Object Oriented Paradigm

How to solve problem using Object Oriented Paradigm?

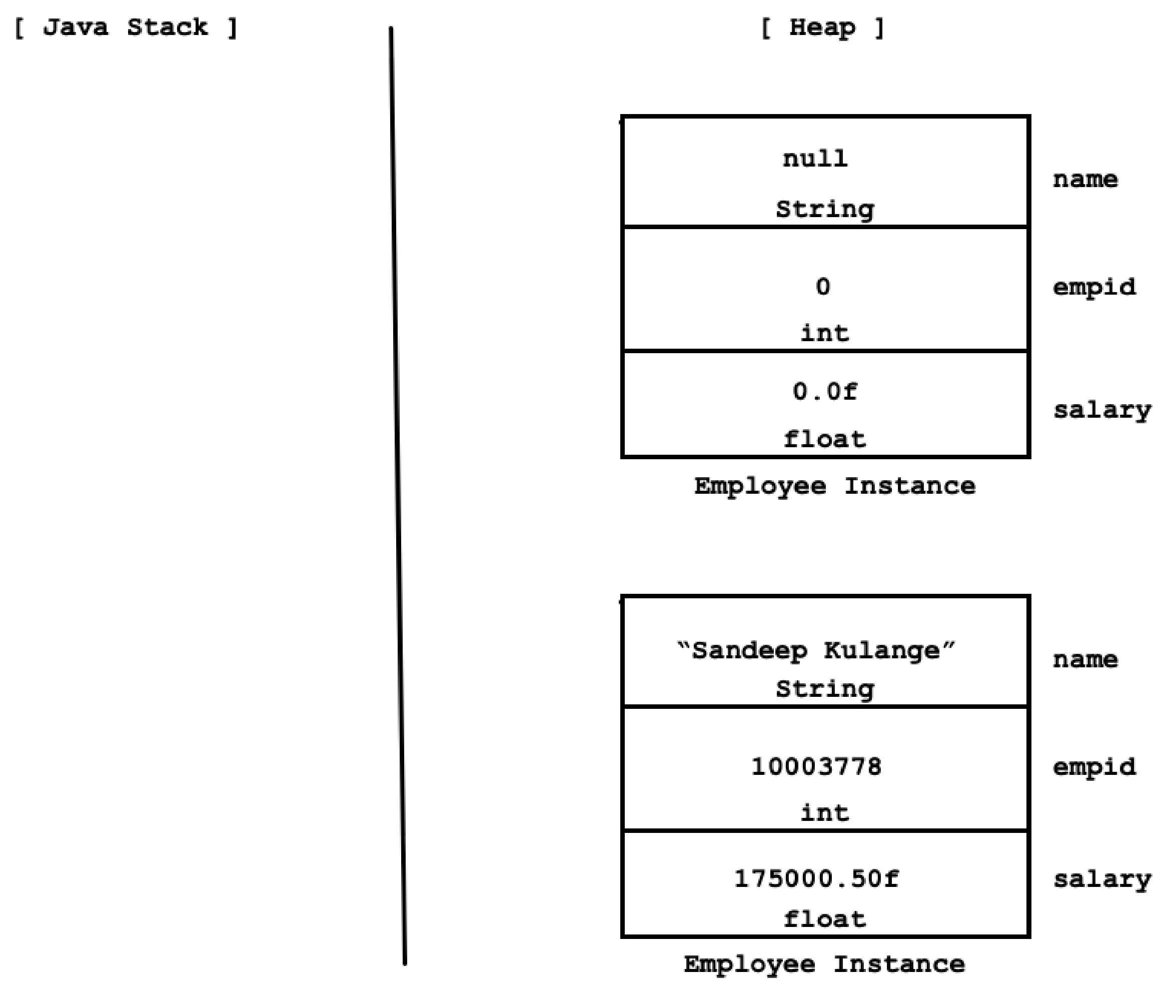
- **Problem Statement:** Write a program to accept and print employee record.
- First analyze problem statement and group related data element(s) together.
  - In Java, to group related data elements together we should use class.
  - class is a keyword in Java.
  - Consider below class definition according to problem statement:

    ```
    class Employee{
      String name;    //Non static field
      int empid;      //Non static field
      float salary;  //Non static field
    };  //Giving semicolon(;) after class definition is optional
    ```

      - **Remember** class definition represents encapsulation in OOPS.
      - If we declare variable inside a method body then it is called as method local variable.
      - If we declare variable(of primitive/non primitive) inside class then it is called as field.
      - Field may be static / non static:
        - Non static field declared inside class is called as instance variable.
          - Non static field declared inside class get space inside instance hence it is called as instance variable.
          - It is designed to access using object reference.
        - Static field declared inside class is called as class level variable.
          - Static field declared inside class do not get space inside instance. Rather all the instances of same class share single copy of it. Hence it is also called as class level variable.
          - We can access it using object reference but it is designed to access using class name.
- To store value inside name, empid and salary, it must get space inside memory.
- Since name, empid and salary are non static field delared inside Employee class, it will get space after creating object/instance of the class.
  - new is operator in java, which is used to create instance of class on heap section of JVM. Consider below code:

    ```
    new Employee( );  //Instance of class Employee
    new Employee( "Sandeep Kulange", 10003778, 175000.50f);  //Instance of class Employee
    ```

      - Process of creating instance from a class is called as instantiation.
      - In above code, class Employee is instantiated.



