CSE310: Programming in Java

Topic: Array and Enum

Outlines

- Introduction
- Array Creation
- Array Initialization
- Enumerations

Array

- Definition:

  An array is a finite collection of variables of the same type that are referred to by a common name.
- Arrays of any type can be created and may have one or more dimensions.
- A specific element in an array is accessed by its index (subscript).
- Array elements are stored in contiguous memory locations.

- Examples:
  - **C**ollection of numbers
  - **C**ollection of names

More points

- In Java all arrays are dynamically allocated.
- Since arrays are objects in Java, we can find their length using the object property *length*.
- The direct superclass of an array type is *Object.*
- If we try to access array outside of its index then *ArrayIndexOutOfBounds* Exception will be raised

One-Dimensional Arrays

- A one-dimensional array is a list of variables of same type.
- The general form of a one-dimensional array declaration is:

type [] arr_ref_var; ***OR***

type [] arr_ref_var= new type[size];

Example:

int [] num = new int [10];

It will create an array of 10 integers.

Syntax and Example

Declaration of array variable:

data-type variable-name[];

eg. *int marks[];*

This will declare an array named ‘marks’ of type ‘int’. But no memory is allocated to the array.

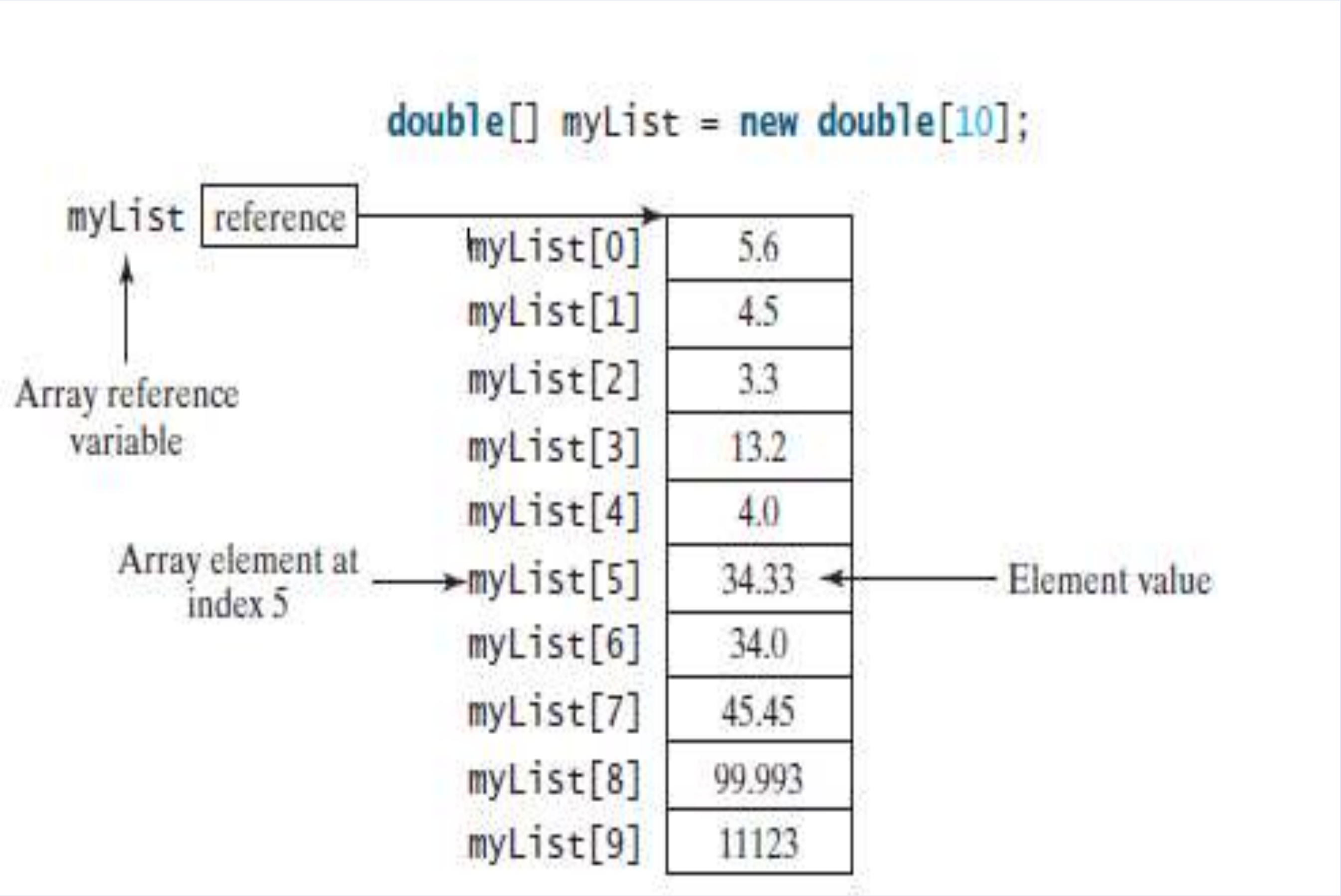
Allocation of memory:

variable-name = new data-type[size];

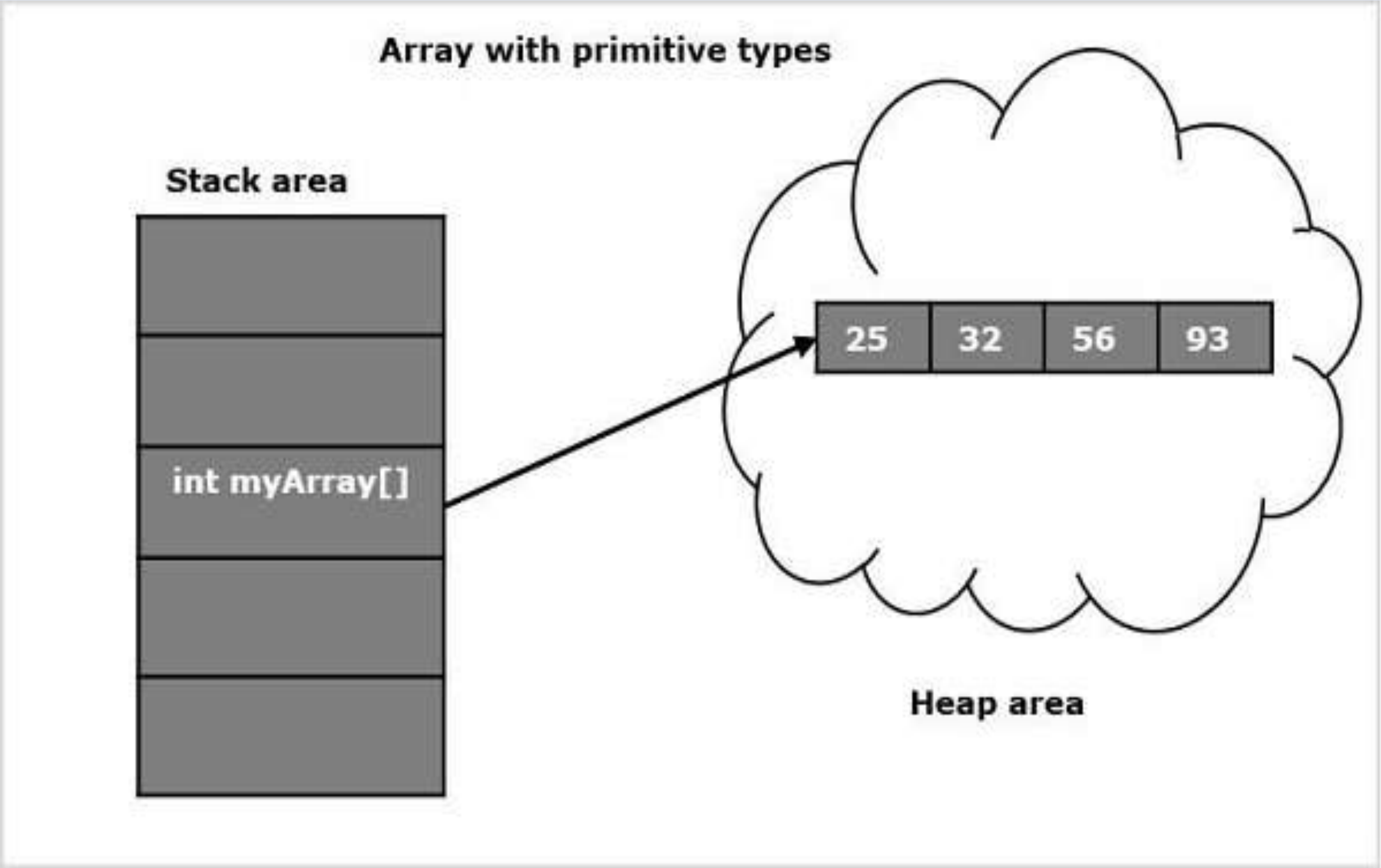
eg. *marks = new int[5];*

This will allocate memory of 5 integers to the array ‘marks’ and it can store upto 5 integers in it. ‘new’ is a special operator that allocates memory.

Another Example of array



Memory allocation



Accessing elements in the array:

• Specific element in the array is accessed by specifying name of the array followed the index of the element.

• All array indexes in Java start at zero.

variable-name[index] = value;

Example:

marks[0] = 10;

This will assign the value 10 to the 1st element in the array.

marks[2] = 863;

This will assign the value 863 to the 3rd element in the array.

Example

STEP 1 : (Declaration)

int marks[];

marks → null

STEP 2: (Memory Allocation)

marks = new int[5];

marks →

0	0	0	0	0
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]

STEP 3: (Accessing Elements)

marks[0] = 10;

marks →

10	0	0	0	0
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]

- Size of an array can’t be changed after the array is created.

- Default values:
  - zero (0) for numeric data types,
  - \u0000 for chars and
  - false for Boolean types

- Length of an array can be obtained as:

  array_ref_var.length

Examples…

```
// to show the working of single dimension array
class Example
{
  public static void main(String args[])
   {
        int a[] = new int[5];
        a[0]=12;a[1]=34;a[2]=56;a[3]=78;a[4]=90;
        System.out.println("Length of the array is "+a.length);
        System.out.println("Printing the elements of array");
        for(int i=0;i<a.length;i++)
        System.out.println(a[i]);
   }
}
```

```java
// to show the working of single dimension array
import java.util.Scanner;
class Example
{
    public static void main(String args[])
      {
        int n;// number of elements in array
        Scanner ob = new Scanner(System.in);
        System.out.println("Enter the number of elements in array");
        n=ob.nextInt();
        int a[] = new int[n];
        System.out.println("Enter"+n+"elements of array");
        for(int i=0;i<a.length;i++)
        a[i]=ob.nextInt();
        System.out.println("Printing the elements of array");
        for(int i=0;i<a.length;i++)
        System.out.println(a[i]);
      }
}
```

Note

- Arrays can store elements of the ***same data type***. Hence an ***int*** array CAN NOT store an element which is not an int.

- Though an element of a compatible type can be converted to int and stored into the int array.

eg. marks[2] = (int) 22.5;

This will convert '22.5' into the int part '22' and store it into the 3rd place in the int array 'marks'.

- Array indexes start from zero. Hence 'marks[index]' refers to the (index+1)th element in the array and 'marks[size-1]' refers to last element in the array.

- For an array of the char[] type, it can be printed using one print statement. For example, the following code displays Dallas:

```
char[] city = {'D', 'a', 'l', 'l', 'a', 's'};
System.out.println(city);
```

- Accessing an array out of bounds is a common programming error that throws a runtime ArrayIndexOutOfBoundsException. To avoid it, make sure that you do not use an index beyond arrayRefVar.length – 1.

Array Initialization

1. data Type [] array_ref_var = {value0, value1, …, value n};
2. data Type [] array_ref_var = new data Type [n];

   array_ref_var [0] = value 0;

   array_ref_var [1] = value 1;

   …

   array_ref_var [n-1] = value n;

3. data type [] array_ref_var = new int[] {value1,value 2..}

Array initialization: Example

```
class Example
{
  public static void main(String args[])
   {
       int [] a = new int [] {1,2,3,4,5};
         for(int i : a)
         System.out.println(i);


   }
}
```

Printing array elements using for each loop

We can also print the Java array using for-each loop. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

```
for(data_type variable:array)
{
//body of the loop
}
```

Example 1

```
class Example
{
    public static void main(String args[])
     {
      int arr[]={33,3,4,5};
      //printing array using for-each loop
      for(int i:arr)
      System.out.println(i);
     }
}
```

Example 2

```
import java.util.Scanner;
class Example
{
    public static void main(String args[])
      {
        int i;
        String s[] = new String[5];
        Scanner ob = new Scanner(System.in);
        System.out.println("Enter the 5 strings");
        for(i=0;i<s.length;i++)
        s[i]=ob.nextLine();
        System.out.println("5 strings are");
        for(String x : s)
        System.out.println(x);
      }
}
```

Exercise

Write a program which prompts the user to enter the number of elements.Now read the marks of all the subjects from the user using Scanner class.Write a method which calculates the percentage of the user.

Multi-Dimensional Array

- Multidimensional arrays are arrays of arrays(2D,3D....)
- Two-Dimensional arrays are used to represent a table or a matrix.
- A two-dimensional array is actually an array in which each element is a one-dimensional array.
- Declaration:

  elementType[][] arrayRefVar; or elementType arrayRefVar[][];

  Example: int[][]a; or int a[][];

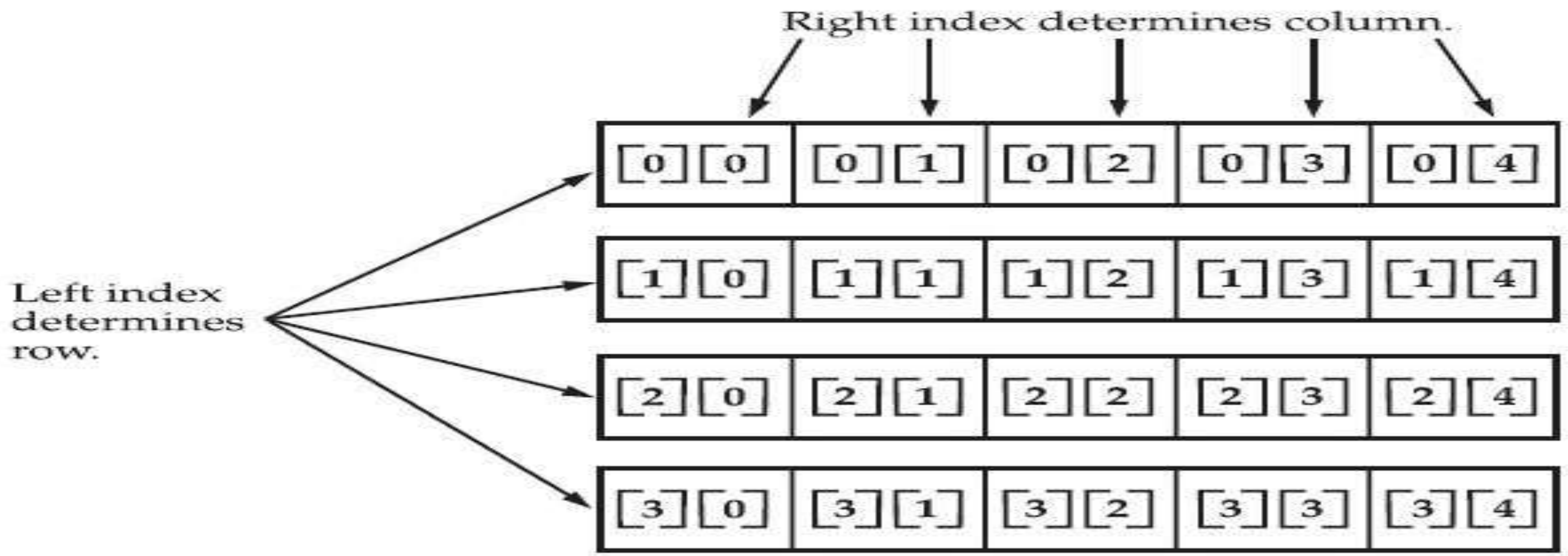
  Creating 2D array:

  elementType[][] arrayRefVar=new elementType[n][m];

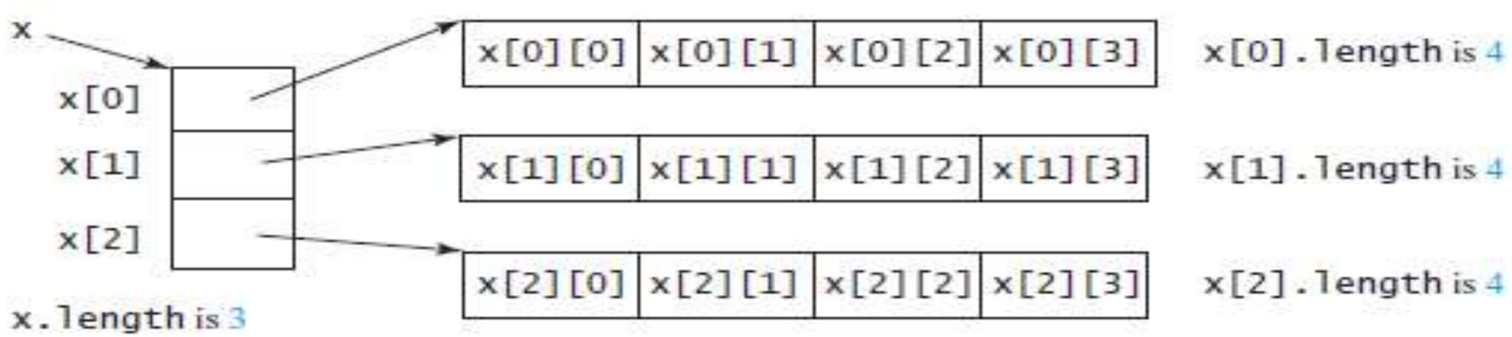
  Example:

  int twoD[][] = new int[4][5];

Conceptual View of 2-Dimensional Array



Given: int twoD [ ] [ ] = new int [4] [5] ;



A two-dimensional array is a one-dimensional array in which each element is another one-dimensional array.

```
class TwoDimArr
    {
      public static void main(String args[])
          {
            int twoD[][]= new int[4][5];
            int i, j, k = 0;
            for(i=0; i<4; i++)
                for(j=0; j<5; j++)
              {
                twoD[i][j] = k;
                k++;
              }
                for(i=0; i<4; i++)
              {
                for(j=0; j<5; j++)
                System.out.print(twoD[i][j] + " ");
                System.out.println();
              }
            }
    }
```

Output:

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

- When we allocate memory for a multidimensional array, we need to only specify the memory for the first (leftmost) dimension.

```
int twoD[][] = new int[4][];
```

- The other dimensions can be assigned manually.

