Fundamentals of Object Oriented Programming

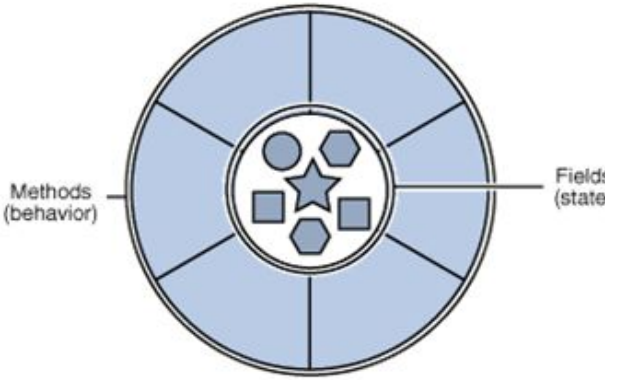
Dr. Ayesha Hakim

MODULE 2

2	**Class, Object, Method and Constructor**		**08**	**CO1,CO2**
	2.1	Class Object and Method: member, method, Modifier, Selector, iterator, State of an object, instanceof operator, Memory allocation of object using new operator.		
	2.2	Method overloading & overriding, constructor, destructor, Types of constructor (Default, Parameterized, copy constructor with object), Constructor overloading, this, final, super keyword, Garbage collection.		

Objects

- **object:** An entity that encapsulates data and behavior.
  - *data*: variables inside the object
  - *behavior*: methods inside the object
    - You interact with the methods; the data is hidden in the object.



- Constructing (creating) an object:

```
Type objectName = new Type(parameters);
```

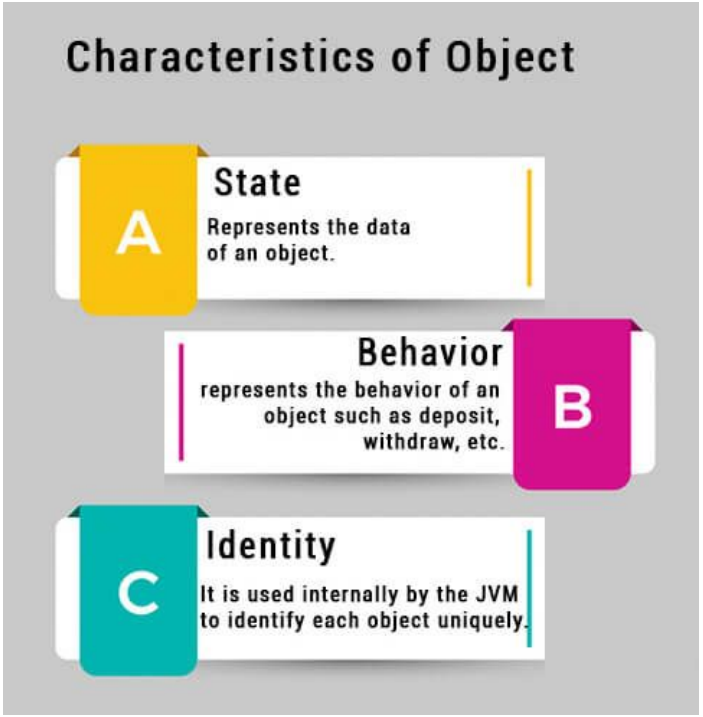
- Calling an object's method:

```
objectName.methodName(parameters);
```

Objects in Java

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. **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.

In Java, when we only declare a variable of a class type, only a reference is created (memory is not allocated for the object). To allocate memory to an object, we must use **new().**



An object has three characteristics:

- **State:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

Classes

- **class**: A program entity that represents either:
  1. A program / module, or
  2. A template for a new type of objects.

– **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects.

  – **abstraction**: Separation between concepts and details.
  Objects and classes provide abstraction in programming.

Accessing the Data Members

- The public data members of objects of a class can be accessed using the direct member access operator (.)

```
#include <iostream>
using namespace std;
class Box{
public:
      double  length;   // Length of a box
      double breadth;  // Breadth of a box
      double height;   // Height of a box
};
  int main( ){
  Box Box1;        // Declare Box1 of type Box
  Box Box2;        // Declare Box2 of type
  Box   double volume = 0.0;     // Store the
  volume of a box here
  // box 1 specification   Box1.height = 5.0;
   Box1.length = 6.0;
  Box1.breadth = 7.0;
  // box 2 specification
  Box2.height = 10.0;
  Box2.length = 12.0;
  Box2.breadth = 13.0;   //
  volume of //box 1
  volume = Box1.height *
  Box1.length * Box1.breadth;
  cout << "Volume of Box1 : " <<
  volume <<endl;    // volume of
  box 2
  volume = Box2.height *
  Box2.length * Box2.breadth;
  cout << "Volume of Box2 : " <<
  volume <<endl;   return 0;}
```

Members of class

- A member function of a class is a function that has its definition or its prototype within the class
- Members are accessed using dot operator(.)

```
class Box
{
public:
  double length;         // Length of a box
  double breadth;        // Breadth of a box
      double height;         // Height of a box
  double getVolume(void);   // Returns box volume
  // member function
};
```

data member

Members of class(continued...)

- Member functions can be defined within the class definition or separately using **scope resolution operator, ::**

```
class Box{
   public:
     double length;      // Length of a box
     double breadth;     // Breadth of a box
     double height;      // Height of a box
     double getVolume(void){
          return length * breadth * height;
     }
};// class end
```

Members of class(continued...)

- If you like you can define same function outside the class using **scope resolution operator**, :: as follows:

```
double Box::getVolume(void){
  return length * breadth * height;
}
```

Classes ARE Object Definitions

- OOP - object oriented programming
- code built from objects
- Java these are called ***classes***
- Each class definition is coded in a separate .java file
- Name of the object must match the class/object name

Classes and Objects

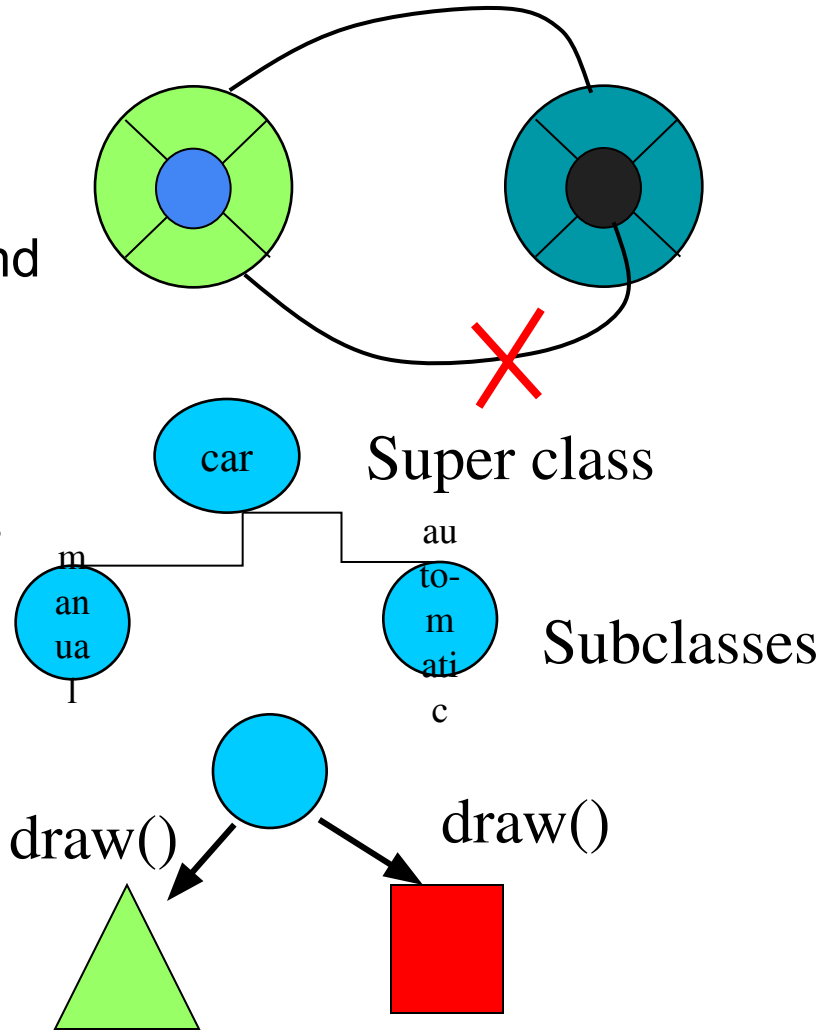
- The ***class*** is the unit of programming
- A Java program is a ***collection of classes***
  - Each class definition (usually) in its own **`.java`** file
  - *The file name must match the class name*
- A class describes ***objects (instances)***
  - Describes their common characteristics: is a *blueprint*
  - Thus all the instances have these same characteristics
- These characteristics are:
  - ***Data fields*** for each object
  - ***Methods*** (operations) that do work on the objects