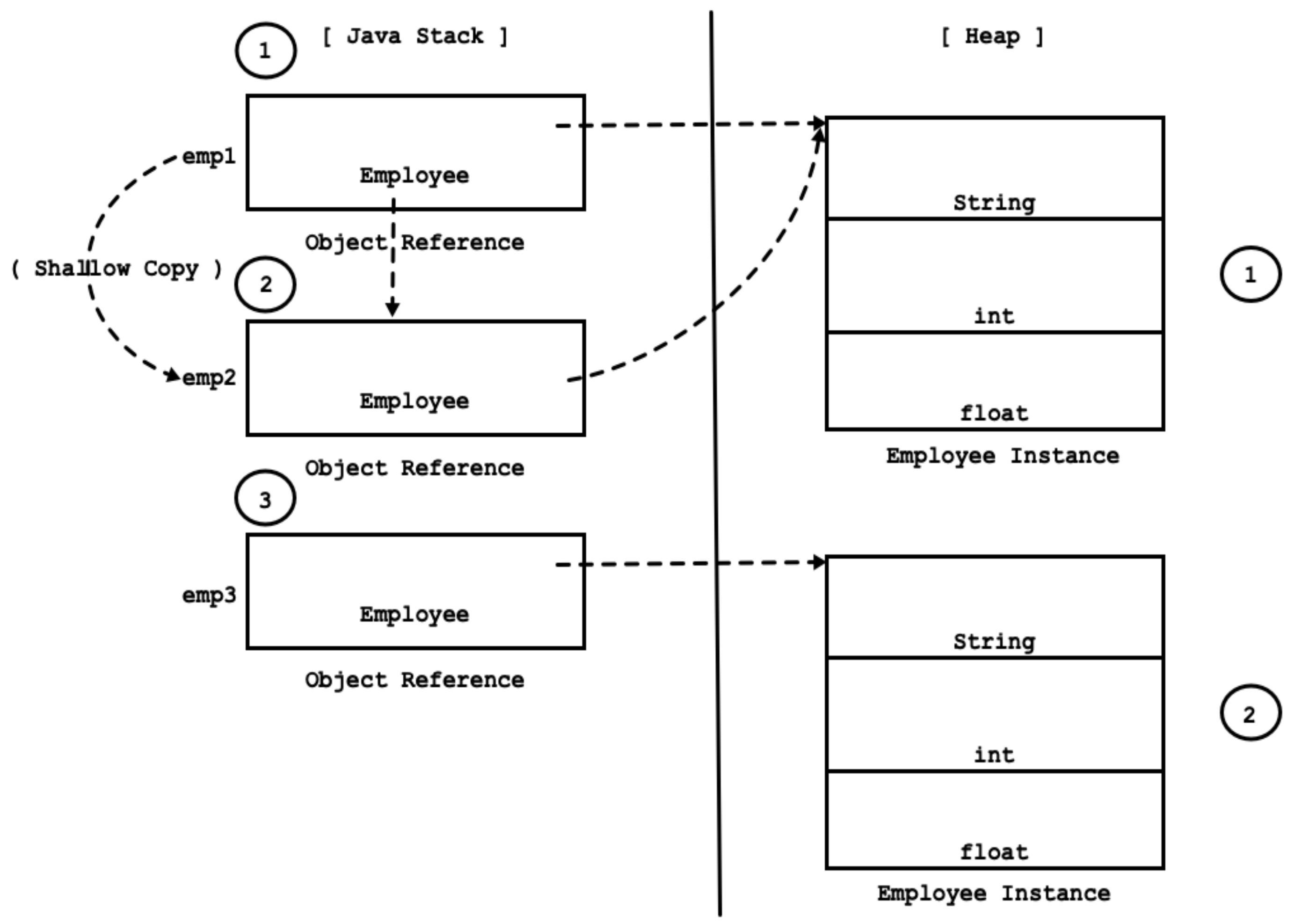
  - In Java, if we create any instance without reference then it is called as anonymous instance.
  - Above instances are anonymous instances.

- If we want to use any instance only once then It should be anonymous.
  - If we want to create instance to pass it as a method argument then it should be anonymous.
  - In case of exception handling, if we want to use instance for throwing then it should be anonymous.
- But if we want to perform some operations on data stored inside instance(i.e field available inside instance) then we should create reference of it.
  - In Java, reference is also called as object reference.
  - We can declare reference as a method local variable / field of the class.
  - Let us discuss how to delare local reference variable:

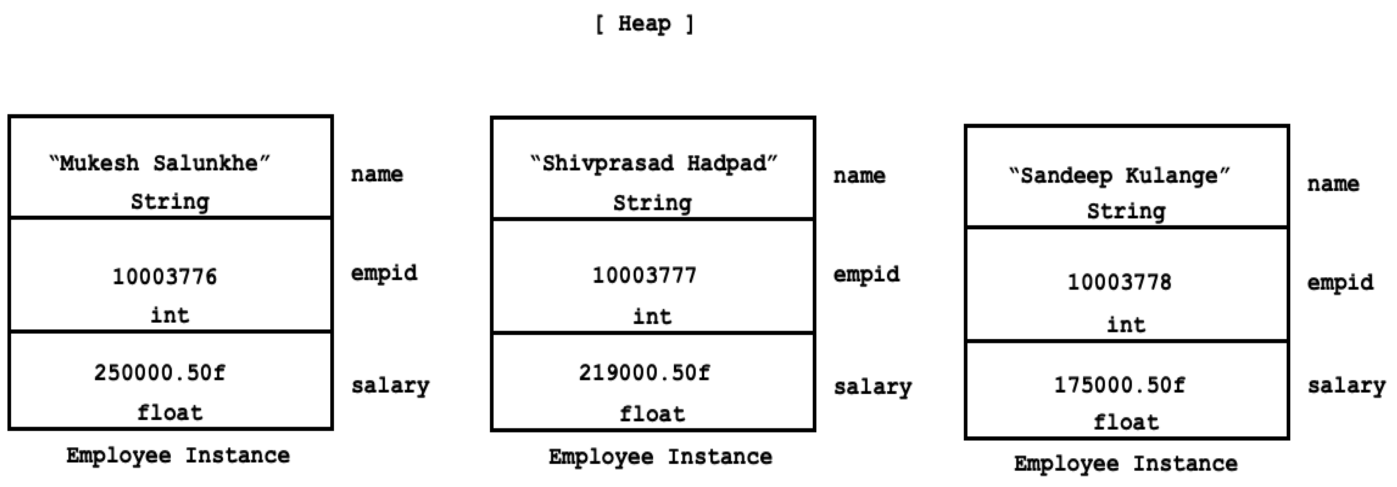
```
Employee emp1;
emp = new Employee( );  //OK
//or
Employee emp2 = new Employee( "Sandeep Kulange", 10003778, 175000.50f); //OK
```

    - In Java, reference is different and instance is different. To create reference, new operator is not require but to create instance new operator is require.
    - Consider Java reference as pointer in C/C++.
    - Instance always get space on Heap section. What about object reference?
      - If reference is method local variable then it gets space in stackframe on Java stack.
      - If reference is non static field of the class then it gets space inside instance on heap section.
      - If reference is static field of the class then it gets space on method area.
  - For more clarity, consider below example. Identify how many instances and how many references?

```
Employee emp1 = new Employee( );
Employee emp2 = emp1;
Employee emp3 = new Employee( );
```



  - **Remember** instance variable get space once per instance. Consider below image



    - Value stored inside instance is called as **state** of that instance.
    - In General, value of the non static field/instance variable represents state of that instance.
- Now, we can access value of instance variable from instance using reference. Consider below code:

```
Employee emp = new Employee( );
emp.name = "Sandeep Kulange"; //OK
emp.empid = 10003778; //OK
emp.salary = 175000.50f;  //OK
```

- Process of giving controlled access to the data is called as **data security**.
- To secure the data first it is necessary to hide that data and to hide data we should use access modifier in Java.
  - There are four access modifiers in Java:
    1. private
    2. package level private
    3. protected
    4. public

	Same Package			Different Package	
Access Modifier	Same Class	Sub Class	Non Sub Class	Sub Class	Non Sub Class
private	A	NA	NA	NA	NA
package level private	A	A	A	NA	NA
protected	A	A	A	A	NA
public	A	A	A	A	A