Initializing Multi-Dimensional Array

```
class Matrix {
  public static void main(String args[]) {
        double m[][] = {
                        { 0*0, 1*0, 2*0, 3*0 },
                        { 0*1, 1*1, 2*1, 3*1 },
                        { 0*2, 1*2, 2*2, 3*2 },
                        { 0*3, 1*3, 2*3, 3*3 }
                  };
        int i, j;
        for(i=0; i<4; i++) {
              for(j=0; j<4; j++)
                    System.out.print(m[i][j] + " ");
              System.out.println();
                    }
              }
        }
```

Syntax—Giving other dimensions manually

Syntax: data_type array_name[][] = new data_type[n][]; //n: no. of rows

array_name[] = new data_type[n1] //n1= no. of colmuns in row-1

array_name[] = new data_type[n2] //n2= no. of colmuns in row-2

array_name[] = new data_type[n3] //n3= no. of colmuns in row-3

----

array_name[] = new data_type[nk] //nk=no. of colmuns in row-n

This type of array is also known as Jagged/ or ragged arrays

Program example-Jagged arrays

```
// Program to demonstrate 2-D jagged array in Java
class Main
{
    public static void main(String[] args)
    {
     int arr[][] = new int[2][];
     arr[0] = new int[3];
     arr[1] = new int[2];
     int count = 0;
     for (int i=0; i<arr.length; i++)
         for(int j=0; j<arr[i].length; j++)
             arr[i][j] = count++;
     System.out.println("Contents of 2D Jagged Array");
     for (int i=0; i<arr.length; i++)
     {
         for (int j=0; j<arr[i].length; j++)
             System.out.print(arr[i][j] + " ");
         System.out.println();
     }
    }
}
```

Output:

```
Contents of 2D Jagged Array
0 1 2
3 4
```

Array Cloning

- To actually create another array with its own values, Java provides the **clone()** method.

- arr2 = arr1; (assignment)

  is not equivalent to

  arr2 = arr1.clone(); (cloning)

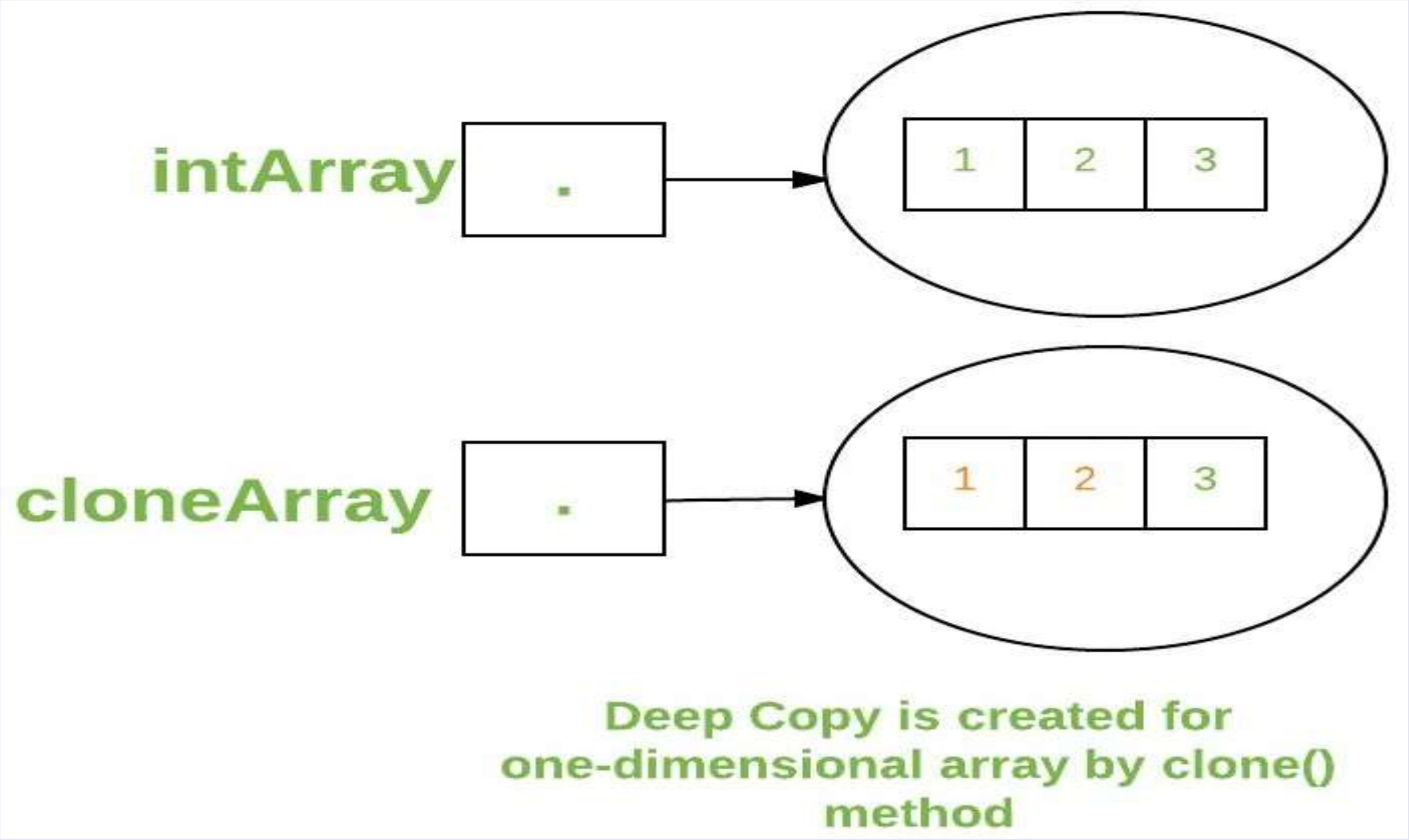
In first case, Only one array is created and two references arr1 and arr2 are pointing to the same array. While in second case two different arrays are created.

Cloning of 1D Array

```java
// Java program to demonstrate
// cloning of one-dimensional arrays
class Test
{
    public static void main(String args[])
    {
      int intArray[] = {1,2,3};
      int cloneArray[] = intArray.clone();
      // will print false as deep copy is created
      // for one-dimensional array
      System.out.println(intArray == cloneArray);
      for (int i = 0; i < cloneArray.length; i++) {
            System.out.print(cloneArray[i]+" ");
      }
    }
}
```

Output:
false
1 2 3

Deep copy is created for ID array



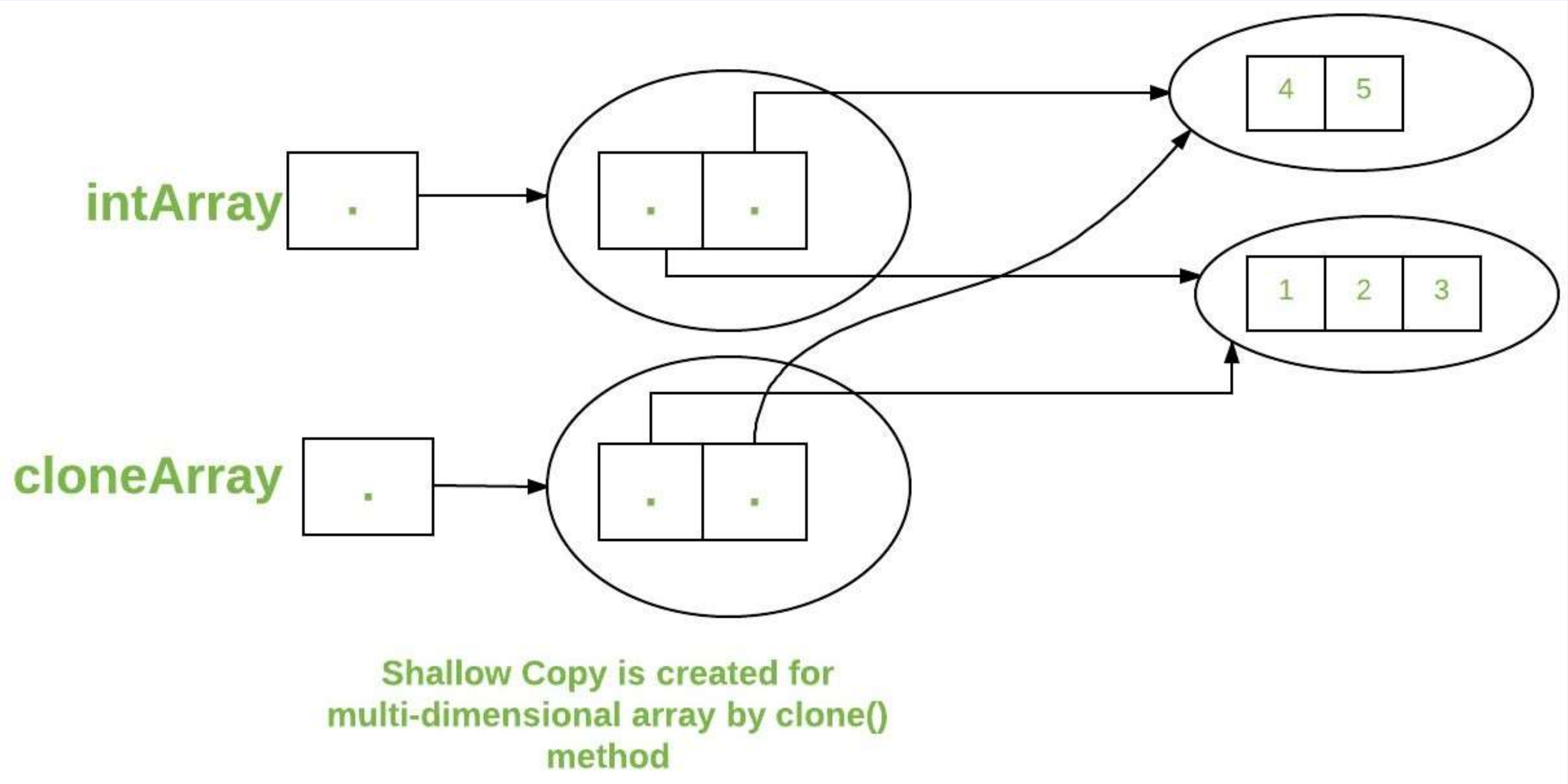
Cloning of 2D Array

A clone of a multi-dimensional array (like Object[][]) is a “shallow copy” however, which is to say that it creates only a single new array with each element array a reference to an original element array, but subarrays are shared.

```
// Java program to demonstrate
// cloning of multi-dimensional arrays
class Test
{
	public static void main(String args[])
	{
	  int intArray[][] = {{1,2,3},{4,5}};
	  int cloneArray[][] = intArray.clone();
	  // will print false
	  System.out.println(intArray == cloneArray);
	  // will print true as shallow copy is created
	  // i.e. sub-arrays are shared
	  System.out.println(intArray[0] == cloneArray[0]);
	  System.out.println(intArray[1] == cloneArray[1]);
	}
}
```

Output:
false
true
true

Shallow copy created for 2D array



Assignment for Practice

- WAP in which prompt the user to enter the number of subjects and number of CA in each subject. Read the marks of each CA and store in a two dimensional array.

Q1

Which of these is the correct syntax for array creation?

a) int arr[] = new arr[5]

b) int [5] arr = new int[]

c) int arr[5] = new int[]

d) int arr[] = new int [5]

Q2

Which of these is an incorrect Statement?
a) It is necessary to use new operator to initialize an array
b) Array can be initialized using comma separated expressions surrounded by curly braces
c) Array can be initialized when they are declared
d) None of the mentioned

Q3

What will be the output of the following Java code?

```
class array_output
{
    public static void main(String args[])
    {
        int array_variable [] = new int[10];
        for (int i = 0; i < 10; ++i)
        {
            array_variable[i] = i;
            System.out.print(array_variable[i] + " ");
            i++;
        }
    }
}
```

a) 0 2 4 6 8

b) 1 3 5 7 9

c) 0 1 2 3 4 5 6 7 8 9

d) 1 2 3 4 5 6 7 8 9 10

Q4

What will be the output of the following Java code?

```
class evaluate
{
    public static void main(String args[])
    {
      int arr[] = new int[] {0 , 1, 2, 3, 4, 5, 6, 7, 8, 9};
      int n = 6;
        n = arr[arr[n] / 2];
      System.out.println(arr[n] / 2);
    }
}
```

a) 3

b) 0

c) 6

d) 1

Q5

What will be the output of the following Java code?

```
class array_output
{
    public static void main(String args[])
    {
        int array_variable[][] = {{ 1, 2, 3}, { 4 , 5, 6}, { 7, 8, 9}};
        int sum = 0;
        for (int i = 0; i < 3; ++i)
            for (int j = 0; j <  3 ; ++j)
                sum = sum + array_variable[i][j];
        System.out.print(sum / 5);
    }
}
```

a) 8

b) 9

c) 10

d) 11

Q6(Output??)

```
public class Main
{
    public static void main(String[] args) {
     int a[][]=new int[2][2];
     System.out.println(a[0][1]);
    }
}
```

A. 0

B. 1

C. Compile time error

D. Run time error

Java Enum

Introduction

- Enum in java is a data type that contains fixed set of constants.
- It can be thought of as a class having fixed set of constants.
- The java enum constants are static and final implicitly. It is available from JDK 1.5.
- It can be used to declare days of the week, Months in a Year etc.

Advantages of Enum

- enum improves type safety
- enum can be easily used in switch
- enum can be traversed
- enum can have fields, constructors and methods

Important

- Enum can not be instantiated using new keyword because it contains private constructors only.

- The enum can be defined within or outside of the class because it is similar to a class.

- Every enum constant is always implicitly **public static final**. Since it is **static**, we can access it by using enum Name.

