Hierarchy

Major and minor elements / pillars /parts of oops

Abstraction

- It is a major pillar of oops.
- Process of getting essential things from instance / system is called as abstraction.
- Abstraction focuses on external behavior of the instance.
- Abstraction changes from entity to entity.
- Abstraction helps to achive simplicity.
- In Java, if we want to achieve abstraction then we should create instance and invoke method on it.
- For Example:

```
class Program{
  public static void main(String[] args) {
    Complex c1 = new Complex( );
    c1.acceptRecord( );
    c1.printRecord( );
  }
```

Encapsulation

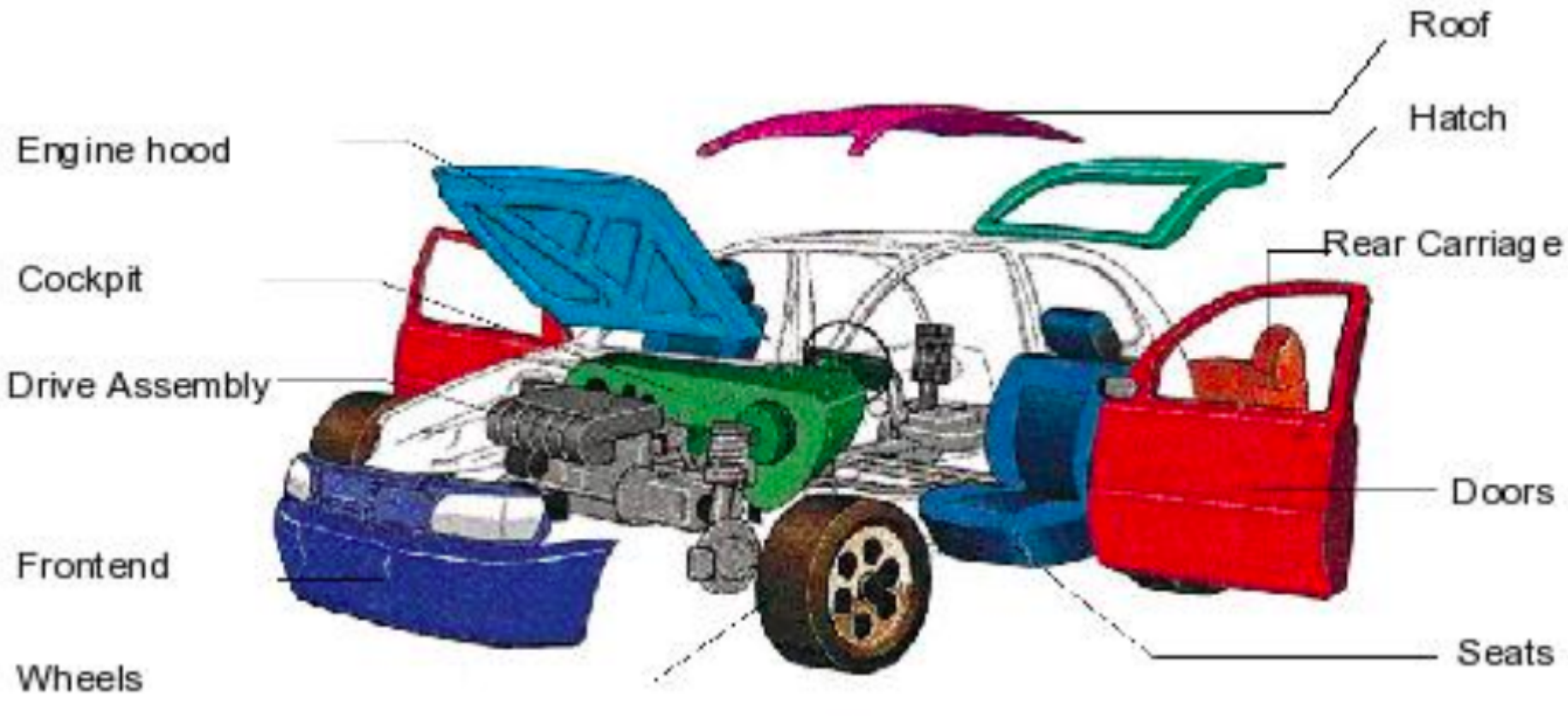
- Definition:
  - Binding of data and code together is called as encapsulation.
  - Implementation of abstraction is called as encapsulation.
- Abstraction and encapsulation are complementry concepts. In other words abstraction focuses on external behavior whereas encapsulation focuses internal behavior.
- Using encapsulation:
  - We can achieve abstraction
  - We can hide data from user:
    - Process of declararing fields private is called as data hiding. Data hiding is also called as data encapsulation.
    - Data hiding helps us to achive data security.
- To achive encapsulation we should define class. Hence class is considered as a basic unit of encapsulation.
- For Example:

```
class Complex{
  private int real;
  private int imag;
  public Complex( ){
    this.real = 0;
    this.imag = 0;
  }
  public void acceptRecord( ){
      //TODO
  }
  public void printRecord( ){
      //TODO
  }
}
```