Grouping Classes: The Java API

- API = *Application Programming Interface*
- Java = small core + extensive collection of packages
- A ***package*** consists of some related Java classes:
  - Swing: a GUI (graphical user interface) package
  - AWT: Application Window Toolkit (more GUI)
  - util: utility data structures
- The ***import*** statement tells the compiler to make available classes and methods of another package
- A ***main*** method indicates where to begin executing a class (if it is designed to be run as a program)

The three principles of OOP

- Encapsulation
  - Objects hide their functions (**methods**) and data (**instance variables**)
- Inheritance
  - Each **subclass** inherits all variables of its **superclass**
- Polymorphism
  - Interface same despite different data types



Simple Class and Method

```
Class Fruit{
    int grams;
    int cals_per_gram;

    int total_calories() {
     return(grams*cals_per_gram);
    }
}
```

Methods

- A method is a named sequence of code that can be invoked by other Java code.
- A method takes some parameters, performs some computations and then optionally returns a value (or object).
- Methods can be used as part of an expression statement.

```
public float convertCelsius(float tempC) {
   return( ((tempC * 9.0f) / 5.0f) + 32.0 );
  }
```

Method Signatures

- A method signature specifies:
  - The name of the method.
  - The type and name of each parameter.
  - The type of the value (or object) returned by the method.
  - The checked exceptions thrown by the method.
  - Various method modifiers.
  - *modifiers type name (parameter list) [throws exceptions]*

```
public float convertCelsius (float tCelsius ) {}
public boolean setUserInfo ( int i, int j, String name ) throws
    IndexOutOfBoundsException {}
```

Public/private

- Methods/data may be declared ***public*** or ***private*** meaning they may or may not be accessed by code in other classes …
- Good practice:
  - keep data private
  - keep most methods private
- well-defined interface between classes - helps to eliminate errors

Using objects

- Here, code in one class creates an instance of another class and does something with it …

```
Fruit plum=new Fruit();
int cals;
cals = plum.total_calories();
```

- ***Dot operator*** allows you to access (public) data/methods inside Fruit class

instanceof operator

- **instanceof** is an operator that determines if an object is an instance of a specified class:

```
Book b1 = new Book("Thinking in Java","Bruce Eckel",1129);

System.out.println(b1 instanceof Book);
```

```
True
```

The instanceof in java is also known as **type *comparison operator*** because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

Java Iterator

An `Iterator` is an object that can be used to loop through collections, like ArrayList and HashSet. It is called an "iterator" because "iterating" is the technical term for looping.

To use an Iterator, you must import it from the `java.util` package.

The `iterator()` method can be used to get an `Iterator` for any collection.

Syntax

```
Iterator itr = c.iterator();
```

Note: *Here "c" is any Collection object. itr is of type Iterator interface and refers to "c".*

Methods of Iterator Interface in Java

The iterator interface defines **three** methods as listed below:

1. hasNext(): Returns true if the iteration has more elements.

```
public boolean hasNext();
```

2. next(): Returns the next element in the iteration. It throws **NoSuchElementException** if no more element is present.

```
public Object next();
```

3. remove(): Removes the next element in the iteration. This method can be called only once per call to next().

```
public void remove();
```

Selector

selector

(or *method selector*) a word (e.g., draw) that performs an operation on a variety classes. A selector describes *what* operation to perform.

selector invocation

a call of a selector. One argument of the call is used for determining which method is used. i.e., a message (consisting of the selector and the other arguments) is sent to the object.

receiving object

the object used for determining the method executed by a selector invocation.

Access Modifiers in Java

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

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

The `public` keyword is an access modifier, meaning that it is used to set the access level for classes, attributes, methods and constructors.

We divide modifiers into two groups:

- Access Modifiers - controls the access level
- Non-Access Modifiers - do not control access level, but provides other functionality

For **classes**, you can use either `public` or *default*:

Modifier	Description
`public`	The class is accessible by any other class
default	The class is only accessible by classes in the same package. This is used when you don't specify a modifier. You will learn more about packages in the Packages chapter

For **attributes, methods and constructors**, you can use the one of the following:

Modifier	Description
`public`	The code is accessible for all classes
`private`	The code is only accessible within the declared class

For **attributes, methods and constructors**, you can use the one of the following:

Modifier	Description
`public`	The code is accessible for all classes
`private`	The code is only accessible within the declared class
default	The code is only accessible in the same package. This is used when you don't specify a modifier. You will learn more about packages in the Packages chapter
`protected`	The code is accessible in the same package and **subclasses**. You will learn more about subclasses and superclasses in the Inheritance chapter

Non-Access Modifiers

For **classes**, you can use either `final` or `abstract`:

Modifier	Description
`final`	The class cannot be inherited by other classes (You will learn more about inheritance in the Inheritance chapter)
`abstract`	The class cannot be used to create objects (To access an abstract class, it must be inherited from another class. You will learn more about inheritance and abstraction in the Inheritance and Abstraction chapters)

For **attributes and methods**, you can use the one of the following:

Modifier	Description
`final`	Attributes and methods cannot be overridden/modified
`static`	Attributes and methods belongs to the class, rather than an object
`abstract`	Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example **abstract void run();**. The body is provided by the subclass (inherited from). You will learn more about inheritance and abstraction in the Inheritance and Abstraction chapters
`transient`	Attributes and methods are skipped when serializing the object containing them
`synchronized`	Methods can only be accessed by one thread at a time
`volatile`	The value of an attribute is not cached thread-locally, and is always read from the "main memory"

The super Reference

- Constructors are not inherited, even though they have public visibility
- Yet we often want to use the parent's constructor to set up the "parent's part" of the object
- The `super` reference can be used to refer to the parent class, and often is used to invoke the parent's constructor

The super Reference

- A child's constructor is responsible for calling the parent's constructor
- The first line of a child's constructor should use the `super` reference to call the parent's constructor
- The `super` reference can also be used to reference other variables and methods defined in the parent's class

Using super

super has two general forms.

✓**The first calls the superclass constructor.**

✓**The second is** used to access a member of the superclass that has been hidden by a member of a subclass

Inheritance

Using super

A first Use for super

```
class a
{
a(){System.out.println("A");}
}
class b extends a
{
b()
{
super();
System.out.println("B");}
}
class c extends b
{
c(){System.out.println("c");}
}
class last
{
public static void main(String args[])
{
c obj = new c();
}
}
```

A Second Use for super

The second form of **super acts somewhat like this, except that it always refers to the superclass** of the subclass in which it is used. This usage has the following general form:

super.*member*

Here, *member can be either a method or an instance variable.*

Most applicable to situations in which member names of a subclass hide members by the same name in the superclass. Consider this simple class hierarchy:

A Second Use for super

```
// Using super to overcome name hiding.
class A {
int i;
}
// Create a subclass by extending class A.
class B extends A {
int i; // this i hides the i in A
B(int a, int b) {
super.i = a; // i in A
i = b; // i in B
}
void show() {
System.out.println("i in superclass: " + super.i);
System.out.println("i in subclass: " + i);
}
}
class UseSuper {
public static void main(String args[]) {
B subOb = new B(1, 2);
subOb.show();
}
}
```

This program displays the following:

```
i in superclass: 1
i in subclass: 2
```

The super keyword in Java

- super keyword is similar to this keyword in Java.
- It is used to refer to the immediate parent class of the object.