Examples…

// A simple enum example where enum is declared outside any class (Note enum keyword instead of class keyword)

```
enum Color
{
    RED, GREEN, BLUE;
}
public class Test
{
    public static void main(String[] args)
    {
     Color c1 = Color.RED;
     System.out.println(c1);
    }
}
```

values(),valueOf() and ordinal() method

- The java compiler internally adds the values(),valueOf() and ordinal() methods when it creates an enum.
- The values() method returns an array containing all the values of the enum.
- **valueOf() method** returns the enum constant of the specified string value, if exists
- By using **ordinal() method**, each enum constant index can be found, just like array index.

Examples….

```
class Example
    {
        public static void main(String args[])
        {
        enum session {WINTER,SUMMER,FALL}
        for(session s : session.values())
        System.out.println(s);
        }
    }
```

Examples….

```
enum session {WINTER,SUMMER,FALL}
class Example
    {
        public static void main(String args[])
        {
        for(session s : session.values())
        System.out.println(s);
        }
    }
```

// Working of values(), valueOf() and ordinal() method

```
class Example
    {
     public enum Season { WINTER, SPRING, SUMMER, FALL }
     public static void main(String[] args) {

     for (Season s : Season.values()){
           System.out.println(s);
           }
     System.out.println("Value of WINTER is: "+Season.valueOf("WINTER"));
     System.out.println("Index of WINTER is: "+Season.valueOf("WINTER").ordinal());
     System.out.println("Index of SUMMER is: "+Season.valueOf("SUMMER").ordinal());
    }
}
```

Enum with switch

```
// An enumeration of apple varieties.
enum Apple {
Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
public class Main {
public static void main(String args[])
{
Apple ap;
ap = Apple.RedDel;
// Use an enum to control a switch statement.
switch(ap) {
case Jonathan:
System.out.println("Jonathan is red.");
break;
case GoldenDel:
System.out.println("Golden Delicious is yellow.");
break;
case RedDel:
System.out.println("Red Delicious is red.");
break;
case Winesap:
System.out.println("Winesap is red.");
break;
}
}
}
```

Java Enumeration are class types(We can give them constructors, add instance variables and methods etc)

```
// Use an enum constructor, instance variable, and method.
enum Apple {
Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);
private int price; // price of each apple
// Constructor
Apple(int p) { price = p; }
int getPrice() { return price; }
}
public class Main {
public static void main(String args[])
{
Apple ap;
// Display price of Winesap.
System.out.println("Winesap costs " +Apple.Winesap.getPrice() +" cents.\n");
// Display all apples and prices.
System.out.println("All apple prices:");
for(Apple a : Apple.values())
System.out.println(a + " costs " + a.getPrice() +" cents.");
}
}
```

Key points….

- enum can contain constructor and it is executed separately for each enum constant at the time of enum class loading.
- We can’t create enum objects explicitly and hence we can’t invoke enum constructor directly.
- Every enum constant represents an **object** of type enum.

Q1(Output??)

```
enum Season
{
   WINTER,SUMMER,SPRING;
}
public class Main
{
      public static void main(String[] args) {
      Season var;
      var=SPRING;
      System.out.println(var);
      }
}
```

A. 0
B. Compile time error
C. -1
D. SPRING

Q2(Output??)

```
enum Flowers
{
   SUNFLOWER,JASMINE,LOTUS;
}
public class Main
{
     public static void main(String[] args) {
     Flowers var[]=Flowers.values();
     for(int i=1;i<2;i++)
     System.out.println(var[i]);
     }
}
```

A. JASMINE

B. LOTUS

C. SUNFLOWER

D. 1

Q3(Output??)

```
enum Colours
{
   WHITE,GREEN,RED,YELLOW
}
public class Main
{
     public static void main(String[] args) {
     System.out.println(Colours.valueOf("YELLOW").ordinal());
}
}
```

A. 0

B. 1

C. 2

D. 3

Q4(Output??)

```
enum Colours
{
  WHITE(23),GREEN(78),RED(7),YELLOW(100);
   int colour_code;
   Colours(int code){
      colour_code=code;
   }
   int get_code(){
      return colour_code;
   }
}
public class Main
{
       public static void main(String[] args) {
       System.out.println(Colours.RED.get_code());
}
}
```

A. 7

B. 0

C. 100

D. 23

CSE310: Programming in Java

Topic: Branching Statements

Outlines

- break Statement
- continue Statement
- return Statement

break Statement

- break statement:
  - terminates a statement sequence in a switch statement
  - used to exit a loop
- The break statement has two forms:
  - labeled
  - unlabeled.

Unlabeled break

- An unlabeled break is used to terminate a for, while, or do-while loop and switch statement.

Example 1:

```
class Example
{
public static void main(String[] args)
{
        for(int i=0; i<100; i++)
                {
                if(i == 10)
                break;
        System.out.println("i: " + i);
                }
        System.out.println("Loop completed");
}
}
```

Labeled break Statement

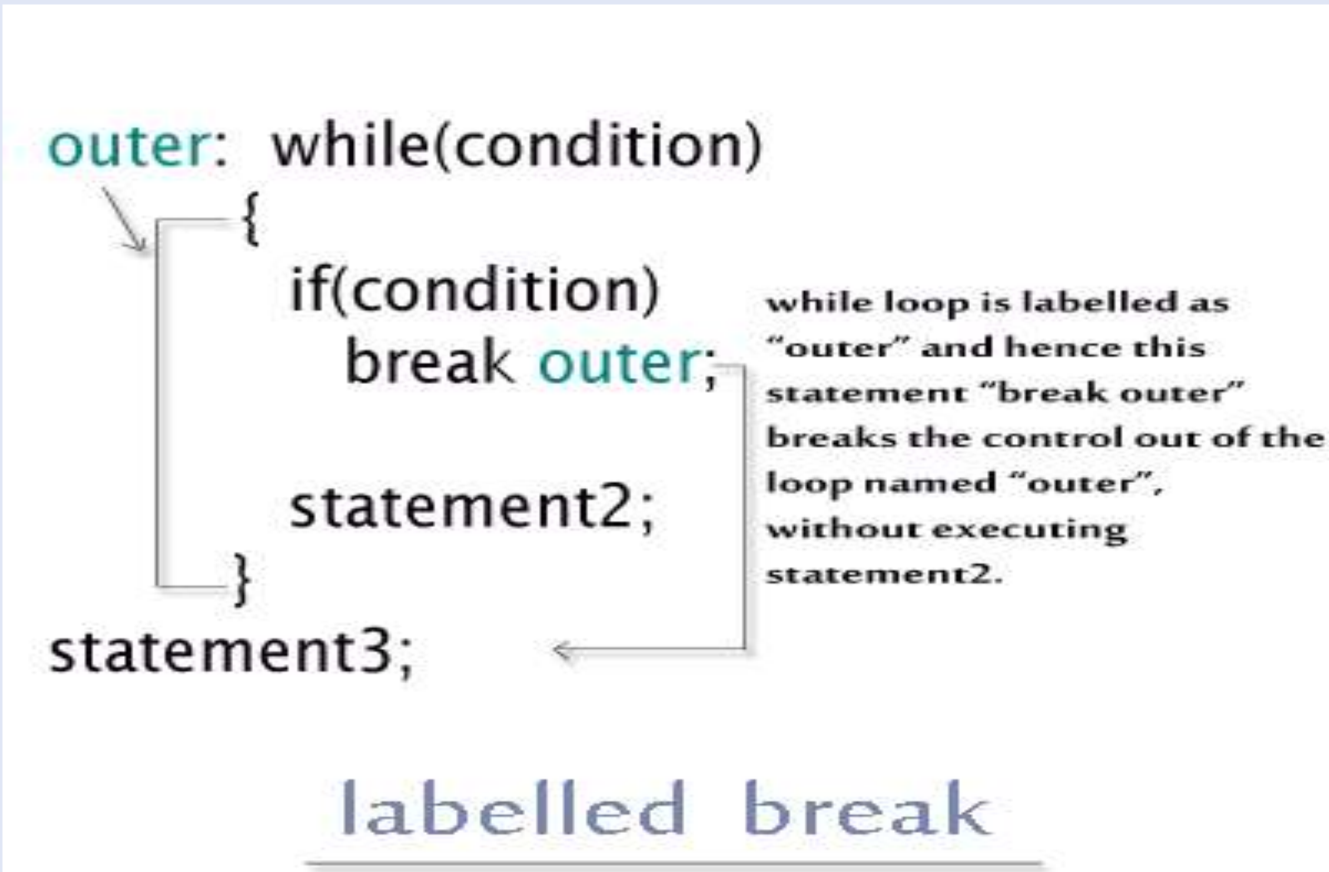
- Java defines an expanded form of the break statement.

  break *label;*

- By using this form of break, we can break out of one or more blocks of code.
- When this form of break executes, control is transferred out of the named block.
- When this **break statement** is encountered with the *label/name of the loop*, it *skips* the execution any statement after it and takes the control right out of this labelled loop.

  And, the control goes to the first statement right after the loop.

Concept



outer: while(condition)
{
if(condition)
break outer;
statement2;
}
statement3;
while loop is labelled as "outer" and hence this statement "break outer" breaks the control out of the loop named "outer", without executing statement2.
labelled break

```
class Example
{
public static void main(String[] args)
        {
        outer:
        for(int i=0; i<3; i++)
        {
        System.out.println("Outer "+ i);
        inner:
        for(int j=0; j<3; j++)
        {
        System.out.println("Inner "+j);
        if(i== j+1)
        break outer;
        System.out.println("Bye");
        }
        }
        }
}
```

Output:

```
Outer 0
Inner 0
Bye
Inner 1
Bye
Inner 2
Bye
Outer 1
Inner 0
```

NOTE

- The break statement terminates the labeled statement; it does not transfer the flow of control to the label.

- Control flow is transferred to the statement immediately following the labeled (terminated) statement.

continue Statement

- The continue statement skips the current iteration of a for, while , or do-while loop.

- The unlabeled form skips to the end of the innermost loop's body and evaluates the boolean expression that controls the loop.

```
class Example
{
public static void main(String[] args)
  {
              int i;
              for(i=1;i<=10;i++)
              {
               if(i==5)
               continue;
              System.out.println(i);
              }
   }
}
```

Output:

1

2

3

4

6

7

8

9

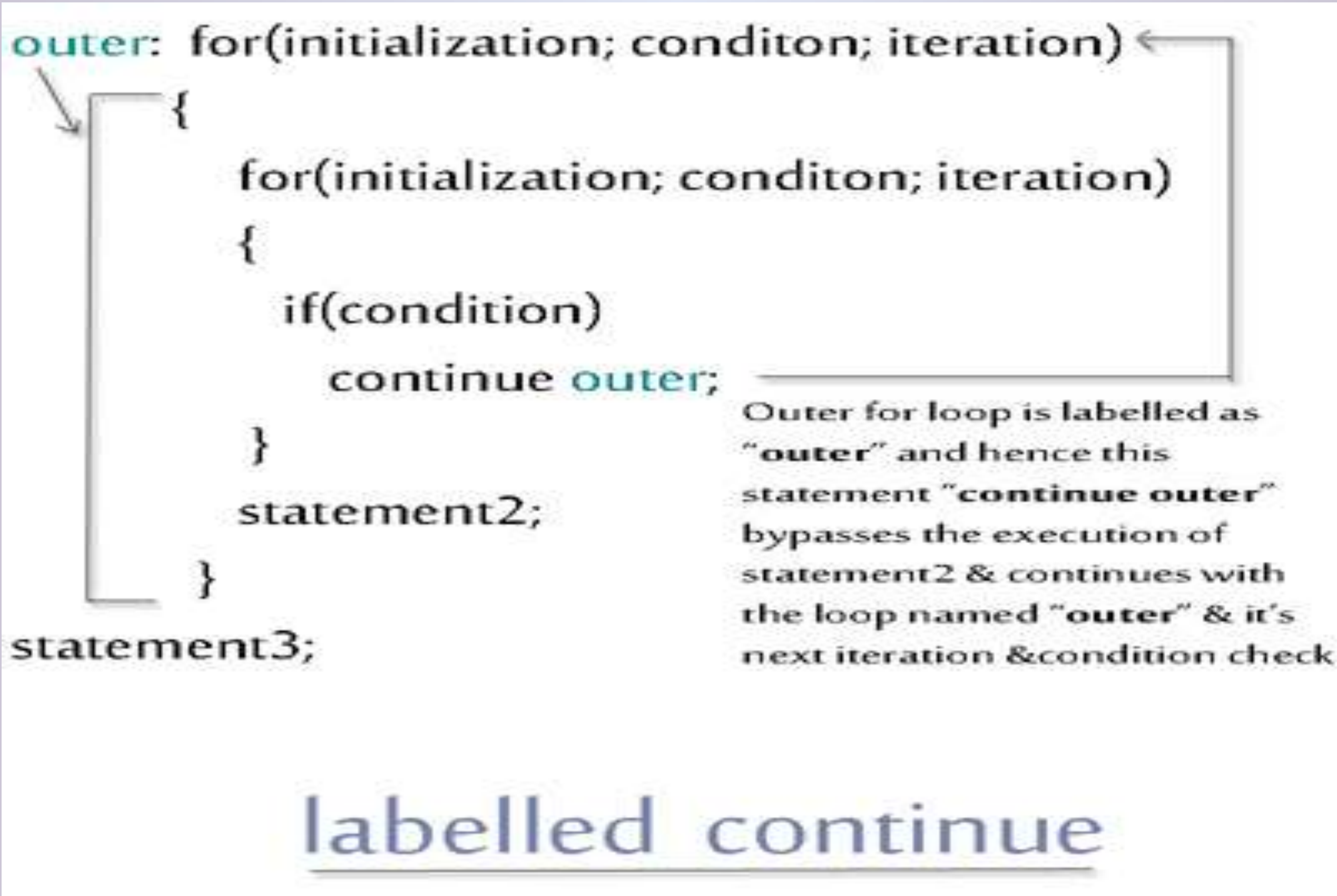
10

Labeled continue Statement

- A labeled continue statement skips the current iteration of an outer loop marked with the given label.
- In **Labelled Continue Statement**, we give a *label/name* to a loop

When this continue statement is encountered with the label/name of the loop, it skips the execution any statement within the loop for the current iteration and *continues* with the next iteration and condition checking in the *labelled loop*.

Concept


outer: for(initialization; conditon; iteration)
{
for(initialization; conditon; iteration)
{
if(condition)
continue outer;
}
statement2;
}
statement3;
Outer for loop is labelled as "outer" and hence this statement "continue outer" bypasses the execution of statement2 & continues with the loop named "outer" & it's next iteration &condition check
labelled continue

Example

```
public class Main
{
  public static void main (String[]args)
  {
  loop:
    for (int i = 0; i < 2; i++)
      {
                for (int j = 0; j < 5; j++)
                  {
                    if (j == 2)
                      continue loop;
                      System.out.println ("i =" + i + " j =" + j);
                  }
      }
    System.out.println ("Out of the loop");
  }
}
```

Output:

i=0 j=0

i=0 j=1

i=1 j=0

i=1 j=1

Out of the loop

return Statement

- The return statement exits from the current method, and control flow returns to where the method was invoked.
- The return statement has two forms: one that returns a value, and one that doesn't.
- To return a value, simply put the value (or an expression that calculates the value) after the return keyword.

```
return ++count;
```

return Statement

- The data type of the returned value must match the type of the method's declared return value.

- When a method is declared void, use the form of return that doesn't return a value.

return;

Output??

```
public class Main
{
  public static void main (String[]args)
  {
  label:
for(int i=0; i<2; i++)
 {
for(int j=0; j<2; j++)
{
System.out.println(i + ", "+ j);
  if(j!=2)
 continue label;
}
 }
  }
}
```

A. 0,0

   1,0

B. 1,1

C. 0,0

D. 0,1

   1,1

Ans: A

Output??

```
public class Main
{
  public static void main (String[]args)
  {
  label:
    for (int i = 0; i < 2; i++)
      {
              for (int j = 0; j < 2; j++)
                {
                  if (j>0)
                    break label;
                  System.out.print(i+" ");
                }
      }
  }
}
```

A. 1
B. 0 1
C. 1 0
D. 0

Ans:D

Output??

```
public class Main
{
  public static void main (String[]args)
  {
  label1:
    for (int i = 1; i <=2; i++)
      {
  label2:
              for (int j = 2; j <=5; j++)
                {
                  System.out.print(j+" ");
                  if (j%2==0)
                    break label1;
                    else
                    continue label2;
                }


      }
  }
}
```

A. 2

B. 1 3 5

C. 1 3

D. 3 5

Ans:A

Output??

```
public class Main
{
 public static void main (String[]args)
 {
    int i=1,j=1;
 label1:
   while(i<=2)
   {
      i++;
                 while(j<=2)
                 {
                 j++;
                 System.out.println(i);
                 if(i==j)
                 break label1;
                 }
     }
 }
}
```

A. 1

B. 2

C. 1 2

D. 2 2

Ans:B

Programming in Java

Topic: Control Flow Statements (Selection and Iteration)

Control Statements

Selection Statements

- Java supports two selection statements: **if** and **switch.**

if statement

if *(condition)* statement1;
else statement2;

- Each statement may be a single statement or a compound statement enclosed in curly braces (block).
- The condition is any expression that returns a **boolean value.**
- **The else** clause is optional.
- If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed.
- In no case will both statements be executed.