Modularity

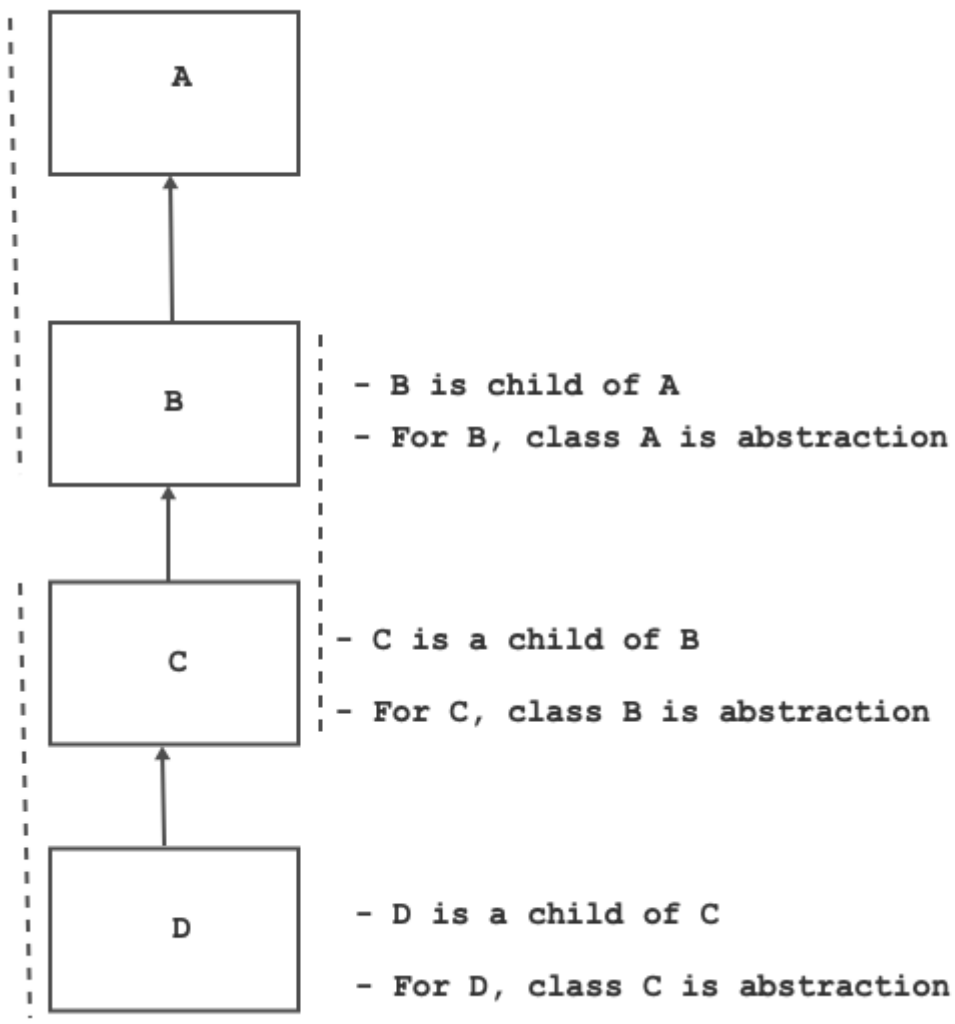
- It is a major pillar of oops.
- It is the process of developing complex system using small parts.
- Modularity helps us to minimize module dependency.

