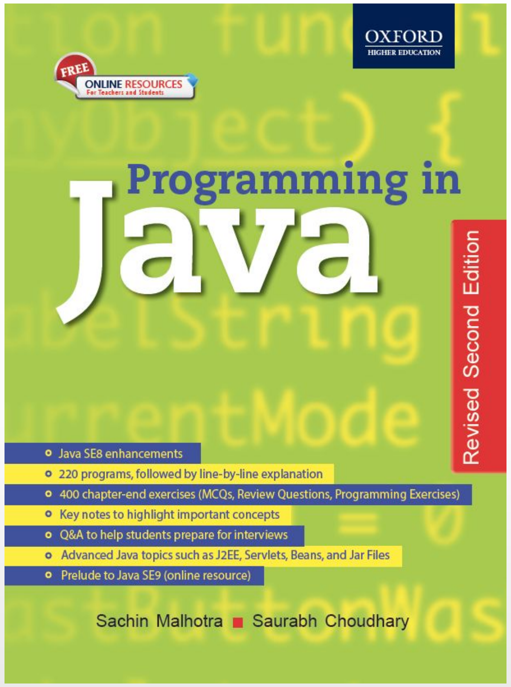
OXFORD
UNIVERSITY PRESS

Programming in Java

Revised 2nd Edition

Sachin Malhotra & Saurabh Choudhary


OXFORD HIGHER EDUCATION
FREE ONLINE RESOURCES For Teachers and Students
Programming in Java
Revised Second Edition
Java SE8 enhancements
220 programs, followed by line-by-line explanation
400 chapter-end exercises (MCQs, Review Questions, Programming Exercises)
Key notes to highlight important concepts
Q&A to help students prepare for interviews
Advanced Java topics such as J2EE, Servlets, Beans, and Jar Files
Prelude to Java SE9 (online resource)
Sachin Malhotra
Saurabh Choudhary

Chapter 4

Classes and Objects

Objectives

- Know how classes and objects are created and applied in Java
- Know how methods are created and used
- Understand polymorphism concepts: Overloading
- Understand what is a Constructor
- Establish familiarity with *static* and *this* keyword
- Know about arrays and command line arguments
- Understand inner classes

Classes and Objects

- A class is a blueprint or prototype that defines the variables and methods common to all objects of a certain kind.
- Class can be thought of as a user-defined data type and an object as a variable of that data type, which can contain data and methods i.e. functions, working on that data.
- All object instances have their own copies of instance variable.
- Each object has its own copy of instance variables which is different from other objects created out of the same class.

How to Declare Class in JAVA

- A class is declared using *class* keyword followed by the name of the class.
- Syntax of declaration of class

```
class classname {
    //Variables declaration
    //Methods declaration
    }
```

- Example

```
class GoodbyeWorld {
public static void main (String args[]) {
    System.out.println("Goodbye World!");
}}
```

Declaring Classes in Java

- Class declaration can specify more about the class, like you can:

❑ declare the class's superclass (discussed in chapter 5)

❑ list the interfaces implemented by the class (discussed in chapter 6)

❑ declare whether the class is public, abstract (discussed in chapter 6), or final (discussed in chapter 5)

- Summary of class declaration syntax as;

```
[modifiers] class ClassName [ extends SuperClassName]
        [ implements  InterfaceNames ] {
............
}
```

Declaring Classes in Java

- Items between [and] are optional. A class declaration defines the following aspects of the class:
  - ❑ *modifiers* declare whether the class is public, protected, default, abstract, or final
  - ❑ *ClassName* sets the name of the class you are declaring
  - ❑ *SuperClassName* is the name of *ClassName*'s superclass
  - ❑ *InterfaceNames* is a comma-delimited list of the interfaces implemented by *ClassName*
- Java compiler assumes the class to be non-final, non-public, non-abstract, subclass of **Object** (chapter 6) that implements no interfaces if no explicit declaration is specified.

Parts of a Class

- Class contains two different sections: variable declarations and method declaration. The variables of a class describe its state and methods describe its behavior.

```
classDeclaration {

memberVariableDeclarations

methodDeclarations

}
```

Example

- Class SalesTaxCalculator for calculating and displaying the tax

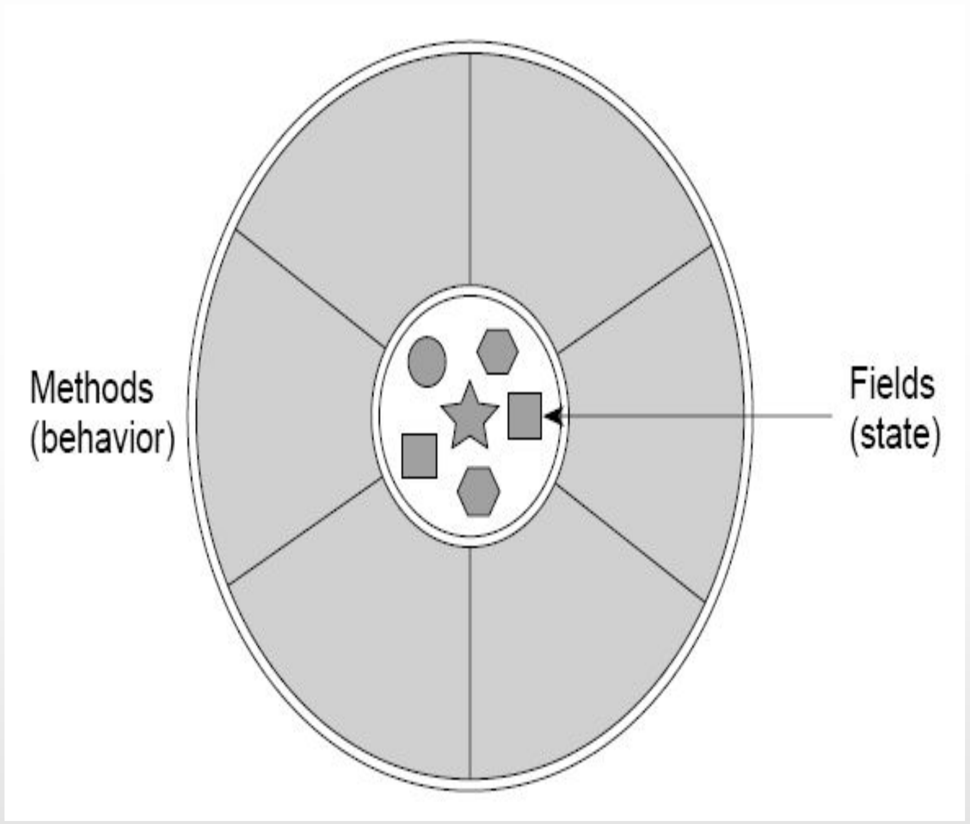
```
class SalesTaxCalculator {
   float amount=100.0f;
   float taxRate=10.2f;
   void calculateTax() {
         float taxAmt = amount*taxRate/100;
         System.out.println(taxAmt);
   }
}
```

Objects

- Real-world objects have *state* and *behavior*.
- For example,
  - horses have state (name, color, breed) and horses have behavior (barking, fetching, and slobbering on your newly cleaned slacks).
  - Bikes have state (gear, accelerator, two wheels, number of gears, brakes) and behavior (braking, accelerating, slowing down and changing gears).
- An object is a software bundle which encapsulates variables and methods, operating on those variables.

Why Should We Use Classes and Objects

Modularity and information hiding i.e. data encapsulation can be incorporated using an object, in software. Classes, being blueprints, provide the benefit of reusability.