  - Lets hide the data from outside world. **Note** we are not yet securing data.

```
class Employee{
  private String name;    //Data Encapsulation
  private int empid;      //Data Encapsulation
  private float salary;  //Data Encapsulation
}
```

    - Process of declaring field private is called as data encapsulation.
    - Now we can not access private fields(static as well as non static) outside class directly. Consider below code:

```
Employee emp = new Employee( ); //OK
emp.name = "Sandeep Kulange"; //Not OK
emp.empid = 10003778; //Not OK
emp.salary = 175000.50f;  //Not OK
```

- To process(acceept/print) state of the instance, we should call method on it. Consider below code:

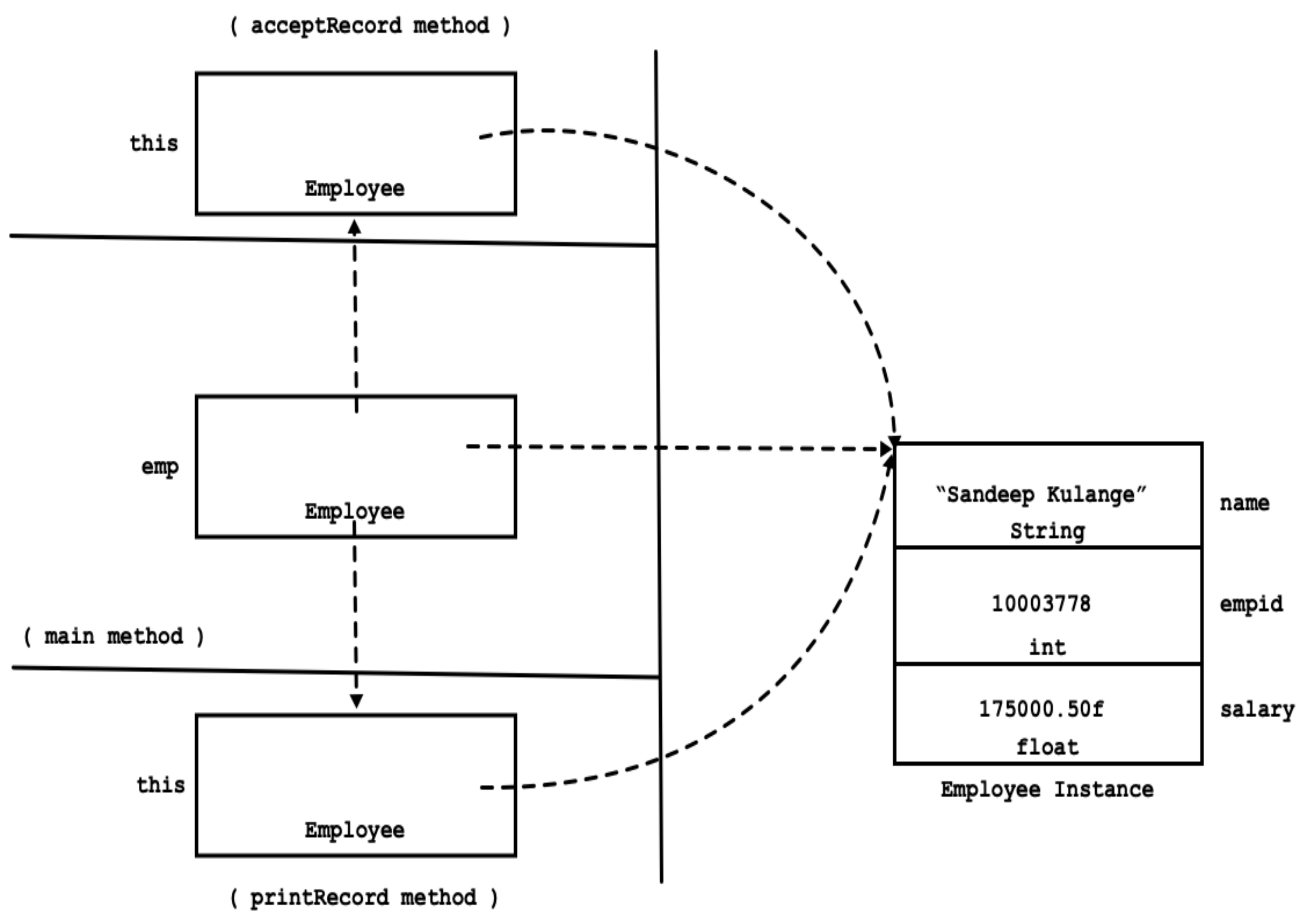
```
Employee emp = new Employee( );
emp.acceptRecord( );  //acceptRecord() method is called on emp  =>  Message Passing
emp.printRecord( );   //printRecord() method is called on emp =>  Message Passing
```

  - Process of calling method on instance(actually object reference) is called as message passing.
  - To call method on instance, first we must define method inside class.
    - Function defined inside class is called method. It can be static or non static.
      - Non static methods are designed to call on instance. Hence it is called as instance method.
      - We can call static methods on instance but it is designed to call on type(class/interface) hence it is called as class level method.
  - Consider below code:

```
class Employee{
  //Fields
  private String name;
  private int empid;
  private float salary;

  //Methods
  void acceptRecord( ){
    //TODO
  }
  void printRecord( ){
      //TODO
    }
  }
```

```
  class Program{
    public static void main( String[] args ){
      Employee emp = new Employee( );
      emp.acceptRecord( );
      emp.printRecord( );
    }
  }
```

- But now question is how to access non static field/instance variable of the instance inside method?
  - If we call method on instance(actually object reference) then compiler implicitly pass reference of the instance as a argument. To catch value of the argument, compiler implicitly declare one parameter is called as this referece.
  - Consider below image for more clarity:



  - We dont need to pass/catch reference explicitly. Compiler implicitly pass reference as well as implicitly declare this reference as a method parameter.
  - Using this reference, we can access instance variable inside method. Consider below code:

```
    class Employee{
      private String name;
      private int empid;
      private float salary;

      void acceptRecord( /* Employee this = emp*/ ){
        Scanner sc = new Scanner( System.in );
        System.out.print("Name  : ");
        this.name = sc.nextLine();
        System.out.print("Empid : ");
        this.empid = sc.nextInt( );
        System.out.print("Salary  : ");
        this.salary = sc.nextFloat();
      }
      void printRecord( /* Employee this = emp*/ ){
        System.out.printf("%-20s%-10d%-10.2f", this.name, this.empid, this.salary);
      }
    }
```