Example

```
// To show the working of  if else statement
class Example
        {
                public static void main(String args[])
                {
                  int number=13;
                  if(number%2==0)
                  {
                    System.out.println("even number");
                  }
                else
                  {
                    System.out.println("odd number");
                  }
                }
        }
```

Java if-else-if ladder Statement

```
if(condition1)
{
//code to be executed if condition1 is true
}
else if(condition2)
{
//code to be executed if condition2 is true
}
else if(condition3)
{
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}
```

Example

```
// To show the working of  if else if else ladder stateent
class Example
        {
                public static void main(String args[])
                {
                  int number=13;
                  if(number>0)
                          System.out.println("Positive number");
                  else if(number<0)
                    System.out.println("Negative Number");
                  else
                       System.out.println("Number is zero");
                }
        }
```

Nested ifs

- A nested if is an if statement that is the target of another if or else.
- In nested ifs an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else.

```
if(i == 10) {
if(j < 20) a = b;
if(k > 100) c = d; // this if is
else a = c; // associated with this else
}
else a = d; // this else refers to if(i == 10)
```

Nested ifs-Example

```
//Java Program to demonstrate the use of Nested If Statement.
public class JavaNestedIfExample {
public static void main(String[] args) {
    //Creating two variables for age and weight
    int age=20;
    int weight=80;
    //applying condition on age and weight
    if(age>=18){
        if(weight>50){
            System.out.println("You are eligible to donate blood");
        }
    }
}}
```

switch

- The switch statement is Java’s multi-way branch statement.
- provides an easy way to dispatch execution to different parts of your code based on the value of an expression.
- provides a better alternative than a large series of **if-else-if statements.**

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

- The expression must be of type byte, short, int, char or String.
- Each of the values specified in the case statements must be of a type compatible with the expression.
- Each case value must be a unique literal (i.e. constant not variable).
- Duplicate case values are not allowed.
- The value of the expression is compared with each of the literal values in the case statements.
- If a match is found, the code sequence following that case statement is executed.
- If none of the constants matches the value of the expression, then the default statement is executed.
- The default statement is optional.

- If no case matches and no default is present, then no further action is taken.
- The break statement is used inside the switch to terminate a statement sequence.
- When a break statement is encountered, execution branches to the first line of code that follows the entire switch statement.

```
class SampleSwitch {
    public static void main(String args[]) {
        for(int i=0; i<6; i++)
            switch(i) {
                case 0:
                    System.out.println("i is zero.");
                    break;
                case 1:
                    System.out.println("i is one.");
                    break;
                case 2:
                    System.out.println("i is two.");
                    break;
                default:
                    System.out.println("i is greater than 2.");
            }
        }
    }
```

It is sometimes desirable to have multiple cases without break statements between them. For example, consider the following program:

```
// In a switch, break statements are optional.
class MissingBreak {
public static void main(String args[]) {
for(int i=0; i<12; i++)
switch(i) {
case 0:
case 1:
case 2:
case 3:
case 4:
System.out.println("i is less than 5");
break;
case 5:
case 6:
case 7:
case 8:
case 9:
System.out.println("i is less than 10");
break;
default:
System.out.println("i is 10 or more");
}
}
}
```

This program generates the following output:

```
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is 10 or more
i is 10 or more
```

As you can see, execution falls through each **case** until a **break** statement (or the end of the **switch**) is reached.

Beginning with JDK 7, you can use a string to control a **switch** statement.

```
// Use a string to control a switch statement.
class StringSwitch {
public static void main(String args[]) {
String str = "two";
switch(str) {
case "one":
System.out.println("one");
break;
case "two":
System.out.println("two");
break;
case "three":
System.out.println("three");
break;
default:
System.out.println("no match");
break;
}
}
}
```

Nested switch Statements

- When a switch is used as a part of the statement sequence of an outer switch. This is called a nested switch.

```
switch(count) {
    case 1:
        switch(target) { // nested switch
            case 0:
                System.out.println("target is zero");
                break;
            case 1: // no conflicts with outer switch
                System.out.println("target is one");
                break;
        }
        break;
    case 2: // ...
```

Difference between ifs and switch

- Switch can only test for equality, whereas if can evaluate any type of Boolean expression. That is, the switch looks only for a match between the value of the expression and one of its case constants.

- A switch statement is usually more efficient than a set of nested ifs.

- Note: No two case constants in the same switch can have identical values. Of course, a switch statement and an enclosing outer switch can have case constants in common.

Q1

What will be the output of following code?

```
public class First
{
public static void main(String[] args)
{
boolean x=true;
int a=10;
if(x)
a++;
else
a--;
System.out.println(a);
}
}
```

A. 10

B. 11

C. 0

D. Error

Q2

OUTPUT??

```
public class First
{
public static void main(String[] args)
{
float x=3.45f;
if(x==3.45)
System.out.println("Hello");
else
System.out.println("World");
}
}
```

A. Hello
B. World
C. Error
D. Nothing will be displayed

Q3

OUTPUT??

```
public class First
{
public static void main(String[] args)
{
int a=10;
if(a==11);
System.out.println(++a);
}
}
```

A. 11
B. 10
C. Error
D. Nothing will be displayed

Q4

OUTPUT??

```
public class First
{
public static void main(String[] args)
{
int a=9;
if(a>9)
if(a%2==0)
System.out.println("Hi");
else
System.out.println("Hello");
else
System.out.println("Bye");
}
}
```

A. Hi
B. Hello
C. Bye
D. Error

OUTPUT??

Q5

```
public class First
{
public static void main(String[] args)
{
int a=1;
switch(a)
{
case 1:
a=a+2;
case 2:
a=a*3;
case 3:
a=a/2;
break;
case 4:
a=100;
break;
default:
a=-99;
}
System.out.println(a);
}
}
```

A. 3
B. 4
C. 100
D. -99

OUTPUT??

Q6

```
public class First
{
public static void main(String[] args)
{
int a=1;
switch(a)
{
case 1:
a=12;
break;
case 2-1:
a=13;
break;
case 3:
a=14;
break;
}
System.out.println(a);
}
}
```

A. 12
B. 13
C. Error
D. 14

Q7

OUTPUT??

```
public class First
{
public static void main(String[] args)
{
char c='B';
switch(c)
{
case 65:
System.out.print("1");
case 66:
System.out.print("2");
case 67:
System.out.print("3");
break;
}
}
}
```

A. 123
B. 12
C. 23
D. Error

ITERATION STATEMENTS (LOOPS)

Iteration Statements

- In Java, iteration statements (loops) are:
  - for
  - while, and
  - do-while

- A loop repeatedly executes the same set of instructions until a termination condition is met.

while Loop

- while loop repeats a statement or block while its controlling expression is true.
- The condition can be any Boolean expression.
- The body of the loop will be executed as long as the conditional expression is true.
- When condition becomes false, control passes to the next line of code immediately following the loop.

```
while(condition)
 {
   // body of loop
 }
```

```
class While
    {
            public static void main(String args[]) {
            int n = 10;
            char a = 'G';
            while(n > 0)
                    {
                     System.out.print(a);
                     n--;
                     a++;
                    }
              }
    }
```

Output:

GHIJKLMNOP

- The body of the loop will not execute even once if the condition is false.
- While loop starts with the checking of condition. If it evaluated to true, then the loop body statements are executed otherwise first statement following the loop is executed. For this reason it is also called **Entry control loop**
- The body of the while (or any other of Java's loops) can be empty. This is because a null statement (one that consists only of a semicolon) is syntactically valid in Java.
- Example in next slide:

while loop no body of execution

```
public class NoBody {
public static void main(String args[]) {
int i, j;
i = 100;
j = 200;
// find midpoint between i and j
while(++i < --j); // no body in this loop
System.out.println("Midpoint is " + i);//150 is output[i=150,j=150]
}
}
```

do-while

- The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop.

```
do {
        // body of loop
    } while (condition);
```

- Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression.
- If this expression is true, the loop will repeat. Otherwise, the loop terminates.
- It is known as **exit controlled** loop

Example

```
// Java program to illustrate do-while loop
class dowhileloopDemo
{
        public static void main(String args[])
        {
                int x = 21;
                do
                {
                        // The line will be printed even
                        // if the condition is false
                        System.out.println("Value of x:" + x);
                        x++;
                }
                while (x < 20);
        }
}
```

Output: Value of x:21

for loop

```
for (initialization; condition; iteration)
    {
      // body
    }
```

for loop provides a concise way of writing the loop structure. Unlike a while loop, a for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping.

- Initialization portion sets the value of loop control variable.
- Initialization expression is only executed once.
- Condition must be a Boolean expression. It usually tests the loop control variable against a target value.
- Iteration is an expression that increments or decrements the loop control variable.

The for loop operates as follows.

- When the loop first starts, the initialization portion of the loop is executed.
- Next, condition is evaluated. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates.
- Next, the iteration portion of the loop is executed.

Note: It is also known as **entry controlled loop**

```
class ForTable
        {
                public static void main(String args[])
                 {
                        int n;
                        int x=5;
                        for(n=1; n<=10; n++)
                          {
                           int p = x*n;
                           System.out.println(x+"*"+n +"="+ p);
                          }
                 }
        }
```

Output:

5*1=5
5*2=10
5*3=15
5*4=20
5*5=25
5*6=30
5*7=35
5*8=40
5*9=45
5*10=50

Example of for loop with no body of execution

```
// The body of a loop can be empty.
class Empty3 {
  public static void main(String args[]) {
    int i;
    int sum = 0;

    // sum the numbers through 5
    for(i = 1; i <= 5; sum += i++) ;        No body in this loop!

    System.out.println("Sum is " + sum);
  }
}
```

The output from the program is shown here:

```
Sum is 15
```

What will be the output?

```
class Loop
  {
      public static void main(String args[])
        {
            int i;
            for(i=0; i<5; i++);
                  System.out.println (i++);
         }
   }
```

Declaring loop control variable inside loop

- We can declare the variable inside the initialization portion of the for.

```
for ( int i=0; i<10; i++)
    {
        System.out.println(i);
    }
```

- Note: The scope of this variable i is limited to the for loop and ends with the for statement.

More points….

Interesting for loop variation. Either the initialization or the iteration expression or both may be absent, as in the following example:

```
// Parts of the for loop can be empty.
class ForVar {
public static void main(String args[]) {
int i;
boolean done = false;
i = 0;
for( ; !done; ) {
System.out.println("i is " + i);
if(i == 10) done = true;
i++;
}
}
}
```

Output: Value of I will be printed from 0 to 10

More points…

- We can intentionally create an infinite loop (a loop that never terminates) if you leave all three parts of the **for** empty. For example:

```
for( ; ; ) {
// ...
}
```

- Java permits you to include multiple statements in both the initialization and iteration portions of the **for**. Each statement is separated from the next by a comma. Using the comma, the preceding **for** loop can be more efficiently coded, as shown here:

```
// Using the comma.
class Comma {
public static void main(String args[]) {
int a, b;
for(a=1, b=4; a<b; a++, b--) {
System.out.println("a = " + a);
System.out.println("b = " + b);
}
}
}
```

Output:

```
a = 1
b = 4
a = 2
b = 3
```

Q1

OUTPUT??

```
public class First
{
public static void main(String[] args)
{
int i=5;
while(i)
{
System.out.println("Hello");
i--;
}
}
}
```

A. Hello will be printed 5 times
B. Compile time error
C. Runtime error
D. Infinite loop

Q2(Output??)

```
public class First
{
public static void main(String[] args)
{
int x=5,sum=0;
boolean y=true;
while(y)
{
sum=sum+x;
x--;
if(x==3)
y=!y;
}
System.out.println(sum);
}
}
```

A. 9
B. Compile time error
C. Infinite loop
D. 12

Q3(Output??)

```
public class First
{
public static void main(String[] args)
{
int x=5,y=1;
while(--x!=++y);
System.out.println(x+y);
}
}
```

A. Compile time error
B. 6
C. 4
D. 2

Q4(Output??)

```
public class First
{
public static void main(String[] args)
{
int k=1;
for(int i=1,j=2;i>=1 & i<=3;i++,j++)
{
k=k*j;
}
System.out.println(k);
}
}
```

A. 24
B. 12
C. Compile time error
D. Runtime error

Q5

How many times “Hello” will be printed in the following code?

```
public class First
{
public static void main(String[] args)
{
int i=24;
for(;i>1;i>>=2)
{
System.out.println("Hello");
}
}
}
```

A. 2 times
B. 3 times
C. 5 times
D. 4 times

Programming in Java

Topic: Date Time API

Contents…

- Introduction
- Local Date
- Local Time
- Local Date Time

Introduction

- New DateTime API is introduced in jdk8.
- LocalDate, LocalTime and LocalDateTime classes are provided in java.time package.

Java Date and Time API goals

- Classes and methods should be straight forward.
- The API should support fluent API approach.
- Instances of Date and Time objects should be immutable.
- Should be thread safe.
- Use ISO standard to define Date and Time.
- API should support strong type checks.
- Allows developers to extend API.

Working with Local Date and Time

- Java.time package provides two classes for working with local Date and Time.
- LocalDate
  - Does not include time
  - A year-month-day representation
  - toString – ISO 8601 format(YYYY-MM-DD)
- LocalTime
  - Does not include date
  - Stores hours:minutes:seconds:nanoseconds
  - toString- (HH:mm:ss.SSS)

LocalDate, LocalTime and LocalDateTime

- They are local in the sense that they represent date and time from the context of one observer, in contrast to time zones.
- All the core classes in the new API are constructed by factory methods.
- When constructing a value through its fields, the factory is called *of*.
- When converting from another type, the factory is called *from*.
- There are also parse methods that take strings as parameters.

LocalDate Class

LocalDate Class

- A date without a time-zone in the ISO-8601 calendar system, such as 2007-12-03.
- LocalDate is an immutable date-time object that represents a date, often viewed as year-month-day.
- Other date fields, such as day-of-year, day-of-week and week-of-year, can also be accessed.
- This class does not store or represent a time or time-zone so its portable across time zones.

Methods of LocalDate

- public static LocalDate now()
- public static LocalDate now(ZoneId zone)

- public static LocalDate of(int year, Month month, int dayOfMonth)

Note:DateTimeException can be thrown

- public static LocalDate of(int year, int month, int dayOfMonth)

Note: *DateTimeException can be thrown.*

- public static LocalDate parse(CharSequence text)

Note: *DateTimeParseException can be thrown.*

Example (now() method)

```
// Java program to demonstrate
// LocalDate.now() method

import java.time.*;
public class Test {
    public static void main(String[] args)
    {
     // create an LocalDate object
     LocalDate lt = LocalDate.now();
     // print result
     System.out.println("LocalDate : "+ lt);
    }
}
```

Example (now(ZoneId zone))

```
// Java program to demonstrate LocalDate.now() method
import java.time.*;
public class Test {
    public static void main(String[] args)
    {
     // create a clock
     ZoneId zid = ZoneId.of("Asia/Kolkata");
     // create an LocalDate object using now(zoneId)
     LocalDate lt = LocalDate.now(zid);
     // print result
     System.out.println("LocalDate : "+ lt);
    }
}
```

Example (of() method)
public static LocalDate of(int year,int month,int dayOfMonth)

```
// Java program to demonstrate LocalDate.of(int month) method
import java.time.*;
public class Test {
    public static void main(String[] args)
    {
     // create LocalDate object
     LocalDate localdate = LocalDate.of(2020, 5, 13);
     // print full date
     System.out.println("Date: " + localdate);
    }
}
```

Output:

Date:2020-05-13

Example (of() method)

public static LocalDate of(int year,Month month,int dayOfMonth)

```
// Java program to demonstrate
// LocalDate.of(Month month) method
import java.time.*;
public class Test {
    public static void main(String[] args)
    {
     // create LocalDate object
     LocalDate localdate = LocalDate.of(2020, Month.MAY, 13);
     // print full date
     System.out.println("Date: "+ localdate);
    }
}
```

Output:

Date: 2020-05-13

Example:parse() method

```
import java.time.*;
public class Test {
    public static void main(String[] args)
    {
     // create an LocalDate object
     LocalDate lt = LocalDate.parse("2020-05-13");
     // print result
     System.out.println("LocalDate : "+ lt);
    }
}
```

Example

```
LocalDate ldt = LocalDate.now();

ldt = LocalDate.of(2015, Month.FEBRUARY, 28);

ldt = LocalDate.of(2015, 2, 13);

ldt = LocalDate.parse("2017-02-28");
```

LocalTime Class

LocalTime Class

- A time without a time-zone in the ISO-8601 calendar system, such as 10:15:30. 13

- LocalTime is an immutable date-time object that represents a time, often viewed as hour-minute-second.

