HEXAWARE

Static/Final keywords/Object class/ String/ StringBuffer/ StringBuilder

Course Objectives

- To Introduce Object Oriented programming
- Classes and Objects
- Data Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Strings in Java

Strings in Java

Strings are an integral part of programming.

In Java, We have a **String class** for creating and manipulating strings.

Also, there is an **interface** called **CharSequence** used for representing a character sequence.

The **String class** is one of the classes which **implements this interface**. Hence, string is basically an object that represents a sequence of char values..

Ways of creating a String

There are two ways to create a string in Java:

- String Literal
- Using new Keyword

String created as Literal

String s=“Welcome”

Whenever a **String** Object is created as a literal, the object will be created in the **String Constant Pool**.

This allows JVM to optimize the initialization of String literal.

Only one object will be created. Initiallly JVM will not find any string object with the value “Welcome” in the string constant pool, so it will create a new object.

String t=“Welcome”

it will find the string with the value “Welcome” in the pool, it will not create a new object but will return the reference to the same instance.

String created using new keyword

- String s = new String(“Welcome”);
- In such a case, JVM will create a new string object in normal (non-pool) heap memory and the literal “Welcome” will be placed in the string constant pool. The variable s will refer to the object in the heap (non-pool)

.

Object

- Any entity that has state(properties) and behavior(actions) is known as an object.
- **State**: Address, Color, Area, Door no
- **Behavior**: Open door, close door
- The '*new*' keyword is used to create the *object*.

```
House myHouse=new House();
```

Real world representation

- Object represents real world entity.

Baby

Name
Sex
Weight
Decibels

String name
boolean isMale
double weight
double decibels

Why Use Classes?
Why Not Primitves?

HEXAWARE

// Data of baby Alex

String **nameAlex**;

double **weightAlex**;

// Data of baby David

String **nameDavid;**

double **weightDavid;**

// Data of baby David

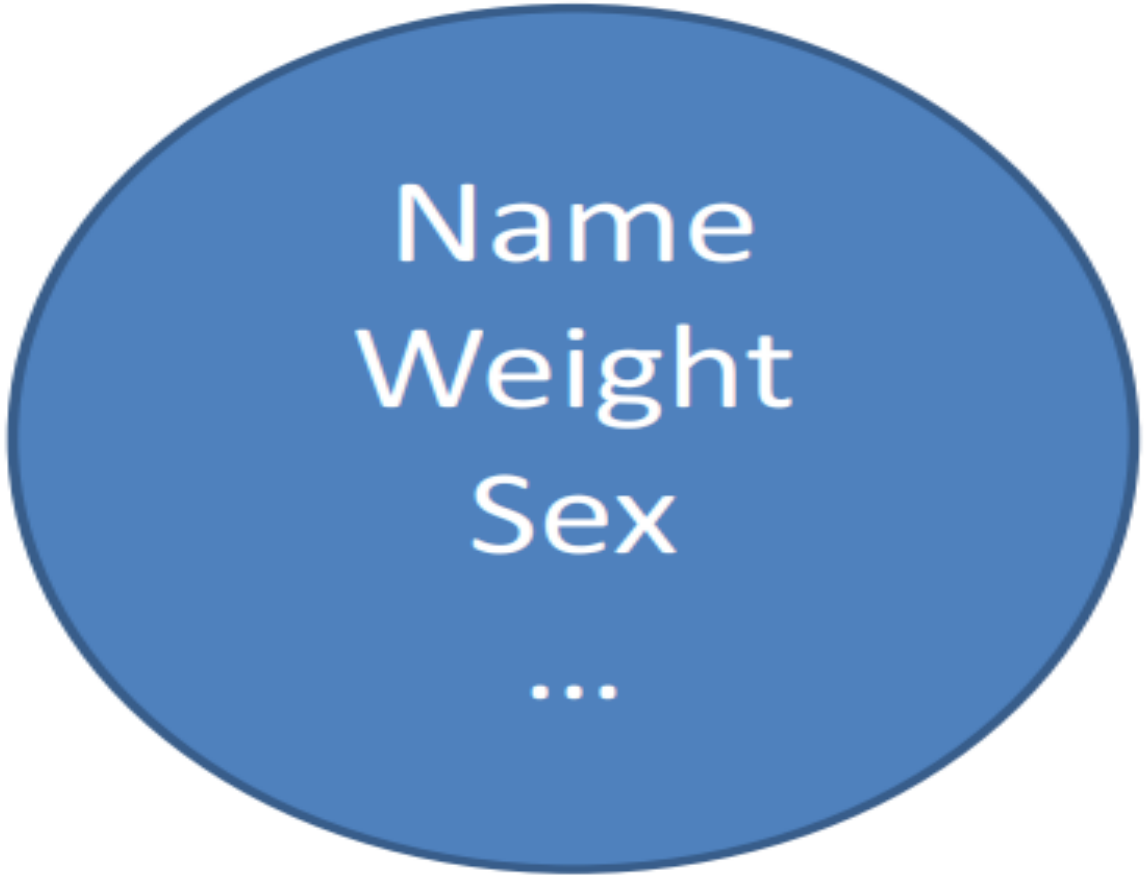
String **nameDavid2;**

double **weightDavid2;**

David2?

Terrible ☹

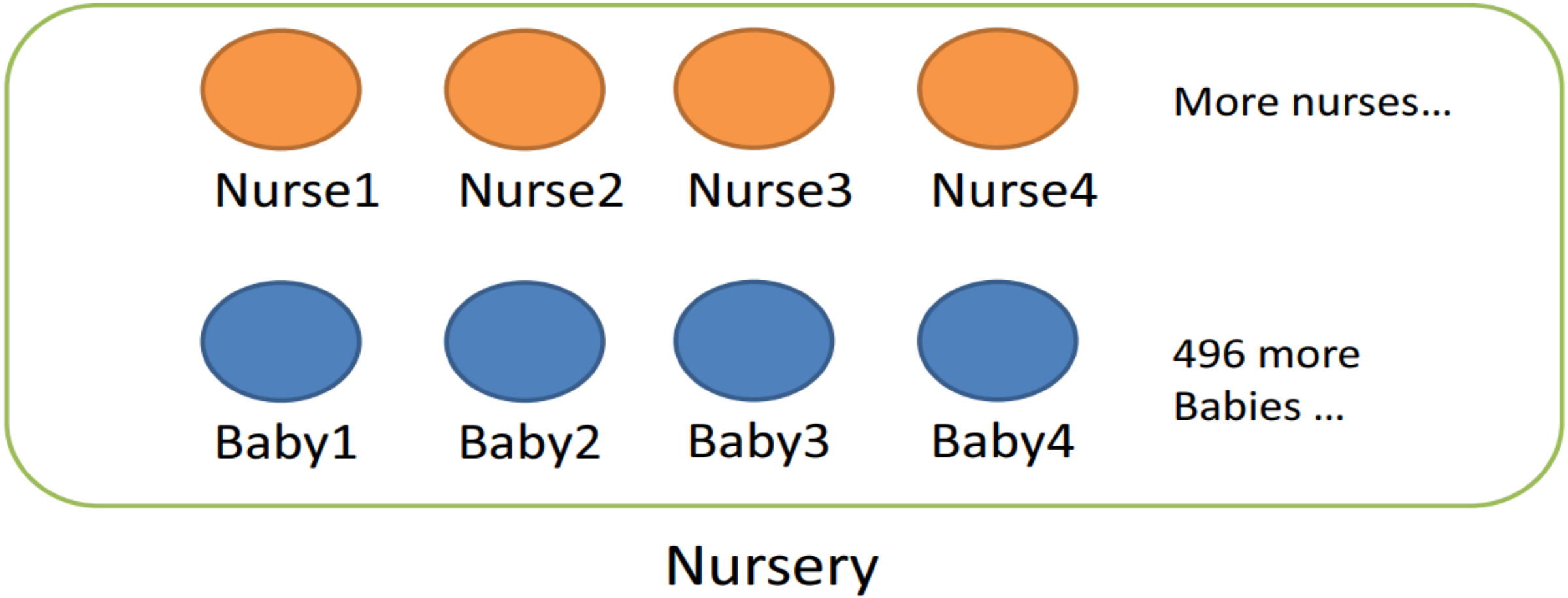
What if there are 500 babies !!!!