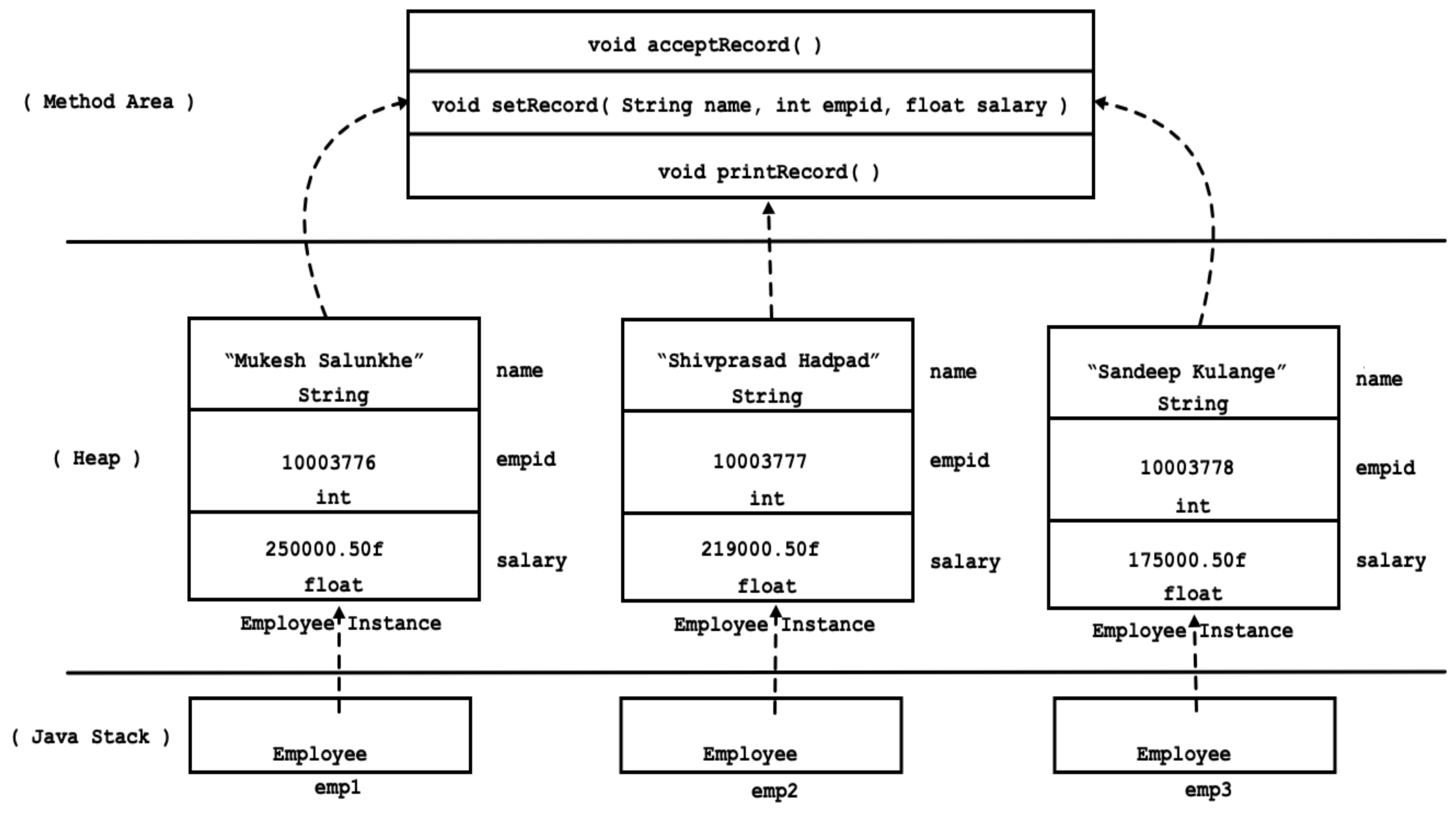
```
    class Program{
      public static void main( String[] args ){
        Employee emp = new Employee( );
        emp.acceptRecord( );
        emp.printRecord( );
      }
    }
```

- **Points to remember:**
  - this is a keyword in Java.
  - Compiler implicitly declare(Programmer need not to declare) this reference as a first implicit parameter in non static method. Static method do not get this reference.(We will discuss it in static topic.)
  - this reference is not a field of the class. It is method parameter of the class. Hence it gets memory space per method call.
  - Using this reference, we can access instance variables of the instance inside method. Hence this reference is considered as a connection / link between instance variable and instance method.
  - Inside method, use of this keyword, before non static members is optional. But in case, name of local variable and name of non static field is same then we must use this before member variable.
  - Consider below code:

```
class Employee{
  private String name;
  private int empid;
  private float salary;

  void setRecord( /* Employee this = emp,*/String name, int empid, float salary ){
    //Here we should use this keyword before non static field
    this.name = name;
    this.empid = empid;
    this.salary = salary;
  }
  void acceptRecord( /* Employee this = emp*/ ){
    //TODO
  }
  void printRecord( /* Employee this = emp*/ ){
    //System.out.printf("%-20s%-10d%-10.2f", this.name, this.empid, this.salary); //OK
    System.out.printf("%-20s%-10d%-10.2f", name, empid, salary);  //OK
  }
}
```

> **Definition:** An implicit reference variable which is available in every non static method of the class and which us used to access instance members(field/method) of the class is called as this reference.

- Method do not get space inside instance. All the instances of same class share single copy of that method.
  - this sharing is done by passing object reference as a argument.
  - Consider below image:



    - Set of / number of operations that we can invoke/call on instance represents behavior the instance.

        - It means, methods defined inside class represents behaviour of the instance.

How to find size of instance in java?

- Reference:
    - https://www.baeldung.com/java-size-of-object
    - https://dzone.com/articles/java-object-size-estimation-measuring-verifying

NullPointerException

- What will be the output of below code?

```
public static void main( String[] args ){
  Employee emp;
  emp.printRecord( );
}
```

- In Java, we can not use any type of local variable without storing value inside it.
- emp is local variable of non primitive type and it isn't containing any value. We are trying to perform operation on uninitialized variable hence compiler will report error.
- emp can contain reference of instance or null.
- null is literal in Java which tails to the compiler that instance creation is pending. Instantiation will done after some time or dyanamically.
- Consider below code:

```
public static void main( String[] args ){
  Employee emp = null;  //OK
  emp.printRecord( ); //No Compilation error
}
```

- If reference variable contains null value then reference variable is called as null reference variable or null object.

```
Employee emp = null;
//null is a literal
//emp is a null object
```

- Using null object, if we try to access any instance variable / instance method then JVM throws NullPointerException
- Consider below code:

```
public static void main( String[] args ){
  Employee emp = null;  //null object
  emp.printRecord( ); //NullPointerException
}
```