- Time is represented to nanosecond precision.
- For example, the value "13:45:30.123" can be stored in a LocalTime.

- This class does not store or represent a date or time-zone.

Methods of LocalTime

Methods

- public static LocalTime now()
- public static LocalTime now(ZoneId zone)

- public static LocalTime of(int hour, int minute)
- public static LocalTime of(int hour, int minute, int second)
- public static LocalTime of(int hour, int min, int sec, int nsec)

- public static LocalTime parse(CharSequence text)

Example(now() method)
public static LocalTime now()

```
// Java program to demonstrate LocalTime.now() method
import java.time.*;
public class Test {
    public static void main(String[] args)
    {
      // apply now() method
      // of LocalTime class
      LocalTime time = LocalTime.now();
      // print time
      System.out.println("Time: "+ time);
    }
}
```

Output: It varies as the time passes.

Time: 20:43:41.453

Example(now(ZoneId zone)

```
// Java program to demonstrate LocalTime.now() method
import java.time.*;
public class Test {
    public static void main(String[] args)
    {
        // create a clock
        ZoneId zid = ZoneId.of("Asia/Kolkata");
        LocalTime time = LocalTime.now();
        // print time
        System.out.println("Time: "+ time);
    }
}
```

Output:

Time: 06:30:45.936

Output may vary with the passage of time

Example(of()) public static LocalTime of(int hour,int minute)

```
// Java program to demonstrate LocalTime of(int hour, int minute) method
import java.time.*;
public class Main {
    public static void main(String[] args)
    {
      // Create LocalTime object
      LocalTime localtime = LocalTime.of(6, 5);
      // Print time
      System.out.println("TIME: "+ localtime);
    }
}
```

Output:

TIME: 06:05

Example:public static LocalTime of(int hour,int minute,int second)

```
// Java program to demonstrate LocalTime of(int hour, int minute, int second) method
import java.time.*;
public class Main {
    public static void main(String[] args)
    {
      // Create LocalTime object
      LocalTime localtime = LocalTime.of(6, 5, 40);
      // Print time
      System.out.println("TIME: "+ localtime);
    }
}
```

Output:

TIME: 06:05:40

Example(public static LocalTime of(int hour,int minute,int second,int nanosecond))

```
// Java program to demonstrate LocalTime of(int hour, int minute, int second, int
    nanosecond) method
import java.time.*;
public class Main {
    public static void main(String[] args)
    {
     // Create LocalTime object
     LocalTime localtime = LocalTime.of(6, 5, 40, 50);
     // Print time
     System.out.println("TIME: "+ localtime);
    }
}
```

Output:

TIME: 06:05:40.000000050

Example(public static LocalTime parse(CharSequence text))

```
// Java program to demonstrate LocalTime.parse() method
import java.time.*;
public class Main {
    public static void main(String[] args)
    {
      // create an LocalTime object
      LocalTime lt = LocalTime.parse("10:15:45");
      // print result
      System.out.println("LocalTime : "+ lt);
    }
}
```

Output:

LocalTime : 10:15:45

LocalDateTime Class

LocalDateTime Class

- A date-time without a time-zone in the ISO-8601 calendar system, such as 2007-12-03T10:15:30.

- LocalDateTime is an immutable date-time object that represents a date-time, often viewed as year-month-day-hour-minute-second.
- Other date and time fields, such as day-of-year, day-of-week and week-of-year, can also be accessed.

- Time is represented to nanosecond precision.
- For example, the value "2nd October 2007 at 13:45.30.123456789" can be stored in a LocalDateTime.

Methods of LocalDateTime

Methods

- public static LocalDateTime now()
- public static LocalDateTime now(ZoneId zone)

- public static LocalDateTime of(int year, int mnth, int day, int hour, int mint)
- public static LocalDateTime of(int year, int mnth, int day, int hour, int mint, int sec)
- public static LocalDateTime of(int year, int mnth, int day, int hour, int mint, int sec, int nsec)
- public static LocalDateTime of(LocalDate d, LocalTime t)
- public static LocalDateTime parse(CharSequence text)

Example—now()

```
// Java program to demonstrate LocalDateTime.now() method
import java.time.*;
public class Test {
    public static void main(String[] args)
    {
      // create an LocalDateTime object
      LocalDateTime lt = LocalDateTime.now();
      // print result
      System.out.println("LocalDateTime : "+ lt);
    }
}
```

Sample output:

LocalDateTime : 2021-02-19T10:03:55.356

Example—now()

```
// Java program to demonstrate LocalDateTime.now() method
import java.time.*;
public class Main {
    public static void main(String[] args)
    {
      // create a clock
      ZoneId zid = ZoneId.of(“Asia/Kolkata");
      // create an LocalDateTime object using now(zoneId)
      LocalDateTime lt   = LocalDateTime.now(zid);
      // print result
      System.out.println("LocalDateTime : "+ lt);
    }
}
```

Sample Output:

LocalDateTime : 2021-02-20T09:37:12.068

Example—of()

```
import java.time.*;
 public class Main {
     public static void main(String[] args)
     {
      // create LocalDateTime object
      LocalDateTime localdatetime1 = LocalDateTime.of(2020, 5, 13, 6, 30);
      // print full date and time
      System.out.println("DateTime: "+ localdatetime1); //DateTime: 2020-05-13T06:30
      LocalDateTime localdatetime2 = LocalDateTime.of(2020, 5, 13, 6, 30,45);
      // print full date and time
      System.out.println("DateTime: "+ localdatetime2); //DateTime: 2020-05-13T06:30:45
      // create LocalDateTime object
       LocalDateTime localdatetime3 = LocalDateTime.of(2020, 5, 13, 6, 30, 45, 20000);
       // print full date and time
       System.out.println("DateTime: "+ localdatetime3); //DateTime: 2020-05-
     13T06:30:45.000020
     }
}
```

Example-of()

```
// Java program to demonstrate LocalDateTime.of(LocalDate date, LocalTime time)
    method
import java.time.*;
public class Main {
    public static void main(String[] args)
    {
      // Create LocalDate object using LocalDate.of() method
      LocalDate date = LocalDate.of(2020, 5, 13);
      // Create LocalTime object using LocalTime.of() method
      LocalTime time = LocalTime.of(6, 30);
      // Create LocalDateTime object
      LocalDateTime localdatetime = LocalDateTime.of(date, time);
      // Print full date and time
      System.out.println( "DateTime: " + localdatetime); //DateTime: 2020-05-13T06:30
    }
}
```

Example-parse()

```
// Java program to demonstrate LocalDateTime.parse() method
import java.time.*;
public class Main {
    public static void main(String[] args)
    {
    // create an LocalDateTime object
     LocalDateTime lt = LocalDateTime .parse("2018-12-30T19:34:50.63");
     // print result
     System.out.println("LocalDateTime : "+ lt);
    }
}
```

Output:

LocalDateTime : 2018-12-30T19:34:50.630

Programming in Java

Object, Classes, Methods and Constructors

Object and classes in java

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

An object is a real-world entity.

An object is a runtime entity.

The object is an entity which has state and behavior.

The object is an instance of a class.

What is a class in Java

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

Fields

Methods

Constructors

Blocks

Nested class and interface

A class is also a data type. You can use it to declare object reference variables. An object reference variable that appears to hold an object actually contains a reference to that object. Strictly speaking, an object reference variable and an object are different, but most of the time the distinction can be ignored.

Syntax to declare a class

```
class <class_name>
{
    field;
    method;
}
```

Class is a keyword in java. <class_name> can be replaced by name of the class we want to give.

Syntax to declare an object using new keyword

Class_name object_name = new class_name();

For example:

rectangle r = new rectangle(); // for single object

rectangle r1 = new rectangle(), r2= new rectangle(); // for two objects

- In java object is just a reference variable for accessing members of the class.
- An object is an instance of a class. You use the new operator to create an object, and the dot operator (.) to access members of that object through its reference variable.

Object and Class Example: main within the class

```
//Java Program to illustrate how to define a class and fields
//Defining a Student class.
class Student{
 //defining fields
 int id;
 String name;
 public static void main(String args[])
{
   Student s1=new Student();//creating an object of Student
  System.out.println(s1.id);//accessing member through reference variable
  System.out.println(s1.name);
 }
}
```

Output

0

Object and Class Example: main outside the class

```
class Student{
 int id;
 String name;
}
//Creating another class Example which contains the main method
class Example{
 public static void main(String args[]){
  Student s1=new Student();
  System.out.println(s1.id);
  System.out.println(s1.name);
 }
}
```

Now your file name should be same as that of class name in which main method is defined.

Method in Java

Making programs modular and reusable is one of the central goals in software engineering. Java provides many powerful constructs that help to achieve this goal. Methods are one such construct.In Java, a method is like a function which is used to expose the behavior of an object.

Advantage of Method

Code Reusability

We will discuss methods in more detail after few slides

Ways to initialize object

There are 3 ways to initialize object in Java.

By reference variable

By method

By constructor

By reference variable

```
class Student{
 int id;
 String name;
}
class Example
{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name=“rohit";
  System.out.println(s1.id+" "+s1.name);
 }
}
```

Initialization through method

```
class Student{
 int rollno;
 String name;
 void insertRecord(int r, String n)
{
  rollno=r;
  name=n;
 }
 void displayInformation()
{
System.out.println(rollno+" "+name);
}
}
```

```
class TestStudent4
{
public static void main(String args[])
{
  Student s1=new Student();
  Student s2=new Student();
  s1.insertRecord(111,"Karan");
  s2.insertRecord(222,"Aryan");
  s1.displayInformation();
  s2.displayInformation();
 }
}
```

Initialization through a constructor

We will discuss the constructor seperately.

Methods

Method

- A method is a construct for grouping statements together to perform a function.
- A method that returns a value is called a value retuning method, and the method that does not return a value is called void method.
- In some other languages, methods are referred to as procedures or functions.
- A method which does not return any value is called a procedure.
- A method which returns some value is called a function.

Defining a Method

Syntax:

modifier returnType methodName (list of parameters)

{

Method Signature

// Body of the method(set of statements);

}

Modifier can be static, public, private or protected.

Example:

int sum(int a, int b)

Method Header

{

…

Method Body

}

- Method header specifies the modifier, return type, method name and parameters of method.

```
public void display()
  {…}
```

- The variables defined in method header are called formal parameters or parameters.

```
void display(int x, int y)
  {…}
```

- When a method is invoked, a value as a parameter is passed which is known as actual parameters or arguments.

```
a.display (3, 5);
```

- Method body contains a set of statements that define the function to be performed by the method.

- A return statement using the keyword *return* is required for a value-returning method to return a result.

Calling a Method

- To use a method, we need to *call* or *invoke* it.
- There are two ways to call a method:
  - If the method returns a value, a call to method is usually treated as a value.

    int area = **rectangleArea** (4,6);

    System.out.println(**rectangleArea** (4,6));

  - If the method returns void, a call to method must be a statement.
  - For example, println method returns void

    System.out.**println**("Hello…");

```
// Defining method with in the class in which main method is defined.

class Example
{
	int max (int a,int b)
	{
		return (a>b?a:b);
	}
	public static void main(String[] args)
	{
	Example ob = new Example();
	 System.out.println("Maximum of 12 and 34 is "+ ob.max(12,34));
	}
}
```

```
// Defining a static method with in the class in which main method is defined.
// When a method is defined as static method with in the same class in which
// main method is defined. We need not to create object to call that method.
class Example
{
	static int max (int a,int b)
	{
		return (a>b?a:b);
	}
	public static void main(String[] args)
	{
	System.out.println("Maximum of 12 and 34 is "+ max(12,34));
	}
}
```

Example of methods defined in the separate class

```
class Rectangle{
 int length;
 int width;
 void insert(int l, int w)// method
{
  length=l;
  width=w;
 }
 void calculateArea() // method
{
System.out.println(length*width);
}
}
```

```
class TestRectangle1{
 public static void main(String args[])
{
  Rectangle r1=new Rectangle();
  Rectangle r2=new Rectangle();
  r1.insert(11,5);
  r2.insert(3,15);
  r1.calculateArea();
  r2.calculateArea();
}
}
```

Calling method using anonymous objects

Anonymous objects are the objects that are instantiated but are not stored in a reference variable.

- They are used for immediate method calling.
- They will be destroyed after method calling.

Example

```
class factorial
{

    double calculate(double x)
    {
        double i,f=1;
        for(i=1;i<=x;i++)
        f*=i;
        return f;
    }
}
```

```
class Example
{
     public static void main(String[] args)
     {
    double f= new factorial().calculate(5);
     System.out.println(f);
     }
}
```

Exercise

Write a program to create a class BankAccount having instance variable *balance*.Implement a method deposit(int amt) which receives the amount to be deposited as an argument and adds to the current balance.

Implement another method int withdraw() which asks the user to enter the amount to be withdrawn and updates the balance if having sufficient balance and return the new balance. Invoke both the methods from TestBankAccount class.

Call by Value and Call by Reference in Java

There is only call by value in java, not call by reference. If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.

However we can implement call by reference method if we pass object of the class rather than primitive data types.