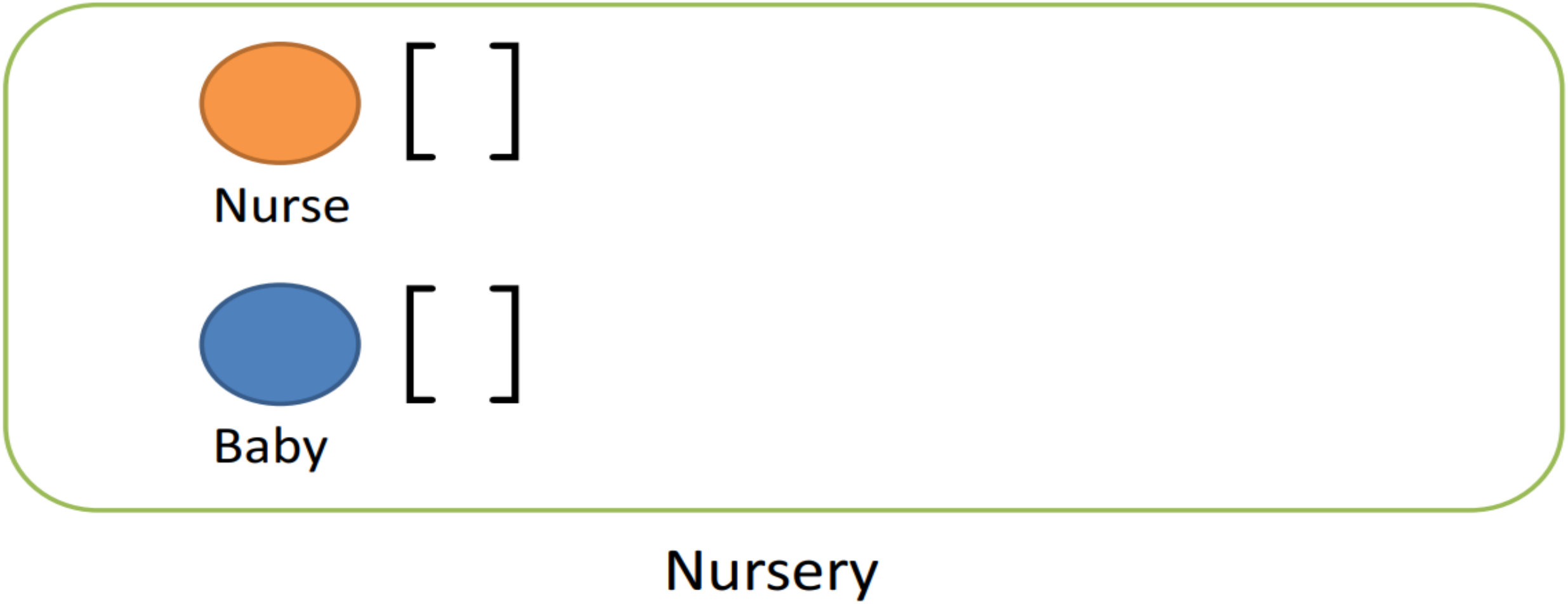
Baby1

Why use Classes?

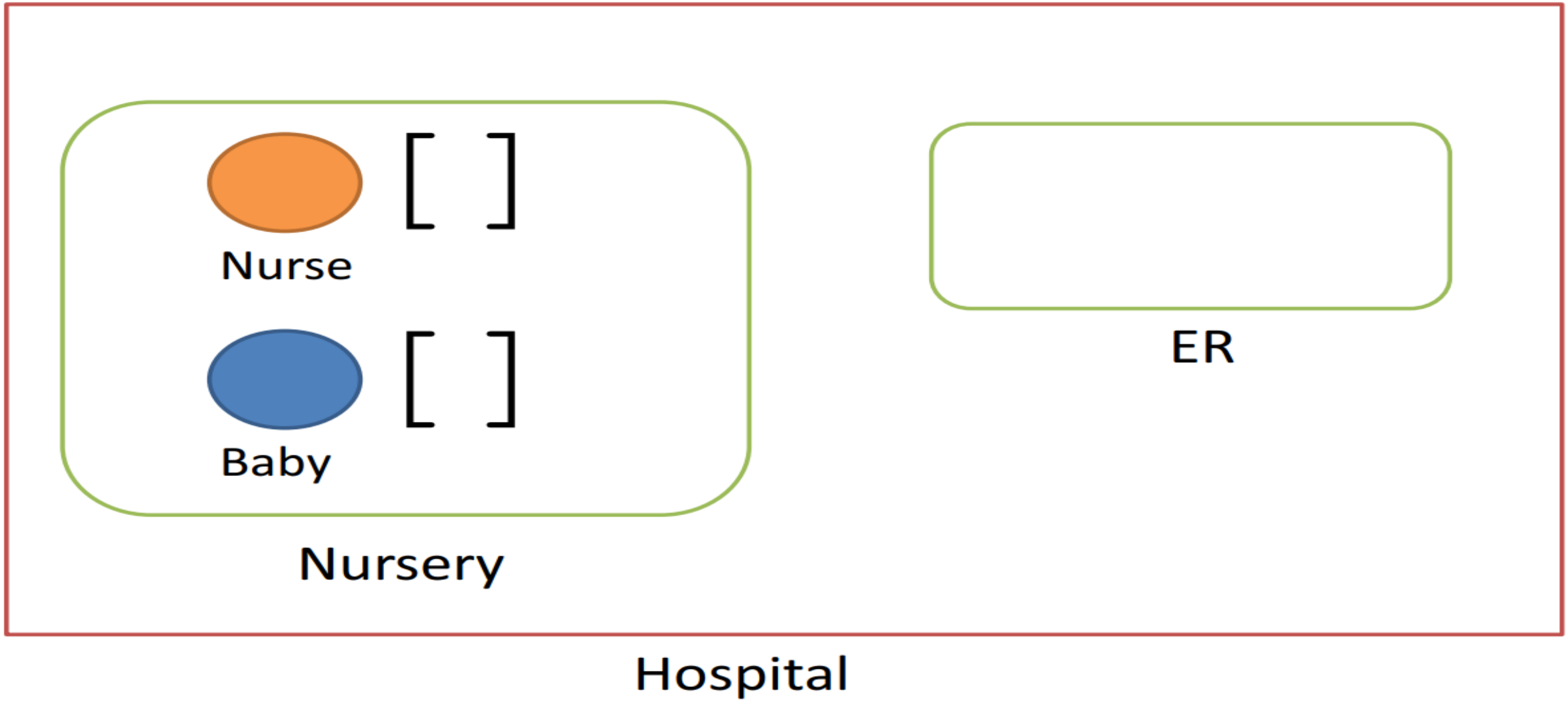
Why use Classes?