Creating Objects

- Java object created with a statement like this one:

```
SalesTaxCalculator obj1 = new SalesTaxCalculator ( );
```

- This statement creates a new SalesTaxCalculator object.
- This single statement declares, instantiates, and initializes the object.

Declaring an Object

- Object declaration is same as variable declaration, e.g.

  - SalesTaxCalculator obj1;

- Generally the declaration is as follows:

  - type name;

  where *type* is the type of the object (i.e. class name) and *name* is the name of the reference variable used to refer to the object

- Difference between variables and objects:

  - A variable holds a single type of literal, while an
  - Object is a instance of a class with a set of instance variables and methods which performs certain tasks depending on what methods have been defined for

Initializing an Object

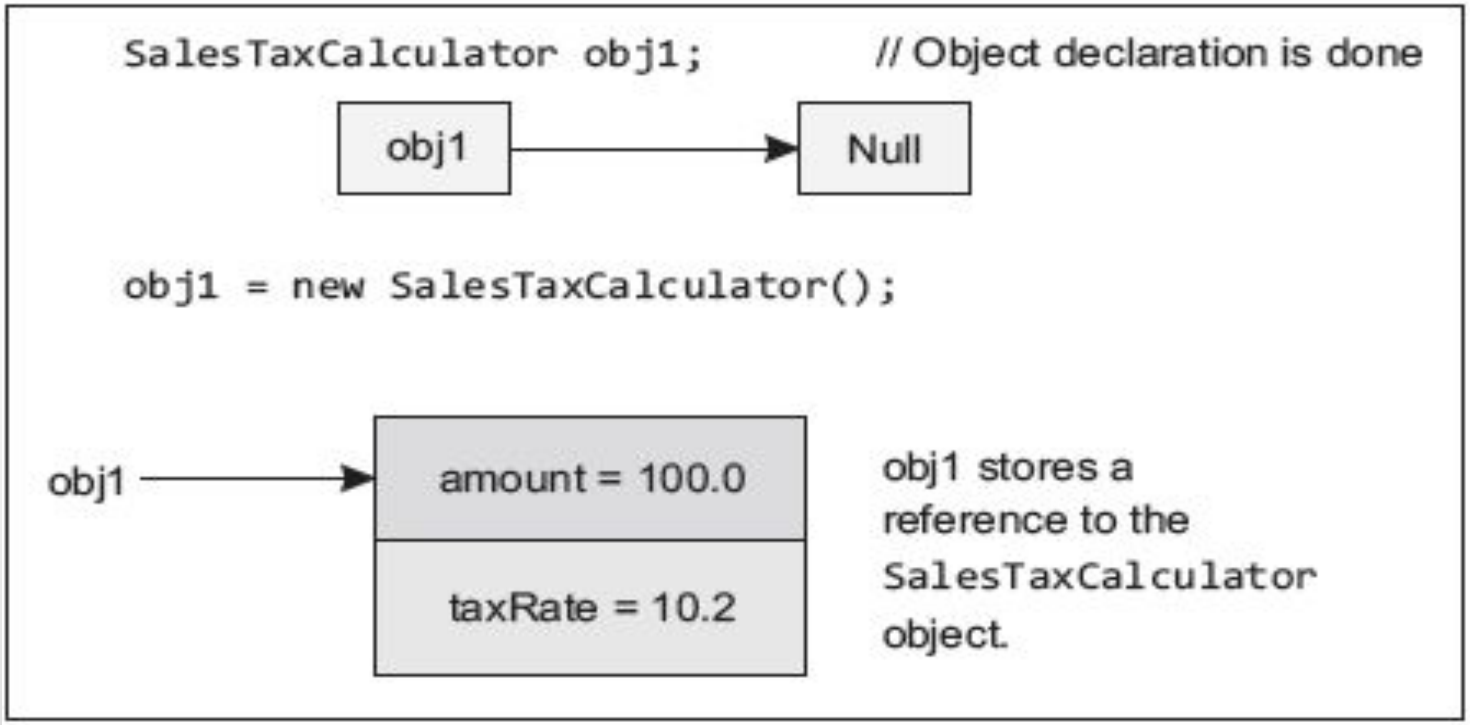
- By initializing an object we mean that the instance variables are assigned some values. This task is accomplished using a constructor.
- The final object creation can be said as complete when the objects are initialized, either with an implicit constructor or an explicit constructor. This object creation can be used in programming code in two ways:

```
SalesTaxCalculator obj1 = new SalesTaxCalculator ( );
```

- Here all the three operations - object declaration, object instantiation, and object initialization - are done by one statement only.

Initializing an Object

- The process actually takes place in following way:



SalesTaxCalculator obj1;
// Object declaration is done
obj1
Null
obj1 = new SalesTaxCalculator();
obj1
amount = 100.0
taxRate = 10.2
obj1 stores a reference to the SalesTaxCalculator object.

Fig. 4.3 Steps in Object Creation

Instance Variable

- A class can have many instances, each instance having its own set of variables. E.g.

```
class SalesTaxCalculator {
    float amount=100.0f; // instance variable
    float taxRate=10.2f; //instance variable
    void calculateTax() {
    float taxAmt = amount*taxRate/100;
    System.out.println(taxAmt);   }
    public static void main (String args[ ])     {
    SalesTaxCalculator obj1 = new SalesTaxCalculator();
    SalesTaxCalculator obj2 = new SalesTaxCalculator();
System.out.println("Amount in Object 1: "+ obj1.amount);
System.out.println("Tax Rate in Object 1: "+ obj1.taxRate);
System.out.println("Amount in Object 2: "+ obj2.amount);
System.out.println("Tax Rate in Object 2: "+ obj2.taxRate);
}}
```

Instance Variable

- Each variable declared inside a class and outside the methods is termed as instance variable **(except static variables)**, because they are created whenever an instance (object) of the class is created and these variables are initialized by the constructors.
- In the above example the two objects obj1 and obj2 will have their own set of instance variables, i.e. *obj1* will have its own *amount* and *taxRate* whereas *obj2* will have its own set of *amount* and *taxRate*.

Accessing Instance Variables

- For accessing value of an object
  - Objectname.variablename
- To assign values to the variables of an object
  - Objectname.variablename = value
- There are three ways of assigning values to the instance variables in the objects:
  - Assigning values directly to the instance variables as shown below:

    float amount=100.0f;
  - Assigning values through a setter method
  - Assigning values using constructors

Methods

- Similar to a function in any other programming language.
- None of the methods can be declared outside the class.
- Why use methods?