//Example of call by value Method

```
public class Main
{
   void byValue(int x)
   {
      x=x+10;
   }
     public static void main(String[] args) {
          Main obj=new Main();
          int x=12;
          System.out.println("Before method call:"+x);
          obj.byValue(x);
          System.out.println("After method call:"+x);
     }
}
```

Before method call:12

After method call:12

In java if we pass an object of the class then we can implement call by reference method. In case of call by reference original value is changed if we made changes in the called method.

```
class Example
{
 int data=50;
 void change(Example op)
{
 op.data=op.data+100;//changes will be in the instance variable
 }
     public static void main(String args[])
 {
      Example op=new Example();
    System.out.println("before change "+op.data);
   op.change(op);//passing object
   System.out.println("after change "+op.data);

 }
}
```

Passing Array to method(1D)

When passing an array to a method, the reference of the array is passed to the method.

```
public class Main
{
  static int method(int arr[])
   {
      int sum=0;
      for(int i=0;i<arr.length;i++)
      {
         sum=sum+arr[i];
      }
      return sum;
   }
    public static void main(String[] args) {
    int a[]={1,2,3,4,5};
    System.out.println("Sum of array elements is:"+method(a));
   }
}
```

Passing anonymous array to a method(1D)[Anonymous array means without creating reference for array]

```
public class Main
{
  static int method(int arr[])
    {
      int sum=0;
      for(int i=0;i<arr.length;i++)
      {
        sum=sum+arr[i];
      }
      return sum;
    }
     public static void main(String[] args) {
     System.out.println("Sum of array elements is:"+method(new int[]{1,2,3,4,5}));
    }
}
```

Passing Array to a method(2D)

```
public class Main
{
   static int method(int arr[][])
     {
        int sum=0;
        for(int i=0;i<arr.length;i++)
        {
           for(int j=0;j<arr[i].length;j++)
           {
           sum=sum+arr[i][j];
           }
        }
        return sum;
     }
      public static void main(String[] args) {
      int a[][]={{1,2},{3,4}};
      System.out.println("Sum of array elements is:"+method(a));
     }
}
```

Returning array from a method[1D]

```
public class Main
{
  static int[] method()
   {
   int a[]={1,2,3,4,5};
   return a;
   }
    public static void main(String[] args) {
    int arr[]=method();
    for(int i=0;i<arr.length;i++)
    {
      System.out.println(arr[i]);
    }
   }
}
```

Returning array from a method[2D]

```
public class Main
{
  static int[][] method()
   {
   int a[][]={{1,2},{3,4}};
   return a;
   }
     public static void main(String[] args) {
     int arr[][]=method();
     for(int i=0;i<arr.length;i++)
     {
         for(int j=0;j<arr[i].length;j++)
         System.out.println(arr[i][j]);
     }
   }
}
```

Using varargs to pass variable number of arguments to a method

```
// Java program to demonstrate varargs
public class Main
{
   // A method that takes variable number of integer arguments.
   static void fun(int ...a)
   {
      System.out.println("Number of arguments: " + a.length);
      // using for each loop to display contents of a
      for (int i: a)
      System.out.print(i + " ");
      System.out.println();
   }
   public static void main(String args[])
   {
      // Calling the varargs method with different number
      // of parameters
      fun(100);        // one parameter
      fun(1, 2, 3, 4);  // four parameters
      fun();           // no parameter
   }
}
```

Output:

Number of arguments: 1

100

Number of arguments: 4

1 2 3 4

Number of arguments: 0

Method Overloading

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- If we have to perform only one operation, having same name of the methods increases the readability of the program.
- **Advantage of method overloading**
- Method overloading *increases the readability of the program*.
- Different ways to overload the method
- There are two ways to overload the method in java
- By changing number of arguments
- By changing the data type
- **Note: Methods are never overloaded by just changing their return types**

Example 1(Changing no. of arguments)

```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}
}
```

Output:

22

33

Example 2(Changing data type of arguments)

```
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

Output:

22

24.9

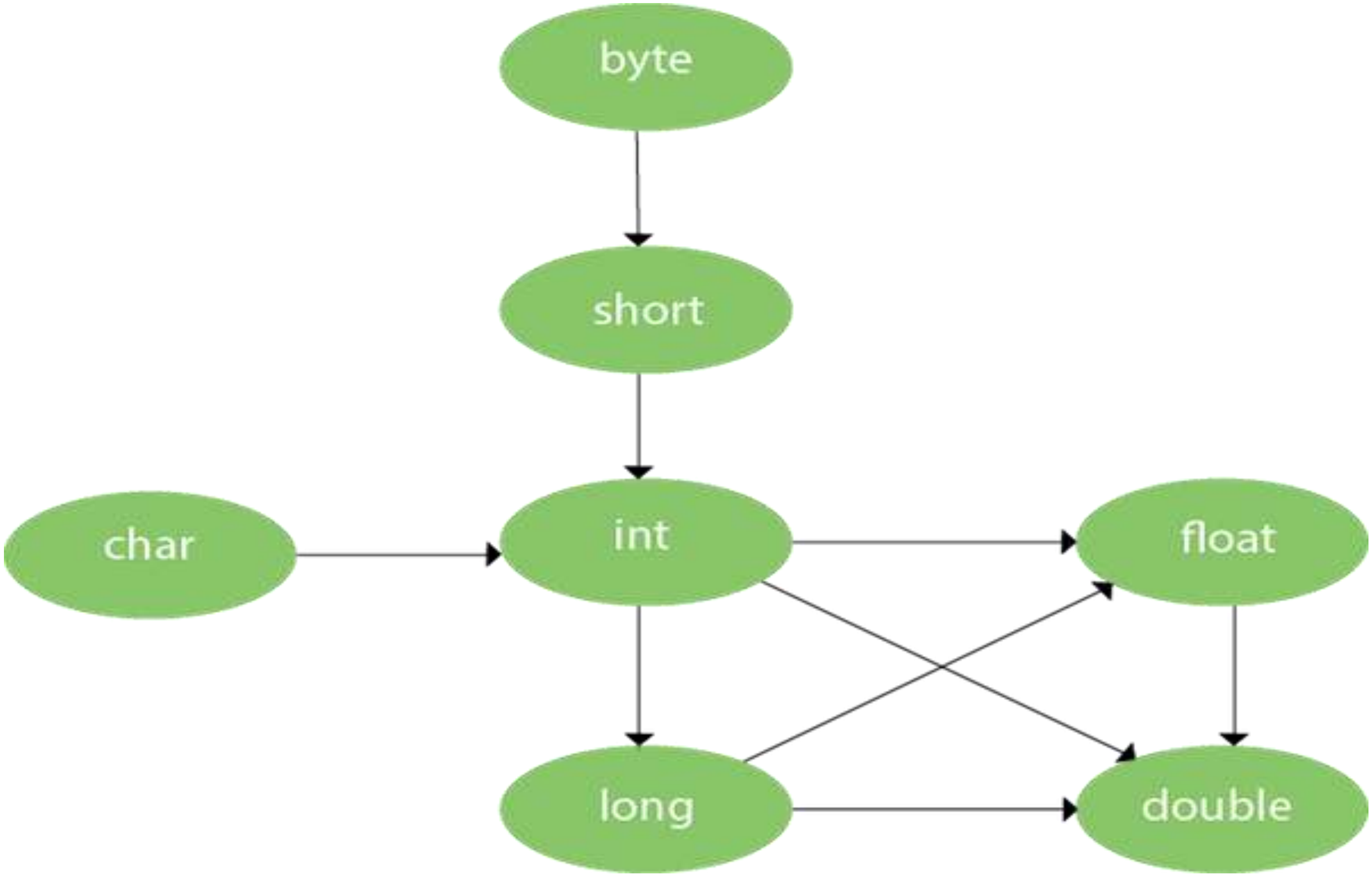
Note: In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

```
class Adder{
static int add(int a,int b){return a+b;}
static double add(int a,int b){return a+b;}
}
class TestOverloading3{
public static void main(String[] args){
System.out.println(Adder.add(11,11));//ambiguity
}}
```

Output:

Compile Time Error: method add(int,int) is already defined in class Adder

Method Overloading and Type Promotion



As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int, long, float or double. The char datatype can be promoted to int,long,float or double and so on.

Example-Type Promotion

```
class OverloadingCalculation1{
  void sum(int a,long b){System.out.println(a+b);}
  void sum(int a,int b,int c){System.out.println(a+b+c);}
  public static void main(String args[]){
  OverloadingCalculation1 obj=new OverloadingCalculation1();
  obj.sum(20,20);//now second int literal will be promoted to long
  obj.sum(20,20,20);
  }
}
```

Output:

40

60

CONSTRUCTOR

Constructors

- A constructor is a special method that is used to initialize a newly created object.
- It is called just after the memory is allocated for the object.
- It can be used to initialize the objects with some defined values or default values at the time of object creation.
- Constructor cannot return values.
- Constructor has the same name as the class name.
- It is not mandatory for the coder to write constructor for the class.

Default Constructor

- A constructor is called "Default Constructor" when it doesn't have any parameter.
- If we do not provide any constructor in the class then default constructor will automatically called to initialize the values of data members.
  - numeric data types are set to 0
  - char data types are set to null character(‘’)
  - reference variables are set to null
  - Boolean are set to false

- In order to create a Constructor observe the following rules:
  - It has the **same name** as the class
  - It should not return a value, not even ***void***

Defining a Constructor

- Like any other method

```
public class ClassName {

    // Data Fields…

    // Constructor
    public ClassName()
    {
        // Method Body Statements initialising Data Fields
    }

    //Methods to manipulate data fields
}
```

- Invoking:
  - There is NO explicit invocation statement needed: When the object is created, the constructor method will be executed automatically.

Constructors

- Constructor name is class name. A constructors must have the same name as the class its in.

- Default constructor. If you don't define a constructor for a class, a default (parameter-less) constructor is automatically created by the compiler.

- The default constructor initializes all instance variables to default value (zero for numeric types, null for object references, and false for booleans).

Key Points

- Default constructor is created only if there are no constructors.
- If you define *any constructor* for your class, no default constructor is automatically created.
- There is *no return type* given in a constructor signature (header).
- There is *no return statement* in the body of the constructor.

Key Points

- The *first line* of a constructor must either be a call on another constructor in the same class (using this), or a call on the super-class constructor (using super).

- If the first line is neither of these, the compiler automatically inserts a call to the parameter-less super class constructor.

Parameterized Constructors

A constructor which has a specific number of parameters is called a parameterized constructor.

```
class rectangle
{
    int l,b;
    rectangle(int x,int y)
    {
        l=x;
        b=y;
    }
    int area()
    {
        return l*b;
    }
}
```

```
class Example
{
public static void main(String[] args) {
        rectangle r1 = new rectangle(12,34), r2= new rectangle(34,56);
        System.out.println(r1.area());
        System.out.println(r2.area());
    }
}
```

Constructor Overloading

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

Example of constructor overloading

```
class sum
{
   sum(int x,int y)
   {
   System.out.println("Sum of 2 integers are "+(x+y));
   }
   sum(int x,int y,int z)
   {
   System.out.println("Sum of 3 integers are "+(x+y+z));
   }
   sum(double x,double y)
   {
   System.out.println("Sum of 2 doubles are "+(x+y));
   }
}
```

```
class Example
{
    public static void main(String[] args) {
        sum s1 = new sum(12,34);
        sum s2 = new sum(12,34,67);
        sum s3 = new sum(12.56,34.78);
    }
}
```

Let’s Do Some thing

Write a program to create a class named Patient which contains:

a. Attributes patient _name, age, contact_number
b. Constructor to initialize all the patient attributes
c. A method to display all the details of any patient

Create a Patient object and display all the details of that patient.

Q1

```
class Test {
  int i;
}
public class Main {
  public static void main(String args[]) {
      Test t = new Test();
      System.out.println(t.i);
   }
}
```

A.-1

B. 0

C. Compiler error

D. Runtime error

Q2

```
class Test {
 int i;
}
pubic class Main {
  public static void main(String args[]) {
    Test t;
    System.out.println(t.i);
}
```

A. 0

B. -1

C. Compiler error

D. Runtime error

Q3

```
public class Main
{
    static void modify(int a)
    {
     a=a+12;
    }
    public static void main(String[] args) {
    int x=12;
    modify(x);
    System.out.println(x);
    }
}
```

A. 24

B. 12

C. 0

D. Compile time error

Q4

```
public class Main
{
   int a,b;
   static void modify(Main obj)
   {
      obj.a=obj.a+2;
      obj.b=obj.b+2;
   }
       public static void main(String[] args)
{
       Main o1=new Main();
       o1.a=13;
       o1.b=15;
       modify(o1);
       System.out.println(o1.a+" "+o1.b);
       }
}
```

A. 13 15

B. 15 17

C. 0 0

D. 1 1

Q5

```
class Dummy
{
    static void show()
    {
        System.out.println("Hello");
    }
}
public class Main
{
        public static void main(String[] args) {
            show();
        }
}
```

A. Hello
B. Compile time error
C. Runtime error
D. Nothing will be displayed

Programming in Java

String Handling

Introduction

- Every string we create is actually an object of type String.
- String constants are actually String objects.
- Example:

String Constant

System.out.println("This is a String, too");

- Objects of type String are immutable i.e. once a String object is created, its contents cannot be altered.
- Because String objects are immutable, whenever we want to modify a String, it will construct a new copy of the string with modifications.

Introduction

- In java, four predefined classes are provided that either represent strings or provide functionality to manipulate them. Those classes are:
  - String
  - StringBuffer
  - StringBuilder
  - *StringTokenizer*

- String, StringBuffer, and StringBuilder classes are defined in java.lang package and all are final.

- All of them implement the CharSequence interface.

Declaring and creating string

To represent a string of characters, use the data type called String. For example, the following code declares message to be a string with the value "Welcome to Java".

```
String message = "Welcome to Java";
```

String is a predefined class in the Java library, just like the classes System and Scanner. The String type is not a primitive type. It is known as a reference type. Any Java class can be used as a reference type for a variable. The variable declared by a reference type is known as a reference variable that references an object. Here, message is a reference variable that references a string object with contents Welcome to Java.

Different ways of creating strings:

There are two ways to create string in Java:

String literal

String s = “Hello”;

Using new keyword

String s = new String (“Hello”);

Simple Methods for String object

Method	*Description*
`length()`	Returns the number of characters in this string.
`charAt(index)`	Returns the character at the specified index from this string.
`concat(s1)`	Returns a new string that concatenates this string with string s1.
`toUpperCase()`	Returns a new string with all letters in uppercase.
`toLowerCase()`	Returns a new string with all letters in lowercase
`trim()`	Returns a new string with whitespace characters trimmed on both sides.

Examples

```
class Example
{
        public static void main(String[] args)
        {
        String s="Hello World";
        System.out.println("Length of the string s is "+ s.length());
        System.out.println("Character at position 4 is "+ s.charAt(4));
        String s1=" Welcome to java";
        System.out.println("String after joining of s and s1"+ s.concat(s1));
        System.out.println("String in upper case letters"+ s.toUpperCase());
        System.out.println("String in lower case letters"+ s.toLowerCase());
        String s2=" Hello ";
        System.out.println("String s2 after trimming white spaces from both ends "+s2.trim());

    }
}
```

Output:

Length of the string s is 11
Character at position 4 is o
String after joining of s and s1: Hello World Welcome to java
String in upper case letters: HELLO WORLD
String in lower case letters: hello world
String s2 after trimming white spaces from both ends Hello

Reading a String

- Two methods can be used.
- next()
- nextLine()

- next() method is used to take input of string that ends with a whitespace character.
- nextLine() You can use the nextLine() method to read an entire line of text. The nextLine() method reads a string that ends with the Enter key pressed. For example, the following statements read a line of text.

Example:

```
//next() method
import java.util.Scanner;
public class Main
{
  public static void main (String[]args)
  {
    Scanner input = new Scanner (System.in);
    System.out.print ("Enter three words separated by spaces: ");
    String s1 = input.next ();
    String s2 = input.next ();
    String s3 = input.next ();
    System.out.println ("s1 is " + s1);
    System.out.println ("s2 is " + s2);
    System.out.println ("s3 is " + s3);
  }
}
```

Output:

```
Enter three words separated by spaces: Hi Hello Bye    //user input
s1 is Hi
s2 is Hello
s3 is Bye
```

Example:

nextLine():

```
import java.util.Scanner;
public class Main
{
  public static void main (String[]args)
  {

    Scanner input = new Scanner (System.in);
    System.out.println ("Enter a line: ");
    String s = input.nextLine ();
    System.out.println ("The line entered is " + s);
  }
}
```

Output:

```
Enter a line:
Hello this is one string   //user input
The line entered is Hello this is one string
```

Comparing Strings

Method	*Description*
`equals(s1)`	Returns true if this string is equal to string `s1`.
`equalsIgnoreCase(s1)`	Returns true if this string is equal to string `s1`; it is case insensitive.
`compareTo(s1)`	Returns an integer greater than 0, equal to 0, or less than 0 to indicate whether this string is greater than, equal to, or less than `s1`.
`compareToIgnoreCase(s1)`	Same as `compareTo` except that the comparison is case insensitive.
`startsWith(prefix)`	Returns true if this string starts with the specified prefix.
`endsWith(suffix)`	Returns true if this string ends with the specified suffix.
`contains(s1)`	Returns true if s1 is a substring in this string.

Examples

```
class Example
{
		public static void main(String[] args)
		{
				String s1="Hello World";
				String s2="Hello World";
				String s3="Welcome to java";
				System.out.println(s1.equals(s2));// true
				System.out.println(s1.equals(s3));// false
				System.out.println(s1.compareTo(s3));// value less than 0
				System.out.println(s1.startsWith("H"));// true
				System.out.println(s3.startsWith("H"));// false
				System.out.println(s1.endsWith("d"));// true
				System.out.println(s3.contains("to"));// true
				System.out.println(s1.contains("to"));// false

		}
}
```

Methods for finding substrings/or characters in a given string

Method	*Description*
index(ch)	Returns the index of the first occurrence of ch in the string. Returns -1 if not matched.
indexOf(ch, fromIndex)	Returns the index of the first occurrence of ch after fromIndex in the string. Returns -1 if not matched.
indexOf(s)	Returns the index of the first occurrence of string s in this string. Returns -1 if not matched.
indexOf(s, fromIndex)	Returns the index of the first occurrence of string s in this string after fromIndex. Returns -1 if not matched.
lastIndexOf(ch)	Returns the index of the last occurrence of ch in the string. Returns -1 if not matched.
lastIndexOf(ch, fromIndex)	Returns the index of the last occurrence of ch before fromIndex in this string. Returns -1 if not matched.
lastIndexOf(s)	Returns the index of the last occurrence of string s. Returns -1 if not matched.
lastIndexOf(s, fromIndex)	Returns the index of the last occurrence of string s before fromIndex. Returns -1 if not matched.

The first method is indexOf(ch)----->Misprinted as index(ch)

Example

```
public class Main
{
  public static void main (String[]args)
  {
     String s = "Welcome to Java";
     System.out.println (s.indexOf ('W'));// returns 0.
     System.out.println (s.indexOf ('o'));  // returns 4.
     System.out.println (s.indexOf ('o', 5));              // returns 9.
     System.out.println (s.indexOf ("come"));          // returns 3.
     System.out.println (s.indexOf ("Java", 5));       // returns 11.
     System.out.println (s.indexOf ("java", 5));       // returns -1.
     System.out.println (s.lastIndexOf ('W'));         // returns 0.
     System.out.println (s.lastIndexOf ('o'));         // returns 9.
     System.out.println (s.lastIndexOf ('o', 5));      // returns 4.
     System.out.println (s.lastIndexOf ("come"));// returns 3.
     System.out.println (s.lastIndexOf ("Java", 5));// returns -1.
     System.out.println (s.lastIndexOf ("Java"));      // returns 11.
  }
}
```

Extracting a substring from a given string

- substring(): used to extract a part of a string.

public String substring (int start_index)

public String substring (int start_index, int end_index)

Example: String s = “ABCDEFG”;

String t = s.substring(2); System.out.println (t);//CDEFG

String u = s.substring (1, 4); System.out.println (u);//BCD

Note: Substring from start_index to end_index-1 will be returned.

replace( ): The replace( ) method has two forms.

- The first replaces all occurrences of one character in the invoking string with another character. It has the following general form:

String replace(char original, char replacement)

- Here, original specifies the character to be replaced by the character specified by replacement.

Example: String s = "Hello".replace('l', 'w');//All occurances of l will be replaced with w and s will take reference of object with value:Hewwo

- The second form of replace( ) replaces one character sequence with another. It has this general form:

String replace(CharSequence original, CharSequence replacement)

Example:

String s = "This is java class".replace("java", "Python"); System.out.println(s);

Output: This is Python class

Q1(Output)??