- In Java, we can achieve Modularity using packages and .jar file.



Hiearchy

- It is a major pillar of oops.
- Level / order / ranking of abstraction is called as hierarchy.



- Using hierarchy, we can achieve code reusability.
  - Development time will be reduce
  - Development cost will be reduce
  - Developers effoort will reduce
- Types of hierarchy
  - Has-a hierarchy
    - Also called as part-of hierarchy.
    - It represents Association.
    - There are 2 specialized form of association:
      - Compositon
      - Aggregation
  - Is-a hierarchy
    - It is also called as Kind-of hierarchy.
    - It represents generalization.
    - Generalization is also called as inheritance.
    - Types of Inheritance:

        - Implementation inheritance
            - Single level inheritance : (Allowed in Java)
            - Multiple inheritance : (Not Allowed in Java)
            - Hierarchical level inheritance : (Allowed in Java)
            - Multilevel inheritance : (Allowed in Java)
        - Interface inheritance
            - Single level inheritance : (Allowed in Java)
            - Multiple inheritance : (Allowed in Java)
            - Hierarchical level inheritance : (Allowed in Java)
            - Multilevel inheritance : (Allowed in Java)
    - Use-a hierarchy
        - It represents dependency.
    - Creates-a hierarchy
        - It represents instantiation.

Typing

- It is a minor pillar of oops.
- Typing is also called as Polymorphism.
- It comes from the greek roots "poly" (many) and "morphe" (form).
- An ability of an instance to take multiple forms is called as Polymorphism.
- If we want to reduce maintainence of the systen then we should use Polymorphism.
- Types of Polymorphism:
    - Compile time Polymorphism
        - We can achieve it using method overloading
    - Runtime Polymorphism
        - In Java, it is also called as dynamic method dispatch.
        - We can achieve it using method overriding.

Concurrency

- It is a minor pillar of oops.
- It is the process of executing multiple task simultaneously.
- Main purpose of concurrency is to utilize h/w resources efficiently.
- In Java we achieve concurrency using multithreading.