Why use Classes?

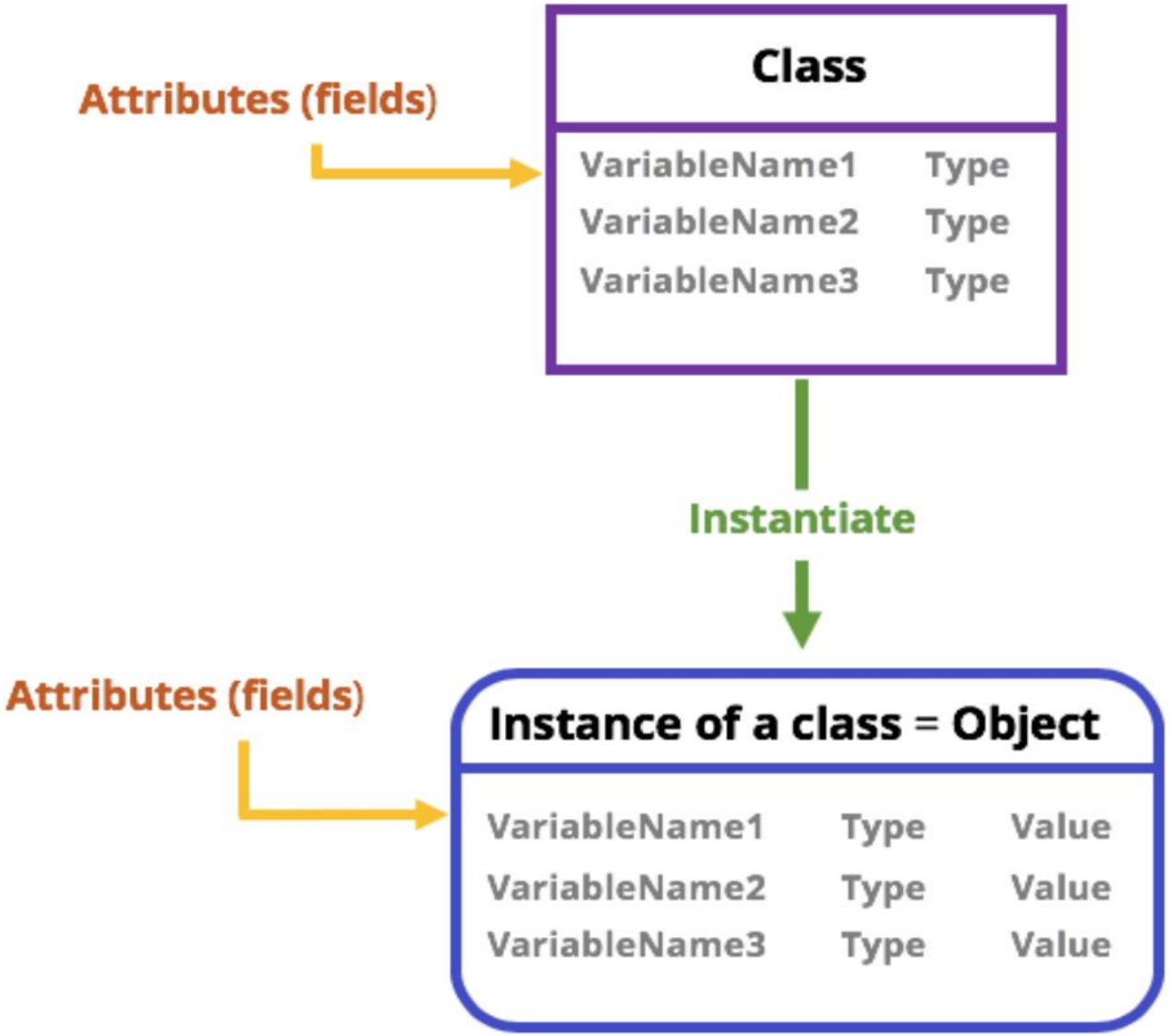


Why use Classes?



Classes and Objects

Class and object



Attributes (which are called *fields* in Java) are the **variables** you **define** when creating a class.

To instantiate an object, you declare a variable of that class.

When you instantiate an object from the class, you also set the **value** for each field in the object.

Sample Entity/ POJO?Bean in Java class/

```java
// Java bean for Person
public class Person {
    // Private variable
    private String fullName;
    // Constructor
    public Person(String fullName) {
        setFullName(fullName);
    }
    // Getter and setter for variable
    public String getFullName() {
        return fullName;
    }
    public void setFullName (String fullName) {
        this.fullName=fullName;
    }
}
```

Primitives Vs References

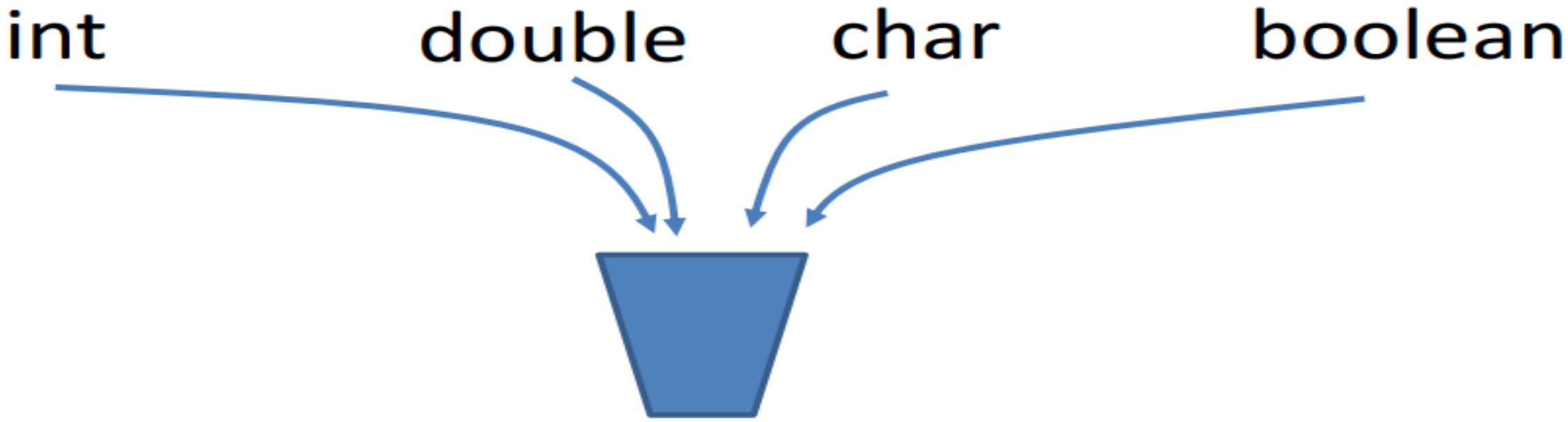
Primitive types are basic java types – **int, long, double, Boolean, char, short, byte, float** – The actual values are stored in the variable