Use of super keyword in Java

- super () calls the parent class constructor with no argument.
- super.methodname calls method from parents class.
- It is used to call the parents class variable.

Example: Sample program for invoking parents class constructor

```
class Animal
{
   public Animal(String str)
   {
      System.out.println("Constructor of Animal: " + str);
   }
}
class Lion extends Animal
{
   public Lion()
   {
      super("super call Lion constructor");
      System.out.println("Constructer of Lion.");
   }
}
public class Test
{
   public static void main(String[] a)
  {
      Lion lion = new Lion();
   }
}
```

Output:

Constructor of Animal: super call from Lion constructor

Constructor of Lion.

Example: Sample program for calling parents class variable

```
class Base
{
   int a = 50;
}
public class Derive extends Base
{
   int a = 100;
   void display()
   {
      System.out.println(super.a);
      System.out.println(a);
   }
   public static void main(String[] args)
   {
      Derive derive = new Derive();
      derive.display();
   }
}
```

Output:

50

100

this keyword

- The keyword “this” is used in an instance method to refer to the object that contains the method, i.e. the 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

Example codes: https://www.javatpoint.com/this-keyword

this keyword (contd.)

- 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;
}
```

Java Garbage Collection

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

Cleaning up unused object

- 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.
- The Java runtime environment has a garbage collector that periodically frees the memory used by 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.

https://www.javatpoint.com/Garbage-Collection

Garbage Collector

- Two basic approaches :
  - R*eference counting* and
  - *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.

Garbage Collector (contd.)

- 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
- The objects that are not marked (not referenced) are assumed to be garbage and their memory is reclaimed.

Garbage Collector (contd.)

- 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 copies 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.
- It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.
- It is automatically done by the garbage collector(a part of JVM) so we don't need to make extra efforts.

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.

How can an object be unreferenced?

1) By nulling a reference:

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

2) By assigning a reference to another:

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;//now the first object referred by e1 is available for garbage collection
```

3) By anonymous object:

```
new Employee();
```

finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

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

Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

```
public static void gc(){}
```

gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

```
public static void gc(){}
```

Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

Simple Example of garbage collection in java

```
public class TestGarbage1{
 public void finalize(){System.out.println("object is garbage collected");}
 public static void main(String args[]){
  TestGarbage1 s1=new TestGarbage1();
  TestGarbage1 s2=new TestGarbage1();
  s1=null;
  s2=null;
  System.gc();
 }
}
```

Test it Now

```
object is garbage collected
object is garbage collected
```

The final keyword in Java

The **final** keyword in Java indicates that no further modification is possible. Final can be Variable, Method or Class

Final Variable

Final variable is a constant. We cannot change the value of final variable after initialization.

Example: Sample program for final keyword in Java

```
class FinalVarDemo
{
    final int a = 10;
    void show()
    {
        a = 20;
        System.out.println("a : "+a);
    }
    public static void main(String args[])
    {
        FinalVarDemo var = new FinalVarDemo();
        var.show();
    }
}
```

Output:
Compile time error

The Final Modifier (Inheritance)

- Methods preceded by the final modifier cannot be overridden
  e.g., public ***final*** void displayTwo ()
- Classes preceded by the final modifier cannot be extended
  -e.g., ***final*** public class ParentFoo

Final method

When we declare any method as final, it cannot be overriden.

Example: Sample program for final method in Java

```
class Animal
{
    final void eat()
   {
        System.out.println("Animals are eating");
    }
}
public class Dear extends Animal
{
    void eat()
    {
        System.out.println("Dear is eating");
    }
    public static void main(String args[])
    {
        Dear dear = new Dear();
        dear.eat();
    }
}
```

Output:
Compile time error

Final Class

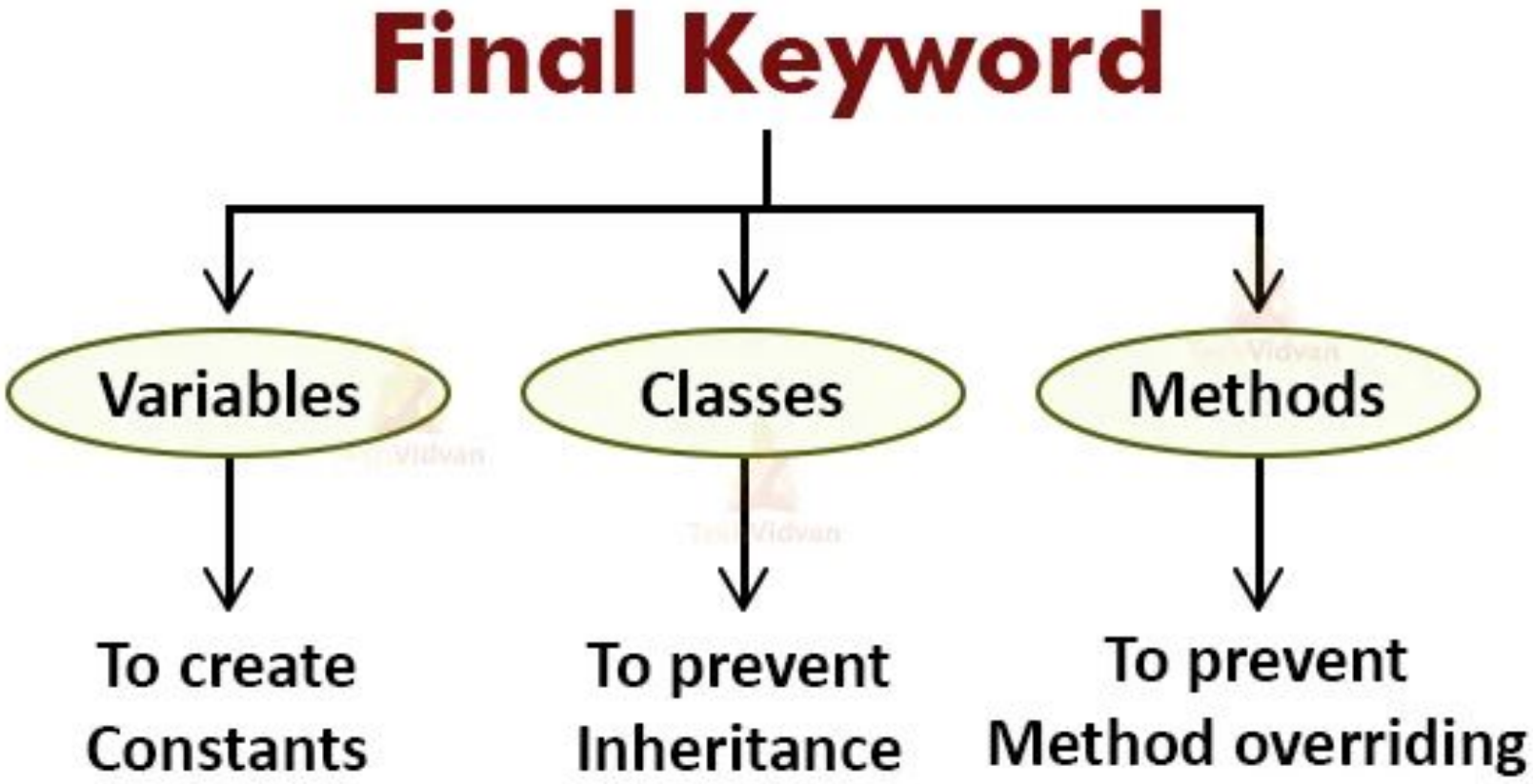
When a class is declared as a final, it cannot be extended.