  - To make code reusable

  - To parameterize code

  - For top-down programming

  - To simplify code

Methods

- General syntax for a method declaration:

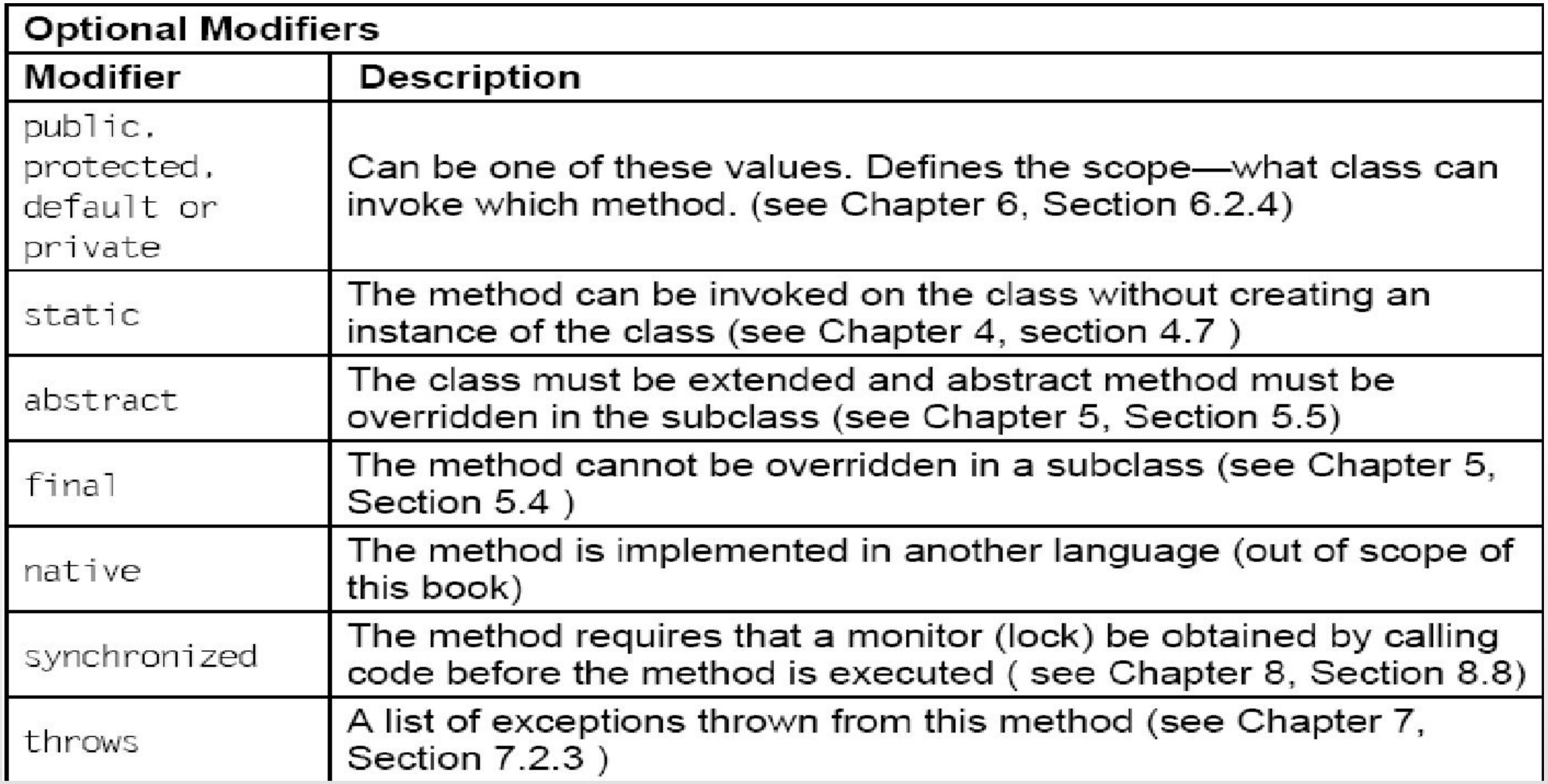
  [modifiers] return_type method_name (parameter_list) [throws_clause] { [statement_list]}

- The method declaration includes:

  Modifiers:

  -Optional. Can be, public, protected, default or private, static, abstract, final, native, synchronized, throws

Optional Modifiers Used with Methods



Optional Modifiers

Modifier	Description
`public`, `protected`, `default` or `private`	Can be one of these values. Defines the scope—what class can invoke which method. (see Chapter 6, Section 6.2.4)
`static`	The method can be invoked on the class without creating an instance of the class (see Chapter 4, section 4.7)
`abstract`	The class must be extended and abstract method must be overridden in the subclass (see Chapter 5, Section 5.5)
`final`	The method cannot be overridden in a subclass (see Chapter 5, Section 5.4)
`native`	The method is implemented in another language (out of scope of this book)
`synchronized`	The method requires that a monitor (lock) be obtained by calling code before the method is executed (see Chapter 8, Section 8.8)
`throws`	A list of exceptions thrown from this method (see Chapter 7, Section 7.2.3)

Methods

- Return Type:
  - Can be either void or if a value is returned, it can be either a primitive type or a class
  - If the method declares a return type, then before it exits it must have a return statement
- Method Name**:**
  - The method name must be a valid Java identifier
- Parameter List**:**
  - Contains zero or more type/identifier pairs make up the parameter list
  - Each parameter in parameter list is separated by a comma
- Curly Braces:
  - The method body is contained in a set of curly braces (opening ‘{‘ and closing ‘}’).

Method Example

```
class Circle {
   float pi = 3.14f;
   float radius;
   void setRadius(float rad)   {
         radius = rad;}
         float calculateArea()   {
         float area = pi* radius*radius;
         return (area);
}  }
```

Method Invocation

- Methods cannot run on their own; they need to be invoked by the objects they are a part of.
- When an object calls a method, it can pass on certain values to the methods (if methods accept them).
- Methods can also return values from themselves if they wish to.
- Data that are passed to a method are known as arguments or parameters.

Method Invocation

You must also know about different types of parameters:

- Formal Parameters: the identifier used in a method to stand for the value that is passed into the method by a caller.
- Actual Parameters: The actual value that is passed into the method by a caller.
- The number and type of the actual and formal parameters should be same for a method.
- In Java, all values are **passed by value**. This is unlike some other programming languages that allow pointers to memory addresses to be passed into methods.

Method Invocation

```
class CallMethod {
public static void main (String args[]) {
  float area1;
  Circle circleobj = new Circle();
  circleobj.setRadius(3.0f);
  area1 = circleobj.calculateArea();
  System.out.println("Area of Circle = " + area1);
 }}
```

Method Overloading

- In method overloading two methods can have the same name but different signature, i.e. different number or type of parameters.
- The concept is advantageous where same kind of activities are to be performed but with different input parameters.

Method Overloading Example

```
class OverloadDemo {
   void max(float a, float b) {
      …} //takes care of which number (float) is greater
   void max(int a, int b) {
      ...} //takes care of which number (int) is greater
public static void main(String args[]) {
   OverloadDemo o=new OverloadDemo();
   o.max(23L,12L);
   o.max(2,3);  } }
```

Constructors

- Java has a mechanism, known as constructor, for automatically initializing the values for an object, as soon as the object is created.
- Constructors have the same name as the class it resides in and is syntactically similar to a method.
- It is automatically called immediately after the object for the class is created by **new** operator.
- Constructors have no return type, not even void, as the implicit return type of a class' constructor is the class type itself.
- Types of constructors: Implicit/Default, Explicit, Parameterized.

Explicit Constructor Example

```
class  Room{
double length, breadth, height, volume;
Room( ) {
length = 14;
breadth = 12;
height = 10;  }
double volComp( ) {
   volume = length * breadth * height;
   return volume;      }
```

Example (contd.)

