.out.println("f/int/int/" + x +
" " + y); }
}
...
SampleOvld ob = new SampleOvld();
int i=10; double d=10.1; byte b=99; short s=10; float
f=11.5F;
// Check following - any type conversions?
Ob.f(i); Ob.f(d); Ob.f(b); Ob.f(s); Ob.f(f); Ob.f(i, b);
```

Parameter vs Instance Variable

Check below program:

```
class Student {
    int id;
    String name;

    Student ( int id, String name ) {
            id = id;
            name = name;
    }
    void display(){ System.out.println (id+" "+name); }

    public static void main( String args[] ) {
            Student s1 = new Student (111,"ABC");
            Student s2 = new Student (222,"XYZ");
            s1.display();
            s2.display();
    }
}
```

Output:

```
0 null
0 null
```

this keyword

Reference variable that refers to the current object

```
class Student{
   int id;
   String name;

   Student( int id, String name ) {
         this.id = id;
         this.name = name;
   }
   void display(){ System.out.println( id+" "+name ); }
   public static void main( String args[] ){
         Student s1 = new Student (111,“ABC");
         Student s2 = new Student (222,“XYZ");
         s1.display();
         s2.display();
   }
}
```

Output:

```
111 ABC
222 XYZ
```

Use of objects as parameter

```
class Rectangle {
    int length; int width;

    Rectangle(int l, int b) {length = l; width = b;}

    void area(Rectangle r1) {
        int areaOfRectangle = r1.length * r1.width;
        System.out.println("Area of Rectangle : " +
            areaOfRectangle);
    }
}

class RectangleDemo {// main class
    public static void main(String args[]) {
        Rectangle r1 = new Rectangle(10, 20);
        r1.area(r1);
    }
}
```

```
Area of Rectangle
: 200
```

Method returning object

```
class Rectangle {
      int length; int breadth;

      Rectangle(int l,int b) {length = l;breadth = b;}

      Rectangle getRectangleObject() {
        Rectangle rect = new Rectangle(10,20); return rect;
      }
}

class RetOb {
      public static void main(String args[]) {
        Rectangle ob1 = new Rectangle(40,50); Rectangle ob2;

        ob2 = ob1.getRectangleObject();
        System.out.println("ob1.length : " + ob1.length +
"ob1.breadth: " + ob1.breadth);
        System.out.println("ob2.length : " + ob2.length +
"ob2.breadth: " + ob2.breadth);
        }
}
```

```
ob1.length : 40
ob1.breadth: 50
ob2.length : 10
ob2.breadth: 20
```

Call by value & Call by reference

Call by value: No change in original value

Call by reference: **Not applicable directly**, however object references passed by value achieves Call By Reference

```
class Operation{
     int data=50;

    void change (int data){
        data = data + 100; //changes will be in the local variable only
    }

    public static void main(String args[]){
        Operation op=new Operation();

        System.out.println("before change "+op.data);
        op.change(500);
        System.out.println("after change "+op.data);
    }
}
```

Output:

```
before change 50
after change 50
```

Call by value & Call by reference (contd)

```
class Operation2{
        int data=50;

        void change(Operation2 op){  op.data=op.data+100; }


        public static void main(String args[]){
                Operation2 op=new Operation2();

                System.out.println("before change "+op.data);
                op.change(op);//passing object
                System.out.println("after change "+op.data);
        }
}
```

```
Output:
before change 50
after change 150
```

Static variables & Methods

Used mainly for memory management

- with variables, methods, blocks and nested class

Static variables:

- Known as Class Variables; used for maintaining information at the class level rather than object
- Such variables get default values based on data type (0 for integer, null for String)
- Common data storage for all the objects of that Class.
- Memory allocation happens only once when the class is loaded in the memory
- Can be accessed in any other class using class name.
- Unlike **non-static variables**, such variables can be accessed directly in static and non-static methods

Static variables

```
class Sample {
    int x;            // different storage for different objects
    static int y;     // common storage across all objects
}

class DemoSample {
    public static void main(String args[]) {
        Sample s1 = new Sample();
        Sample s2 = new Sample();

        s1.x = 100;
        s1.y = 200;

        s2.x = 40;
        s2.y = 50;

        System.out.println(s1.x + " " + s1.y);
        System.out.println(s2.x + " " + Sample.y);
    }
}
```

Output:

```
100 50
40 50
```

Static Methods

- Belongs to the class rather than object of a class; called independently of any object
- Can be invoked without the need for creating an instance of a class
- Can access static data member, but canoot access non-static member

