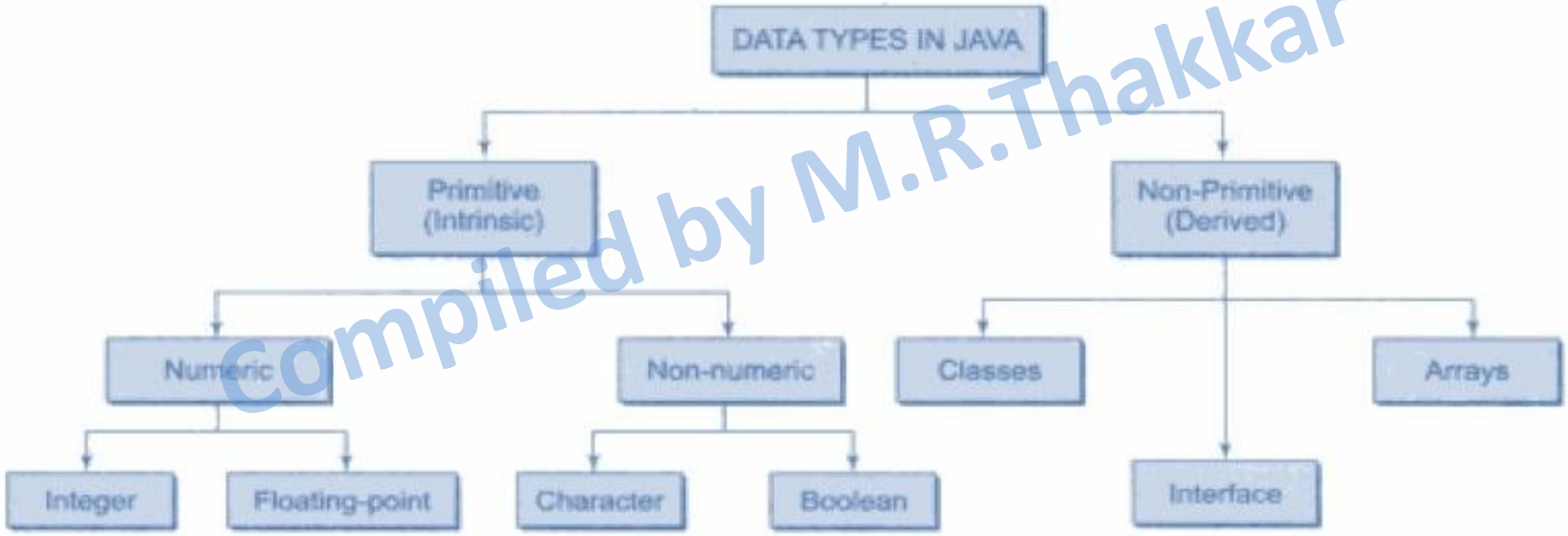
UNIT – II

BUILDING BLOCKS OF THE LANGUAGE

2.1 Primitive Data Types

- There are two data types available in Java:

1. Primitive Data Types
2. User Defined Data Types



2.1 Primitive Data Types

- Primitive data types are **predefined types of data**, which are supported by the programming language.

- There are **eight** primitive data types supported by Java:
    1. **byte**
    2. **short**
    3. **int**
    4. **long**
    5. **float**
    6. **double**
    7. **boolean**
    8. **char**

2.1 Primitive Data Types

- 2.1.1 byte

- Byte data type is an **8-bit signed integer**.
- Minimum value is **-128**
- Maximum value is **127**
- Default value is **0**
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is **four times smaller** than an int.

- Example:
- **byte a = 100 , byte b = -50 ;**

2.1 Primitive Data Types

- 2.1.2 short

- Short data type is a **16-bit signed integer**.
- Minimum value is **-32,768**
- Maximum value is **32,767**
- Short data type can also be used to save memory as byte data type. A short is **2 times smaller than an int**
- Default value is **0**.
- Example:
- **short s = 10000, short r = -20000 ;**

2.1 Primitive Data Types

- 2.1.3 int

- Int data type is a **32-bit signed integer**.
- Minimum value is **- 2,147,483,648**.
- Maximum value is **2,147,483,647**
- Int is generally used as the **default data type for integral values** unless there is a concern about memory.
- The default value is **0**.
- Example:

  int a = 100000, int b = -200000;

2.1 Primitive Data Types

- 2.1.4 long

- Long data type is a **64-bit signed integer**.
- Minimum value is **-9,223,372,036,854,775,808**.
- Maximum value is **9,223,372,036,854,775,807** .
- This type is used when a **wider range than int is needed**.
- Default value is **0L**.
- Example:

  long a = 100000L, int b = -200000L ;

2.1 Primitive Data Types

- 2.1.5 float

- Float data type is a **32-bit floating point**.
- Default value is **0.0f**.
- Float data type **is not used for precise values such as currency**.
- Example:

```
float  f1 = 234.5f ;
```

2.1 Primitive Data Types

- 2.1.6 double

- double data type is a **64-bit** floating point.
- This data type is generally used as the **default data type for decimal values**, generally the default choice.
- Default value is **0.0d**.
- Example:

  double d1 = 123.4 ;

2.1 Primitive Data Types

- 2.1.7 boolean

- boolean data type represents **one bit of information**.
- There are only **two possible values**: true and false.
- This data type is used for simple true/false conditions.
- Default value is **false**.
- Example:

  boolean one = **true ;**

2.1 Primitive Data Types

- 2.1.8 char

- char data type is a single **16-bit Unicode character**.
- Minimum value is '\u0000' (or 0).
- Maximum value is '\uffff' (or 65,535 inclusive).
- Char data type is used to store any character.
- Example:

  char letter = 'A';

2.2 User Defined Data Type

- User defined (Non-primitive) data types are not defined by the programming language, but are instead **created by the programmer**.