Reference types are arrays and objects – **String, int[], Baby**, …

How java stores primitives

Variables are like fixed size cups

Primitives are small enough that they just fit into the cup

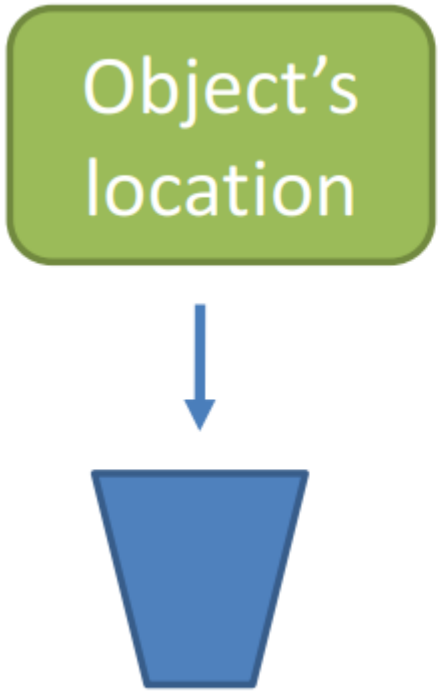


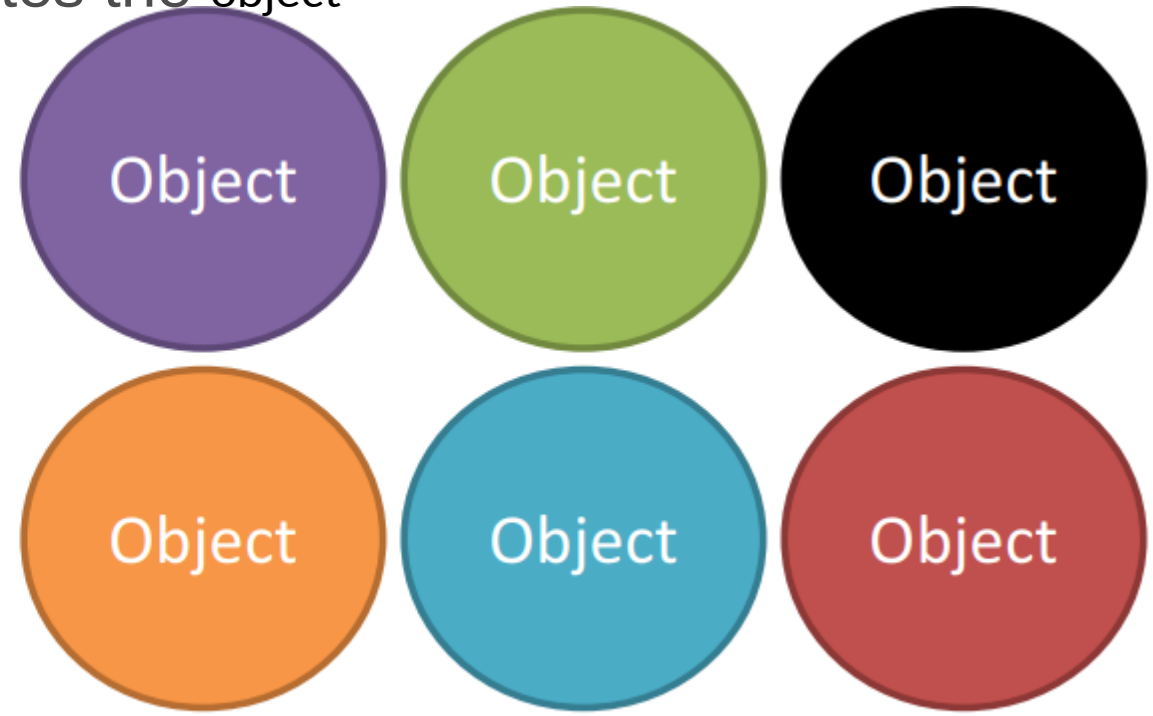
How java stores objects

- Objects are too big to fit in a variable
- Stored somewhere else
- Variable stores a number that locates the object

How Java stores objects

Objects are too big to fit in a variable
– Stored somewhere else
– Variable stores a number that locates the object





References

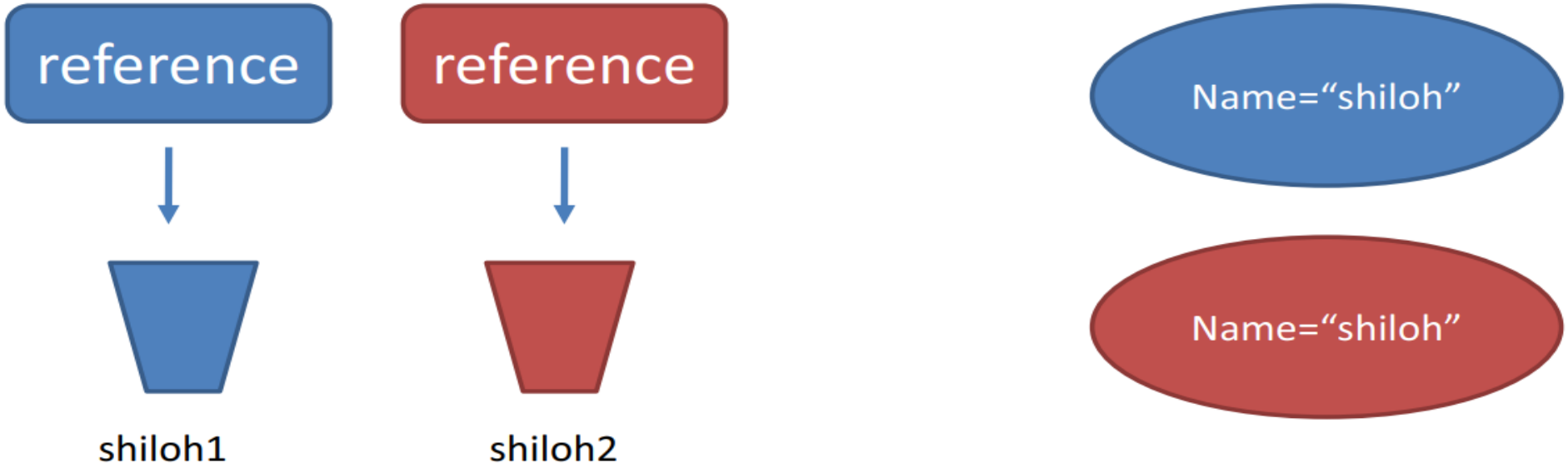
- The object’s location is called a reference
- == compares the references Baby shiloh1 = new Baby(“shiloh”);

Baby shiloh2 = new Baby(“shiloh”);

Does shiloh1 == shiloh2?