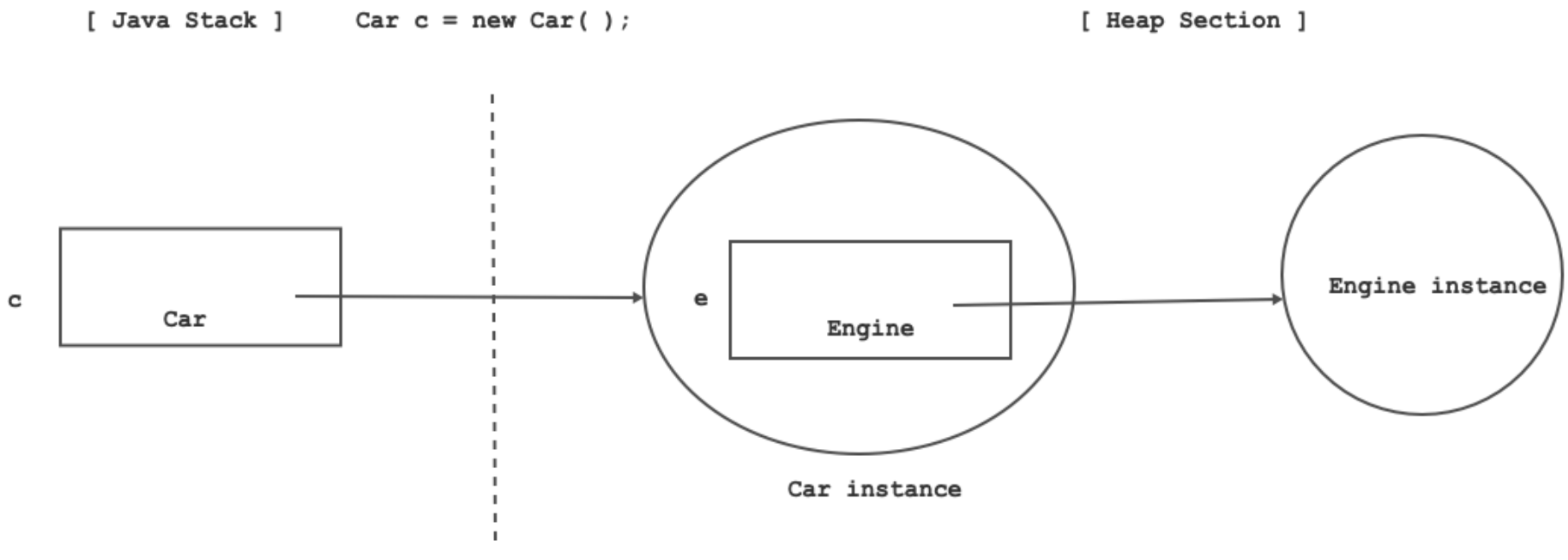
Persistence

- It is a minor pillar of oops.
- It is process of maintaining state of instance of secondary storage(HDD / DB).
- In Java, we can achieve Persistence using serialization , JDBC etc.
- Persistence helps to achieve reliablity

Association

- If has-a relationship is exist between the type then we should use association.
- Consider following examples:
    - Car has-a engine / wheel.
    - Car has-a music system.
    - Room has-a wall/window
    - Room has-a chair
    - System unit has-a motherboard
    - System unit has-a modem
- Has-a hierarchy is also called as part-of hierarchy.
- Consider example of Car and Engine
    - Car has-a engine
    - Engine is part of car
- When onject/instance is part/component of another object/instance then it is called as association.
- Consider following example:

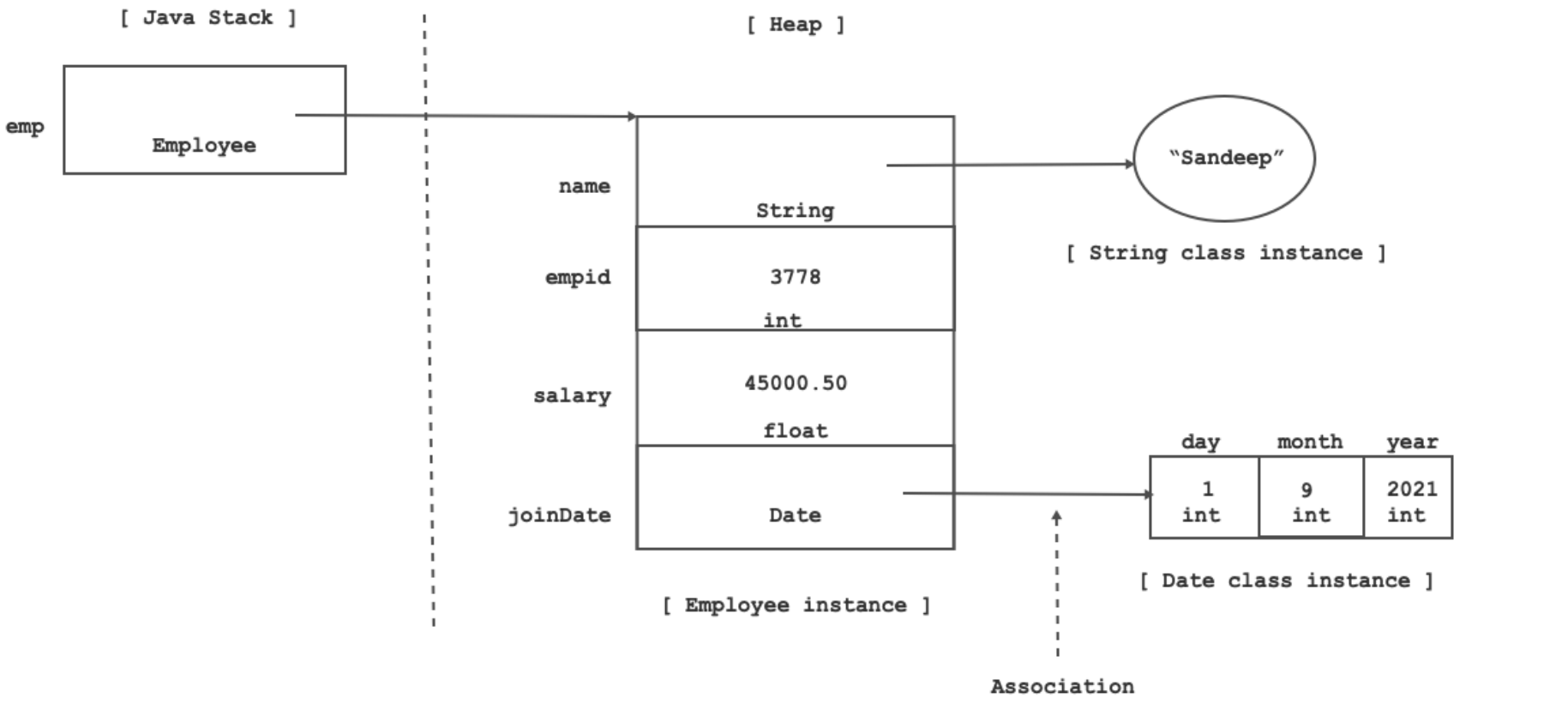
```
class Engine{
  //Fields
  //Constructors
  //Methods
}
class Car{
  //Car has a engine
  private Engine e = new Engine();  //Association
}
```