- There are **three types** of user defined data types in java:

1. class
2. interface
3. array

2.2 User Defined Data Type

- 2.2.1 class

```
class classname
{
    type instance-variable1;
    type instance-variable2;
    // ...
    type instance-variableN;

    type methodname1(parameter-list)
    {
       // body of method
    }
    type methodname2(parameter-list)
    {
       // body of method
    }
       // ...
    type methodnameN(parameter-list)
    {
       // body of method
    }
}
```

2.2 User Defined Data Type

- 2.2.1 class Example

```
class   Test
{
    int   i;

    void  display()
    {
       System.out.print(“ i = “ + i);
    }
}
```


- **Object:**

```
Test  obj1 =  new Test();
```

2.2 User Defined Data Type

- 2.2.2 interface

```
Interface interfacename
{

        type final-varname1 = value;
        type final-varname2 = value;
        // ...
        type final-varnameN = value;

    return-type methodname1( parameter-list );
    return-type methodname2( parameter-list );
                    // ...
    return-type methodnameN( parameter-list );


}
```

2.2 User Defined Data Type

- 2.2.2 interface Example

```
interface Test
{
    void display(void);
}
```

2.2 User Defined Data Type

- 2.2.3 Array

- type var-name[ ] = new type[size];

- **Example:**

```
int a[] = new int[5]
```

2.3 Identifiers & Literals

- Identifiers

- All Java components require **names**. Name used for **classes, methods, interfaces and variables** are called **Identifier**.

- An identifier is a sequence of characters, comprising **uppercase and lowercase letters** (**a-z, A-Z**), **digits** (**0-9**), **underscore** "**_**", and **dollar sign** "**$**".

- **Identifier must follow some rules:**

1. All identifiers must **start with** either a letter(**a to z or A to Z**) or an **underscore**.
2. After the first character, an identifier can have any combination of characters.
3. A Java **keyword** cannot be used as an identifier.
4. Identifiers in Java are **case sensitive**, **foo** and **Foo** are two different identifiers.

2.3 Identifiers & Literals

- Literals

- A literal is a **specific constant value or data** that is used in program source code, such as 123, 3.14, ‘a’, “Hello”, true .

- **Example:**

```
int i = 555;
float  f = 10.2;
char  c = ‘A’ ;
boolean  b = true ;
String s  = “Hello” ;
```

2.3 Identifiers & Literals

- Literals

- Types of Literal :

1. Integer Literal

```
byte b = 100;
short s = 1000;
int i = 555;
long  l = 35L ;
```

2. Floating-Point Literal

```
float  f = 10.2f;
double  d = 23.57d ;
```

2.3 Identifiers & Literals

- Literals

3. Character Literal

char c = '**A**' ;

4. Boolean Literal

boolean b = **true** ;

5. String Literal

String s = "**Hello**" ;

2.4 Declaration of Constants & Variables

- Variable
- Computer program process data. A variable is used to **store a piece of data** for processing. It is called variable, because you can change the stored value.

- A variable has a name , e.g. radius, area, age. The name is used to uniquely identify each variable.

2.4 Declaration of Constants & Variables

- Declaration of Variable
- In Java, all variables **must be declared before they can be used**.
- The basic form of a variable declaration is shown here:

  data-type identifier [= value][, identifier [= value] ...] ;

- **Examples:**

  int a;

  int a=10;

  int a, b, c;

  int a=10, b=20, c=30;

  int a, b=20, c, d=40;

2.4 Declaration of Constants & Variables

- Constant
- In Java, Constants are declared with keyword **final**. Their values **cannot be changed during execution**.
- **Example:**

```
final double PI = 3.1415926;  // Need to initialize
```


- It is a common coding convention to choose **all uppercase identifiers** for constant variables.
- **Example:** MIN_VALUE, MAX_SIZE

2.5 Type Conversion & Casting

- In Java, you will get a *compilation error,* if you try to assign a floating-point value to an int variable.

- For Example :

```
int i = 15.8;
```

- The compiler issues an error "possible loss in precision".
- This is because the fractional part would be lost. Java is strict type language.

- To assign the double value to an int variable, you need to invoke the ***type-casting operator*** - in the form of prefix (int).

```
int i = (int) 15.8;
```

2.5 Type Conversion & Casting

- There are two kinds of type-conversion in Java:

    1. ***Implicit (automatic) type-conversion***
    2. ***Explicit type-conversion***

2.5 Type Conversion & Casting

1. Implicit (automatic) type-conversion

- Implicit type-conversion is performed by compiler automatically. There is no loss of precision.

- **For example:**

  double f = 3;

- An implicit *type-conversion will take place if the following two conditions are met:*
  1. The two types are compatible.
  2. The destination type is larger than the source type.

2.5 Type Conversion & Casting

1. Explicit type-conversion

- Explicit type-casting is performed using type-casting operator in the prefix form of operand.

- **Syntax:**

  new_value = **(typecast)** value;

- **For example:**

  int i = **(int)** 15.8;