References

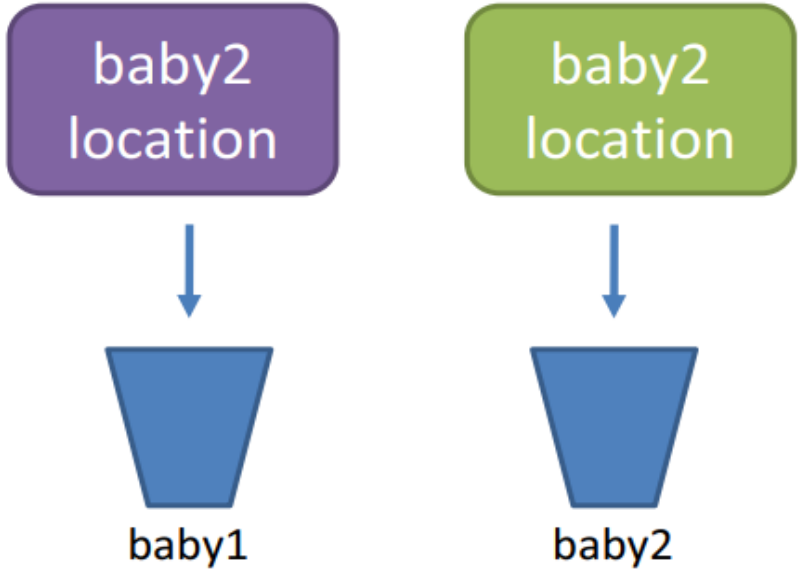
```
Baby shiloh1 = new Baby("shiloh");
Baby shiloh2 = new Baby("shiloh");
```

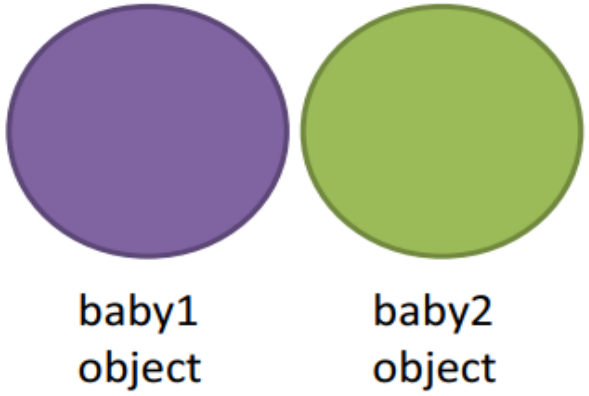


References

- Using = updates the reference.

```
baby1 = baby2
```

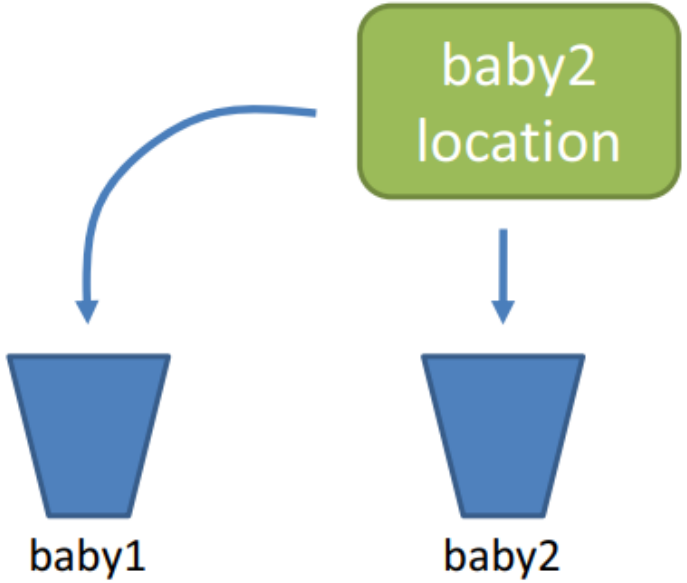


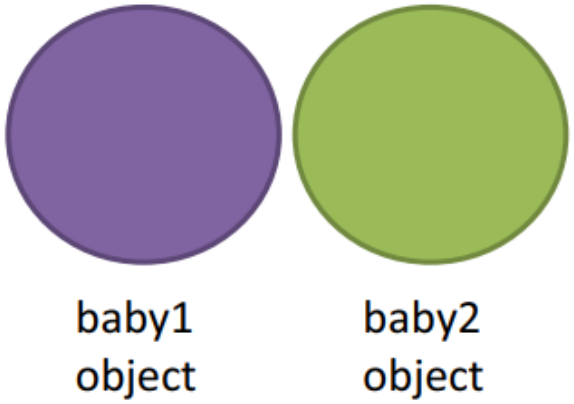


References

- Using = updates the reference.

```
baby1 = baby2
```





Static Fields

- Applies to fields and methods
- Means the field/method
  - Is defined for the class declaration
  - Is not unique for each instance

Demo on Creating Entity Class

OOP Terminologies

OOP Terminologies

- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Inheritance[IS-A Relation]

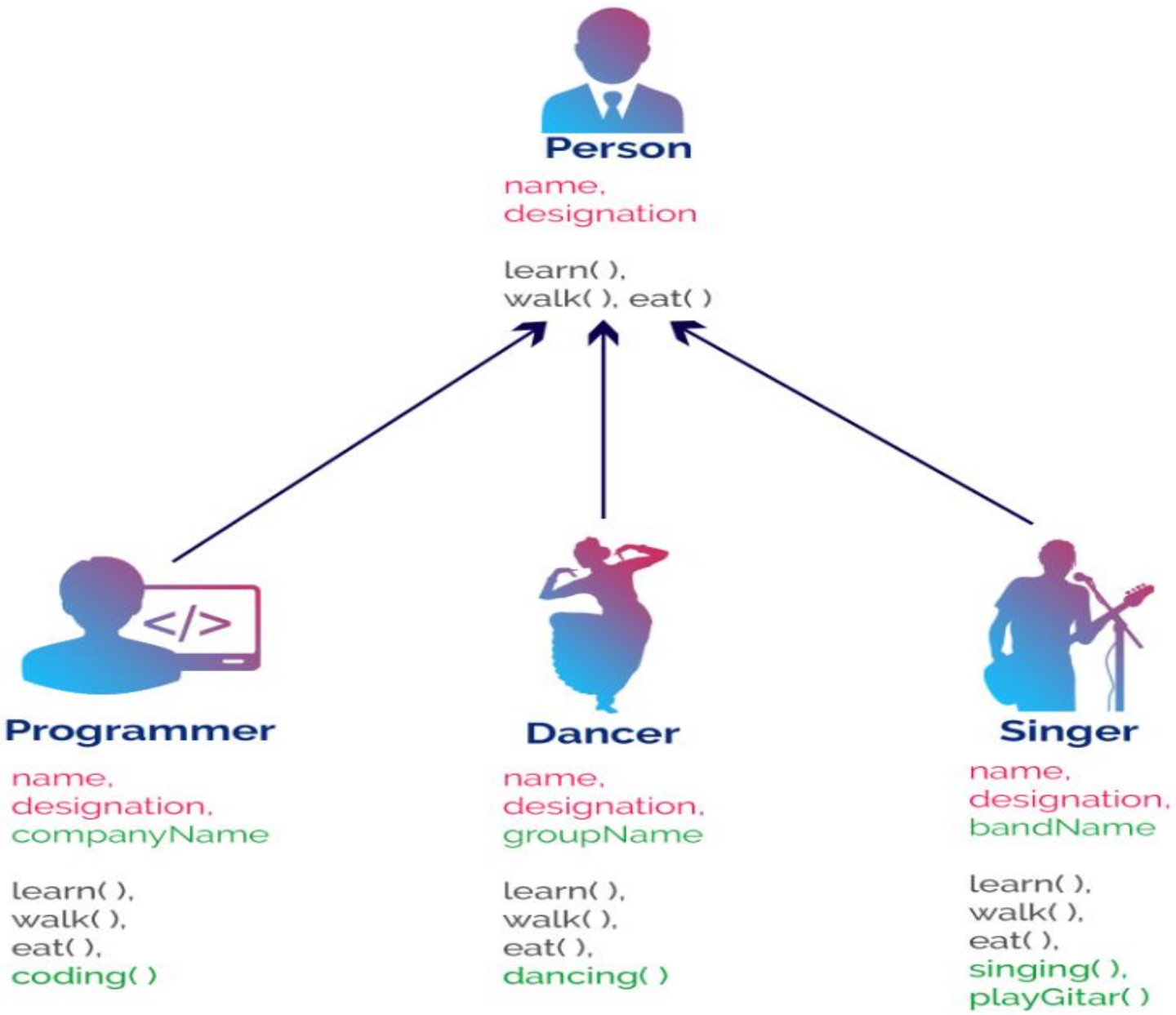
Inheritance refers to acquiring the attributes and functionalities by a class from another class.