    - Car Instance is dependent instance
    - Engine instance dependency instance
- If we declare instane of a class as a field inside another class then it is considered as association.
- In Java, association do not represent physical containment. In other words, instance can not be part of another instance directly. Association is maintained through reference variable.

```
Class Date{
   //TODO: Member declaration
}
class Employee{
   String name;
   int empid;
   float salary;
   Date joinDate;
}
```

```
class Program{
   public static void main( String[] args ){
      Employee emp = new Employee("Sandeep",3778,45000.50f,new Date(1,9,2021));
      //TODO
   }
}
```



Composition

- Consider example of car and wheel & engine
    - Car has a wheel

```
class Wheel{
  //TODO
}
class Engine{

}
class Car{
  private Wheel[] wheels; //Association --> Composition
  private Engine e; //Association --> Composition
}
```

        - Car instance is dependent instance
        - Wheel & Engine instance is dependency instance

- In case of association, if dependency instance can not exist without dependent instance then it reprsents composition.
- Composition represents tight coupling.

Aggregation

- Consider example of University and Student

```
class Student{
  //TODO
}
class University{
  private Stundet[] students; //Association --> Aggregation
  //TODO
}
```

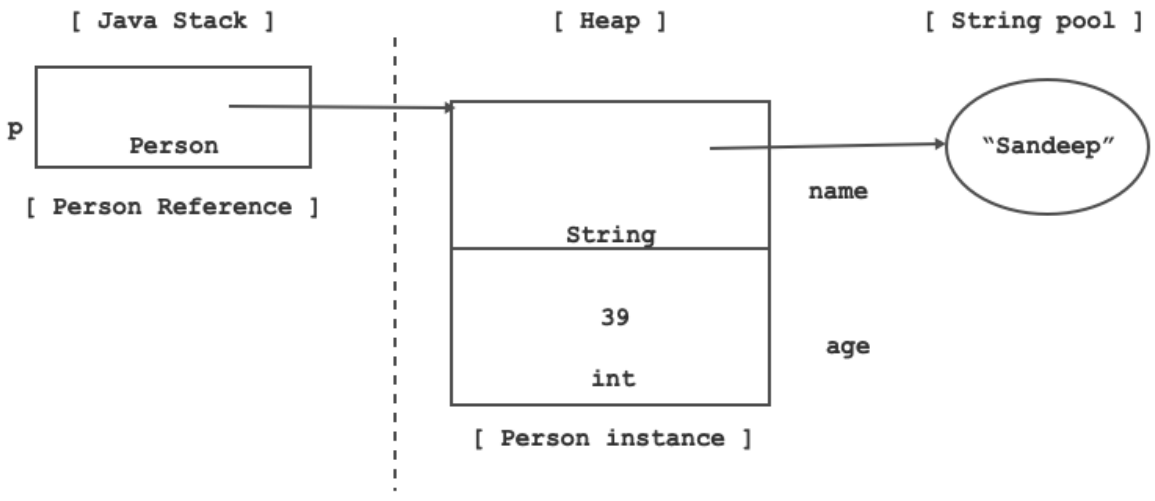
  - University instance is dependent instance.
  - Student instance is dependency instance.
- In case of association, if dependency instance exist without dependent instance then it represents aggregation.
- Aggregation represents loose coupling.