2.6 Scope of Variables & Default Value of Variables Declared

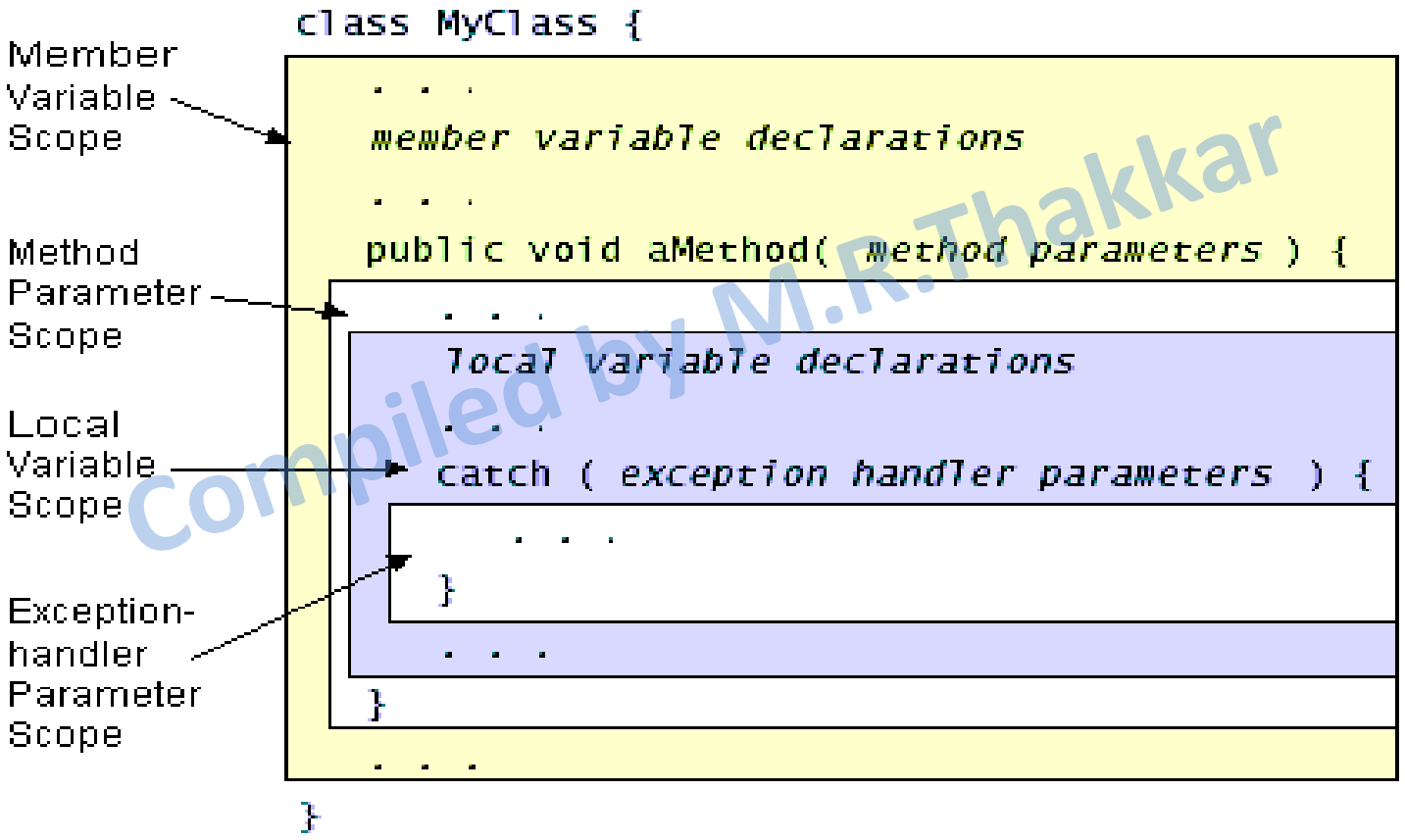
- Scope of Variables

- A variable's scope is the region of a program within which the variable can be accessed.
- Scope determines when the system creates and destroys memory for the variable.

- The location of the variable declaration within your program establishes its scope in one of these categories:
    1. member variable
    2. local variable
    3. method parameter
    4. exception-handler parameter

2.6 Scope of Variables & Default Value of Variables Declared

- Scope of Variables



2.6 Scope of Variables & Default Value of Variables Declared

- **Default Value of Variables Declared**

Data Type	Default Value
boolean	false
char	\u0000
int,short,byte / long	0 / 0L
float /double	0.0f / 0.0d
any reference type	null

2.7 Wrapper Classes

- As we are not able to use primitive data types as objects in Java, we can use ***Wrapper Class*** to wrap the primitive data types into objects.

- In Java, there is a wrapper class for every primitive data type.

Primitive type	Wrapper class	Constructors
byte	Java.lang.Byte	Byte(byte), Byte(String)
short	Java.lang.Short	Short(short), Short(String)
int	Java.lang.Integer	Integer(int), Integer(String)
long	Java.lang.Long	Long(long), Long(String)
float	Java.lang.Float	Float(float), Float(String)
double	Java.lang.Double	Double(double), Double(String)
boolean	Java.lang.Boolean	Boolean(boolean), Boolean(String)
char	Java.lang.Character	Character(char)

2.7 Wrapper Classes

- Below is wrapper class hierarchy in Java :

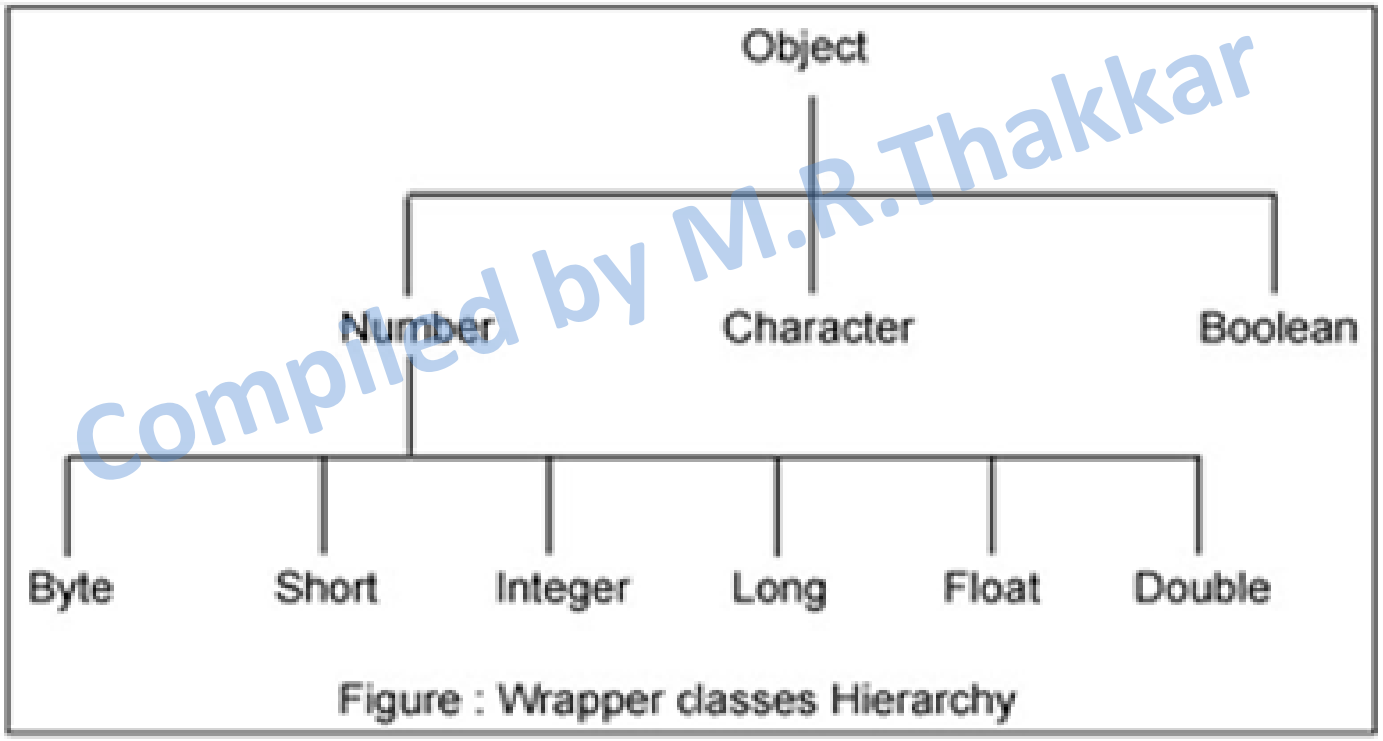


Figure : Wrapper classes Hierarchy

2.7 Wrapper Classes

- There are two ways to create wrapper objects:

    1. Using wrapper class constructors
    2. Using valueOf() methods

2.7 Wrapper Classes

1. Using wrapper class constructors

- Each wrapper classes provide two constructors except **Character** type. One for its **primitive type** and other one for **String** representation.

- **Example :**

```
int i = 10;
Integer x1 = new Integer(i);
```

2.7 Wrapper Classes

2. Using valueOf() methods

- This is the second method to create wrapper classes. This is a static method. So it can be invoked directly on the class.

- **Syntax :**

```
static Integer valueOf(int i)
static Integer valueOf(String s)
```

- **Example :**

```
int i = 10;
Integer x1  = Integer.valueOf(i);
Integer x2 = Integer.valueOf("10");
```

2.7 Wrapper Classes

- Retrieving the value wrapped by a wrapper class object
- Each of the eight wrapper classes have a method to retrieve the value that was wrapped in the object.
- These methods have the form : *Value() , where * refers to the corresponding data type (for Example intValue()).
- **Example :**

```
Integer  x1  =  Integer.valueOf(i);

Int  i = x1.intValue();
```

2.8 Comment Syntax

- Comments are used to document and explain your codes and program logic.

- Comments are ignored by the compiler, but they are very important for documentation and later readability and understanding of program.

Type of Comment	Comment Example
End-of-line Comment	int x; // a comment
Multi-line Comment	/* The variable x is an integer: */ int x;
Documentation comment	/** x -- an integer representing the x coordinate */ int x;

2.9 Garbage Collection

- In java, garbage means unreferenced objects.
- Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.
- In java garbage collection is performed automatically. The garbage collector automatically runs periodically.

2.9 Garbage Collection

- **How can an object be unreferenced?**

  1. **By nulling a reference**

     ```
     Employee e=new Employee();
     e=null;
     ```

  2. **By assigning a reference to another**

     ```
     Employee e1=new Employee();
     Employee e2=new Employee();
     e1=e2;
     ```

  3. **By anonymous object**

     ```
     new Employee();
     ```

2.9 Garbage Collection

- **gc() method**

- You can run the garbage collector on demand by calling the **gc( )** method.

- Java.lang.System.gc() method runs garbage collector. This method recycles unused objects to free memory.

2.9 Garbage Collection

- **finalize() method**

- if an object is holding some non-Java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed.

- The finalize() method is invoked each time before the object is garbage collected.

- This method is defined in Object class as:

```
protected void finalize()
{    --------  }
```

- Inside the finalize( ) method you will specify those actions that must be performed before an object is destroyed.

2.9 Garbage Collection

- **Advantages**

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.

- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

2.9 Garbage Collection

- **Example**

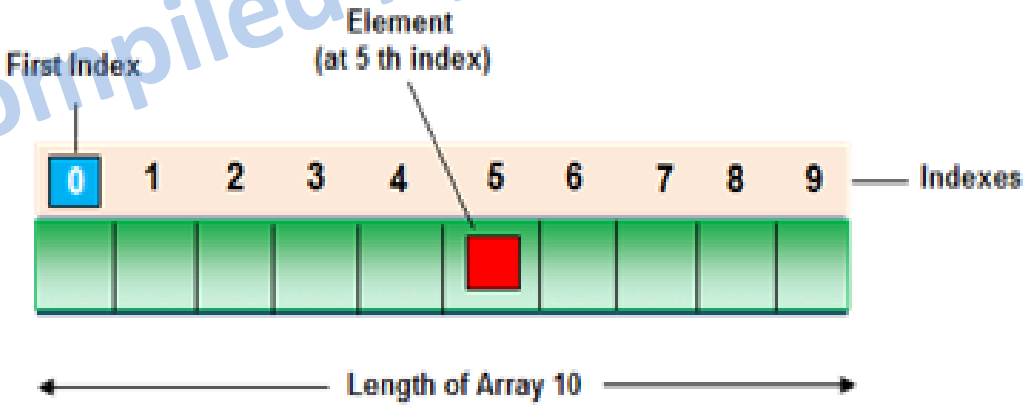
```
public class Example
{
    public static void main(String args[])
    {
                Example e1=new  Example();
                e1=null;
                System.gc();
    }

    protected void finalize()
    {
                System.out.println("Object cleaned");
    }

}
```

2.10 Array of Primitive Data Types

- An array is a group of similar typed variables that are referred to by a common name.
- A specific element in an array is accessed by its index.
- An array may have one or more dimensions.



2.11 Types of Arrays

- There are two types of array in java.

1. Single Dimensional Array : A *single-dimensional array is a group of same-typed variables*.

2. Multidimensional Array: A *multi-dimensional arrays are actually arrays of arrays*.

2.11 Types of Arrays

1. Single Dimensional Array

- **Array Declaration:**
  - int a[];
- **Array creation:**
  - a = new int[5];
- **Combined Statement:**
  - int a[] = new int[5];

2.11 Types of Arrays

1. Single Dimensional Array (Alternative Declaration)

- **Array Declaration:**
  - int[] a;
- **Array creation:**
  - a=new int[5];
- **Combined Statement:**
  - int[] a = new int[5];

2.11 Types of Arrays

2. Multi Dimensional Array (2-Dimensional)

- Rectangular Arrays
- Non Rectangular Arrays

2.11 Types of Arrays

2. Multi Dimensional Array (2-Dimensional)

➢ Rectangular Arrays

- **Array Declaration:**
  - int a[][];
- **Array creation:**
  - a = new int[4][5];
- **Combined Statement:**
  - int a[][] = new int[4][5];