```
class Student {
    int rollno;  String name;  static String college = "???";
    static void change() {   college = "TMSL";   }

    Student (int r, String n) {  rollno = r; name = n;   }

    void display () {System.out.println(rollno+" "+name+" "+college);}

    public static void main(String args[]) {
        Student.change();  // assign college
        Student s1 = new Student (10,"Abhi");
        Student s2 = new Student (20,"Ipsita");
        Student s3 = new Student (30,"Sania");

        s1.display();  s2.display();    s3.display();
    }
}
```

```
Output:
10 Abhi TMSL
20 Ipsita TMSL
30 Sania TMSL
```

Static Methods

Restrictions:

- Can not use non static data member or call non-static method directly
- **this** and **super** cannot be used in static context

What is the ouput of following program?

```
class X {
        int a;
        public static void main( String args[] ) {
                a = 40; System.out.println(a/3);
        }
}
```

Why java main method is static?

- Object creation is not required to call a static method
- If it were a non-static method, JVM will need to create object first before calling main() method → meaning extra and unnecessary memory allocation

Static Block

```
class stat {
        static int i;                // static variable
        static String str;           // static variable

         static {      // static block
                  System.out.println("Executing static block..");
                  i = 100;
                  str = "TMSL";
         }

         public static void main(String args[]) { // static method
            System.out.println("STARTING from main()..");
            System.out.println("i: "+i+" str: " + str);
         }
}
```

Output:

```
Executing static block..
STARTING from main()..
i: 100 str: TMSL
```

Notes:

1) Click here for more details on static block.

2) You can also have a static class. But, outer class cannot be a static class (click for details); only nested (inner) classes can be static.

Nested classes

- Class defined within another Java class → logical grouping of classes & increased encapsulation

- Part of its enclosing class → both enclosing and enclosed class can access members of each other

- Allows nesting classes either as members or within blocks of code

- Both classes and interfaces can be nested

Nested class Example

```
public class OuterClass {

  private int outerMem = 10;

  class InnerClass {
    private int innerMem = 99;
    public void getOuterMem() { System.out.println(outerMem); }
  }

  public static void main(String[] args) {
      OuterClass oc = new OuterClass();
      OuterClass.InnerClass ic =
          new OuterClass().new InnerClass();
      ic.getOuterMem();
  }

}
```

Static Nested classes

- Static nested class can be accessed without outer class instance

- For instantiating a static nested class, no reference is needed to the outer class object

- Non-static nested classes are called Inner Classes

- All inner classes are nested classes, but not all nested classes (e.g. static nested classes) are inner classes

Command line arguments

```
public class ParseInt {
   static int sum = 0;
   public static void main( String[] args) {
     for ( int i=0; i < args.length; i++ ) {
        System.out.println( args[i] );
        sum = sum + Integer.parseInt(args[i]);
     }
     System.out.println( "sum: "+ sum );
   }
}
$ java ParseInt 12 34
```

Output:

```
12
34
sum: 46
```

Basics of I/O operations

- Java Uses ‘stream’ for faster I/O operations
  - Stream: an abstraction for producing or consuming information with sequence of data bytes

  - Built-in streams (← java.lang package):

    System.out: standard output (console *by default)*;
    object of type PrintStream class
    System.in: standard input (keyboard *by default*);
    object of type InputStream class
    System.err: standard error (console *by default*);
    object of type PrintStream class
    in, out & err: declared as public, final & static… ***Why?***

  - Can be redirected to any compatible I/O device

    Quiz: What is expected from the following code snippet?

    ```
    System.setOut( new PrintStream (new
    BufferedOutputStream( new FileOutputStream(
    "op.log" ))));
    ```

Basics of I/O operations (contd.)

- java.io package defines stream implementation within class-hierarchies

- Byte-oriented I/O uses following types:

  (i) Byte Stream

  - reading / writing binary data
  - specially useful for file handling

  (ii) Character Stream

  - reading / writing characters
  - can be internationalized leveraging Unicode
  - more efficient than Byte streams

- Primitive (lowest level) I/O is Byte-oriented

- Separate sets of class hierarchies for each stream

Keyboard input using BufferedReader

- Reading characters from the keyboard is more convenient than byte streams

- System.in is byte stream → need to be wrapped unde Reader: use InputStreamReader for converting bytes to characters

- Use constructor as below:

  BufferedReader(Reader *inputReader*), where *inputReader* is the stream linked to the instance of **BufferedReader** being created, e.g.