```
import java.util.Scanner;
public class Main
{
  public static void main (String[]args)
  {
    String s=" Test ";
    System.out.print(s.length()+”,”);
    String s1=s.trim();
    System.out.print(s1.length());
  }
}
```

A. 6 6
B. 6 4
C. 4 4
D. 6 5

Q2(Output??)

```
import java.util.Scanner;
public class Main
{
  public static void main (String[]args)
  {
    String s1="Polling";
    String s2="Question";
    String s3=s1.concat(s2);
    System.out.println(s3.charAt(8));
  }
}
```

A. Q
B. u
C. Runtime error
D. g

Q3(Output??)

```
import java.util.Scanner;
public class Main
{
  public static void main (String[]args)
  {
    String s1="Hello";
    String s2="Halogen";
    System.out.println(s1.compareTo(s2));
  }
}
```

A. 5
B. 4
C. -4
D. 0

Q4(Output??)

```
public class Main
{
        public static void main(String[] args) {
                String s1="TESTING";
                String s2="testing";
System.out.println(s1.compareToIgnoreCase(s2))
;
        }
}
```

A. 0
B. -1
C. 1
D. false

Q5(Output??)

```
public class Main
{
        public static void main(String[] args) {
                String s1="This is the test phase";

        System.out.println(s1.lastIndexOf('t',11));
        }
}
```

A. 8
B. 12
C. -1
D. 7

Q6(Output??)

```
public class Main
{
        public static void main(String[] args) {
                String s1="Best among the Best";
        System.out.println(s1.indexOf("Best"));
        }
}
```

A. 0
B. 15
C. -1
D. Error

Q7(Output??)

```
public class Main
{
            public static void main(String[] args)
{
                        String s1="Programming
Skills";
            System.out.println(s1.substring(3,7))
;
            }
}
```

A. grammin
B. gram
C. gramm
D. ogram

Programming in Java

Topic: StringBuilder Class

Introduction

- Java StringBuilder class is used to create mutable (modifiable) string.
- We can add, insert or append new contents into a string builder, whereas the value of a String object is fixed, once the string is created.
- It is available since JDK 1.5.

StringBuilder Constructors

Following are some of the constructors defined for StringBuilder class:

- StringBuilder( )

  It creates an empty string Builder with the initial capacity of 16.
- StringBuilder(int length)

  It creates an empty string Builder with the specified capacity as length.
- StringBuilder(String str)

  It creates a string Builder with the specified string.

- The default constructor reserves room for 16 characters without reallocation.

Some Examples

```
StringBuilder sb = new StringBuilder();
System.out.println(sb.capacity());//Default capacity:16

StringBuilder sb = new StringBuilder(65);
System.out.println(sb.capacity());//Specified capacity:65

StringBuilder sb = new StringBuilder("A");
System.out.println(sb.capacity());
//Capacity: Default+No. of characters in the string, i.e. 16+1=17

StringBuilder sb = new StringBuilder('A');
System.out.println(sb.capacity());//Capacity:65[ASCII code of 'A']
```

StringBuilder Methods

length( ) and capacity( )

The current length of a StringBuilder can be found via the length( ) method, while the total allocated capacity can be found through the capacity( ) method.

int length( )

int capacity( )

The capacity() is the number of characters it is able to store without having to increase its size

The length() method returns the number of characters actually stored in the string builder

Example:

```
class StringBuilderDemo {
        public static void main(String args[]) {
                StringBuilder sb = new StringBuilder(“New Zealand");
                System.out.println("length = " + sb.length());//11
                System.out.println("capacity = " + sb.capacity());//27[16+11]
            }
        }
```

Important

- When the length of StringBuilder becomes larger than the capacity then memory reallocation is done:
- In case of StringBuilder, reallocation of memory is done using the following rule:
- If the new demand is exceeding the current capacity then new capacity will be:
- new_capacity = 2*(original_capacity + 1)
- If new_capacity can accommodate new demand, then it will remain as it is, otherwise new_capacity value will be set to the value of new demand.

```
public class StringBuilderCapacityExample3 {
  public static void main(String[] args) {
    StringBuilder sb=new StringBuilder();
    System.out.println(sb.capacity());//default 16
    sb.append("Hello");    5 characters took the space
    System.out.println(sb.capacity());//now 16   [Capacity will not change]
    sb.append("java is my favourite language");//After appending the current capacity will be exceeded, so reallocation will be performed
    System.out.println(sb.capacity());//now 2*(16+1)=34
    sb.append("string");   //It also exceeds the current capacity, so reallocation will be performed
    System.out.println(sb.capacity());//now 2*(34+1)=70
  }
}
```

ensureCapacity( )

- If we want to preallocate room for a certain number of characters after a StringBuilder has been constructed, we can use ensureCapacity( ) to set the size of the buffer.

- This is useful if we know in advance that we will be appending a large number of small strings to a StringBuilder.

void ensureCapacity(int capacity)

Example: ensureCapacity()

```
public class Main
{
        public static void main(String[] args) {
        StringBuilder sb=new StringBuilder(12);
  System.out.println(sb.capacity());//12
  sb.ensureCapacity(18);
  System.out.println(sb.capacity());//2*(12+1)=26
        }
}
```

setLength( )

- used to set the length of the buffer within a StringBuilder object.

  void setLength(int length)

- Here, length specifies the length of the buffer.

- When we increase the size of the buffer, null characters are added to the end of the existing buffer.

- If the new length is less than the current length of the string builder, then string builder is truncated to contain exactly the number of characters given in the new length.

```
public class Main
{
        public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Hello");
        System.out.println(sb.length());
        sb.setLength(2);
    System.out.println("New length is:"+sb.length()+" with content:"+sb);
        }
}
```

Output:

5

New length is:2 with content He

charAt( ) and setCharAt( )

- The value of a single character can be obtained from a StringBuilder via the charAt( ) method.

- We can set the value of a character within a StringBuilder using setCharAt( ).

char charAt(int index)

void setCharAt(int index, char ch

Example:

```
public class Main
{
        public static void main(String[] args) {
        StringBuilder str = new StringBuilder("Welcome");
        System.out.println("String = " + str);
        // set char at index 2 to 'L'
        str.setCharAt(2, 'L');
        // print string
        System.out.println("After setCharAt() String = "+ str); //WeLcome
        System.out.println(str.charAt(0));//W
                }
}
```

getChars( )

- getChars( ) method is used to copy a substring of a StringBuilder into an array.
- void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) method of StringBuilder class copies the characters starting at the given index:srcBegin to index:srcEnd-1 from String contained by StringBuilder into an array of char passed as parameter to function.

```
public class Main
{
		public static void main(String[] args) {
		StringBuilder str = new StringBuilder("WelcomeJava");
	char[] array = new char[7];
	str.getChars(0, 7, array, 0);
	System.out.print("Char array contains : ");
	for (int i = 0; i < array.length; i++) {
	   System.out.print(array[i] + " "); //W e l c o m e
	}
		}
}
```

append() Method

The StringBuilder append() method concatenates the given argument with this string.

```
class Example
{
public static void main(String args[])
{
StringBuilder sb=new StringBuilder("Hello ");
sb.append("Java");//now original string is changed
System.out.println(sb);//prints Hello Java
}
}
```

insert Method

The StringBuilder insert() method inserts the given string with this string at the given position.

```
class Example
{
public static void main(String args[])
{
StringBuilder sb=new StringBuilder("Hello ");
sb.insert(1,"Java");//now original string is changed
System.out.println(sb);//prints HJavaello
}
}
```

replace Method

The StringBuilder replace() method replaces the given string from the specified beginIndex and endIndex.

StringBuilder replace(int startIndex, int endIndex, String str)

Thus, the substring at startIndex through endIndex–1 is replaced.

```
class Example
{
public static void main(String args[])
{
StringBuilder sb=new StringBuilder("Hello");
sb.replace(1,3,"Java");
System.out.println(sb);//prints HJavalo
}
}
```

delete( ) and deleteCharAt( )

The delete() method of StringBuilder class deletes the string from the specified beginIndex to endIndex.

StringBuilder delete(int startIndex, int endIndex)
StringBuilder deleteCharAt(int index)

- The delete( ) method deletes a sequence of characters from the invoking object (from startIndex to endIndex-1).
- The deleteCharAt( ) method deletes the character at the specified index.
- It returns the resulting StringBuilder

Example

```
public class Main
{
        public static void main(String[] args) {
StringBuilder sb = new StringBuilder("WelcomeJava");
        sb.delete(3, 7);
        System.out.println("After delete: " + sb);//WelJava
        sb.deleteCharAt(2);
System.out.println("After deleteCharAt: " + sb);//WeJava
        }
}
```

substring( )

- Used to obtain a portion of a StringBuilder by calling substring( ).

String substring(int startIndex)
String substring(int startIndex, int endIndex)

- The first form returns the substring that starts at startIndex and runs to the end of the invoking StringBuilder object.

- The second form returns the substring that starts at startIndex and runs through endIndex–1

Example:

```
public class Main
{
        public static void main(String[] args) {
         StringBuilder str = new StringBuilder("WelcomeJava");
    System.out.println("SubSequence = " + str.substring(7)); //Java
    System.out.println("SubSequence = " + str.substring(0,7)); //Welcome
        }
}
```

reverse() Method

The reverse() method of StringBuilder class reverses the current string.

```
class Example
{
public static void main(String args[])
{
StringBuilder sb=new StringBuilder("Hello");
sb.reverse();
System.out.println(sb);//prints olleH
}
}
```

Q1(Output)

```
public class Main
{
        public static void main(String[] args) {
         StringBuilder sb=new StringBuilder(2);
         sb.append("Exam");
         System.out.println(sb.capacity()+" "+sb.length());
        }
}
```

A.2   2

B.4   2

C.6   4

D.2   4

Q2(Output)

```
public class Main
{
        public static void main(String[] args) {
         StringBuilder sb = new StringBuilder("Programming");
         sb.setLength(7);
   System.out.println(sb.length()+” “+sb);

        }
}
```

A.7   Programming

B.11 Program

C.7 Program

D.11 Programming

Q3(Output)

```
public class Main
{
		public static void main(String[] args) {
		 StringBuilder str = new StringBuilder("Evaluation");
	System.out.println(str.substring(1));
		}
}
```

A.v

B.valuation

C.va

D.valuatio

Q4(Output)

```
public class Main
{
        public static void main(String[] args) {
    StringBuilder str = new StringBuilder("Programming");
    char[] array = new char[5];
    str.getChars(0, 5, array, 0);
    System.out.print("Char array contains : ");
    for (int i = 0; i < array.length; i++) {
      System.out.print(array[i]);
    }
        }
}
```

A.Progr

B.Progra

C.Program

D.Programming

Q5(Output)

```
public class Main
{
        public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("PollingQuestion");
        sb.delete(1, 4);
        System.out.println(sb);
        }
}
```

A.PngQuestion

B.PingQuestion

C.Polling

D.Question

Q6(Output)

```
public class Main
{
        public static void main(String[] args) {
        StringBuilder sb=new StringBuilder("Object");
      sb.insert(6,"ive");
      System.out.println(sb);
        }
}
```

A.Objecivet

B.Objective

C.ive

D.Object

Wrapper classes

Wrapper Class

- Wrapper classes are classes that allow primitive types to be accessed as objects.

- Wrapper class in java provides the mechanism to convert *primitive into object* and *object into primitive*.

- Wrapper class is wrapper around a primitive data type because they "wrap" the primitive data type into an object of that class.

Wrapper Classes

- Each of Java's eight primitive data types has a class dedicated to it.
- They are one per primitive type: Boolean, Byte, Character, Double, Float, Integer, Long and Short.
- Wrapper classes make the primitive type data to act as objects.

Primitive Data Types and Wrapper Classes

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

Why Wrapper Class?

- Most of the objects collection store objects and not primitive types.
- Primitive types can be used as object when required.
- As they are objects, they can be stored in any of the collection and pass this collection as parameters to the methods.
- Wrapper classes are used to be able to use the primitive data-types as objects.
- Many utility methods are provided by wrapper classes.

To get these advantages we need to use wrapper classes.

Why Wrapper classes??

- They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
- The classes in java.util package handles only objects and hence wrapper classes help in this case also.
- Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.

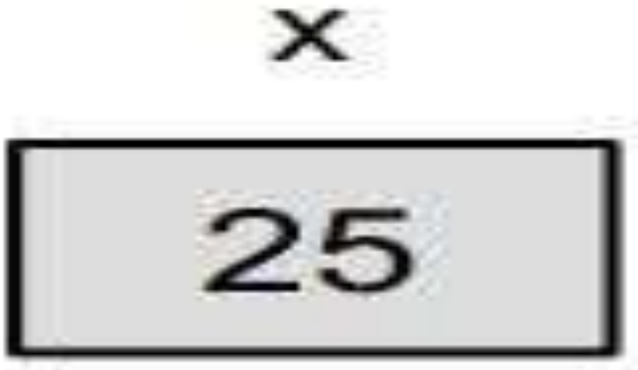
Difference b/w Primitive Data Type and Object of a Wrapper Class

- The following two statements illustrate the difference between a primitive data type and an object of a wrapper class:

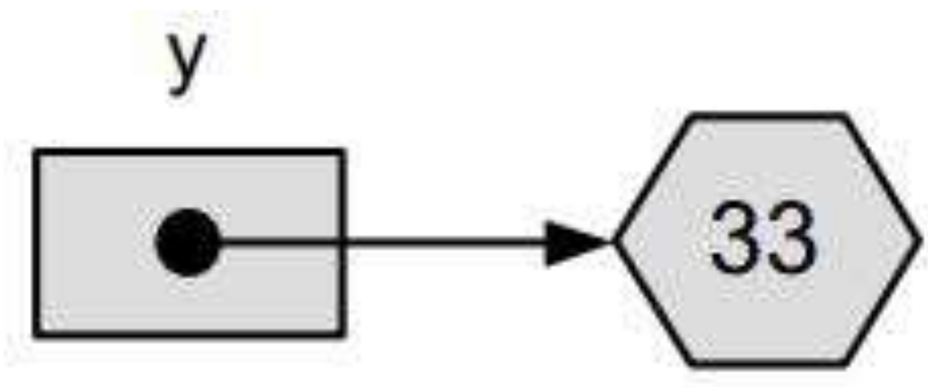
```
int x = 25;