2.11 Types of Arrays

➢ Rectangular Arrays

- **Example**

```
public class Example
{
    public static void main(String args[])
    {
        int a[ ][ ]=new int[2][3];

        a[0][0]= 10;
        a[0][1]= 20;
        a[0][2]= 30;

        a[1][0]= 40;
        a[1][1]= 50;
        a[1][2]= 60;
```

2.11 Types of Arrays

- **Rectangular Arrays**

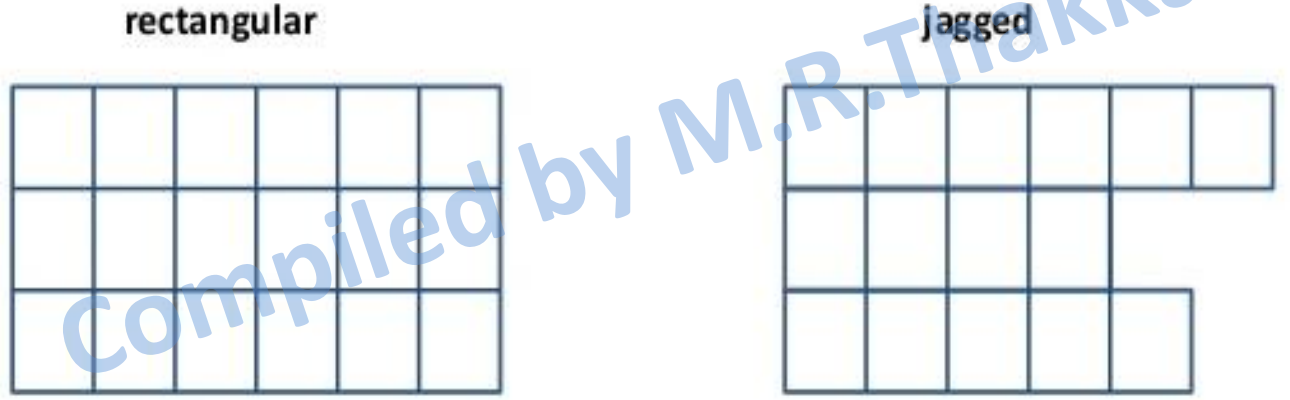
- **Example**

```
        for(int i=0; i<=1; i++)
        {
                for(int j=0; j<=2; j++)
                {
                        System.out.println (a[i][j]);
                }
        }
    }
}
```

2.11 Types of Arrays

➢ Non Rectangular Arrays

- In non-rectangular arrays number of rows elements are fixed, but number of columns for each row are different.



- In Java it is compulsory to allocate memory for the first dimension in multi dimensional arrays. We can allocate memory to remaining dimension afterwards.

2.11 Types of Arrays

➢ Non Rectangular Arrays

- **Example**

```
public class Example
{
        public static void main(String args[])
        {
                int a[ ][ ]=new int[3][ ];

                a[0]=new int[1];
                a[1]=new int[2];
                a[2]=new int[3];

    a[0] [0]= 1;

    a[1] [0]= 1;
    a[1] [1]= 2;

   A[2] [0]= 1;
   A[2] [1]= 2;
   A[2] [2]= 3;
```

2.11 Types of Arrays

- **Non Rectangular Arrays**

- **Example**

```
for(int i=0;i<=2;i++)
{
        for(int j=0;j<=i; j++)
        {
                System.out.println (arr[i][j]);
        }
 }
}
}
```

2.11 Types of Arrays

- **Length of Array**

- In Java, all arrays have one instance variable length, which holds the number of elements or size of array.

2.11 Types of Arrays

- **Length of Array**

- **Example**

```
public class Example
{
        public static void main(String args[])
        {
                int a[ ]={1,2,3,4,5};
                int b[ ]={1,2,3,4,5,6,7};
                int c[ ]={1,2,3};

                System.out.println (“no of elements of A : “ + a.length);
                System.out.println (“no of elements of B : “ + b.length);
                System.out.println (“no of elements of C : “ + c.length);
        }
}
```

2.12 String Operations

- **String**

- String is a sequence of characters enclosed within double quotes.

  E.g. “JAVA PROGRAMMING”, “123”, “Me2” etc…

- But, unlike many other languages that implement strings as character arrays, Java implements strings as objects of String class.

- Once a String object has been created, you cannot change the characters that comprise that string. Each time you need an altered version of an existing string, a new String object is created that contains the modifications. The original string is left unchanged.

- This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones.

2.12 String Operations

- **2.12.1 String Creation**

- There are two ways to create String object:

  1. Using String Literal
  2. Using String Constructor

2.12 String Operations

- ## 2.12.1 String Creation

1. Using String Literal

- String literal is created by double quote.

- **Example:**

```
String s  =  "Hello";
```

- Each time you create a string literal, the JVM checks the string constant pool in memory first. If the string already exists in the memory pool, a reference to the pooled instance returns. If the string does not exist in the pool, a new String object is created and placed in the pool.

- **Example:**

```
String s1="Welcome";     // new object will be created
String s2="Welcome";     //no new object will be created
```

2.12 String Operations

- ## 2.12.1 String Creation

2. Using String Constructor

- The String class supports several constructors.
- **Example:**

  - String s = new String("Welcome");
  - String s = new String(); //String with no character
  - char Name[ ]={ ‘W’, ‘e’, ‘l’, ‘c’, ‘o’, ‘m’, ‘e’ };
    String s = new String(Name);

2.12 String Operations

- **2.12.2 Concatenation and Conversion of String**

- Strings Concatenating form a new string i.e. the combination of multiple strings.

- There are two ways to concat string objects:

  1. Using concat() method
  2. Using + (string concatenation) operator

2.12 String Operations

- **2.12.2 Concatenation and Conversion of String**

1. Using concat() method

- **Description:** concat() method concatenates the specified string to the end of current string.
- **Syntax:** public String concat(String obj)
- **Example:**

```
String s1="Indian ";
String s2="Cricketer";

String s3 = s1.concat(s2);

System.out.println(s3);        //IndianCricketer
```

2.12 String Operations

- **2.12.2 Concatenation and Conversion of String**

2. Using + (string concatenation) operator

- **Description:** The **+** (string concatenation) operator concatenates two strings, producing a String object as the result.