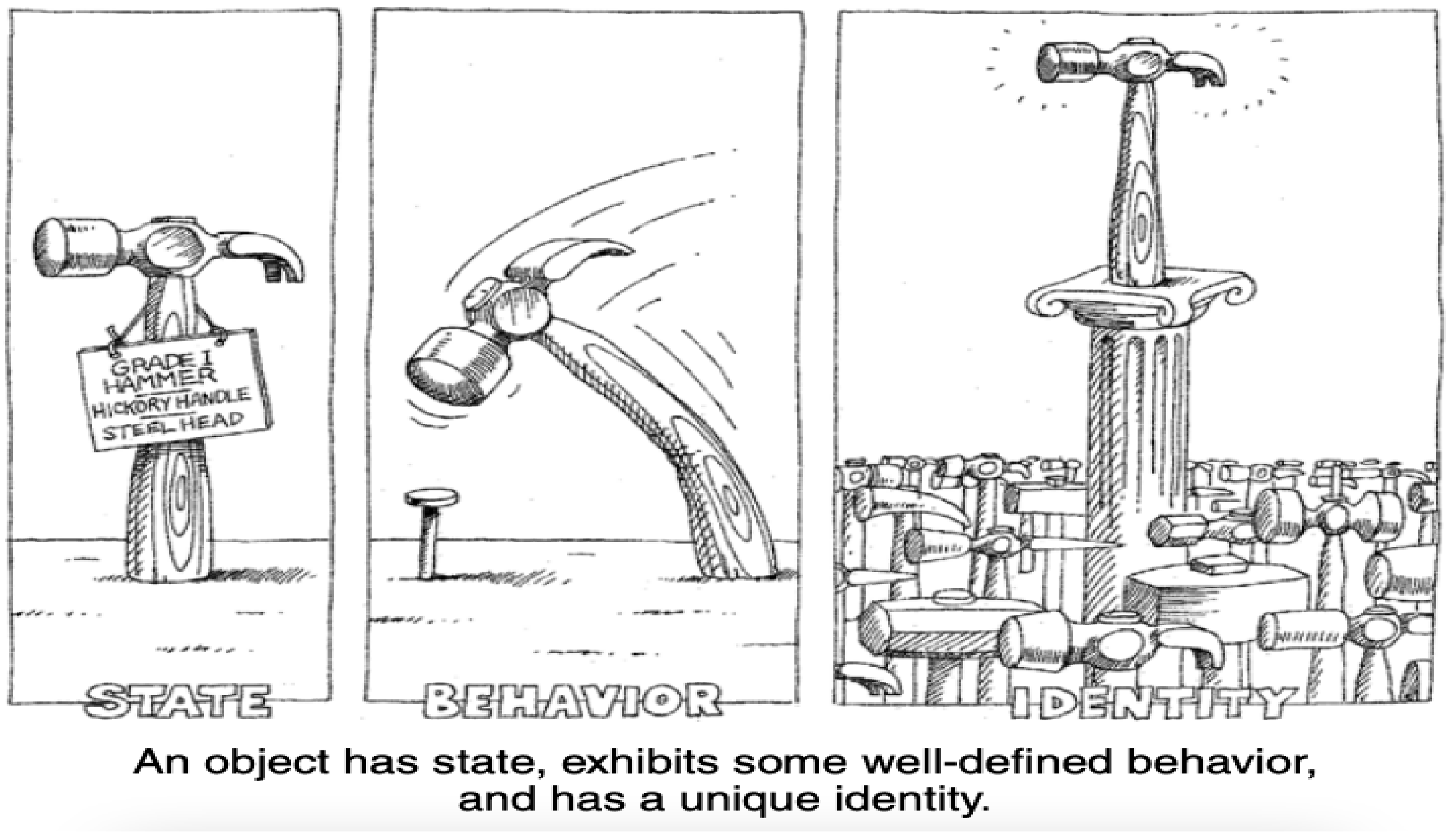
- How to fix it?
- Consider below code:

```
public static void main( String[] args ){
  Employee emp = null;  //null object
  emp = new Employee( );
  emp.printRecord( ); //OK
}
```

Characteristics of instance

- If we create instance of the class then only non static fields declared inside class get space inside it. Hence size of instance depends on size of fields declared inside class.
- In other words, instance variable get space once per instance.
- Method do not get space inside instance. Rather all the instances of same class share single copy of the method.
- Below are the characteristics of instance:
    - State
    - Behavior

- Identity



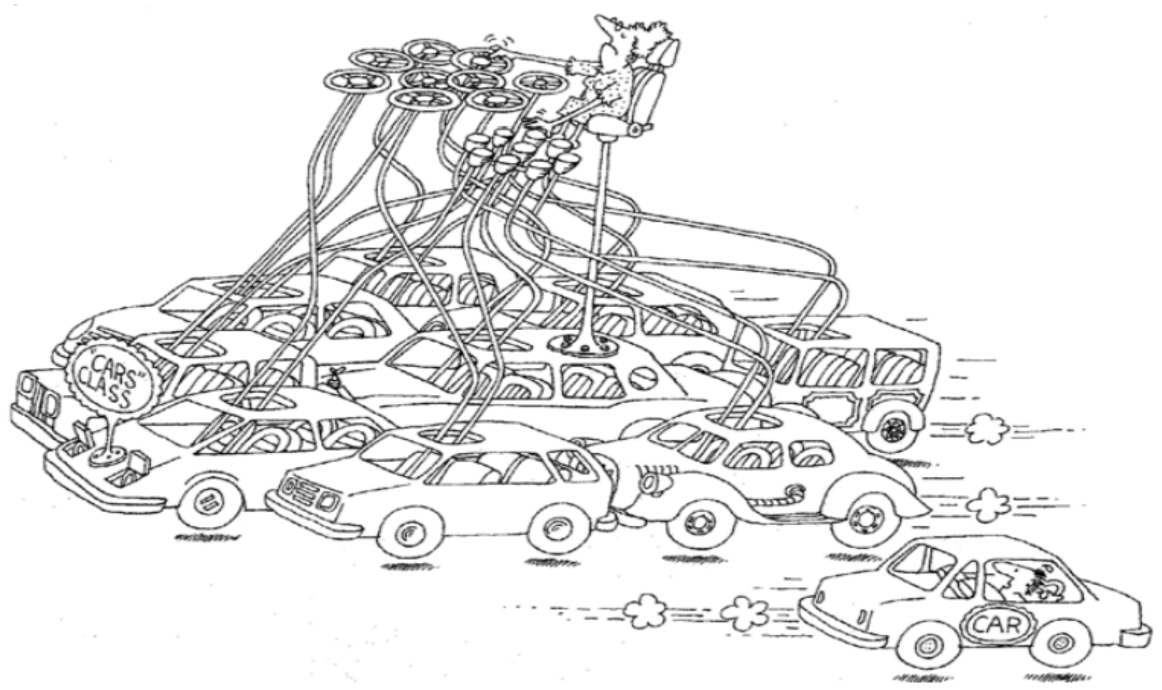
An object has state, exhibits some well-defined behavior, and has a unique identity.