Example: Sample program for final class in Java

```
final class Animal
{
    void eat()
    {
        System.out.println("Animals are eating");
    }
}
class Dear extends Animal
{
    void eat()
    {
        System.out.println("Dear is eating");
    }
    public static void main(String args[])
   {
        Dear dear = new Dear();
        dear.eat();
    }
}
```

Output:

Compile time error



https://www.javatpoint.com/final-keyword

Characteristics of final keyword in java:

In Java, the final keyword is used to indicate that a variable, method, or class cannot be modified or extended. Here are some of its characteristics:

- **Final variables:** When a variable is declared as final, its value cannot be changed once it has been initialized. This is useful for declaring constants or other values that should not be modified.
- **Final methods**: When a method is declared as final, it cannot be overridden by a subclass. This is useful for methods that are part of a class's public API and should not be modified by subclasses.
- **Final classes:** When a class is declared as final, it cannot be extended by a subclass. This is useful for classes that are intended to be used as is and should not be modified or extended.
- **Initialization:** Final variables must be initialized either at the time of declaration or in the constructor of the class. This ensures that the value of the variable is set and cannot be changed.
- **Performance:** The use of final can sometimes improve performance, as the compiler can optimize the code more effectively when it knows that a variable or method cannot be changed.
- **Security:** final can help improve security by preventing malicious code from modifying sensitive data or behavior.

Practise

- Java Program to Add Two Integers (entered by user)
- Java Program to Swap Two Numbers
- Java Program to Check Whether a Number is Even or Odd
- Java Program to Check Whether an Alphabet is Vowel or Consonant
- Java Program to Find Factorial of a Number
- Java Program to Check Whether a Number is Prime or Not
- Java Program to Find Factorial of a Number **Using Recursion**

$ Java Cheat Sheet

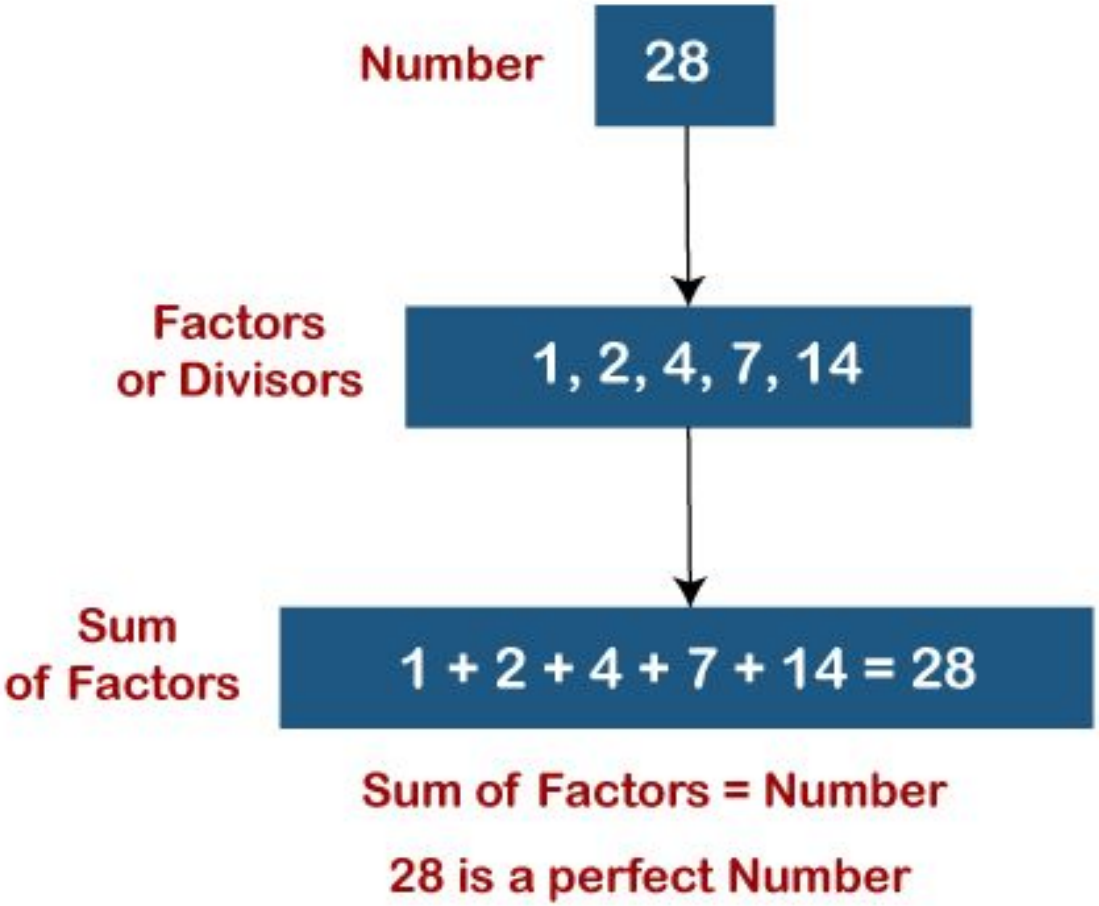
Expt. 1: Perfect Number

- A number whose sum of factors (excluding the number itself) is equal to the number is called a perfect number. In other words, if the sum of positive divisors (excluding the number itself) of a number equals the number itself is called a perfect number.

EXAMPLE:

- Let's take the number 496 and check it is a perfect number or not.
- First, we find the factors of 496 i.e. 1, 2, 4, 8, 16, 31, 62, 124, and 248. Let's find the sum of factors (1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 +248 = 496).
- We observe that the sum of factors is equal to the number itself. Hence, the number 496 is a perfect number.

Perfect Number in Java



Perfect Number	Positive Factors	Sum of all factors excluding itself
6	1, 2, 3, 6	6
28	1, 2, 4, 7, 14, 28	28
496	1, 2, 4, 8, 16, 31, 62, 124, 248, 496	496
8,128	1, 2, 4, 8, 16, 32, 64, 127, 254, 508, 1016, 2032, 4064, 8128	8,128

$$
\begin{aligned}
6 = 2^1(2^2-1) &= 1+2+3,\\
28 = 2^2(2^3-1) &= 1+2+3+4+5+6+7 = 1^3+3^3,\\
496 = 2^4(2^5-1) &= 1+2+3+\cdots+29+30+31\\
&= 1^3+3^3+5^3+7^3,\\
8128 = 2^6(2^7-1) &= 1+2+3+\cdots+125+126+127\\
&= 1^3+3^3+5^3+7^3+9^3+11^3+13^3+15^3,\\
33550336 = 2^{12}(2^{13}-1) &= 1+2+3+\cdots+8189+8190+8191\\
&= 1^3+3^3+5^3+\cdots+123^3+125^3+127^3.
\end{aligned}
$$

Steps to Find Perfect Number

- Read or initialize a number (n).
- Declare a variable (s) for storing sum.
- Find the factors of the given number (n) by using a loop (for/ while).
- Calculate the sum of factors and store it in a variable s.
- Compare the sum (s) with the number (n):
- If both (s and n) are equal, then the given number is a perfect number.
- Else, the number is not a perfect number.
- **Implement the above steps in a Java program to find Perfect Numbers Between Given Range**

Algorithm to Find Perfect Number in Java

- Insert an integer from the user.
- Set the initial value of a different variable to 0 to store the sum of the appropriate positive divisors.
- Check each number from 1 to n/2 to see if it is a divisor. Count the sum of all the divisors. If the sum is exactly n, the number is a perfect number; otherwise, it is not.