- **Example:**

```
String s = "Indian"+" Cricketer";
System.out.println(s);  //IndianCricketer
```

2.12 String Operations

- **2.12.2 Concatenation and Conversion of String**

2. Using + (string concatenation) operator

- The + (string concatenation) operator can concatenate not only string , but primitive values also.

- **Example:**

```
int age = 9;
String s = "He is “  +  age  +  " years old.";
System.out.println(s);   // He is 9 years old.
```

```
String s = "four: " + 2 + 2;
System.out.println(s);   // four: 22
```

2.12 String Operations

- ## 2.12.3 Changing case of String

- **Description:** The method **toLowerCase( )** method converts all the characters in a string from uppercase to lowercase. The **toUpperCase( )** method converts all the characters in a string from lowercase to uppercase.

- Nonalphabetical characters, such as digits, are unaffected.

- **Syntax:**

  String toLowerCase( )

  String toUpperCase( )

2.12 String Operations

- **2.12.3 Changing case of String**

- **Example:**

```
String s = "Indian";

System.out.println(s.toUpperCase());   //INDIAN
System.out.println(s.toLowerCase());   //indian
```

2.12 String Operations

- **2.12.4 Character Extraction**

- The String class provides a several methods in which characters can be extracted from a String object.

1. **Substring()**
2. **charAt()**
3. **getChars()**

2.12 String Operations

- **2.12.4 Character Extraction**

➢ **substring() method**

- **Description:** You can extract a substring from given string using **substring( ).** It has two forms.

- **Syntax:** String substring(int *startIndex*)
  String substring(int *startIndex*, *int endIndex*)

➢ substring(int *startIndex*) : This form returns a copy of the substring that begins at *startIndex* and runs to the *end* of the string.

2.12 String Operations

- **2.12.4 Character Extraction**

➢ substring(int *startIndex*) : This form returns a copy of the substring that begins at *startIndex* and runs to the *end* of the string.

- substring(int *startIndex*, *int endIndex*) : Here, *startIndex specifies the beginning index, and endIndex specifies the stopping* point. The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

2.12 String Operations

- **2.12.4 Character Extraction**

➢ **substring() method**

- **Example:**

```
String s="IndianCricketer";

System.out.println(s.substring(6));    //Cricketer
System.out.println(s.substring(0,6));   //Indian
```

2.12 String Operations

- **2.12.4 Character Extraction**

➢ **charAt() method**

- **Description:** The charAt() method returns single character at a specific index in string.
- **Syntax:** char charAt(int index)
- **Example:**

```
String s="Indian";
System.out.println(s.charAt(0));    //I
System.out.println(s.charAt(3));   //i
```

2.12 String Operations

- ## 2.12.4 Character Extraction

➢ getChars() method

- **Description:** If you need to extract more than one character at a time, you can use the **getChars( )** method.

- **Syntax:** void getChars (int start, int end, char target[ ], int tstart);

- **start:** specifies the index of the beginning of the substring.
- **end:** specifies the index of the ending of the substring.
- **target[ ]:** specifies the array, where you want to receive the characters, and
- **tstart:** specifies the index within array target[] to copy the substring.

2.12 String Operations

- **2.12.4 Character Extraction**

➢ **getChars() method**

- **Example:**

```
String s = "IndianCricketer";

char target[]=new char[9];
s.getChars(6,15,target,0);

for(int i=0; i< target.length; i++)
        System.out.print(target[i]);  //Cricketer
```

2.12 String Operations

- ## 2.12.5 String length

- **Description:** The length of a string is the number of characters that it contains. It can be obtained using **length**() method.
- **Syntax:** int length( )
- **Example:**

```
String s="Indian";
System.out.println(s.length());     //6
```

2.12 String Operations

- **2.12.6 String Comparison**

- The String class includes several methods that compare two strings.

1. **equals()**
2. **equalsIgnoreCase()**
3. **compareTo()**
4. **compareToIgnoreCase()**

2.12 String Operations

- **2.12.6 String Comparison**

➢ **equals() method**

- **Description:** The equals() method , returns true if the strings contain the same characters in the same order, and false otherwise.
- The comparison is case-sensitive.
- **Syntax:** boolean equals(String str)
- **Example:**

```
String s1="Indian";
String s2="Indian";
String s3="Rahul";

System.out.println(s1.equals(s2));     // true
System.out.println(s1.equals(s3));     // flase
```

2.12 String Operations

- **2.12.6 String Comparison**

➢ **equalsIgnoreCase() method**

- **Description:** The **equalsIgnoreCase( )** performs a comparison of strings that ignores case differences**.**
- **Syntax:** boolean equalsIgnoreCase(String *obj*)
- **Example:**

```
String s1="Indian";
String s2="Indian";
String s3="Rahul";
String s4="INDIAN";

System.out.println(s1.equalsIgnoreCase (s2));     // true
System.out.println(s1.equalsIgnoreCase (s3));    // flase
System.out.println(s1.equalsIgnoreCase (s4));     // true
```

2.12 String Operations

- **2.12.6 String Comparison**

➢ **compareTo() method**

- **Description:** Often, it is not enough to simply know whether two strings are identical.
- For sorting applications, you need to know which is *less than, equal to, or greater than the next.*
- *A* string is less than another if it comes before the other in dictionary order.
- A string is greater than another if it comes after the other in dictionary order.
- The String method **compareTo( )** serves this purpose**.**

- **Syntax:** **int** compareTo(String *str*)

- The result of the comparison is returned as integer value and is interpreted as shown here:

2.12 String Operations

- **2.12.6 String Comparison**

➢ **compareTo() method**

Value	Meaning
• Less than zero	The invoking string is less than *str.*
• Greater than zero	The invoking string is greater than *str.*
• Zero	The two strings are equal.