Inheritance

- Consider Person class

```
class Person{
  private String name;
  private int age;
  public Person( ) {
    this( "", 0); //Constructor chaining
  }
  public Person( String name, int age ) {
    this.name = name;
    this.age = age;
  }
  public void showRecord( ) {
    System.out.println("Name  :   "+this.name);
    System.out.println("Age   :   "+this.age);
  }
}

public class Program {
  public static void main(String[] args) {
    //Person p = new Person( );
    Person p = new Person( "Sandeep", 39 );
    p.showRecord( );
  }
}
```



- Consider Employee class

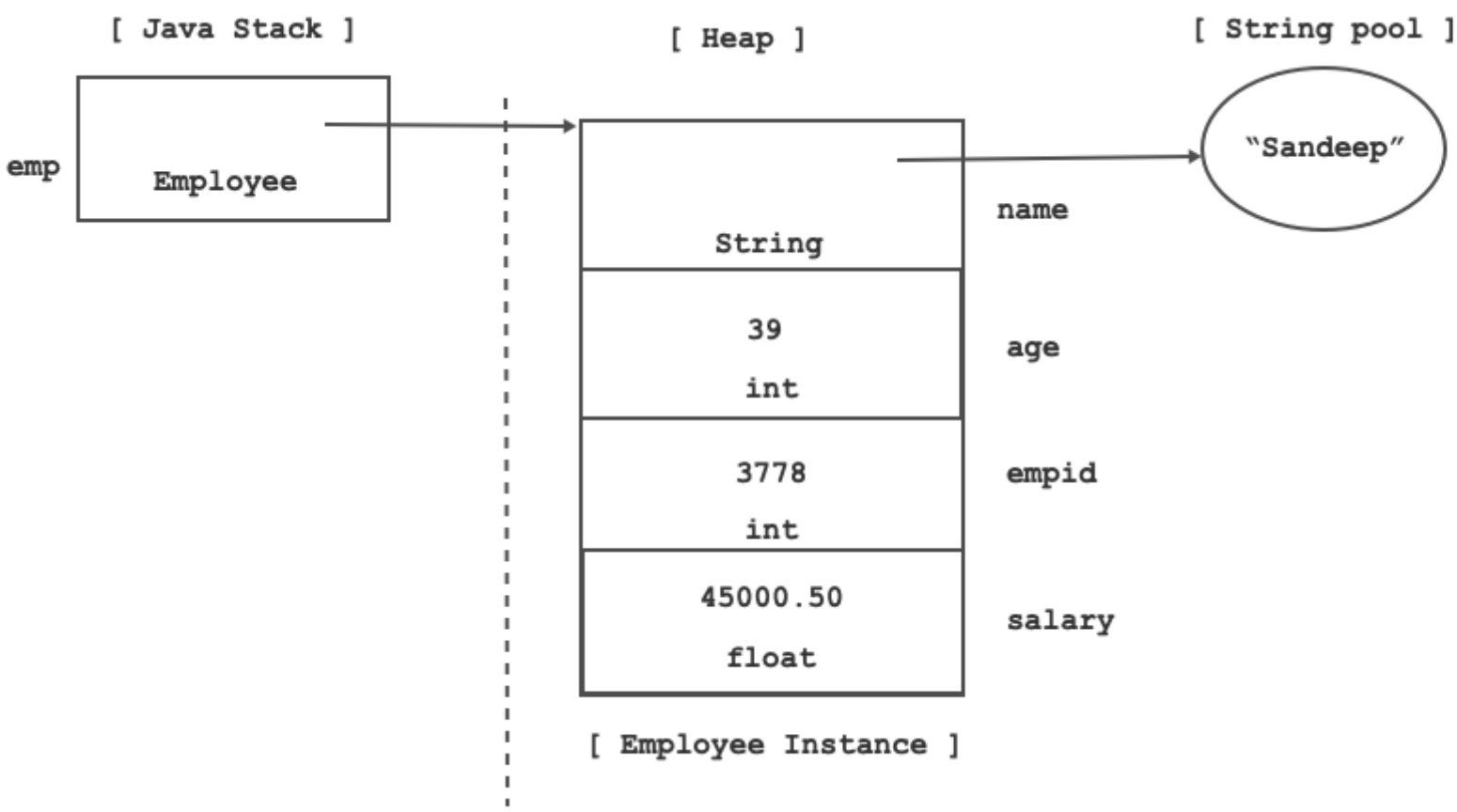
```
class Employee{
  private String name;
  private int age;
  private int empid;
  private float salary;
```

```
    public Employee() {
      this("",0,0,0.0f);
    }

    public Employee(String name, int age, int empid, float salary) {
      this.name = name;
      this.age = age;
      this.empid = empid;
      this.salary = salary;
    }

    public void displayRecord( ) {
      System.out.println("Name  :   "+this.name);
      System.out.println("Age   :   "+this.age);
      System.out.println("Empid :   "+this.empid);
      System.out.println("Salary    :   "+this.salary);
    }
}

public class Program {
  public static void main(String[] args) {
    //Employee emp = new Employee();
    Employee emp = new Employee("Sandeep", 39, 3778, 45000.50f);
    emp.displayRecord();
  }
}
```