Integer y = new Integer(33);
```

- The first statement declares an int variable named x and initializes it with the value 25.



- The second statement instantiates an Integer object. The object is initialized with the value 33 and a reference to the object is assigned to the object variable y.



- Clearly x and y differ by more than their values:

  x is a variable that holds a value;

  y is an object variable that holds a reference to an object.

Boxing and Unboxing

- The wrapping is done by the compiler.
- if we use a primitive where an object is expected, the compiler boxes the primitive in its wrapper class.
- Similarly, if we use a number object when a primitive is expected, the compiler un-boxes the object.

Example of boxing and unboxing:

- Integer x, y;   x = 12; y = 15;   System.out.println(x+y);
- When x and y are assigned integer values, the compiler boxes the integers because x and y are integer objects.
- In the println() statement, x and y are unboxed so that they can be added as integers.
- Boxing and unboxing can happen automatically, hence they are also known as AutoBoxing and Auto-UnBoxing

- **Autoboxing:** Converting a primitive value into an object of the corresponding wrapper_class is called autoboxing. For example, converting int to Integer_class.

The Java compiler applies autoboxing when a primitive value is:

- ➢ Passed as a parameter to a method that **expects an object** of the corresponding wrapper class.
- ➢ Assigned to a variable of the corresponding **wrapper class**.

- **Unboxing:** Converting an object of a wrapper type to its corresponding primitive value is called unboxing. For example conversion of Integer to int.

The Java compiler applies unboxing when an object of a wrapper class is:

- ➢ Passed as a parameter to a method that **expects a value** of the corresponding primitive type.
- ➢ Assigned to a variable of the corresponding **primitive type**.

Example

```
// Java program to illustrate the concept  of Autoboxing and Unboxing
class Example
{
	public static void main (String[] args)
	{
		// creating an Integer Object
		// with value 10.
		Integer i = new Integer(10);
		// unboxing the Object
		int i1 = i;
		System.out.println("Value of i: " + i);
		System.out.println("Value of i1: " + i1);
		//Autoboxing of char
		Character ch1 = 'a';
		// Auto-unboxing of Character
		char ch2 = ch1;
		System.out.println("Value of ch1: " + ch1);
		System.out.println("Value of ch2: " + ch2);


	}
}
```

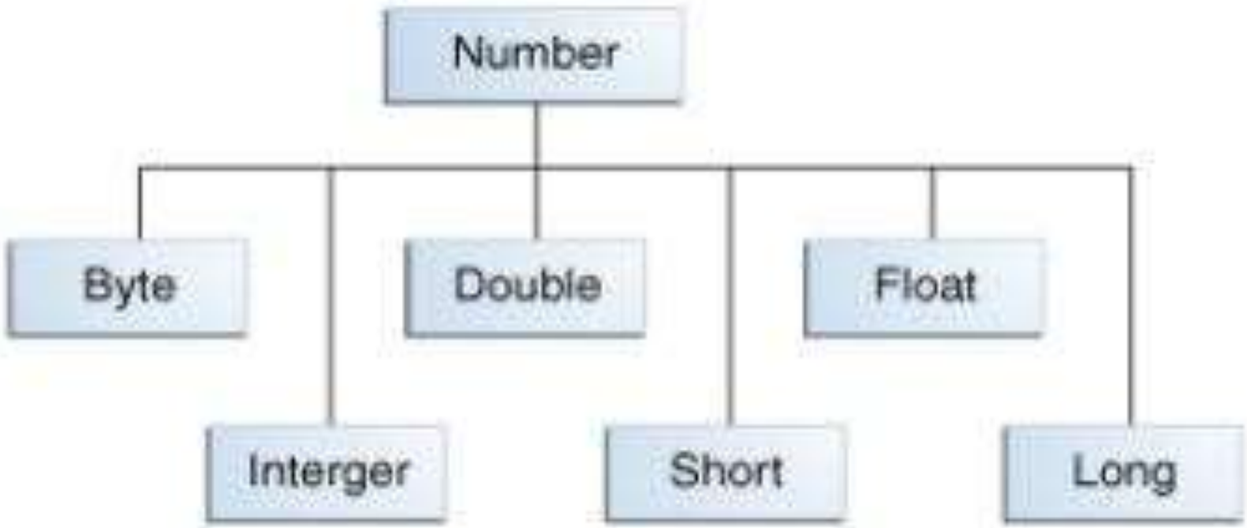
Advantages of Autoboxing / Unboxing:

- Autoboxing and unboxing lets developers write cleaner code, making it easier to read.
- The technique let us use primitive types and Wrapper class objects interchangeably and we do not need to perform any typecasting explicitly.

Numeric Wrapper Classes

- All of the numeric wrapper classes are subclasses of the abstract class Number .

- All of them implements Comparable



Example

```
//Java program to demonstrate typeValue() method
public class Test
{
		public static void main(String[] args)
		{
		// Creating a Double Class object with value "6.9685"
				Double d = new Double("6.9685");
		// Converting this Double(Number) object to
				// different primitive data types
				byte b = d.byteValue();
				short s = d.shortValue();
				int i = d.intValue();
				long l = d.longValue();
				float f = d.floatValue();
				double d1 = d.doubleValue();
System.out.println("value of d after converting it to byte : " +b);
System.out.println("value of d after converting it to short : " + s
		System.out.println("value of d after converting it to int : " + i);
		System.out.println("value of d after converting it to long : " + l);
		System.out.println("value of d after converting it to float : " + f);
		System.out.println("value of d after converting it to double : " + d1);
		}
}
```

Output values:

6

6

6

6

6.9685

6.9685

Features of Numeric Wrapper Classes

- All the numeric wrapper classes provide a method to convert a numeric *string into a primitive value*.

public static type parseType (String Number)

- parseInt()
- parseFloat()
- parseDouble()
- parseLong()

…

Example 1

```
//Java program to demonstrate Integer.parseInt() method
public class Test
{
        public static void main(String[] args)
        {
                // parsing different strings
                int z = Integer.parseInt("654",8);
                int a = Integer.parseInt("-FF", 16);
        long l = Long.parseLong("2158611234",10);
                System.out.println(z);
                System.out.println(a);
                System.out.println(l);

// run-time NumberFormatException will occur here
                // “Hello" is not a parsable string
                int x = Integer.parseInt(“Hello",8);
// run-time NumberFormatException will occur here
        // (for octal(8),allowed digits are [0-7])
                int y = Integer.parseInt("99",8);


        }
}
```

```
428
-255
2158611234
Exception in thread "main"
java.lang.NumberFormatException:
For input string: Hello"
```

Example 2

```
//Java program to demonstrate Integer.parseInt() method
public class Test
{
		public static void main(String[] args)
		{
			// parsing different strings
			int z = Integer.parseInt("654");
		long l = Long.parseLong("2158611234");

			System.out.println(z);
			System.out.println(l);

// run-time NumberFormatException will occur here
			// “Hello" is not a parsable string
			int x = Integer.parseInt(“Hello");

// run-time NumberFormatException will occur here
		// (for decimal(10),allowed digits are [0-9])
			int a = Integer.parseInt("-FF");

		}
}
```

Output:

654

2158611234

Exception in thread "main" java.lang.NumberFormatException : For input string: “Hello"

Features of Numeric Wrapper Classes

- All the wrapper classes provide a static method toString to provide the *string representation of the primitive values.*

public static String toString (type value)

Example:

public static String toString (int a)

```java
// Java program to illustrate toString()
class Test {
public static void main(String[] args)
        {
Integer I = new Integer(10);
        String s = I.toString();
        System.out.println(s);
        }
}
```

```java
// Java program to illustrate toString()
class Test {
    public static void main(String[] args)
    {
            String s = Integer.toString(10);
            System.out.println(s);
            String s1 = Character.toString('a');
            System.out.println(s1);
    }
}
```

Features of Numeric Wrapper Classes

- All numeric wrapper classes have a static method valueOf, which is used to *create a new object initialized to the value* represented by the specified string.

  public static DataType valueOf (String s)

Example:

Integer i = Integer.valueOf (“135”);

Double d = Double.valueOf (“13.5”);

Example 1

```
// Java program to illustrate valueof()

class Test {
        public static void main(String[] args)
        {
                Integer I = Integer.valueOf("10");
                System.out.println(I);
                Double D = Double.valueOf("10.0");
                System.out.println(D);
                Boolean B = Boolean.valueOf("true");
                System.out.println(B);

                // Here we will get RuntimeException
                Integer I1 = Integer.valueOf("ten");
        }
}
```

Example 2

```
// Java program to illustrate valueof()

class Test {
        public static void main(String[] args)
        {
                Integer I = Integer.valueOf(10);
                Double D = Double.valueOf(10.5);
                Character C = Character.valueOf('a');
                System.out.println(I);
                System.out.println(D);
                System.out.println(C);
        }
}
```

Methods implemented by subclasses of Number

- Compares this Number object to the argument.

  int compareTo(Byte anotherByte)
  int compareTo(Double anotherDouble)
  int compareTo(Float anotherFloat)
  int compareTo(Integer anotherInteger)
  int compareTo(Long anotherLong)
  int compareTo(Short anotherShort)

- returns int

Methods implemented by subclasses of Number

boolean equals(Object obj)

- Determines whether this number object is equal to the argument.
- The methods return true if the argument is not null and is an object of the same type and with the same numeric value.

Example

```
//Java program to demonstrate compareTo() method
public class Test
{
        public static void main(String[] args)
        {
// creating an Integer Class object with value "10"
        Integer i = new Integer("10");

        // comparing value of i
                System.out.println(i.compareTo(7));
                System.out.println(i.compareTo(11));
                System.out.println(i.compareTo(10));
        }
}
```

Output:

1

-1

0

Example

```
//Java program to demonstrate equals() method
public class Test
{
        public static void main(String[] args)
        {
// creating a Short Class object with value "15"
                Short s = new Short("15");
// creating a Short Class object with value "10"
                Short x = 10;
// creating an Integer Class object with value "15"
                Integer y = 15;
// creating another Short Class object with value "15"
                Short z = 15;
//comparing s with other objects
                System.out.println(s.equals(x));
                System.out.println(s.equals(y));
                System.out.println(s.equals(z));
        }
}
```

Output:

false

false

true

Character Class

- Character is a wrapper around a char.
- The constructor for Character is :

  Character(char ch)

  Here, ch specifies the character that will be wrapped by the Character object being created.

- To obtain the char value contained in a Character object, call charValue( ), shown here:

  char charValue( );

-

Boolean Class

- Boolean is a wrapper around boolean values.

- It defines these constructors:

  Boolean(boolean boolValue)

  Boolean(String boolString)

- In the first version, boolValue must be either true or false.

- In the second version, if boolString contains the string “true” (in uppercase or lowercase i.e TrUE, trUE), then the new Boolean object will be true. Otherwise, it will be false.

- To obtain a boolean value from a Boolean object, use booleanValue( ), shown here:

    boolean booleanValue( )

- It returns the boolean equivalent of the invoking object.

ARRAY LIST

ArrayList

ArrayList is a part of collection framework and is present in java.util package. It provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed. This class is found in java.util package.

Basic Example

```
import java.util.*;
class ArrayListExample {
        public static void main(String[] args)
        {
                int n = 5;
                ArrayList<Integer> arrli = new ArrayList<Integer>(n);
                for (int i = 1; i <= n; i++)
                arrli.add(i);
                System.out.println(arrli);
                arrli.remove(3);
                System.out.println(arrli);
                for (int i = 0; i < arrli.size(); i++)
                System.out.print(arrli.get(i) + " ");
        }
}
```

Various methods of ArrayList

Adding Elements: In order to add an element to an ArrayList, we can use the add() method. This method is overloaded to perform multiple operations based on different parameters. They are:

add(Object): This method is used to add an element at the end of the ArrayList.

add(int index, Object): This method is used to add an element at a specific index in the ArrayList.

Example of add Elements in ArrayList

```
// Java program to add elements
// to an ArrayList

import java.util.*;
public class ABC {
        public static void main(String args[])
        {
                ArrayList<String> al = new ArrayList<>();
                al.add("Welcome");
                al.add("Java");
                al.add(1, "to");
                System.out.println(al);
        }
}
```

Changing Elements: After adding the elements, if we wish to change the element, it can be done using the set() method. Since an ArrayList is indexed, the element which we wish to change is referenced by the index of the element. Therefore, this method takes an index and the updated element which needs to be inserted at that index.

```java
// Java program to change elements
// in an ArrayList
import java.util.*;
public class ABC {
		public static void main(String args[])
		{
				ArrayList<String> al = new ArrayList<>();

				al.add("welcome");
				al.add("java");
				al.add(1, "to");

				System.out.println("Initial ArrayList " + al);

				al.set(1, "the");

				System.out.println("Updated ArrayList " + al);
		}
}
```

Removing Elements: In order to remove an element from an ArrayList, we can use the remove() method. This method is overloaded to perform multiple operations based on different parameters. They are:

remove(Object): This method is used to simply remove an object from the ArrayList. If there are multiple such objects, then the first occurrence of the object is removed.

remove(int index): Since an ArrayList is indexed, this method takes an integer value which simply removes the element present at that specific index in the ArrayList. After removing the element, all the elements are moved to the left to fill the space and the indices of the objects are updated.

```
// Java program to remove elements
// in an ArrayList

import java.util.*;
public class ABC {

        public static void main(String args[])
        {
                ArrayList<String> al = new ArrayList<>();

                al.add("Welcome");
                al.add("java");
                al.add(1, "to");
                System.out.println("Initial ArrayList " + al); // Welcome to java
                al.remove(1);
                System.out.println("After the Index Removal " + al); //Welcome java
                al.remove("java");
                System.out.println("After the Object Removal " + al); //Welcome
        }
}
```

Iterating the ArrayList: There are multiple ways to iterate through the ArrayList. The most famous ways are by using the basic for loop in combination with a get() method to get the element at a specific index and the advanced for loop.

```
import java.util.*;
public class ABC {
	public static void main(String args[])
	{
		ArrayList<String> al = new ArrayList<>();
		al.add("Welcome");
		al.add("java");
		al.add(1, "to");

		for (int i = 0; i < al.size(); i++) {

			System.out.print(al.get(i) + " ");
		}

		System.out.println();
		for (String str : al)
			System.out.print(str + " ");
	}
}
```

Important Features

- ArrayList inherits AbstractList class and implements List interface.
- ArrayList is initialized by the size. However, the size is increased automatically if the collection grows or shrinks if the objects are removed from the collection.
- Java ArrayList allows us to randomly access the list.
- ArrayList can not be used for primitive types, like int, char, etc. We need a wrapper class for such cases.
- ArrayList in Java can be seen as a vector in C++.
- ArrayList is not Synchronized. Its equivalent synchronized class in Java is Vector.

clear method()

The clear() method of ArrayList in Java is used to remove all the elements from a list. The list will be empty after this call returns.

Example:

```
import java.util.ArrayList;
public class ABC {
        public static void main(String[] args)
        {
                ArrayList<Integer> arr = new ArrayList<Integer>(4);
                arr.add(1);
                arr.add(2);
                arr.add(3);
                arr.add(4);
                System.out.println("The list initially: " + arr);
                arr.clear();
                System.out.println("The list after using clear() method: " + arr); //[]
        }
}
```

contains() Method

ArrayList contains() method in Java is used for checking if the specified element exists in the given list or not.

Example:

```
import java.util.ArrayList;
class ABC {
        public static void main(String[] args)
        {
                ArrayList<Integer> arr = new ArrayList<Integer>(4);
                arr.add(1);
                arr.add(2);
                arr.add(3);
                arr.add(4);
                boolean ans = arr.contains(2);
                if (ans)
                        System.out.println("The list contains 2");
                else
                        System.out.println("The list does not contains 2");
        }
}
```

Some other important Methods

int indexOf(Object o)
Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain the element.

int lastIndexOf(Object o)
Returns the index in this list of the last occurrence of the specified element, or -1.

void ensureCapacity(int minCapacity)
Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.

boolean isEmpty()
used to check whether the Arraylist is empty or not?

trimToSize() Method

The **trimToSize()** method of **ArrayList** in Java trims the capacity of an ArrayList instance to be the list’s current size. This method is used to trim an ArrayList instance to the number of elements it contains.