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

   Room r1 = new Room();

   Room r2 = new Room();

   System.out.println("The volume of the room is "+r1.volComp( ));

   System.out.println("The volume of the room is "+r2.volComp( ));

   }
```

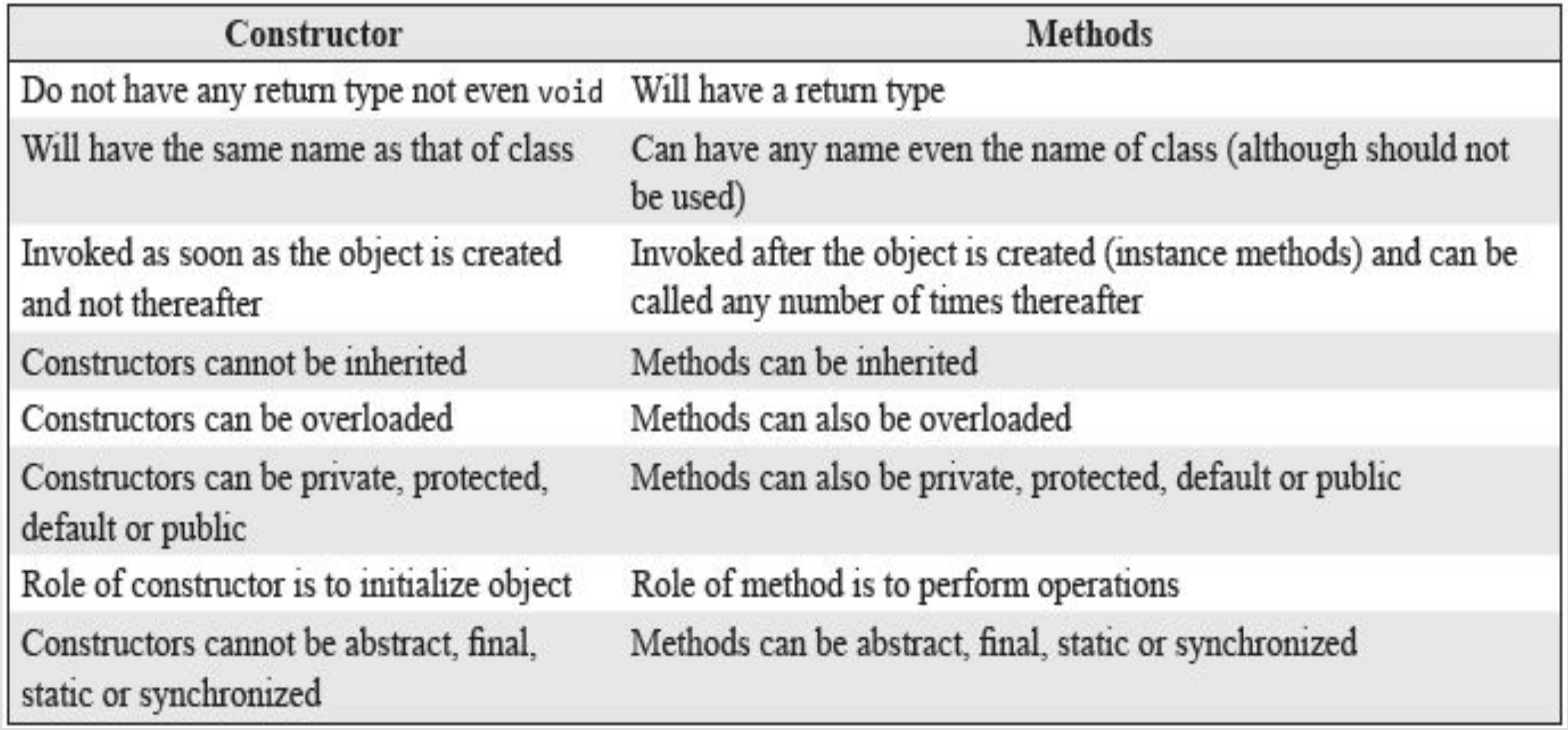
Parameterized Constructor Example

```
class  Room2{
double length, breadth, height, volume;
Room2(double l, double b, double h) {
length = l;
breadth = b;
height = h;   }
// Computation of volume of the room
double volComp( ) {
volume = length * breadth * height;
return volume; }
```

Example (contd.)

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

Room2 r1 = new Room2(14, 12, 10 );

Room2 r2 = new Room2(16, 15, 11 );

System.out.println("The volume of the room is "+r1.volComp( ));

System.out.println("The volume of the room is "+r2.volComp( ));

} }
```

Constructor vs Methods



Constructor	Methods
Do not have any return type not even `void`	Will have a return type
Will have the same name as that of class	Can have any name even the name of class (although should not be used)
Invoked as soon as the object is created and not thereafter	Invoked after the object is created (instance methods) and can be called any number of times thereafter
Constructors cannot be inherited	Methods can be inherited
Constructors can be overloaded	Methods can also be overloaded
Constructors can be private, protected, default or public	Methods can also be private, protected, default or public
Role of constructor is to initialize object	Role of method is to perform operations
Constructors cannot be abstract, final, static or synchronized	Methods can be abstract, final, static or synchronized

Constructor Overloading

- Constructors for a class have the same name as the class but they can have different signature, i.e. different types of arguments or different number of arguments. Such constructors can be termed as overloaded constructors.
- In the example given on next slide, we have two different classes, *Rectangle* and *ConstOverloading.* Rectangle class has two constructors, both with same names but different signatures. Second class *ConstOverloading* has the *main( )* method inside it.

Constructor Overloading Example

```
class Rectangle{
int l, b;
Rectangle(){
   l = 10;
   b = 20;
}
Rectangle(int x, int y){
   l = x;
   b = y;
}
int area() {
   return l*b;
}}
```

Constructor Overloading Example

```
class ConstOverloading   {
public static void main(String args[]) {
  Rectangle rectangle1=new Rectangle();
  System.out.println("The area of a rectangle using first
  constructor is:     "+rectangle1.area());
  Rectangle rectangle2=new Rectangle(4,5);
  System.out.println("The area of a rectangle using
  second constructor is:   "+rectangle2.area();
} }
```

The Output

```
The area of a rectangle using first constructor is: 200
The area of a rectangle using second constructor is: 20
```

this keyword

- “this” keyword is used in an instance method to refer to the object that contains the method, i.e. it refers to the current object.
- Whenever and wherever a reference to an object of the current class type is required, ‘this’ can be used.
- It has two other usages:
  - Differentiating between instance variables and local variables
  - Constructor chaining

this keyword

- To differentiate between instance variables and local variables

```
Room2( double l, double b, double h) {
  this.length = l;
  this.breadth = b;
  this.height = h;
}
```

Constructor Chaining

- It means a constructor can be called from another constructor.

```
/*First Const.*/ Room2()
{
   // constructor chained
   this(14,12,10);
}
/*Second Const.*/ Room2( double l, double b, double h) {
   length = l;
   breadth = b;
   height = h;
}
```

Cleaning up Unused Objects

- Java allows a programmer to create as many objects as he/she wants but frees him/her from worrying about destroying them. The Java runtime environment deletes objects when it determines that they are no longer required.
- Java runtime environment has a garbage collector that periodically frees the memory used by the objects that are no longer needed.
- Before an object gets garbage collected, the garbage collector gives the object an opportunity to clean up itself through a call to the object's finalize() method. This process is known as finalization.

Garbage Collector

- Two basic approaches :
  - *Reference counting*
  - *Tracing*
- Reference counting maintains a reference count for every object.
- It is incremented and decremented as and when the references on objects increase or decrease (leave an object).
- When reference count for a particular object is 0, the object can be garbage collected.
- Tracing technique traces the entire set of objects (starting from root) and all objects having reference on them are marked in some way a.k.a *mark and sweep* garbage
- Objects that are not marked (not referenced) are assumed to be garbage and their memory is reclaimed.

Garbage Collector

- Mark and sweep collectors further use the techniques of compaction and copying for fragmentation problems that may arise once you sweep the unreferenced objects.
- Compaction moves all the live objects towards one end making the other end a large free space and copying techniques copy all live objects besides each other into a new space and the old space is considered free now.