- **State**
  - Value stored inside instance is called as state of the instance. In short, value of the non static field represents state of the instance.
  - Consider example of Car instance: value of fields company, model, color, fuel type, speed, price etc. represents state.
- **Behavior**
  - By sending message to the instance for changing state how instance acts/reacts is called as behavior of the instance.
  - For example, let's consider a car object. It might have methods such as "start," "accelerate," "brake," and "stop." Each of these methods represents a specific action that the car can perform. When the "start" method is called, the car's engine might start, and when the "accelerate" method is called, the car's speed might increase. Similarly, when the "brake" method is called, the car might slow down, and when the "stop" method is called, the car might come to a complete stop.
- **Identity**
  - Identity is that property of an instance which distinguishes it from all other instances.
  - In the context of a car object, the identity might be represented by a unique identifier, such as a Vehicle Identification Number (VIN).

Class versus Instance

Class

- Definition:
  - Class is collection of fields and methods.
  - Structure of instance depends on fields declared inside class and Behavior of instance depends on methods defined inside class. Hence class is considered as a template/model/blueprint for an instance.
  - Class represents collection of such instances which are having common structure and common behavior.
- class is virtual or imaginary entity. Example Car, Book, Laptop, Mobile Phone etc.
- In Java class can contain:
  - Nested Type(Interface/Class/Enum)
  - Field
  - Constructor
  - Method
- **Note:** class implementation represents encapsulation.



Instance

- Definition:
  - An entity which is having physical existance is called as instance.
  - An entity which is having state, behavior and identity is called as instance.
- Instance is real time entity. Example "Skoda Kushaq", "The Secret", "MacBook Air", "iPhone 15 Pro" etc.
- In Java, only non static field get space inside instance.

Addition of instances

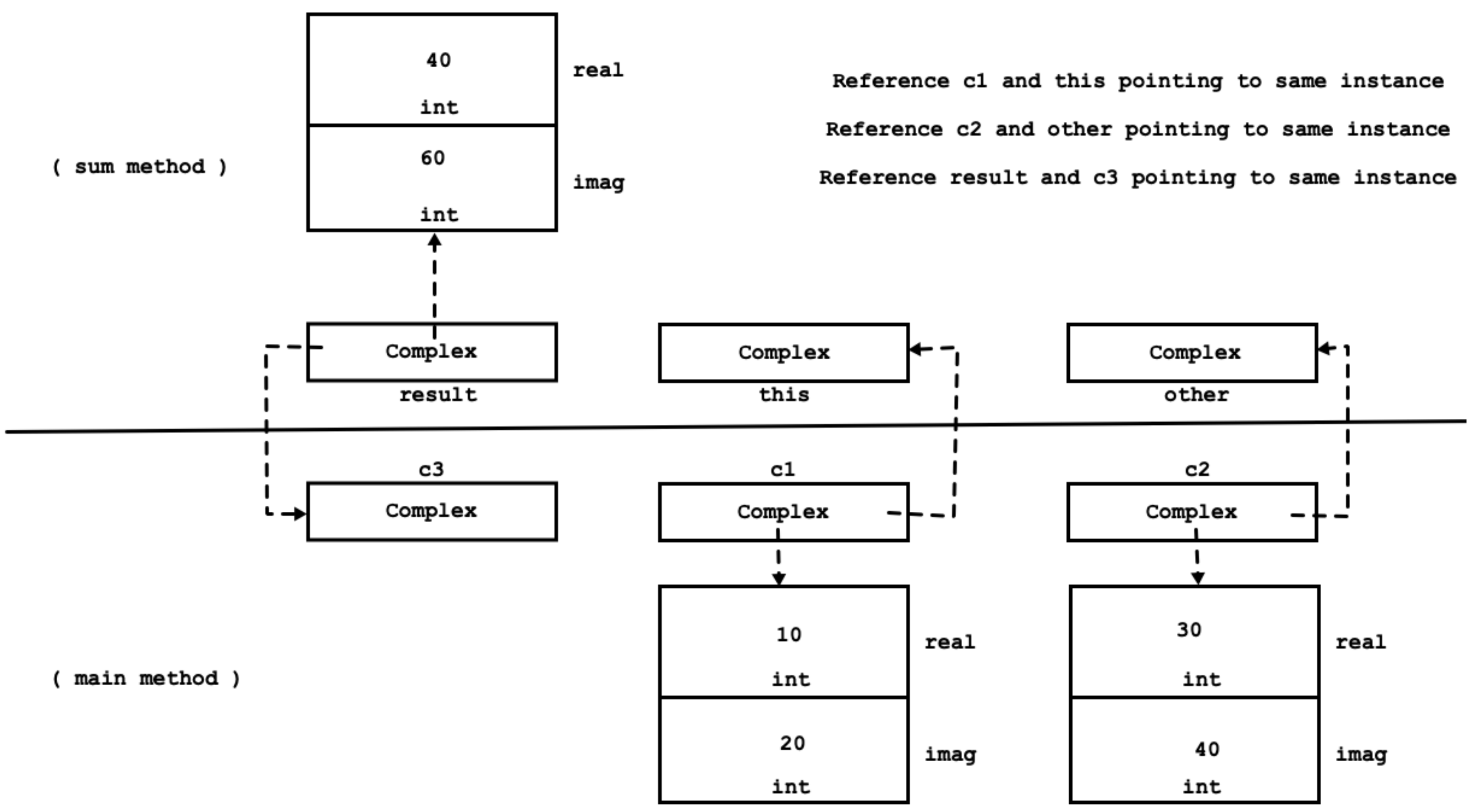
- Consider method call:

```
  public static void main( String[] args ){
      Complex c1 = new Complex( );
      c1.acceptRecord( ); //10, 20

      Complex c2 = new Complex( );
      c2.acceptRecord( ); //30, 40

      Complex c3 = c1.sum( c2 );
      c3.printRecord( );  //40, 60
  }
```

- Consider method definition:

```
class Complex{
  private int real;
  private int imag;
  //Complex this = c1;
  //Complex other = c2
  Complex Sum( Complex other ){
    Complex result = new Complex( );
    result.real = this.real + other.real;
    result.imag = this.imag + other.imag;
    return result;
  }
}
```

- Consider memory representation:



Difference between Value Type and Reference Type

Sr.No.	Value Type	Reference Type
1	Primitive type is also called as value type.	Non primitive type is also called as reference type.
2	boolean, byte, char, short, int, float, double, long are primitive / value types in Java.	Interface, class, enum and array are non primitive / reference types in Java.
3	Variable of value type contains value.	Variable of reference type contains reference.
4	In case of initialization / assignment value gets copied.	In case of initialization / assignment reference gets copied.
5	For the field, default value of primitive /value type variable is zero.	For the field, default value of non primitive /refrence type variable is null.
6	Variable of primitive / value type do not contain null value.	Variable of non primitive / reference type can contain null value
7	To create variable of primitive / value type new operator is not required.	To create instance of non primitive type / reference type new operator is required.
8	Variable of value type get space on Java Stack	Instance of reference type get space on Heap.