- **Example:**

```
String s1="Indian";
String s2="Indian";
String s3="Rahul";

System.out.println(s1.compareTo(s2));  //0
System.out.println(s1.compareTo(s3));   // -9 (because s1<s3)
System.out.println(s3.compareTo(s1));   //  9 (because s3 > s1 )
```

2.12 String Operations

- **2.12.6 String Comparison**

➢ **compareToIgnoreCase() method**

- **Description:** If you want to ignore case differences when comparing two strings, use **compareToIgnoreCase( ).**

- **Syntax:** int compareToIgnoreCase(String *str*)

- **Example:**

```
String s1="Indian";
String s2="INDIAN";

System.out.println(s1.compareToIgnoreCase(s2)); // 0
```

2.12 String Operations

- **2.12.7 String Buffer**

- The StringBuffer class is used to created mutable (modifiable) string.
- The StringBuffer class is same as String except it is mutable i.e. it can be changed.
- Commonly used Constructors of StringBuffer class:
  - **StringBuffer():** creates an empty string buffer with the initial capacity of 16.
  - **StringBuffer(String str):** creates a string buffer with the specified string.
  - **StringBuffer(int capacity):** creates an empty string buffer with the specified capacity as length.

2.12 String Operations

- **2.12.7 String Buffer**

- **Example:**

```
StringBuffer s = new StringBuffer("India");
System.out.println(s.length());    //  5
```

2.12 String Operations

- **2.12.7 String Buffer**
- **String Vs StringBuffer:**

	String	StringBuffer
1	The data which enclosed within double quote (" ") is by default treated as String class.	The data which enclosed within double quote (" ") is not by default treated as StringBuffer class
2	String class object is immutable	StringBuffer class object is mutable
3	When we create an object of String class by default no additional character memory space is created.	When we create an object of StringBuffer class by default we get 16 additional character memory space.

2.13 Operators

- Java provides a rich operator environment. Most of its operators can be divided into the following four groups:

1. Arithmetic
2. Bitwise
3. Relational
4. Logical

2.13 Operators

- **Arithmetic Operator**

Operator	Description	Usage	Examples
*	Multiplication	expr1 * expr2	2 * 3 → 6
/	Division	expr1 / expr2	4 / 2 → 2
%	Remainder (Modulus)	expr1 % expr2	5 % 2 → 1 42.5 % 10 → 2.5
+	Addition	expr1 + expr2	1 + 2 → 3
-	Subtraction	expr1 - expr2	1 - 2 → -1

2.13 Operators

- **Arithmetic Operator (Example)**

```
public class Example
{
    public static void main(String args[])
    {
        int i = 22;
        int j = 8;

        System.out.println("   i + j = " + (i + j));        // 30
        System.out.println("   i - j = " + (i - j));        // 14
        System.out.println("   i * j = " + (i * j));        // 176
        System.out.println("   i / j = " + (i / j));        // 2
        System.out.println("   i % j = " + (i % j));        // 6
    }
}
```

2.13 Operators

- **Bitwise Operator**

Operator	Name	Example	Description
&	and	3 & 5 → 1	1 if both bits are 1.
|	or	3 | 5 → 7	1 if either bit is 1.
^	xor	3 ^ 5 → 6	1 if both bits are different.
~	not	~3 → -4	Inverts the bits.
<<	left shift	3 << 2 → 12	Shifts the bits of *n* to left *p* positions. Zero bits are shifted into the low-order positions.
>>	right shift	5 >> 2 → 1	Shifts the bits of *n* to right *p* positions. If *n* is a 2's complement signed number, the sign bit is shifted into the high-order positions.
>>>	Right shift	-4 >>> 28 → 15	Shifts the bits of *n* right *p* positions. Zeros are shifted into the high-order positions.

2.13 Operators

- **Bitwise Operator**

A	B	~A	A & B	A ^ B	A | B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	0	1

2.13 Operators

- **Bitwise Operator (Example)**

```
public class Example
{
    public static void main(String args[])
    {
            int a = 3;  // 0 1 1
            int b = 6; //  1 1 0

            System.out.println(" a & b = " + (a & b));  // a & b = 2
            System.out.println(" a | b = " + (a | b));   // a | b = 7
            System.out.println(" a ^ b = " + (a ^ b));   // a ^ b = 5
            System.out.println(" ~a = " + (~a));         //  ~a = -4
    }
}
```

2.13 Operators

- **Bitwise Operator (Example)**

```
public class Example
{
    public static void main(String args[])
    {
        int a = 4;  // 0 0 0 0 0 1 0 0
        int b = 1; //  0 0 0 0 0 0 0 1
        int c = -4; // 1 1 1 1 1 1 0 0

        System.out.println("a << b : "+ (a << b) );       // a << b : 8
        System.out.println("a >> b : "+ (a >> b) );       // a >> b : 2
        System.out.println("a >>> b : "+ (a >>> b) );   // a >>> b : 2

        System.out.println("c >> b : "+ (c >> b) );   // c >> b : -2
        System.out.println("c >>> b : "+ (c >>> b) );   // c >>> b : 2147483646

    }
}
```

2.13 Operators

- **Relational Operator**

- The relational operators determine the relationship that one operand has to the other.
- In Java, relational operator returns a boolean value of either true or false.
- Java provides six relational operators:

Operator	Description	Usage	Example
==	Equal to	expr1 == expr2	(x == y) → false
!=	Not Equal to	expr1 != expr2	(x != y) → true
>	Greater than	expr1 > expr2	(x > y) → false
>=	Greater than or equal to	expr1 >= expr2	(x >= 5) → true
<	Less than	expr1 < expr2	(y < 8) → false
<=	Less than or equal to	expr1 >= expr2	(y <= 8) → true

2.13 Operators

- **Relational Operator**

```
public class Example
{
        public static void main(String args[])
        {
                int a=2  ,b=5;

                System.out.println("a < b : “ + (a<b));          //  a < b : true
                System.out.println("a > b : “ + (a>b));          //  a > b : false
                System.out.println("a <= b : “ + (a<=b));        // a <= b : true
                System.out.println("a >= b : “  + (a>=b));       // a >= b : false
                System.out.println("a != b : “ + (a!=b));        // a != b : true
                System.out.println("a == b : “  + (a==b));       //  a == b : false
        }
}
```

2.13 Operators

- **Logical Operator**
- The Logical operators operates only on **boolean operands** to form a **resultant boolean value.**
- Java provides four logical operators:

Operator	Description	Usage
!	Logical NOT	! booleanExpr
||	Logical OR	booleanExpr1 || booleanExpr2
&&	Logical AND	booleanExpr1 && booleanExpr2
^	Logical XOR	booleanExpr1 ^ booleanExpr2