  ```
  BufferedReader br = new BufferedReader(new
  InputStreamReader(system.in) );
  ```
  → *br will be a character-based stream linked to console thru System.in*

Keyboard input using BufferedReader (contd.)

Characters can be read from System.in using read() method defined by BufferedReader. Three versions:

int read() throws IOException:

- reads a single character
- returns -1 on end of stream

int read(char[] buffer) throws IOException

- reads characters and puts them into buffer until either the array is full or EOF or an error
- Returns number of characters read or -1 on end of stream

int read(char[] buffer, int offset, int numChars) throws IOException

- Similar as 2nd version
- starts at buffer location specified as *offset*
- store characters upto *numChars*

Keyboard input using BufferedReader (contd.)

```
import java.io.*;
public class ReadChars {
     public static void main ( String[] args )
throws IOException {
          char c;
          BufferedReader br = new BufferedReader(
                    new InputStreamReader
(System.in) );
          System.out.println("Enter characters,
period to quit");

          // Read characters
          do {
               c = (char) br.read();
               System.out.println( c );
          } while ( c != '.' );
     }
```

Keyboard input using BufferedReader (contd.)

```
import java.io.*;
class ReadLine {
     public static void main ( String[] args ) throws
IOException {
            String str;
            BufferedReader br = new BufferedReader(
                        new InputStreamReader (System.in) );
            System.out.println("Enter characters, period to
quit");

            // Read Lines
            do {
                  str = br.readLine();
                  System.out.println( str );

            } while ( ! str.equals(".") );
     }
}
```

Keyboard input using Scanner

- **Scanner class methods packaged in java.util**

- **Used to read input from console / files for converting formatted string into binary form.**

  Steps:

  1. Create a Scanner linked to input

  ```
  Scanner inp = new Scanner( System.in ); // Console OR
  FileReader fin = new FileReader("Input.txt"); // File
  Scanner inp = new Scanner( fin );
  ```

  2. Use *Scanner object* to read input:

  i) Check whether desired type of input (say X) is available thru *inp*.hasNext*X* method

  ii) If desired input is available, read it by calling *inp*.next*X* method, e.g.