Advantages

- Free from worrying about deallocation of memory
- Helps in ensuring integrity of programs
- No way by which Java programmers can knowingly or unknowingly free memory incorrectly

Disadvantages

- Overhead to keep track of which objects are being referenced by the executing program and which are not being referenced
- Overhead is also incurred on finalization and freeing memory of the unreferenced objects
- These activities will incur more CPU time than would have been incurred if the programmers would have explicitly deallocated memory

Your Turn

- What is method overloading?
- What is a constructor?
- Why do we overload constructors?
- Explain the use of *this* keyword?
- What is garbage collection?

static Keyword

- Different objects, variables, and methods will occupy different areas of memory when created/called. Sometimes we would like to have multiple objects, share variables or methods. The *static* keyword effectively does this for us.
- Static keyword can be applied to variables/methods and blocks of code.
- Java supports three types of variables: Local, Instance, and Class variables.
  - ❑ Local variables are declared inside a method, constructor, or a block of code
  - ❑ Instance variables are declared inside a class, but outside a method

static Keyword

- Class/static variables' declaration is preceded with a *static* keyword. They are also declared inside a class, but outside a method.

- The most important point about static variables is that there exists only a single copy of static variables per class.

- The effect of doing this is that when we create multiple objects of that class, every object shares the static variable i.e. there is only one copy of the variable declared as static. We can declare variables as static as:

```
static int var = 0;
```

Instance Variables vs. Class Variables

- All instances of the class share the *static* variables of the class.
- A class variable can be accessed directly with the class name, without the need to create an instance.
- Without the *static* keyword, it's called “instance variable”, and each instance of the class has its own copy of the variable.

Static Methods

- Like static variables, we do not need to create an object to call our static method. Simply using the class name will suffice.
- Static methods however can access static variables directly.
- Variables that have not been declared static cannot be accessed by the static method directly, i.e. the reason why we create an object of the class within the main (which is static) method to access the instance variables and call instance methods.
- To make a method static, we simply precede the method declaration with the static keyword.

Static Methods

```
static void aMethod(int param1) {
.....
.....
}
```

- When we declare a method as static, we are basically saying that there should only be one instance of this method within our program (e.g., as in the *main* method)
- Methods can also be declared with the static keyword.
  - static int computeArea(int length, int width) { }

Example

```
class Area {
static int area; // class variable
static int computeArea (int width, int height) {
area = width * height;
return area;
} }
class callArea {
public static void main(String args[]) {
System.out.println(Area.computeArea(4,3);
} }
```

Static Initialization Block

- A block of statements with static keyword applied to it.
- Used for initializing static or class variables.
- In case some logic is used for assigning values to the variables, static blocks can be used.
- The syntax for static block is as follows:

```
static{
...
}
```

Static Initialization Block

- The *static* executes as soon as the class loads even before the JVM executes the *main* method.
- There can be any number of static blocks within the class and they will be executed in the order in which they have appeared in the source code.

Instance Initialization Block

- In case the *static* keyword is dropped from this block, it becomes an instance initialization block.
- Actually the code of instance initialization block is placed in the <init> method, which is created for every constructor by the compiler, before the source code mentioned by programmer in the constructor.

Static and Instance Block Example

```
class StaticBlockDemo {
int x=10;
static {
int z=10; // local variable
System.out.println(“In static block”);          }
{
System.out.println(“In Instance Initialization block”);
System.out.println(“Printing Instance variable
Initializer value through Block: “ +x);
}
```

Example (contd.)

```
// Constructor
StaticBlockDemo(int y) {
System.out.println(“Within Constructor”);
System.out.println(“Instance variable printed
using constructor: “+x);
 x=y;
 System.out.println(“Instance variable
 initialized using constructor: “+x);
}
```

Example (contd.)

```
StaticBlockDemo()    {
System.out.println(“Within Constructor”);
 System.out.println(“Instance variable printed using
 constructor: “+x);     }
 public static void main(String[] args)   {
 System.out.println(“In main”);
 StaticBlockDemo st = new StaticBlockDemo(100);
 System.out.println(“--------------------------------------   ”);
 StaticBlockDemo st1 = new StaticBlockDemo();
  }   }
```

The Output

```
In static block
In main
In Instance Initialization block
Printing Instance variable Initializer value through Block: 10
Within Constructor
Instance variable printed using constructor: 10
Instance variable initialized using constructor: 100
----------------------------------------
In Instance Initialization block
Printing Instance variable Initializer value through Block: 10
Within Constructor
Instance variable printed using constructor: 10
```

Arrays

- There are situations where we might wish to store a group of similar type of values in a variable.
- Array is a memory space allocated, which can store multiple values of same data type, in contiguous locations.
- This memory space, which can be perceived to have many logical contiguous locations, can be accessed with a common name.

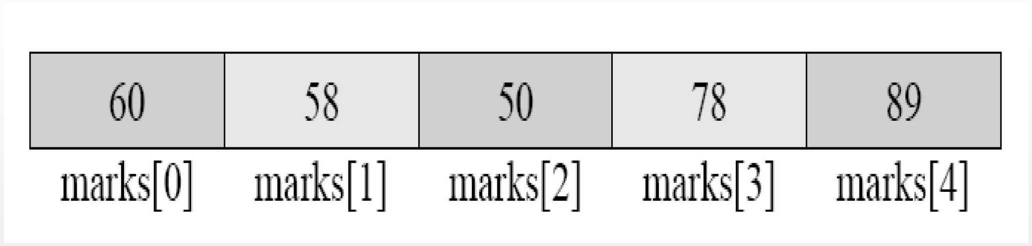
Arrays

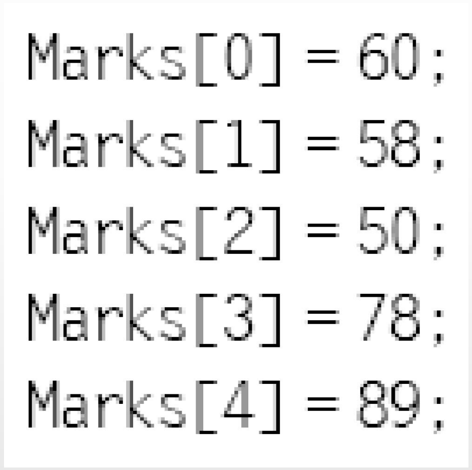
- A specific element in an array is accessed by the use of a subscript or index used in brackets, along with the name of the array.
    - For example, marks[5] would mean marks of 5th student.
- While the complete set of values is called an array, the individual values are known as *elements*.
- Arrays can be of two types:
    - One-dimensional array
    - Multi-dimensional array

1-D Arrays

- In one-dimensional arrays, a single subscript or index is used, where each index value refers to individual array element.
- Index starts from 0 and goes up to n-1, i.e. the first value of the array will have an index of 0 and the last value will have an index of n–1, where n is the number of elements in the array.
- So, if an array named *marks* has been declared to store the marks of 5 students, the computer reserves five contiguous locations in the memory, as shown in next slide.

1-D Arrays





```
Marks[0] = 60;
Marks[1] = 58;
Marks[2] = 50;
Marks[3] = 78;
Marks[4] = 89;
```

Creation of Arrays

- Creating an array, similar to an object creation, involves three steps:
  - Declaring an array
  - Creating memory locations
  - Initializing/assigning values to an array

Declaring an Array

- Declaring an array is same as declaring a normal variable except that you must use a set of square brackets with the variable type.
- There are two ways in which an array can be declared:
  - type arrayname[ ];
  - type[ ] arrayname;
- For example

  int marks[ ]; or

  int[ ] marks;

Creating Memory Locations

- An array is more complex than a normal variable, so we have to assign memory to the array when we declare it.
- You assign memory to an array by specifying its size. Interestingly, our same old *new* operator helps in doing the job, just as shown below:
  - Arrayname = new type [size];
- So, allocating space and size for the array named as **marks** can be done as,
  - marks = new int[5];

Initializing/Assigning Values to an Array

- Assignment of values to an array, which can also be termed as initialization of arrays, can be done as follows:
    - Arrayname[index] = value;
- The creation of list of marks to be assigned in array, named as *marks* has already been shown. Here is an example of how to set the values for an array of 5 marks.