Program to Find Perfect Number in Java

There are three ways to find the perfect number in Java:

- Using for Loop
- Using while Loop
- Using recursion in Java

https://www.javatpoint.com/perfect-number-program-in-java

Static Method

```
import java.io.*;

public class StaticExample {

    static int num = 100;
     static String str = "you can call a static method without creating an object of the class";

    // This is Static method
    static void display()
    {
        System.out.println("static number is " + num);
        System.out.println("static string is " + str);
    }
```

```
// non-static method
    void nonstatic()
    {
        // our static method can accessed in non static method
        display();
    }

    // main method
    public static void main(String args[])
    {
        StaticExample obj = new StaticExample();

        // This is object to call non static function
        obj.nonstatic();

        // static method can called directly without an object
        display();
    }
}
```

Output

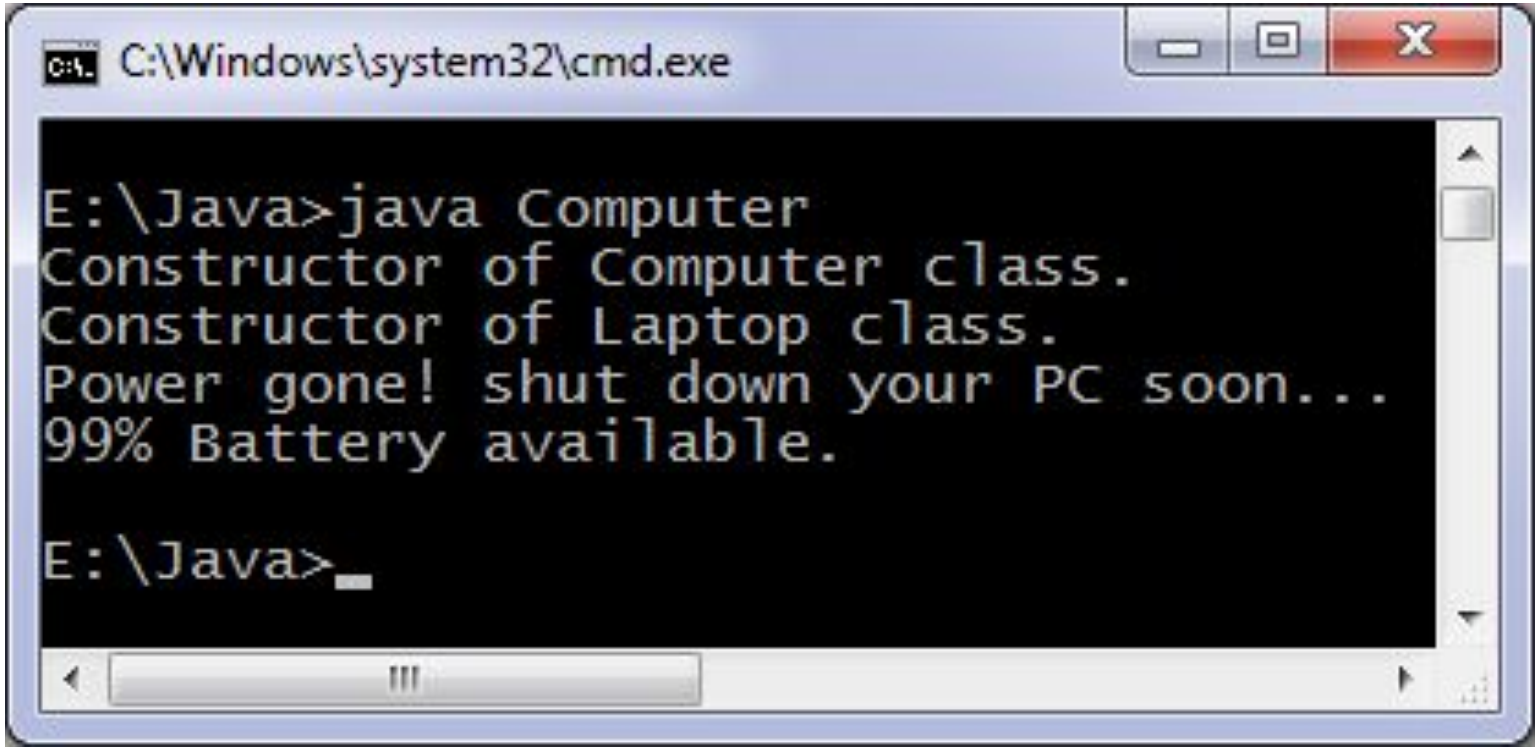
Output Clear

```
java -cp /tmp/qQIhOHaeFM StaticExample
static number is 100
static string is you can call a static method without creating an
    object of the class
static number is 100static string is you can call a static method
    without creating an object of the class
```

Using two classes in Java program

```
class Computer {
  Computer() {
    System.out.println("Constructor of Computer class.");
  }

  void computer_method() {
    System.out.println("Power gone! Shut down your PC soon...");
  }

  public static void main(String[] args) {
    Computer my = new Computer();
    Laptop your = new Laptop();

    my.computer_method();
    your.laptop_method();
  }
}

class Laptop {
  Laptop() {
    System.out.println("Constructor of Laptop class.");
  }

  void laptop_method() {
    System.out.println("99% Battery available.");
  }
}
```

OUTPUT


C:\Windows\system32\cmd.exe
E:\Java>java Computer
Constructor of Computer class.
Constructor of Laptop class.
Power gone! shut down your PC soon...
99% Battery available.
E:\Java>

You can also create objects in a method of Laptop class. When you compile the above program, two .class files are created, which are Computer.class and Laptop.class. Its advantage is you can reuse your .class file somewhere in other projects without recompiling the code.

In a nutshell number of .class files created are equal to the number of classes in the program. You can create as many classes as you want, but writing many classes in a single file isn't recommended, as it makes code difficult to read.

Instead, you can create a separate file for every class. You can also group classes in packages for efficiently managing the development of your application.

Constructors

- The line

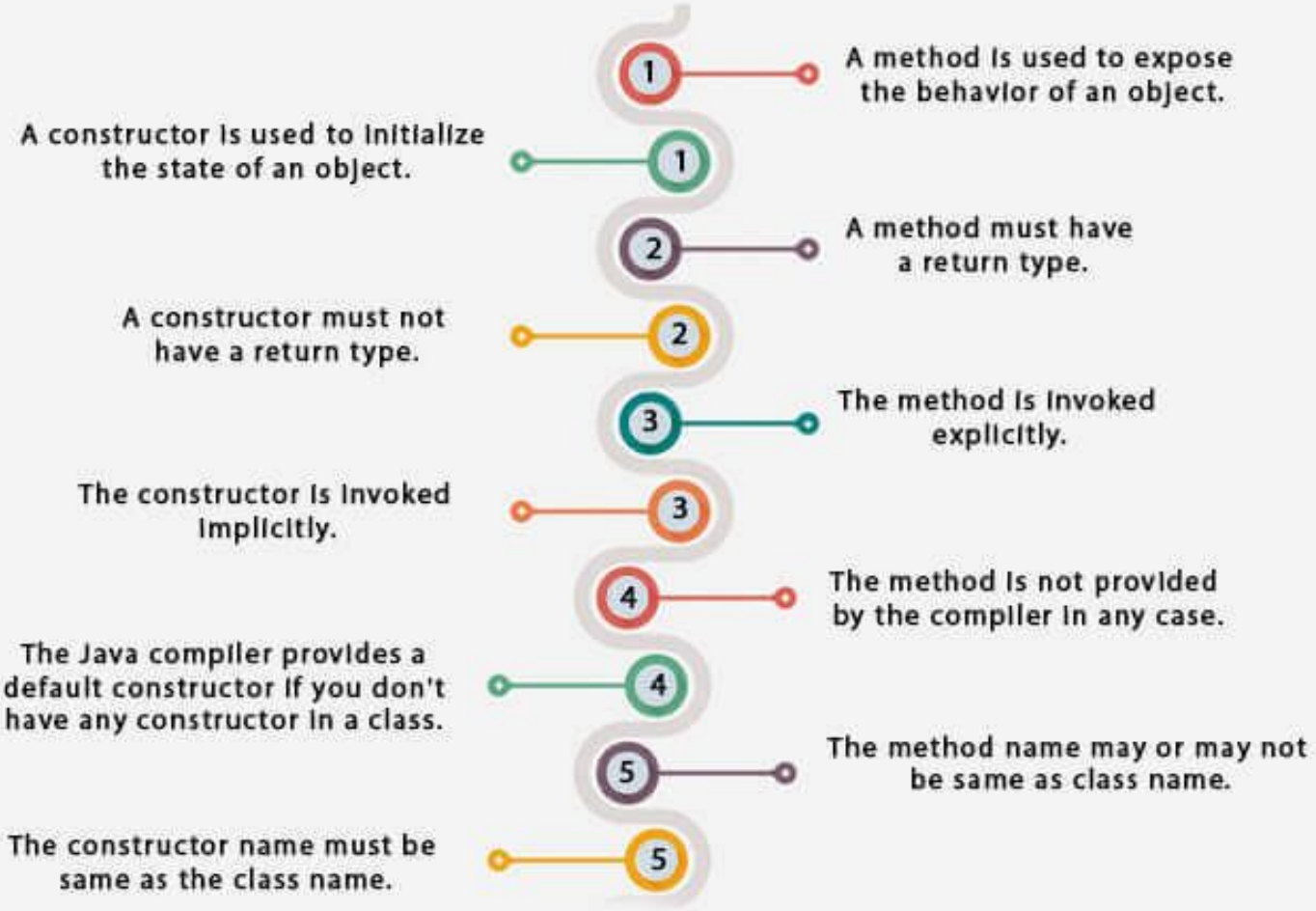
  plum = new Fruit();

- invokes a constructor method with which you can set the initial data of an object
- You may choose different types of constructor with different argument lists

  eg Fruit(), Fruit(a) ...