  ```
  int i; if (inp.hasNextInt()) i = inp.nextInt();
  ```

Keyboard input using Scanner (contd.)

3. Repeat step 2 until *tokens* (delimited by whitespace characters or matching regular expressions) are over

4. Finally close the scanner, e.g.
inp.close();

- If **next*X*** (nextInt or nextDouble or nextLine for example) cannot find the desired input type, it will throw an exception: InputMismatchException

- Use of **hasNext*X*** method recommended along with **next*X*** method.. **Why?**

- **Additional reference:**
  Delimiter may be changed by calling **useDelimeter()** method.. passing something as simple as a comma or any regular expression, e.g. "*Techno.*?*"

Difference between BufferedReader & Scanner

- **Input process:** Scanner treated inputs are automatically parsed as tokens, whereas BufferedReader yields to stream lines or Strings

- **Use of regular expression:** Scanner can also produce tokens using regular expression. Achieving same using BufferedReader will need extra codes

- **Performance:** BufferedReader is faster than Scanner as no parsing is involved

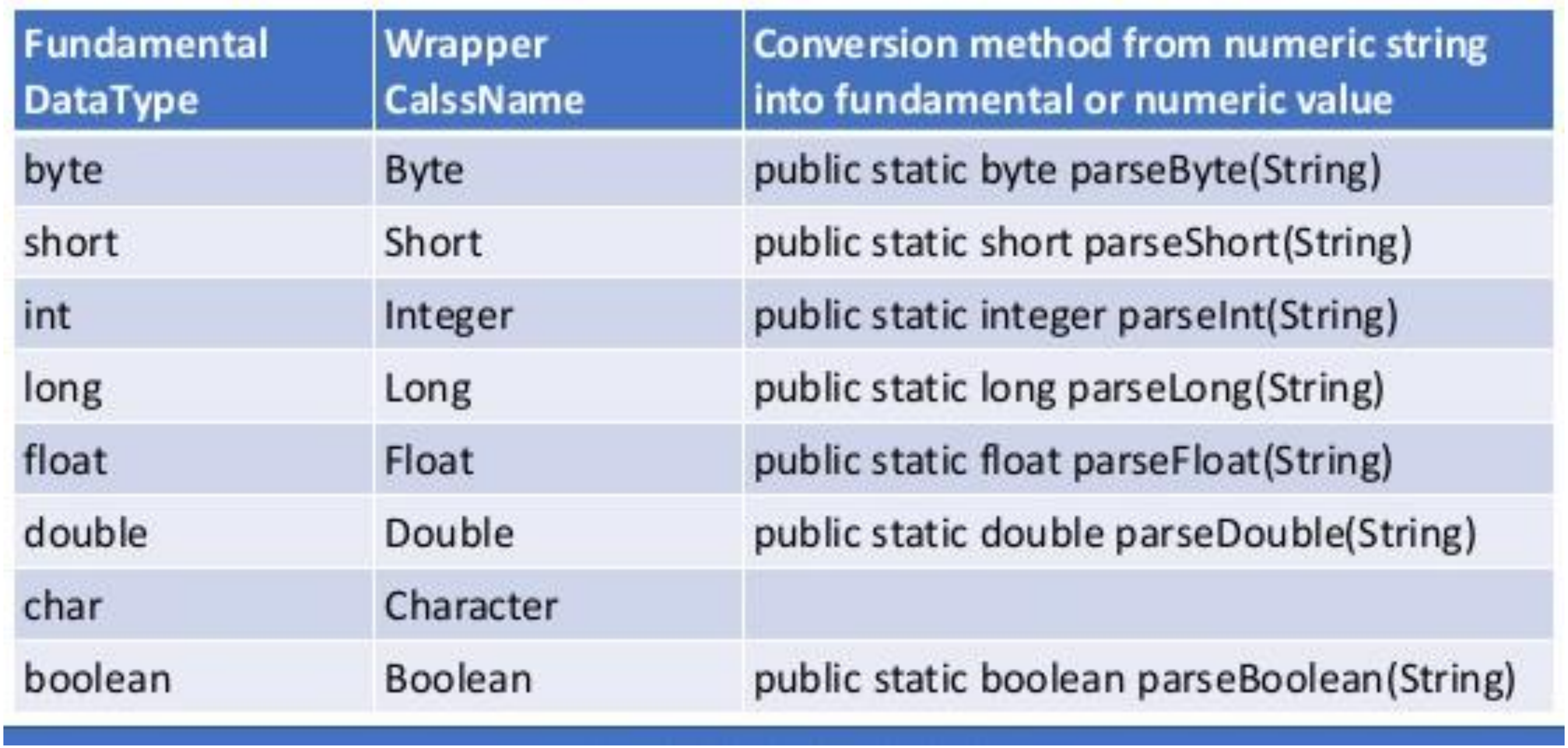
- **Thread safety:** BufferedReader is synchronized (thread-safe), whereas Scanner is not synchronized (thread-unsafe)

- **Usage:** BufferedReader is typically used for efficiently reading lines / strings from a file, whereas Scanner can be used for formatted inputs or XML parser.

Wrapper Class

- Fundamental or Primitive types are not objects → Wrapper classes encapsulate or wrap primitive types → Can now be passed by reference
- Wrapper classes or Type wrappers under **java.lang**
- Various storage mechanisms can work on these objects, e.g. map, lists, sets
- Various useful methods can work on these objects, e.g. compareTo(), equal(), parseInt(), isInfinite(), toBinaryString() etc.

Wrapper Class



Fundamental DataType	Wrapper CalssName	Conversion method from numeric string into fundamental or numeric value
byte	Byte	public static byte parseByte(String)
short	Short	public static short parseShort(String)
int	Integer	public static integer parseInt(String)
long	Long	public static long parseLong(String)
float	Float	public static float parseFloat(String)
double	Double	public static double parseDouble(String)
char	Character	
boolean	Boolean	public static boolean parseBoolean(String)

Wrapper Class

```
Integer intObj1 = new Integer (25);
Integer intObj2 = new Integer ("25");
Integer intObj3 = new Integer (35);
//compareTo demo
System.out.println("Comparing Obj1 and Obj2: " +
intObj1.compareTo(intObj2));
System.out.println("Comparing Obj1 and Obj3: " +
intObj1.compareTo(intObj3));
//Equals demo
System.out.println("Comparing Obj1 and Obj2: " +
intObj1.equals(intObj2));
System.out.println("Comparing Obj1 and Obj3: " +
intObj1.equals(intObj3));
```

Additional Reading

+++

Additionals: Non-static Nested class (Inner Class)

```
public class OuterClass {//OuterClass start
  private int outerMem = 10;
  public void createInnerClassInst(){
    InnerClass ic = new InnerClass();
    System.out.println("InnerMem accessed from OuterClass: " +
ic.innerMem);
    ic.getOuterMem();
  }

  class InnerClass {//InnerClass start
    private int innerMem = 99;
    public void getOuterMem() {
      System.out.println("OuterMem accessed from InnerClass: " +
outerMem);
    }
  } //InnerClass closed

  public static void main(String[] args) {
      OuterClass oc = new OuterClass();
      oc.createInnerClassInst();  // indirect approach