Setting Values in An Array

```
public class Array {
public static void main(String[] args) {
int[] marks = new int[5];
marks[0] = 60;
marks[1] = 58;
marks[2] = 50;
marks[3] = 78;
marks[4] = 89;
} }
```

Alternatively you can also use:

```
int marks[] = {60, 58, 50, 78, 89};
```

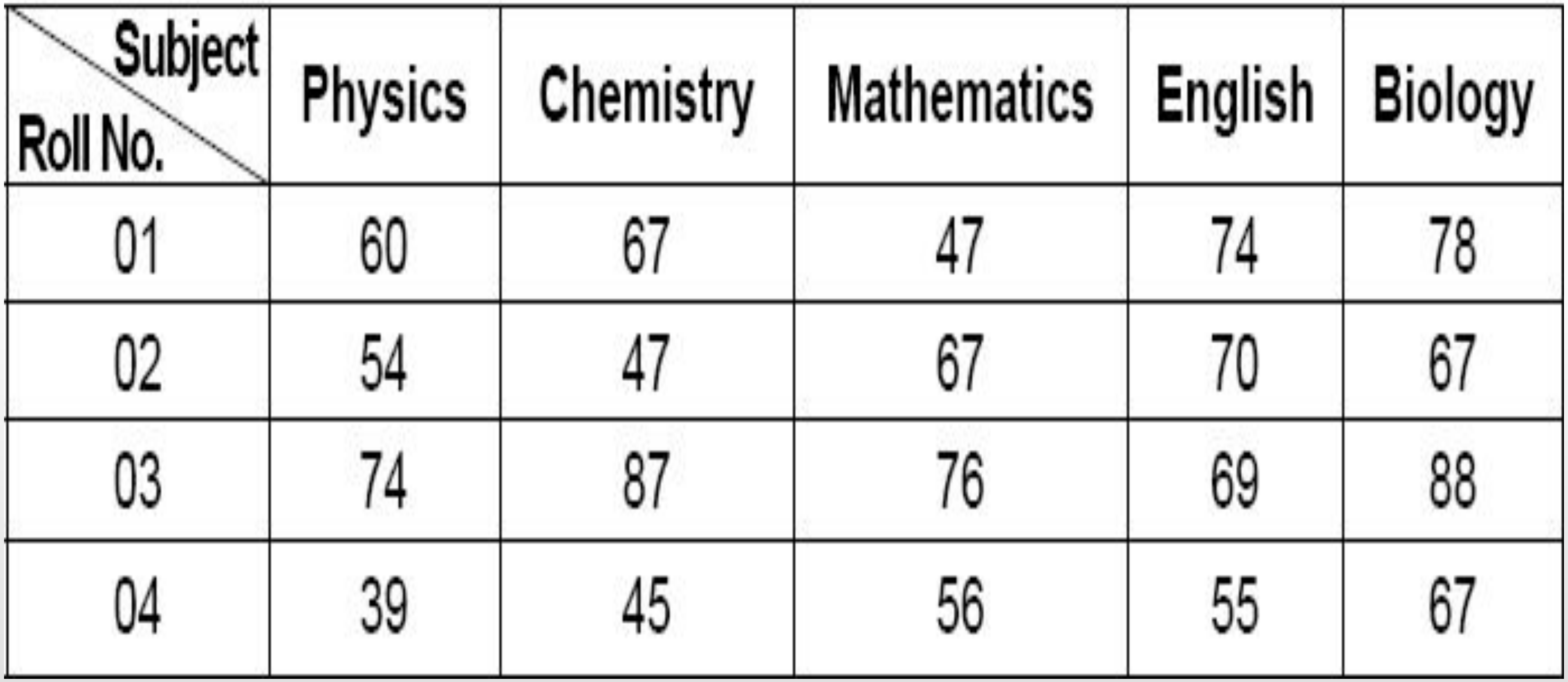
Sorting Arrays (SortArray.java)

```
class SortArray          {
public static void main(String[] args) {
int[] marks = {3, 5, 1, 2, 4};
int temp, n;
n = marks.length;
System.out.print("The list of marks is: ");
for(int i = 0; i< n; i++){
System.out.print(marks[i]+ " ");    }
for (int i = 0; i< n; i++){
for (int j = i+1; j < n; j++){
if (marks[i] < marks[j])        {
temp = marks[i];
marks[i] = marks[j];
marks[j] = temp;                 }}}
System.out.print("\nList of marks sorted in descending order is: ");
for (int i = 0; i< n; i++){
System.out.print(marks[i]+" ");    } } }
```

2-D Arrays

- Sometimes values can be conceptualized in the form of table that is in the form of rows and columns.
- Suppose we want to store the marks of different subjects. We can store it in a one-dimensional array.
- Now if we want to add a second dimension in the form of roll no. of the student. This is possible only if we follow a tabular approach of storing data

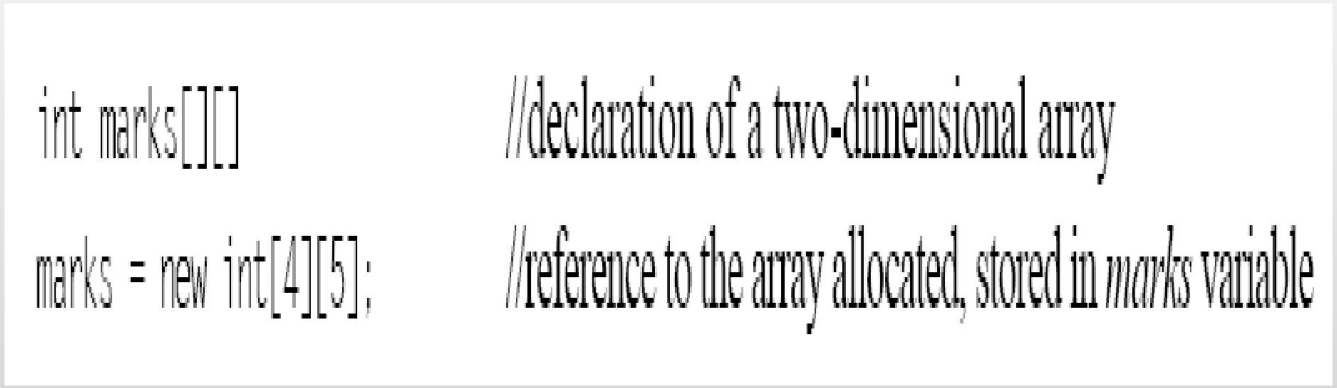
2-D Arrays



Subject Roll No.	Physics	Chemistry	Mathematics	English	Biology
01	60	67	47	74	78
02	54	47	67	70	67
03	74	87	76	69	88
04	39	45	56	55	67

2-D Arrays

- If you want a multidimensional array, the additional index has to be specified using another set of square brackets.
- Following statements create a two-dimensional array, named as **marks**, which would have 4 rows and 5 columns.