The class which acquires the properties is termed **as sub class/child class/derived class.**

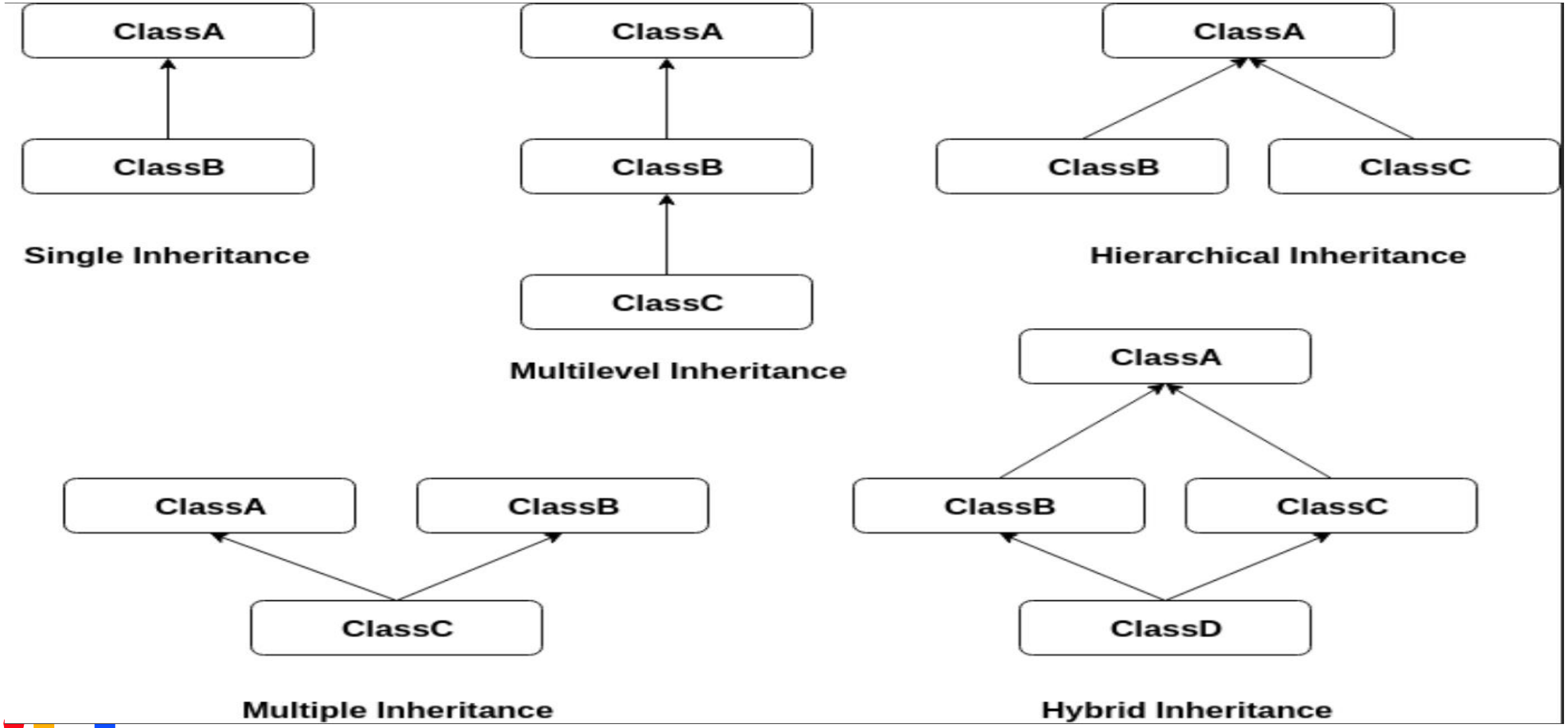
The class from which the properties are acquire is termed as **super class/parent class/base class**.

It reduces the line of codes and helps in code reusablility.