  //Creating InnerClass's instance by an instance of OuterClass oc
```

Additionals: Non-static Nested class (Inner Class)

```
        // Continuation of code in previous slide
        OuterClass.InnerClass icc = oc.new InnerClass();  // Direct
way for creating InnerClass object
        icc.getOuterMem();

        // One liner way to create an InnerClass object
        OuterClass.InnerClass iccc = new OuterClass().new
InnerClass();
        iccc.getOuterMem();
   }
} //OuterClass closed
```

Output:

```
OuterMem accessed from InnerClass: 10
InnerMem accessed from OuterClass: 99
OuterMem accessed from InnerClass: 10
OuterMem accessed from InnerClass: 10
```

Additionals: Static Nested classes

```
class OuterStatic {
    private int mem = 20;
    private static int smem = 50;

    static class InnerStatic {
        public void accessMembers () {
            // System.out.println(mem);
            // Error: non-static reference in static class ^^^
            System.out.println(smem);  // Allowed
        }
    }
}

public class StaticClassDemo
{
    public static void main(String[] args) {
        // Instantiate inner class only with the outer class name
        OuterStatic.InnerStatic is = OuterStatic.new
InnerStatic();            is.accessMembers();
    }
}
```

Output:
50

Additionals: Ready reference for Byte and Character stream class

Byte Stream Class	**Meaning**	**Character Stream Class**	**Meaning**
BufferedInputStream	Buffered input stream	BufferedReader	Buffered input character stream
BufferedOutputStream	Buffered output stream	BufferedWriter	Buffered output character stream
ByteArrayInputStream	Input stream that reads from a byte array	CharArrayReader	Input stream that reads from a character array
ByteArrayOutputStream	Input stream that writes to a byte array	CharArrayWriter	Output stream that writes to a character array
DataInputStream	An input stream that contains methods for reading the primitive data types	InputStreamReader	Input stream that translates bytes to characters
DataOutputStream	An input stream that contains methods for writing the primitive data types	OutputStreamWriter	Output stream that translates bytes to characters
FileInputStream	Input stream that reads from a file	FileReader	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file	FileWriter	Output stream that writes to a file
FilterInputStream	Filtered InputStream	FilterReader	Filtered reader
FilterOutputStream	Filtered OutputStream	FilterWriter	Filtered writer
InputStream	**Abstract class that describes stream input**	**Reader**	**Abstract class that describes character stream input**
ObjectInputStream	Input stream for objects	StringReader	Input stream that reads from a string
ObjectOutputStream	Output stream for objects	StringWriter	Output stream that writes to a string
OutputStream	**Abstract class that describes stream output**	**Writer**	**Abstract class that describes character stream output**
PipedInputStream	Input pipe	PipedReader	Input pipe
PipedOutputStream	Output pipe	PipedWriter	Output pipe

Additionals: Garbage Collection

Garbage (automatic vs. Manual clean)

Unreferenced objects. How it is created?

i) By setting the reference to null, e.g.

```
Student s = new Student();
s = null;
```

i) By assigning a reference to another, e.g.

```
Student s1 = new Student();
Student s2 = new Student();
s1 = s2;
// What happens to object referred by s1?
```

i) By anonymous object etc., e.g.

```
new Student();
```

Garbage Collection (gc): Process of reclaiming the runtime unused memory (→ memory efficiency) automatically (→ no extra efforts) from Heap Memory → More efficient than explicit use of free() in C and delete() in C++.

finalize() invoked each time before garbage is collected

Additionals: Garbage Collection (contd)

finalize() method

- Invoked each time before the object is garbage collected ← no more references to the object exist
- Can be used (overridden) to perform cleanup processing for non-Java resources, e.g. closing files, sockets etc.
- Syntax: `protected void finalize() { .. }`

gc() method

- Runs the garbage collector
- Recycle unused objects in order to free up the memory for quick reuse
- Syntax: `public static void gc() { … }`

Additionals: Garbage collection demo

Demo:

```
import java.util.*;
public class TestGarbage1{
        static int countGC = 0;
        public void finalize() { System.out.print("GC" + ++countGC + " " ) ; }

        public static void main(String args[]){
                int i, loopTimes = 150000;

                for (i=0; i < loopTimes; i++) {
                        TestGarbage1 t1=new TestGarbage1();
                        t1=null;  /* t1 object: unreferenced or garbage */
                }

                System.out.println( "*** gc called ***" ); System.gc();
        }
}
```