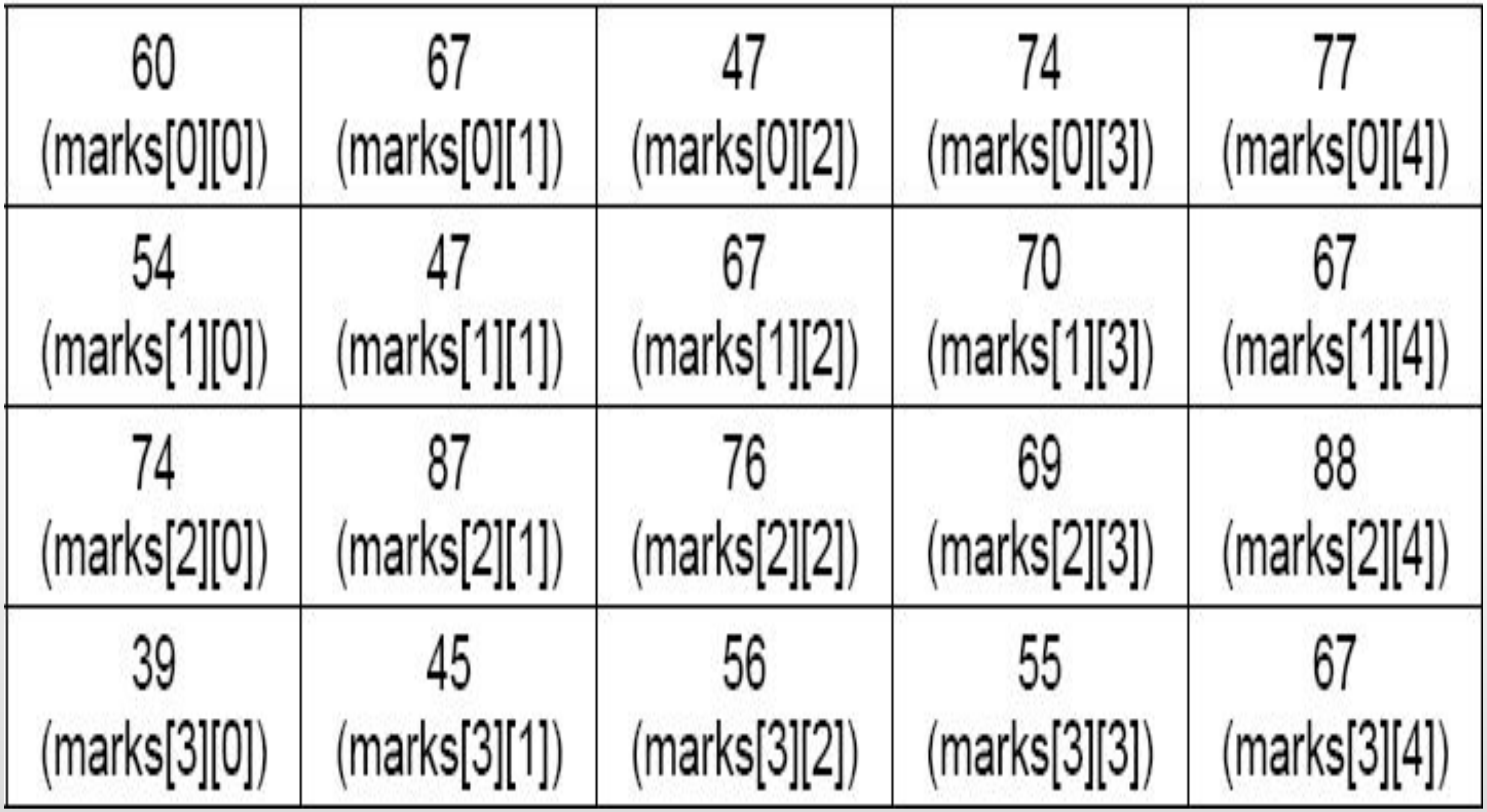


```
int marks[][]              //declaration of a two-dimensional array
marks = new int[4][5];     //reference to the array allocated, stored in marks variable
```

2-D Arrays

- This statement just allocates a 4 × 5 array and assigns the reference to array variable **marks**.
- The first subscript inside the square bracket signifies the number of rows in the table or matrix and the second subscript stands for the number of columns.
- This 4 × 5 table can store 20 values altogether.
- Its values might be stored in contiguous locations in the memory, but logically, the stored values would be treated as if they are stored in a 4 × 5 matrix.
- The following table shows how the **marks** array is conceptually placed in the memory by the above array creation statement.

2-D Arrays



60 (marks[0][0])	67 (marks[0][1])	47 (marks[0][2])	74 (marks[0][3])	77 (marks[0][4])
54 (marks[1][0])	47 (marks[1][1])	67 (marks[1][2])	70 (marks[1][3])	67 (marks[1][4])
74 (marks[2][0])	87 (marks[2][1])	76 (marks[2][2])	69 (marks[2][3])	88 (marks[2][4])
39 (marks[3][0])	45 (marks[3][1])	56 (marks[3][2])	55 (marks[3][3])	67 (marks[3][4])

2-D Arrays

- Like a one-dimensional array, two-dimensional array may be initialized with values, at the time of its creation. For example,
  - int marks[2][4] = {2, 3, 6, 0, 9, 3, 3, 2};
- This declaration shows that the first two rows of a 2 × 4 matrix have been initialized by the values shown in the list above. It can also be written as,
  - int marks[ ][ ] = {(2, 3, 6, 0), (9, 3, 3, 2)};

Example

```
class MatrixMul {
public static void main(String args[]) {
int array[][] = {{3,7},{6,9}};
int array1[][] = {{5,4},{3,6}};
int array2[][] = new int[2][2];
int x = array.length;
System.out.println("Matrix 1: ");
for (int i=0; i<array.length; i++) {
for (int j=0; j<array[i].length; j++) {
System.out.print(" "+array[i][j]);
}
System.out.println();
}
```

Example (contd.)

```
int y = array1.length;
System.out.println("Matrix 2: ");
for (int i=0; i<array1.length; i++) {
for (int j=0; j<array1[i].length; j++)   {
System.out.print(" "+array1[i][j]);    }
System.out.println();     }
for (int i=0; i<x; i++) {
for (int j=0; j<y; j++) {
for(int k=0; k<y; k++) {
array2[i][j] += array[i][k]*array1[k][j];    }    }    }
System.out.println("Multiplication of both matrices: ");
```

Example (contd.)

```
for (int i=0; i<x; i++) {
for (int j=0; j<y; j++) {
System.out.print(" "+array2[i][j]);    }
System.out.println();
}}}
```

Passing Arrays to Methods

```
static void show(int[][] a){
for(int i=0;i<a.length;i++){
for(int j=0;j<2;j++){
System.out.print(" " +a[i][j]);
}
System.out.println();
} }
```

Returning Arrays from Methods

```
 static int[][] show()
 {
 int a[][]={{1,2},{2,3}};
  return a;
}
```

Using for-each with Arrays

The enhanced for loop, i.e. for-each was introduced in Java 5 to enhance iterations over arrays and other collections easier. The format of for-each is as follows:

```
for (type var : arr){
// Body of loop
}
```

Using for-each with Arrays

- For example, to calculate the sum of the elements of the array, for-each can be used as follows:

```
int[] arr= {2,3,4,5,6};
int sum = 0;
for( int a : arr)
// a gets successively each value in arr
{
   sum += a;
}
```

- The disadvantage of for-each approach is that it is possible to iterate in forward direction only by single steps.