Method Overloading

- Consider below examples:

```
int a = 10;
int b = 20;
int c = a + b;
```

```
double x = 10.1d;
double y = 20.2d;
double z = x + y;
```

```
String s1 = "sandeep";
String s2 = "kulange";
String s3 = s1 + s2;
```

- To perform operations on integer, double and String, we have used same operator i.e +.
- Same Concept we can use it in programming too. Consider below example:

```
private static int sum( int num1, int num2 ){
  int result = num1 + num2;
  return result;
}
```

```
private static int sum( double num1, double num2 ){
  double result = num1 + num2;
  return result;
}
```

- **Concept:** If implementation of method is logically equivalant or same then we should overload the method i.e. we should give same name to that method.
- Consider below points while overloading method:
    - If name of the methods and type of parameters passed to the method are same then number of arguments passed to the method must be different.

```
public class Program {
  private static int sum(int num1, int num2) {  //Method parameters are 2
    return num1 + num2;
  }

  private static int sum(int num1, int num2, int num3) {  //Method parameters are 3
    return num1 + num2 + num3;
  }

  public static void main(String[] args) {
    System.out.println("Sum   :   " + sum(10, 20));

    System.out.println("Sum   :   " + sum(10, 20, 30));
  }
}
```

    - If name of the methods and number of parameters passed to the method are same then type of at least one parameter must be different.

```
public class Program {
  private static int sum(int num1, int num2) {  //Method parameter types are int,int
    return num1 + num2;
  }

  private static double sum(int num1, double num2) {  //Method parameter types are int,double
    return num1 + num2;
  }

  public static void main(String[] args) {
    System.out.println("Sum   :   " + sum(10, 20));

    System.out.println("Sum   :   " + sum(10, 20.1d));
  }
}
```

    - If name of the methods and number of parameters passed to the method are same then order of type of parameters must be different.

```
public class Program {
  private static float sum(int num1, float num2) {  //Method parameter types are int,float
    return num1 + num2;
  }

  private static float sum(float num1, int num2) {  //Method parameter types are float,int
    return num1 + num2;
  }

  public static void main(String[] args) {
    System.out.println("Sum   :   " + sum(10, 20.2f));

    System.out.println("Sum   :   " + sum(10.1f, 20));
  }
}
```

    - Only on the basis of different return type we can not give same name to the method i.e. we can not overload method. Below example is not a example of method overloading.

```
  public class Program {
    private static float sum(int num1, int num2) {  //Error: Duplicate method sum(int, int) in type
Program
      float result = num1 + num2;
      return result;
    }

    private static double sum(int num1, int num2) { //Error: Duplicate method sum(int, int) in type
Program
      double result = num1 + num2;
      return result;
    }

    public static void main(String[] args) {
      System.out.println("Sum   :   " + sum(10, 20));

      System.out.println("Sum   :   " + sum(10, 20));
    }
  }
```

    - Only on the basis of different access modifier, we can not give same name to the method.

- Process of defining method with same name and different signature is called as method overloading. Methods which take part in method overloading are called as overloaded method.
- Example:
    - In java.lang.Object class, wait() method is overloaded method.
    - In java.lang.String class, valueOf() method is overloaded method.
    - In java.io.PrintStream class print(), println() and printf() are overloaded methods.
    - In above examples, sum() method is overloaded method.
- In Java, it is not required but we can overload main method:

```
public class Program {
  public static void main( ) {
    System.out.println("Hello from overloaded main method");
  }
  public static void main(String[] args) {
    main();
  }
}
```

Constructor

- Initialization is the process of storing value inside variable during its declaration. Consider below example:

```
int number = 123; //Initialization
```

- In a given scope and in its life time, we can initialize any variable only once.
- Assignment is the process of storing value inside variable after its declaration / initialization. Consider below example:

```
int num1;
int num2 = 10;  //Initialization

num1 = 20;  //Assignment
num2 = 30;  //Assignment

num1 = 40;  //Assignment
num2 = 50;  //Assignment
```

- Consider below code and understand the problem of init style method:

```
class Complex{
  private int real;   //Default value is 0
  private int imag;   //Default value is 0
  public void initComplex( ) {
      this.real = -1;
      this.imag = -1;
    }
    public void acceptRecord( ) {
      Scanner sc = new Scanner(System.in);
      System.out.print("Real Number :   ");
      this.real = sc.nextInt();
      System.out.print("Imag Number :   ");
      this.imag = sc.nextInt();
    }
    public void printRecord( ) {
      System.out.println(this.real+","+this.imag);
    }
  }
  public class Program {
    public static void main(String[] args) {
      Complex c1 = new Complex();
      c1.printRecord(); //0,0

      c1.initComplex();
      c1.printRecord(); //-1,-1

      c1.acceptRecord();    //10 20
      c1.printRecord(); //10,20

      c1.initComplex();
      c1.printRecord(); //-1,-1
    }
  }
```

- In Java, constructor is a method like syntax which is used to initialize instance.
  - new is operator which is used to create instance on heap.
  - Constructor do not create instance. It only initializes instance.
- Consider example of constructor:

```
  class Complex{
    private int real;   //Default value is 0
    private int imag;   //Default value is 0

    public Complex( ) {   //Constructor
      this.real = -1;
      this.imag = -1;
    }
  }
```

- Constructor do not get call on reference. It gets called on instance that too implicitly.
- Due to below reasons, constructor is considered as special in Java:
  - Its name is always same as class name.
  - It doesn't have any return type.
  - It gets called implicitly. (No need to call it on instance explicitly.)
  - In the lifetime of instance it gets called only once.
- **Remember:** constructor gets called once per instance.
- We can use any access modifier on constructor. If constructor is public then we can create it's instance inside method of same class as well as different class. Consider below code.

```
  class Complex{
    private int real;
    private int imag;
    public Complex( ) {
      this.real = 10;
      this.imag = 20;
    }
    public void printRecord( ) {
      System.out.println(this.real+","+this.imag);
    }
    public static void displayRecord( ) {
      Complex c1 = new Complex();   //OK
      System.out.println(c1.real+","+c1.imag);  //OK
    }
  }
  public class Program {
    public static void main(String[] args) {
      Complex.displayRecord();

      Complex c2 = new Complex( );
      c2.printRecord();
    }
  }
```

- If constructor is private then we can create it's instance inside method of same class but not in different class. Consider below code:

```
  class Complex{
    private int real;
    private int imag;
    private Complex( ) {
      this.real = 10;
      this.imag = 20;
    }
    public void printRecord( ) {
      System.out.println(this.real+","+this.imag);
    }
    public static void displayRecord( ) {
      Complex c1 = new Complex();   //OK
      System.out.println(c1.real+","+c1.imag);  //OK
    }
  }
  public class Program {
    public static void main(String[] args) {
      Complex.displayRecord();

      //Complex c2 = new Complex( );    //Not OK
    }
  }
```

- **Note:** this concept we will require for defining singleton class.
- On instance, we can not call constructor explicitly:

```
  public class Program {
    public static void main(String[] args) {
      Complex c1 = new Complex( );  //OK

      c1.Complex( );    //Error: The method Complex() is undefined for the type Complex
    }
  }
```