Inheritance



Types of inheritance



Single Inheritance

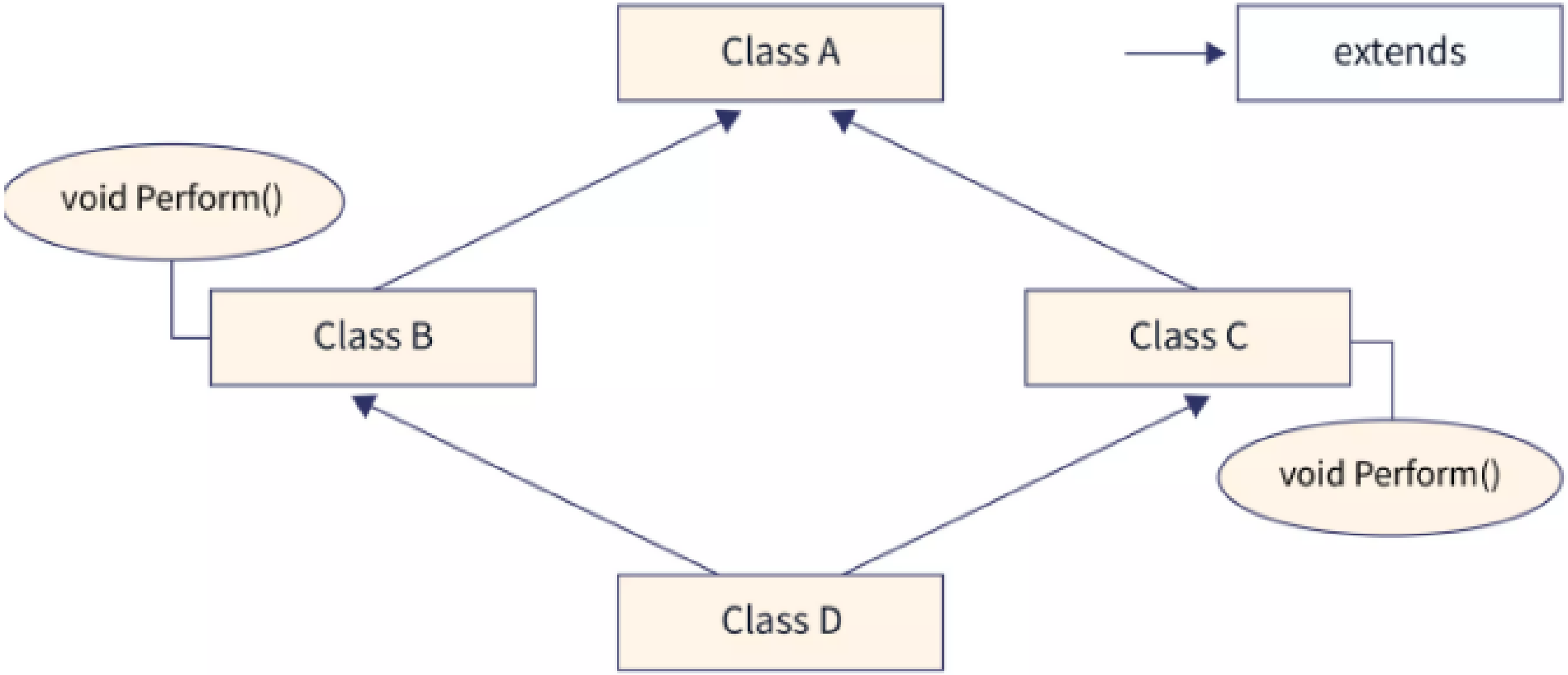
```java
package com.hexaware.entities;
//Class Object is the root of the class hierarchy.Every class has Object as a superclass.
public class Person extends Object {
    // instance/member variables
    private int uId;
    private String name;
    private int age;
    private long mobileNumber;
    // class level variable
    public static String collegeName = "SSN";
    // default constructor
    public Person() {
        super();

    }
    // parameterised
    public Person(int uId, String name, int age, long mobileNumber) {
        super();
        this.uId = uId;
        this.name = name;
        this.age = age;
        this.mobileNumber = mobileNumber;
    }
```

Single Inheritance

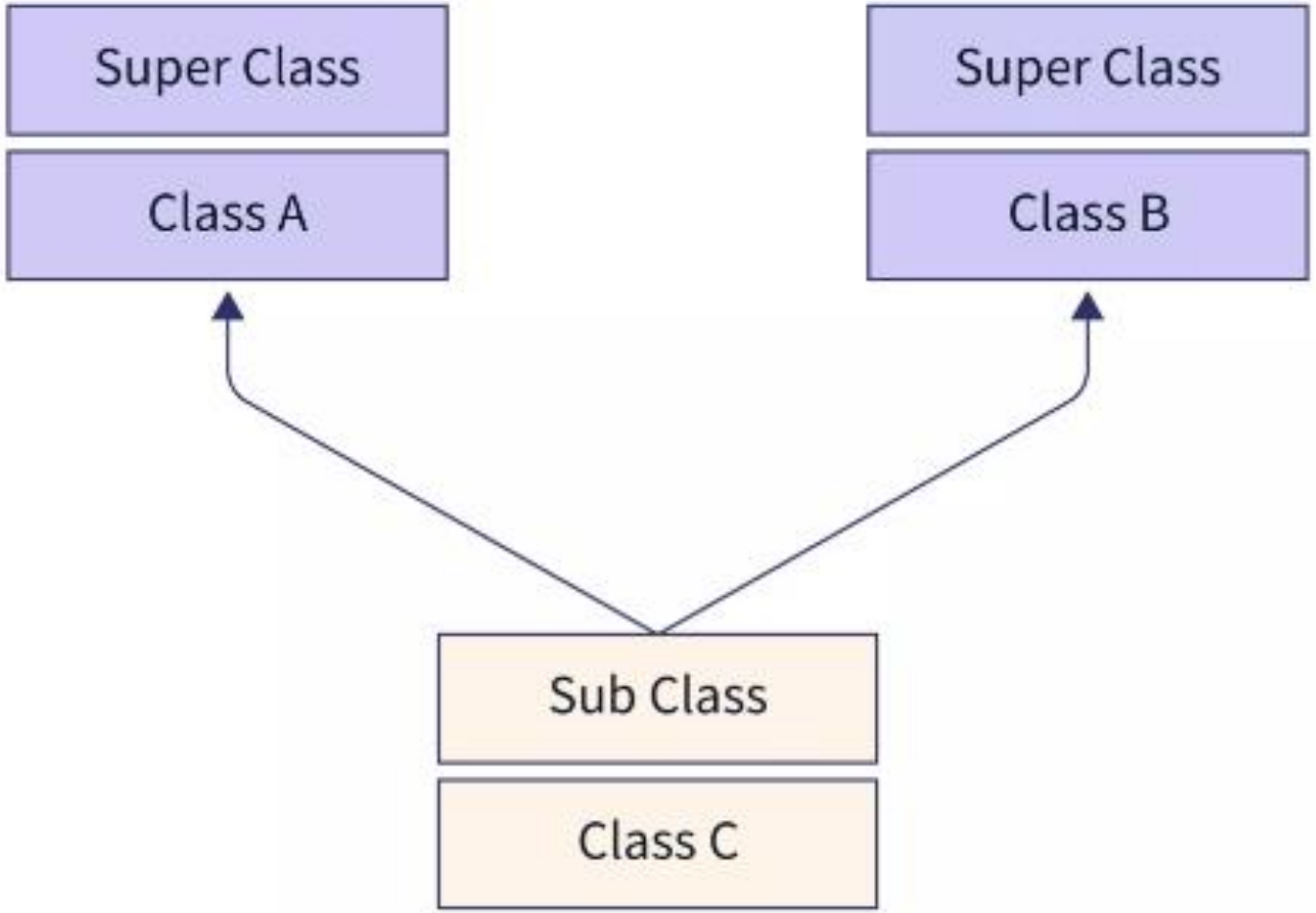
```java
package com.hexaware.entities;
public class Employee extends Person {
    private int employeeId;
    private int cabinNumber;
    private String designation;
    private double salary;
    // al 8 variables and one static variable  gets allocated with the memory
    public Employee() {
        super();
        // TODO Auto-generated constructor stub
    }

    public Employee(int employeeId, int cabinNumber, String designation, double salary,
            int uId, String name, int age,long mobileNumber) {
        // call to superclass parameterised constructor. static binding/compiletime
        // polymorphism
        super(uId, name, age, mobileNumber);
        this.employeeId = employeeId;
        this.cabinNumber = cabinNumber;
        this.designation = designation;
        this.salary = salary;
```