- If is a relationship is exist between the types then we should use inheritance.

```
class Person{ //Parent class / Super class

}
class Employee extends Person{  //Child class / Sub class
  //Here Employee class can reuse fields/constructors/methods and nested types of Person class
}
```

- extends is a keyword which is used to create child class.
- In Java, parent class is called as super class and child is called as sub class.
- java.lang.Object is super class of all the classes in Java language.

```
class Person extends Object{

}
class Employee extends Person {
```

```
}
```

  - Now class Person is direct super class of class Employee and class Object is indirect super class of class Employee.

- In Java, any class can extend only one class.

```
class A{  } //OK
class B{  } //OK
class C extends B{  } //OK
class D extends A, B{ } //Not OK
```

- During inheritance, members of sub class do not inherit into super class. Hence using super class instance, we can access members of super class only.

```
  class Person{ //Super class
    String name;
    int age;

    public void showRecord( ) {
      System.out.println("Name  :   "+this.name);
      System.out.println("Age   :   "+this.age);
    }
  }

  class Employee extends Person{    //Sub class
    int empid;
    float salary;

    public void displayRecord( ) {
      System.out.println("Empid :   "+this.empid);
      System.out.println("Salary    :   "+this.salary);
    }
  }

  public class Program {
    public static void main(String[] args) {
      Person p = new Person();
      p.name = "Sandeep";   //OK
      p.age = 39;   //OK
      //p.empid = 3778; //Not OK
      //p.salary = 45000.50f;   //Not OK
      p.showRecord();
      //p.displayRecord();  //Not OK

    }
  }
```