Run example programs from: https://www.javatpoint.com/java-constructor

Difference between constructor and method in Java



The Class Constructor

- A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.

- **A constructor will have exact same name as the class** and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.

- **0-argument constructor**
- **Parameterized constructor**
- **Copy constructor**

Copy Constructor with Object - Algorithm

- **Define a class:** Create a class that represents the object you want to manage.
- **Define instance variables:** Within the class, define instance variables that represent the data you want to manage.
- Define a constructor: Define a constructor for the class that takes an instance of the same class as its argument. This constructor will be used to create a copy of the object.
- **Initialize the instance variables**: Within the constructor, initialize the instance variables with the values from the argument object.
- Use the this keyword to refer to the instance variables: To refer to the instance variables of the class within the constructor, use the this keyword.
- **Check for null values**: If the argument object is null, return a new instance of the class with default values for the instance variables.
- **Implement deep copying:** If the instance variables are objects, create new instances of those objects within the constructor and initialize them with the values from the argument object. This is called deep copying and ensures that changes to the copied object do not affect the original object.

Code: https://www.geeksforgeeks.org/copy-constructor-in-java/

https://docs.google.com/document/d/1tM6La9VpnZNpBbQ9tcV11IMaogQTkrpxjKrWp1_O1EU/edit?usp=sharing

The Class Destructor

- A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

- **A destructor will have exact same name as the class prefixed with a tilde (~)** and it can neither return a value nor can it take any parameters.
- Destructor can be very **useful for releasing resources** before coming out of the program like closing files, releasing memories etc.

Overloading

- Can have several versions of a method in class with different types/numbers of arguments

```
Fruit() {grams=50;}
Fruit(a,b) { grams=a; cals_per_gram=b;}
```

- By looking at arguments Java decides which version to use

overloading methods and its purpose

✓If a class have multiple methods by same name but different 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.

✓Different ways to overload the method:

1. By changing number of arguments
2. By changing data type

Method overloading with an example:

```
// Demonstrate method overloading.
class OverloadDemo
{
void test()
{
System.out.println("No parameters"); }
// Overload test for one integer parameter.
void test(int a) {
 System.out.println("a: " + a); }
// Overload test for two integer parameters.
void test(int a, int b) {
 System.out.println("a and b: " + a + " " + b); }
// overload test for a double parameter
double test(double a) {
System.out.println("double a: " + a); return a*a; }
}
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
 System.out.println("Result of ob.test(123.25): " +
result); } }
```

This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

Why method overloading is not possible by changing the return type?
Can we overload main() method?

✓Method overloading is not possible by changing the return type of method because there may occur ambiguity.

✓Yes, we can have any number of main methods in a class by method overloading

Constructor overloading

What is constructor overloading ?

✓Just like in case of method overloading you have multiple methods with same name but different signature , in Constructor overloading you have multiple constructor with different signature with only difference that Constructor doesn't have return type in Java.

✓Those constructor will be called as **overloaded constructor**. Overloading is also another form of polymorphism in Java which allows to have multiple constructor with different name in one Class in java.

Recursion

Understanding static

Using command line arguments

Why do you overload Constructor?

- Allows flexibility while create array list object.
- It may be possible that you don't know size of array list during creation than you can simply use default no argument constructor but if you know size then its best to use overloaded
- Constructor which takes capacity. Since Array List can also be created from another Collection, may be from another List than having another overloaded constructor makes lot of sense.
- By using overloaded constructor you can convert your Array List into Set or any other collection.

Constructor overload with an example

```
/* Here, Box defines three constructors to initialize
the dimensions of a box various ways. */
class Box {
double width;
double height;
double depth;
 // constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
 height = h;
depth = d; }
// constructor used when no dimensions specified
 Box() {
width = -1;          // use -1 to indicate
height = -1;         // an uninitialized
depth = -1;          // box }
 // constructor used when cube is created
Box(double len) {
width = height = depth = len; }
// compute and return volume
double volume() {
return width * height * depth; } }
```

```
class OverloadCons {
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
double vol;
// get volume of first
box vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second
box vol = mybox2.volume();
System.out.println is " + vol);
// get volume of
cube vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
} }
```

The output produced by this program is shown here:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

Recursion

✓Recursion is the process of defining something in terms of itself.

✓As it relates to Java programming, recursion is the attribute that allows a method to call itself.

✓A method that calls itself is said to be recursive.

The classic example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N.
For example, 3 factorial is 1 × 2 × 3, or 6. Here is how a factorial can be computed by use of a recursive method:

Recursion with example

```
// A simple example of recursion.
class Factorial {
// this is a recursive method
int fact(int n) {
 int result;
if(n==1)
return 1;
result = fact(n-1) * n; return  result; } }
class Recursion { public static void main(String args[]) {
Factorial f = new Factorial(); System.out.println("Factorial of 3 is " +
f.fact(3)); System.out.println("Factorial of 4 is " + f.fact(4));
System.out.println("Factorial of 5 is " + f.fact(5)); } }
```

The output from this program is shown here: Factorial of 3 is 6 Factorial of 4 is 24 Factorial of 5 is 120

Overriding Methods

- A child class can *override* the definition of an inherited method in favor of its own
- The new method must have the same signature as the parent's method, but can have a different body
- The type of the object executing the method determines which version of the method is invoked

Overriding

- A parent method can be invoked explicitly using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

Overloading vs. Overriding

- Overloading deals with multiple methods with the same name in the same class, but with different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for different data
- Overriding lets you define a similar operation in different ways for different object types

Method Overriding

- Different versions of a method can be implemented in different ways by the parent and child class in an inheritance hierarchy.
- Methods have the same name and parameter list (identical signature) but different bodies

```
public class Parent                    public class Child extends Parent
{                                      {
        :      :                       :          :
        public void method ()          public void method ()
    {                                  {
            System.out.println("m1");          num = 1;
        }                                  }
}                                      }
```

Method Overloading Vs. Method Overriding

- Method Overloading
  - -Multiple method implementations for the same class
  - -Each method has the same name but the type, number or order of the parameters is different (signatures are not the same)
  - -The method that is actually called is determined at program *compile time* (early binding).
  - -i.e., <reference name>.<method name> (parameter list);

Distinguishes overloaded methods

Method Overloading Vs. Method Overriding (2)

- Example of method overloading:

```
public class Foo
{
    public void display () { }
    public void display (int i) { }
    public void display (char ch) { }
}
```

```
Foo f = new Foo ();
f.display();
f.display(10);
f.display('c');
```

Method Overloading Vs. Method Overriding (3)

- Method Overriding
  - -The method is implemented differently between the parent and child classes.
  - -Each method has the same return value, name and parameter list (identical signatures).
  - -The method that is actually called is determined at program *run time* (late binding).
  - -i.e., <reference name>.<method name> (parameter list);

The type of the reference (implicit parameter "this") distinguishes overridden methods

Method Overloading Vs. Method Overriding (4)

- Example of method overriding:

```
public class Foo
{
    public void display () { … }
        : :
}
public class FooChild extends Foo
{
     public void display () { … }
}

Foo f = new Foo ();
f.display();

FooChild fc = new FooChild ();
fc.display ();
```

No	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

Java Method Overloading example

```
class OverloadingExample{

static int add(int a,int b){return a+b;}

static int add(int a,int b,int c){return a+b+c;}

}
```

Java Method Overriding example

```
class Animal{

void eat(){System.out.println("eating...");}

}

class Dog extends Animal{

void eat(){System.out.println("eating bread...");}

}
```

Q1.