2.13 Operators

- **Logical Operator**
- **Truth tables :**

A	B	!A	A || B	A &&B	A ^ B
False	False	True	False	False	False
False	True	True	True	False	True
True	False	False	True	False	True
True	True	False	True	True	False

2.13 Operators

- **Logical Operator**

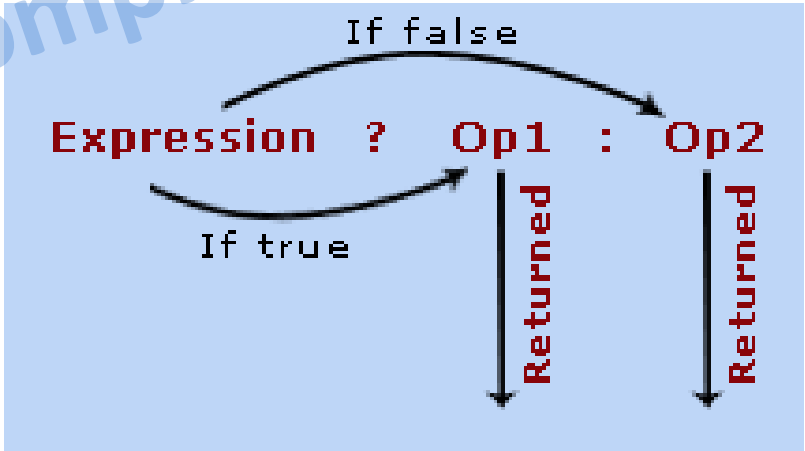
```
public class Example
{
    public static void main(String args[])
    {
        boolean a=true  , b=false;

        System.out.println(" ! a : " + (!a));              //  ! a : false
        System.out.println("a || b : " + (a||b));        //  a || b : true
        System.out.println("a && b : " + (a&&b));   // a && b : false
        System.out.println("a ^ b : "  + (a^b));         // a ^ b : true
    }
}
```

2.13 Operators

- **Ternary Operator (?)**
- The Ternary Operator (?) has following general form:
- booleanExpr ? True-part : false-part ;

- Here, expression1 can be any expression that evaluates to a boolean value. If expression1 is true, then True-part is evaluated; otherwise, False-part is evaluated.



2.13 Operators

- **Ternary Operator (?)**

```
public class Example
{
        public static void main(String args[])
        {
                int x=10, y=5, z=0;
                 String  z = x>y ? "x is maximum" : "Y is maximum";
                System.out.println (z);
        }
}
```

2.13 Operators

- **Increment (++) and Decrement (--) operator**
- The increment operator **(++)** increases its operand by one.
- The decrement operator **(--)** decreases its operand by one.

- For example:

```
x = x + 1;
```

can be rewritten like this by use of the increment operator:

```
x++;
```



```
x = x - 1;
```

can be rewritten like this by use of the increment operator:

```
X--;
```

2.13 Operators

- **Increment (++) and Decrement (--) operator**
- These operators are unique in that they can appear both in *postfix form, where* they follow the operand (x++ / x--) as just shown, and *prefix form, where they precede the* operand (++x / --x).

- In the prefix form, the operand is incremented or decremented before the value is obtained for use in the expression.

- In postfix form, the value is obtained for use in the expression, and then the operand is modified.

2.13 Operators

- **Increment (++) and Decrement (--) operator**
- For example:

```
int x = 42;
Int y = ++x;   // y=43
```

```
int x = 42;
Int y = x++;   // y=42
```



```
int x = 42;
Int y = --x;   // y=41
```

```
int x = 42;
Int y = x--;   // y=42
```

2.14 Decision & Control Statements

- **Selection (Decision) Statements**
- Java supports two selection statements: **if and switch.**
- These statements allow you to control the flow of your program's execution based upon conditions known only during run time.

2.14 Decision & Control Statements

- **If**

```
if (Expression )
{
  true-block ;
}
```

- **if-then-else**

```
if (Expression )
{
  true-block ;
}
else
{
  false-block ;
}
```

2.14 Decision & Control Statements

- **nested-if**

```
if (Expression )
{
        if (Expression )
        {
                true-block ;
        }
        else
        {
                false-block ;
        }
}
else
{
  false-block ;
}
```

2.14 Decision & Control Statements

- **if-else-if Ladder**

```
if(condition)
        statement;
else if(condition)
    statement;
else if(condition)
    statement;
...
else
statement;
```

2.14 Decision & Control Statements

- **switch**

```
switch (expression)
{
        case value1:
                // statement sequence
                break;
        case value2:
                // statement sequence
                break;
        ...
        case valueN:
                // statement sequence
                break;
        default:
                // default statement sequence
}
```

2.14 Decision & Control Statements

- **Loops**

- while
- do-while
- for

2.14 Decision & Control Statements

- **Loops (while)**

```
while (conditional expression)
{
    statements block
}
```

2.14 Decision & Control Statements

- **Loops (do-while)**

```
do
{
    Statement bolck;

} while (condition);
```

2.14 Decision & Control Statements

- **Loops (for)**

```
for ( initialization; termination; increment)
{
      statements;
}
```

2.14 Decision & Control Statements

- **Jump statements**

- Continue
- break
- return
- exit

2.14 Decision & Control Statements

- **Jump statement (continue)**

- The continue statement causes the loop to exit its current trip through the loop and start over at the first statement of the loop.

- **Example:**

```
for(int  i=1 ; i<= 5 ; i++ )
{
      if( i == 3 )
      {
        continue;
      }
   System.out.println(i);
}
```

2.14 Decision & Control Statements

- **Jump statement (break)**

- Break statement ends a loop immediately even if the condition being tested is still true.

- **Example:**

```
for(int  i=1 ; i<= 5 ; i++ )
{
      if( i == 3 )
      {
        break;
      }
   System.out.println(i);
}
```

2.14 Decision & Control Statements

- **Jump statement (return)**

- A return keyword is used to finish the execution of a method.
- You use return to exit from the current method and jump back to the statement within the calling method.

- **Example:**

```
return ;
```

2.14 Decision & Control Statements

- **Jump statement (exit)**
- **System.exit()** method terminates the currently running Java program.
- **Example:**

```
for(int  i=1 ; i<= 5 ; i++ )
{
      if( i == 3 )
      {
        System.exit(0);
      }
    System.out.println(i);
}
```

****