- During inheritance, members of super class inherit into sub class. Hence using sub class instance, we can access members of super class as well as sub class.

```
  class Person{ //Super class
    String name;
    int age;

    public void showRecord( ) {
      System.out.println("Name  :   "+this.name);
      System.out.println("Age   :   "+this.age);
    }
  }

  class Employee extends Person{    //Sub class

    int empid;
    float salary;

    public void displayRecord( ) {
      System.out.println("Empid :   "+this.empid);
      System.out.println("Salary    :   "+this.salary);
    }
  }

  public class Program {
```

```
    public static void main(String[] args) {
      Employee emp = new Employee();
      emp.name = "Sandeep";
      emp.age = 39; //OK
      emp.empid = 3778; //OK
      emp.salary = 45000.45f;   //OK
      emp.showRecord();
      emp.displayRecord();
    }
  }
```

- If we create instance of sub class then all the non static fields declared in super class and sub class get space inside it. In other words, non static fields of super class inherit into sub class.
-
- Using sub class name, we can use/access static fields of super class. It means that static field of super class inherit into sub class.
- All(static and non static) the fields of super class inherit into sub class but only non static fields get space inside instance of sub class.
- We can call/invoke, non static method of super class on instance of sub class. In other words, non static method of super class inherit into sub class.

```
    public static void main(String[] args) {
      Employee emp = new Employee();
      emp.showRecord();
      emp.displayRecord();
    }
```

- We can invoke, static method of super class on sub class name. In other words, static method of super class inherit into sub class.
- Except constructor, all(static and non static) the methods of super class inherit into sub class.
- If we create instance of sub class then first super class constrcutor gets called and then sub class constructor gets called.
- From any constructor of sub class, by default, super class's parameterless constructor gets called.
- If we want to call any constructor of super class from constructor of sub class then we should use super statement.
- super is keyword in java.
- super statement must be first statement inside constructor body.

```
  class Person{ //Super class
    String name;
    int age;
    public Person( ) {
      this.name = "";
      this.age = 0;
    }
    public Person(String name, int age) {
      this.name = name;
      this.age = age;
    }
  }
  class Employee extends Person{    //Sub class
    int empid;
    float salary;
    public Employee( ) {
      this.empid = 0;
      this.salary = 0.0f;
    }
    public Employee( String name, int age, int empid, float salary ) {
      super( name, age ); //Super statement
      this.empid = empid;
      this.salary = salary;
    }
  }
```

- Except constrcutor, all the members of super class inherit into sub class.
- Definition of Inheritance
  - Journey from generalization to specialization is called inheritance.
  - Process of acquiring/accessing members of parent class into child class is called as inheritance.

- Interface inheritance:
    - During inheritance, if super type and sub types are interfaces then it is called as interface inheritance.

```
//Single Interface inheritance
interface A{
  //TODO
}
interface B extends A{  //Single Interface inheritance  : OK
  //TODO
}
```

```
//Multiple Interface inheritance
interface A{
  //TODO
}
interface B{
  //TODO
}
interface C extends A, B{  //Multiple Interface inheritance : OK
  //TODO
}
```

```
//Hierarchical interface inheritance
interface A{
  //TODO
}
interface B extends A{  //Single interface inheritance : OK
  //TODO
}
interface C extends A{  //Single Interface inheritance : OK
  //TODO
}
```

```
// Multilevel interface inheritance
interface A{
  //TODO
}
interface B extends A{  //Single interface inheritance : OK
  //TODO
}
interface C extends B{  //Single Interface inheritance : OK
  //TODO
}
```

- Implementation inheritance
    - During inheritance, if super type and sub type is class then it is called as implementation inheritance.

```
//Single implementation inheritance
class A{
  //TODO
}
class B extends A{  //Single implementation inheritance : OK
  //TODO
}
```

```
//Multiple implementation inheritance
class A{
  //TODO
}
class B{
  //TODO
}
class C extends A, B{  //Multiple implementation inheritance : NOT OK
```

```
  //TODO
}
```

```
//Hierarchical implementation inheritance
class A{
  //TODO
}
class B extends A{  //Single Interface inheritance : OK
  //TODO
}
class C extends A{  //Single Interface inheritance : OK
  //TODO
}
```

```
//Multilevel implementation inheritance
class A{
//TODO
}
class B extends A{  //Single Interface inheritance : OK
//TODO
}
class C extends B{  //Single Interface inheritance : OK
//TODO
}
```