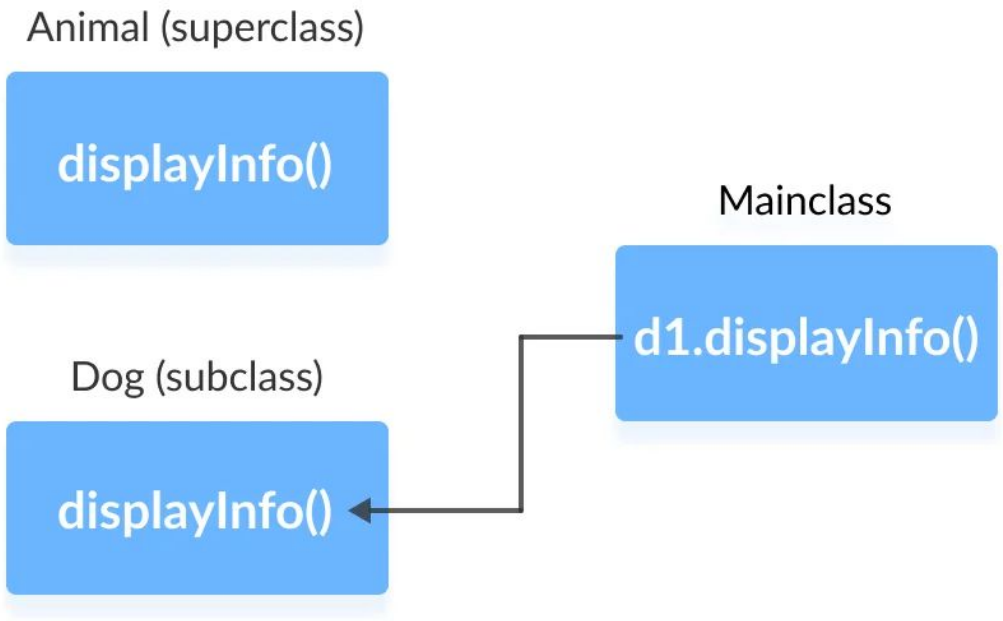
```
class Animal {
   public void displayInfo() {
      System.out.println("I am an animal.");
   }
}

class Dog extends Animal {
   @Override
   public void displayInfo() {
      System.out.println("I am a dog.");
   }
}

class Main {
   public static void main(String[] args) {
      Dog d1 = new Dog();
      d1.displayInfo();
   }
}
```

Output:
I am a dog.

WHY?

In the above program, the displayInfo() method is present in both the Animal superclass and the Dog subclass.

When we call displayInfo() using the d1 object (object of the subclass), the method inside the subclass Dog is called. The displayInfo() method of the subclass overrides the same method of the superclass.



Here, the @Override annotation specifies the compiler that the method after this annotation overrides the method of the superclass. It is not mandatory to use @Override.

Java Overriding Rules

- Both the superclass and the subclass must have the same method name, the same return type and the same parameter list.
- We cannot override the method declared as final and static.

Constructor Overloading

Defining two or more constructors with the **same name in a class** but with **different signatures** is called constructor overloading. We arrange them in a such a way that each constructor performs a different task.

Java compiler differentiates these constructors based on the number of the parameter lists and their types. Therefore, the **signature** of each constructor **must be different.**

The signature of a constructor consists of its name and sequence of parameter types.

If two constructors of a class have the same signature, it represents **ambiguity**. In this case, Java compiler will generate an error message because Java compiler will be unable to differentiate which form to execute.

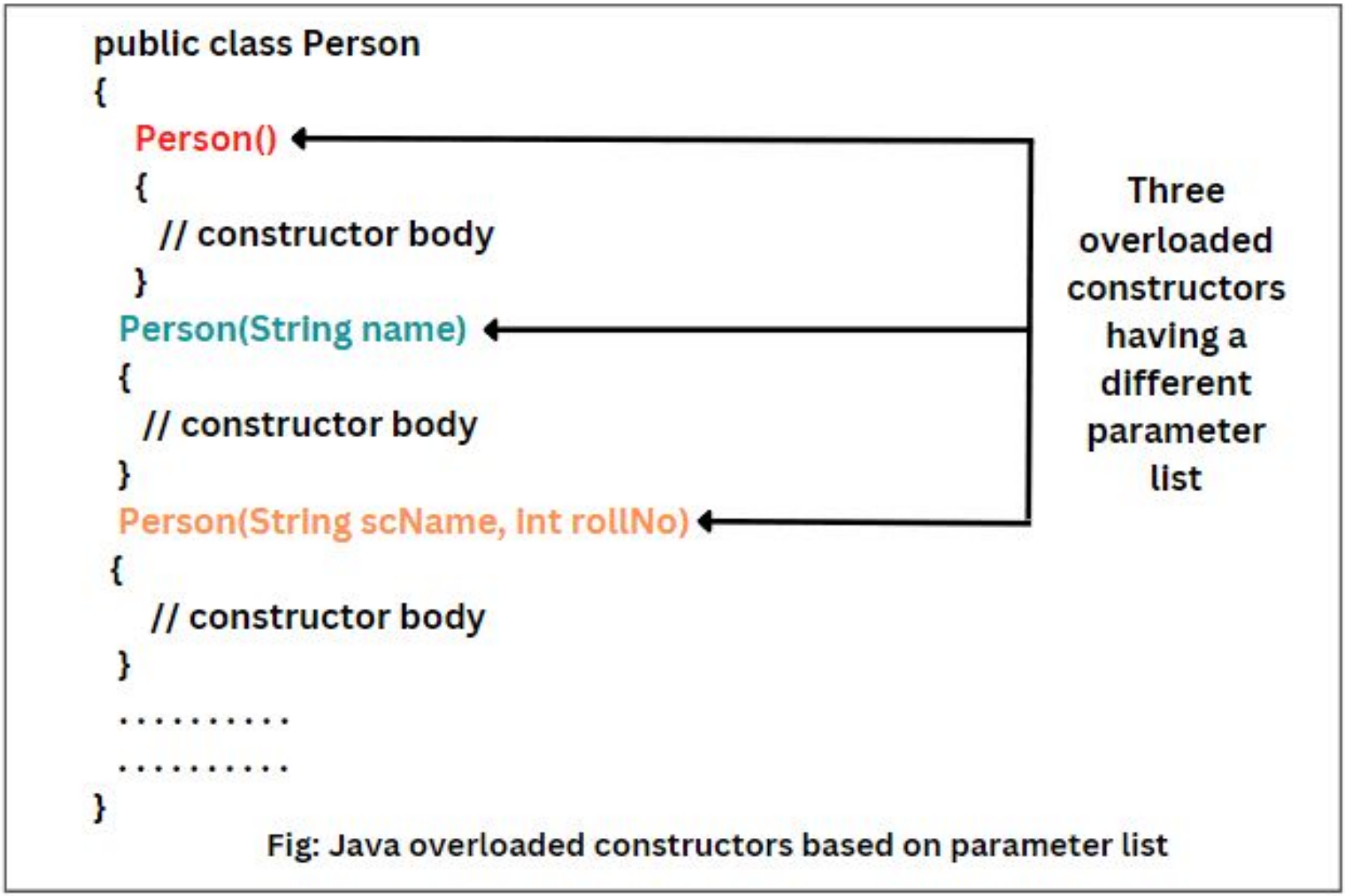


Fig: Java overloaded constructors based on parameter list

FLOW OF EXECUTION OF PROGRAM

```
public class Employee {
 Employee(){
        System.out.println("Employee Details:");
 }
 Employee(String name){
        System.out.println("Employee name: " +name);
 }
 Employee(String nCompany, int id){
        System.out.println("Company name: " +nCompany);
        System.out.println("Employee id: " +id);
 }
}
public class Myclass {
public static void main(String[] args) {
  Employee emp=new Employee();
  Employee emp2=new Employee("Deep");
  Employee emp3=new Employee("HCL", 12234);
      }
}
```