Variable Arguments

- Variable arguments can be used when the number of arguments that you want to pass to a method are not fixed.
- The method can accept different number of arguments of same type whenever they are called.
- The generic syntax for this notation is:

```
 returntype methodName(datatype...arrayname)
```

Variable Arguments

```
static void add(int...a)
{
 int sum=0;
 for(int i=0;i<a.length;i++)
 sum=sum+a[i];
 System.out.println("SUM = "+sum);
 }
```

- Can be invoked as

  `add(2,3,4,5)` Or `add(2,3,4);`

Command Line Arguments

- Suppose you have a Java application, called **sort**, which sorts the lines in a file named Sort.text. You would invoke the Sort application as:
  - java Sort Example.txt
- When the application is invoked, the runtime system passes the command line arguments to the application's main( ) method via an array of string.
- In the statement above, command line arguments (Example.txt) passed to the Sort application contain a single string, i.e. Example.txt.
- This String array is passed to the main method and it is copied in *args*.

```
public static void main (String args[]) {
. . . . . .
} // end of the main( ) method
```

Command Line Arguments

- In this case, each of the elements in the array named *args* (including the elements at position zero) is a reference to one of the command line arguments, each of which is a string object.
- The number of elements in an array can be obtained from the **length** property of the array.
- Therefore in the signature of main( ), it is not necessary in Java to pass a parameter specifying the number of arguments.

Example

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

- After compiling the program, when it is executed, you can pass the command line arguments as follows:

```
C>java Echo A B C
```

The Output

- A
- B
- C

If the command

```
c:\java Echo "A is first letter" "B is second" "C is third"
```

is invoked

The output is as follows:

- A is first letter
- B is second
- C is third

Your Turn

- What is difference between local, instance, and class variables?
- What is the difference between static and instance initialization block?
- How are arrays created in Java?
- What are variable arguments
- What are command line arguments?

Nested Classes

- Nested class is a class within a class. Nested classes are of the following types:
  - ❑ Non-static inner classes
  - ❑ Static nested classes
  - ❑ Local classes
  - ❑ Anonymous Inner classes

Why do We Create Nested Classes?

- Lets you turn logic into their own classes which normally you would not turn into, thus allowing even more object orientation into your programming.
- Lets you encapsulate logic into classes.
- Inner classes provide a structured hierarchy.
- Allows callbacks to be defined conveniently.
- Callback allows an object to call back the originating object at a later point in time.

Why Nested Classes?

- Very effective in implementing event handling in Java.
- Another advantage of nested classes would be to group classes that would be required at one place only.
- If you are certain that a class will be useful to only one class then it is better to embed a class into another.
- The obvious advantages would be ease of readability and ease of maintaining the code.

Non Static Inner Class

- Is a member of the outer class declared outside the functions within a class.
- Bound to the instance of the enclosing class.
- Has access to all the members of the enclosing class even the parent's this reference and private members.
- Can be defined as private, default, protected, public, final and even abstract.

Non Static Inner Class

- Inner class instance has a reference to an enclosing outer instance.
- A reference to the outer class instance can be obtained through OuterClassName.this.
- Cannot have static variables or methods except for compile-time constant variables, i.e., static constant.
- Inner class's objects can be created within instance methods, constructors of the outer class or through an instance of outer class.

Inner Class Example

```
class InnerClassTest
{
int y=20; // instance variable of outer class
static int a=30; // class variable of outer class
class InnerClass // inner class begins
{
int x=10;
final static int z=50;
void show() {
System.out.println("Within Non static Inner Class");
```

Example (contd.)

```
System.out.println("Can Access Inner class variable "+x);
System.out.println("Can Access Outer class variables "+y);
System.out.println("Can Access Outer class static variables "+a);
System.out.println("Inner class instance accessed using this:"+this);
System.out.println("Outer class referred from inner class using InnerClassTest.this: "+ InnerClassTest.this);
outerInstanceMethod();
outerClassMethod();
}
```

Example (contd.)

```
void outerInstanceMethod() {
System.out.println("Outerclass Instance method called from Inner class"); }
static void outerClassMethod() {
System.out.println("Outerclass static method Called from Inner class"); }
void createInnerObject() {
new InnerClass().show(); }
public static void main(String args[]) {
InnerClassTest object = new InnerClassTest();
object.createInnerObject();
//Another way: object.newInnerClass().show(); } }
```

The Output

```
Within Non static Inner Class
Can Access Inner class variable 10
Can Access Outer class variables 20
Can Access Outer class static variables 30
Inner class instance accessed using this:
InnerClassTest$InnerClass@f72617
Outer class referred from inner class using InnerClassTest.this:
InnerClassTest@1e5e2c3
Outerclass Instance method Called from Inner class
Outerclass static method Called from Inner class
```

Static Nested Class

- Is a static member of a class just like normal static members of any class.
- Have access to all static methods of the enclosing parent class.
- Cannot directly refer to instance variables and method of the outer class, similar to static parts of any class; access it through an object of the outer class.
- Unlike the inner classes, the static nested classes can have static members.

Static Nested Class Example

```
class StaticNestedClassTest {
int y; // instance variable
static int z=100; // class variable
static class StaticNestedClass {
int x;
static int staticinner=200;
void nestedClassNonStaticMethod() {
// y cannot be referenced here
System.out.println("Accessing static variable of outer class within Static
Inner Class "+z);
//outerClassInstanceMethod();
outerClassStaticMethod();          }
```

Example (contd.)

```
static void nestedClassStaticMethod() {
System.out.println("Within Static method of Inner Class ");
//outerClassInstanceMethod();
outerClassStaticMethod();}
} // static nested class ends here
static void outerClassStaticMethod(){
System.out.println("Outer Class Static method");}
void outerClassInstanceMethod(){
System.out.println("Outer Class Instance method");  }
```

Example (contd.)

```
public static void main(String args[])
{
StaticNestedClassTest.StaticNestedClass object = new
StaticNestedClassTest.StaticNestedClass();
object.nestedClassNonStaticMethod();
object.nestedClassStaticMethod();
}
}
```

The Output

```
D:\javabook\programs\chap 4>java StaticNestedClassTest
Accessing static variable of outer class within Static Inner
Class 100
Outer Class Static method
Within Static method of Inner Class
Outer Class Static method
```

Local and Anonymous Classes

- *Local inner* classes are declared within a block of code and are visible only within that block, just as any other method variable.
- These classes are declared within a function. They can use only final (constant) local variables and parameters of the function.
- An *anonymous inner* class is a local class that has no name.
- They are extensively used in Event Handling.

Summary

- A class provides a sort of template or blueprint for an object.
- An object is a software bundle which encapsulates variables and methods, operating on those variables.
- A Java object is defined as an *instance of a class*.
- A class can have many instances of objects, each having its own copy of variables and methods. The variables declared inside a class (but outside a method) are termed as instance variables. Attributes of a class are defined by instance variables, while its behavior is defined by methods.

Summary

- These methods, declared as part of one object, can be invoked or called from another object. The method or variable belonging to a particular class can be accessed by specifying the name of the object to which they belong.
- A special type of variable, whose value remains the same throughout all the objects of a class, is known as class or static variable. Likewise a method can also be declared as static, which exists within the one location in memory no matter how many times it is called from multiple objects. Both static variable and static method can be called directly from anywhere inside a class, without specifying any object name.

Summary

- Method overloading and overriding are two ways of implementing polymorphism in Java.
- Command line arguments is a way of passing input to Java programs.
- Arrays are used to hold data of similar type. These can be one-dimensional or multidimensional arrays.
- Nested classes are classes within class. They can be static, non-static, local and anonymous.