- **Remember** constructor calling sequence depends on order of instance creation.

```
  public static void main( String[] args ){
    Complex c1 = new Complex( );
    Complex c2 = new Complex( );
    Complex c3 = new Complex( );
  }
```

  - Here constructor will first call for c1,c2 and then c3
- In Java there is no destructor. But you will get equivalent "protected void finalize() throws Throwable" method.
- **Note:** Below modifiers are not allowed on constructor in Java:
  - static
  - final
  - abstract
  - native
  - synchronized
  - strictfp
- Reference: https://docs.oracle.com/javase/tutorial/java/javaOO/constructors.html

Types Of Constructor

- There are three types of constructor in Java:
    - Parameterless constructor (It is user user defined constructor.)
    - Parameterized constructor
    - Default constructor (It is compiler generated parameterless constructor.)
- **Parameterless constructor**
    - We can define constructor without any parameter. Such constructor is called as parameterless / user defined default constructor. Consider below syntax:

```
class Complex{
  private int real;
  private int imag;

  public Complex( ){  //Parameterless constructor
    this.real = -1;
    this.imag = -1;
  }
}
```

    - If we try to create instance without passing arguments then parameterless constructor gets called. Consider below code:

```
Complex c1; //On reference constructor never gets called
new Complex( ); //Anonymous instance. Here on instance, parameterless constructor will call.
Complex c2 = new Complex( ); //Here also, on instance, parameterless constructor will call.
```

- **Parameterized constructor**
    - We can define constructor with parameter(s). Such constructor is called as parameterized constructor. Consider below code:

```
class Complex{
  private int real;
  private int imag;

  public Complex( int real, int imag ){  //Parameterized constructor
    this.real = real;
    this.imag = imag;
  }
}
```

    - If we try to create instance by passing arguments then parameterized constructor gets called. Consider below code:

```
new Complex( 10, 20); //Anonymous instance. Here on instance, parameterized constructor will call.
Complex c1 = new Complex( ); //Here also, on instance, parameterized constructor will call.
Complex c2 = new Complex( );  //Error. Class do not contain parameterless constructor.
```

    - We can define parameterless as well as parameterized constructor inside class. It is called as constructor overloading and constructors are called overloaded constructors.

```
class Complex{
  private int real;
  private int imag;

  public Complex(  ){  //Parameterless constructor
    this.real = -1;
    this.imag = -1;
  }

  public Complex( int real, int imag ){  //Parameterized constructor
    this.real = real;
    this.imag = imag;
  }
}
```

- **Default constructor**
  - If we do not define any constructor(no parameterless & no parameterized constructor) inside class then compiler provide one constructor for the class by default. It is called as default constructor.
  - In short, when we say deafult constructor, It is compiler provided parameterless constructor.
  - Consider below code:

```
class Complex{
  private int real;
  private int imag;
}
```

    - For above class, compiler will provide default constructor.
    - **Note:** compiler never provide default parameterized constructor. It provides only default parameterless constructor.
  - Consider below Code:

```
Complex c1 = new Complex( );  //Here on instance, default constructor will call.
Complex c2 = new Complex( 10, 20 ); //Compiler Error. Default paramerized constructor does not exist
```

  - **Remember** If class contains only parameterized constructor then compiler never generate default constructor for the class. In this case defining parameterless constructor is programmers responsibility. Consider below code:

```
class Complex{
  private int real;
  private int imag;

  public Complex( int real, int imag ){  //Parameterized constructor
    this.real = real;
    this.imag = imag;
  }
}
```

```
Complex c1 = new Complex( 10, 20 ); //OK
Complex c2 = new Complex( );  //Error.
//Since class contains parameterized constructor, default constructor will not be available for the
class.
```

Constructor Chaining

- To reuse, initialization logic, we can call constructor from another constructor. It is called as constructor chaining.
- Dont confuse. We have already learned, we can not call constructor on instance explicitly and it is true but it is possible to give call to the construcor from another constructor.
- For constructor chaining, we should use "this statement". Consider below code:

```
class Complex{
  private int real;
  private int imag;

  public Complex(  ){
    //Constructor chaining
    this(-1, -1); //Call to parameterized constructor
  }
  public Complex( int real, int imag ){
    this.real = real;
    this.imag = imag;
  }
}
```

- **Remember** "this statement" must be the first statment inside constructor body.

```
class Complex{
  private int real;
  private int imag;

  public Complex(  ){
    System.out.println("Inside parameterless constructor.");
    this(-1, -1); //Not OK: It must be first statement inside constructor body.
  }
  public Complex( int real, int imag ){
    System.out.println("Inside parameterized constructor.");
    this.real = real;
    this.imag = imag;
  }
}
```

- **Note** we cannot write "this statement" in any method. We can write it inside constructor body only.

Instance Field Initializer versus **Instance Initializer/Initialization Block** versus **Constructor**

- Consider example of instance field initializer:

```
class Stack{
  private int top = -1; //instance field initializer
}
```

  - Instance field initializers are expressions that are directly assigned to instance variables at the time of declaration.
  - They are executed when an instance of the class is created and before any constructor is called.
  - They are part of the variable declaration itself.
  - They are executed in the order they are defined in the class.
  - They are concise and provide a convenient way to initialize instance variables with constant values or expressions

- Consider examplex of instance initialization block:

```
class Stack{
  private int top;

  {//Instance initialization block
    this.top = -1;
  }
}
```

  - Instance initializer blocks are code blocks enclosed in curly braces {} that are directly placed within a class, outside of any method or constructor.
  - They are executed when an instance of the class is created, after instance field initializers and before any constructor is called.
  - They provide a way to initialize instance variables with more complex or dynamic initialization logic, including statements and code blocks.
  - They are executed in the order they appear in the class, after instance field initializers.
  - They are particularly useful when initialization logic cannot be expressed using a single expression.

- Consider example of constructor:

```
class Stack{
  private int top;
  public Stack( ){
    this.top = -1;
  }
}
```

- Consider example with instance field initializer, initialization block and constructor:

```
public class Person {
  private String name;
  private int age;
  private boolean isAdult = false;  //// Instance field initializer

  { // Instance initializer block
      if (age >= 18) {
          isAdult = true;
      }
```

```
    }

    public Person( ){
      //TODO
    }
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    // Getter and Setter methods
}
```

```
  class Program{
    public static void main(String[] args) {
        Person person = new Person("Sandeep Kulange", 40);

        System.out.println("Name: " + person.getName());
        System.out.println("Age: " + person.getAge());
        System.out.println("Is Adult: " + person.isAdult());
    }
  }
```

- Reference:
  - https://docs.oracle.com/javase/tutorial/java/javaOO/initial.html
  - https://blogs.oracle.com/javamagazine/post/java-instance-initializer-block