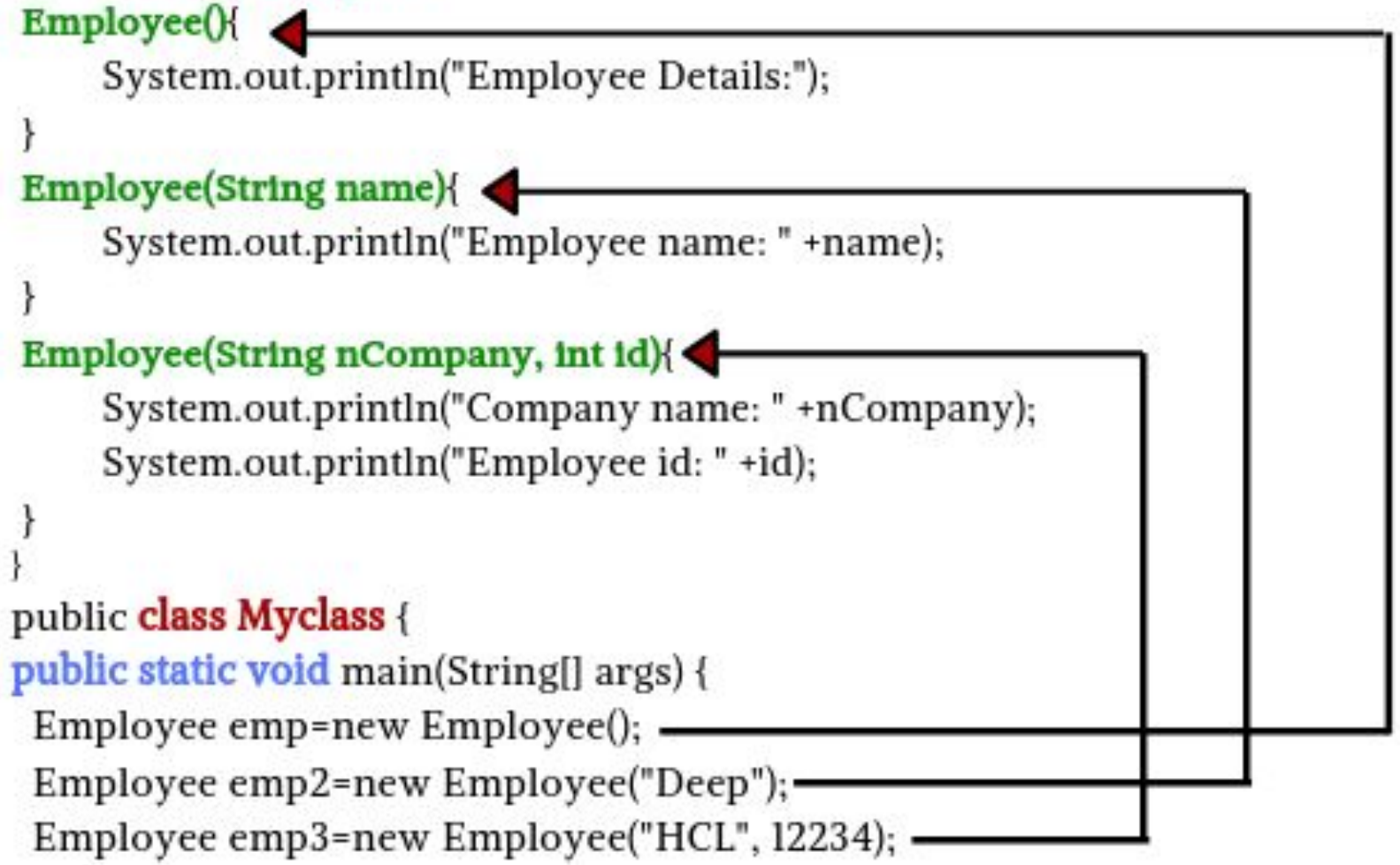


Fig: Overloaded constructors based on parameter list.

using PI in Java

```
import java.lang.Math.*;
public class Pie {
  public static void main(String[] args) {
   //radius and length
   double radius = 5;
   double len = 15;
   // calculate the area using PI
   double area = radius * radius * Math.PI;
   //and now volume
   double volume = area * len;
   System.out.println("Volume of cylinder is: " + volume);
  }
}
```

CODING ACTIVITY

#1. WAJP to find area of circle and rectangle using a) method overloading, b) method overriding

#2. WAJP to demonstrate Simple Parameterized Constructor For i) Finding Prime Number **or** ii) finding area of rectangle

#3. WAJP to illustrate constructor overloading to print the names of students by creating a Student class. If no name is passed while creating an object of Student class, then the name should be "Unknown", otherwise the name should be equal to the String value passed while creating object of Student class.

#1

```
class OverridingDemo
{
public int length=10;
}
class Rectangle extends OverridingDemo
{
void area()
{
int breadth=7;
int area=length*breadth;
System.out.println("The area of the rectangle is"+area+"sq
units");
}
}
class Circle extends OverridingDemo
{
void area()
{
double area =3.14*length*length;
System.out.println("The area of the circle is"+area+"sq units");
}
}
class Inheritence
{
public static void main(String args[])
{
Rectangle rec= new Rectangle();
Circle cir= new Circle();
rec.area();
cir.area();
}
}
```

```
OUTPUT
java Inheritence
The area of the rectangle is 70 sq units
The area of the circle is 314.0 sq units
```

```
class OverloadDemo
{
void area(float x,float y)
{
System.out.println("The area of the rectangle is "+x*y+"sq
units");
}
void area(double x)
{
double z= 3.14*x*x;
System.out.println("The area of the circle is "+z+"sq units");
}
}
class Overload
{
public static void main(String args[])
{
OverloadDemo ob= new OverloadDemo();
ob.area(10,20);
ob.area(2.5);
}
}
```

```
OUTPUT:
java Overload
The area of the rectangle is 200.0sq units
The area of the circle is 19.625sq units
```

Example: How to check number is prime or not using constructor in java.

```java
import java.util.Scanner;

class Test {

    int n, i, p = 1;

    Scanner sc = new Scanner(System.in);

    Test() {
        System.out.print("Enter a number:");
        n = sc.nextInt();
    }

    void checkPrime() {
        for (i = 2; i < n; i++) {
            if (n % i == 0) {
                p = 0;
                break;
            }
        }

        if (p == 1) {
            System.out.println("Number is prime:" + n);
        } else {
            System.out.println("Number is not prime:" + n);
        }

    }
}
```

```java
class Main {

    public static void main(String args[]) {

        Test obj = new Test();
        obj.checkPrime();

    }
}
```

Output:

```
Enter a number:13
Number is prime:13
```

EXTRA EXAMPLE

#2

```
1  class Prime{
2       Prime(int n){
3          int count=0;
4          for(int i=2;i<=n;i++){
5              if(n%i==0){
6                  count=count+1;
7              }
8              if(count==0){
9                  System.out.println("its a prime");
10                 break;
11             }
12             else{
13                 System.out.println("its not a prime");
14                 break;
15             }
16         }
17     }
18 }
19
20 class main {
21     public static void main(String[] args) {
22         Prime p =new Prime(7);
23
24
25     }
26 }
27
```

Output

Clear

```
java -cp /tmp/FUwutTEtp6 main
its a prime
```

Example: How to find the area of a rectangle using parameterized constructor in Java

#2

```java
import java.util.Scanner;

class Test {

    int area;

    Test(int l, int w) {
        area = l * w;
    }

    void display() {
        System.out.println("Area of rectangle is: " + area);
    }
}

class Main {

    public static void main(String args[]) {
        int length, width;
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter length of rectangle:");
        length = sc.nextInt();
        System.out.print("Enter width of rectangle:");
        width = sc.nextInt();

        Test t = new Test(length, width);
        t.display();
    }
}
```

Output:

```
Enter length of rectangle:12
Enter width of rectangle:3
Area of rectangle is: 36
```

#3

```java
class Student{
  String name;
  public Student(String s){
    name = s;
  }
  public Student(){
    name = "Unknown";
  }
}

class Ans{
  public static void main(String[] args){
    Student s = new Student("xyz");
    Student a = new Student();

    System.out.println(s.name);
    System.out.println(a.name);
  }
}
```


The End!