Demo on Single Inheritance

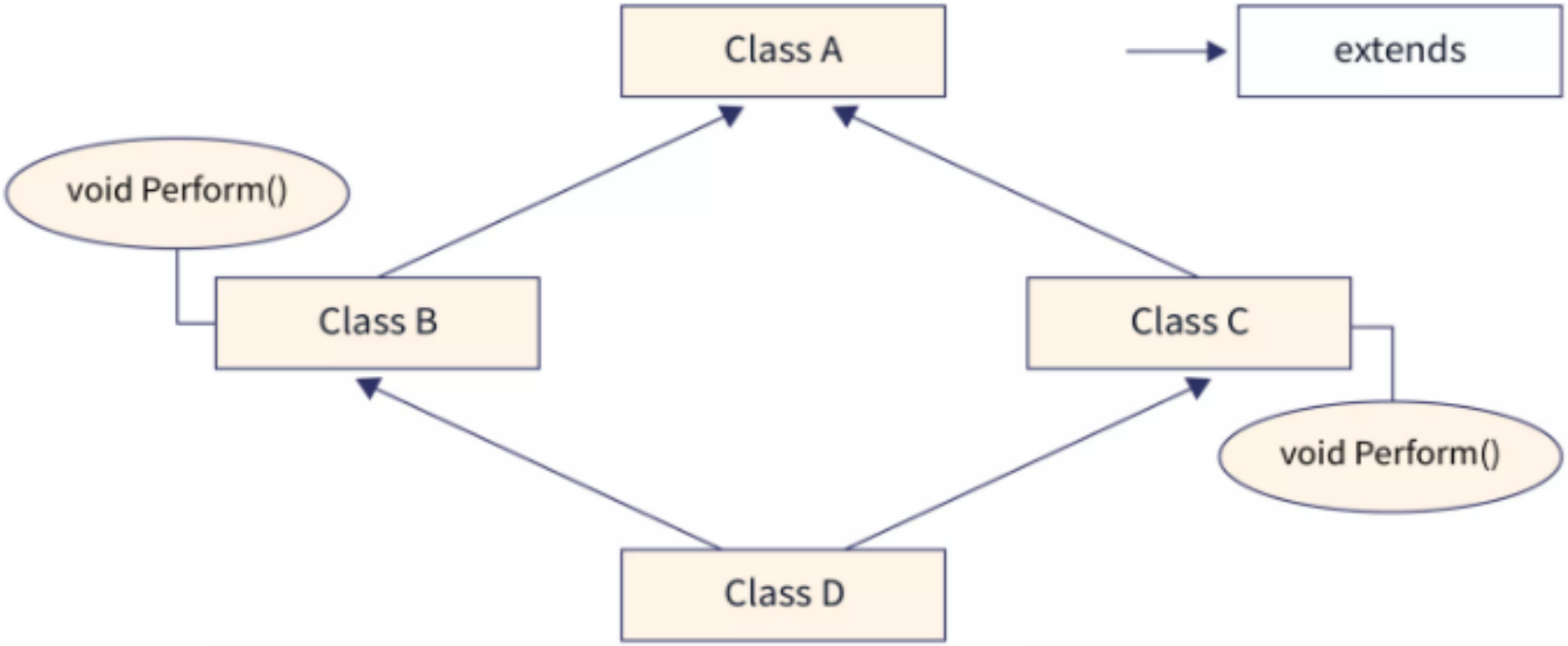
Multiple Inheritance



Multiple Inheritance



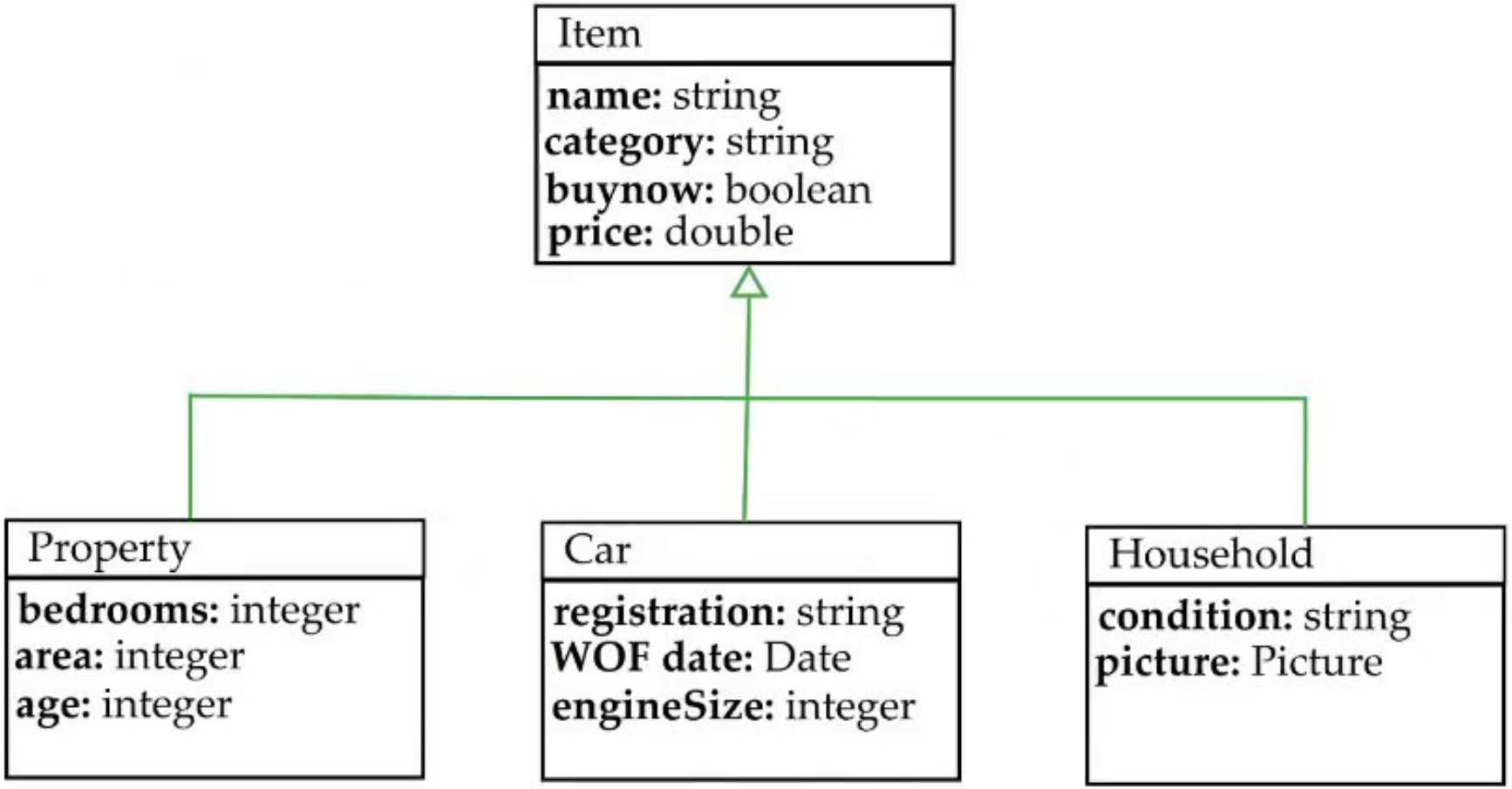
Multiple Inheritance



Multiple Inheritance

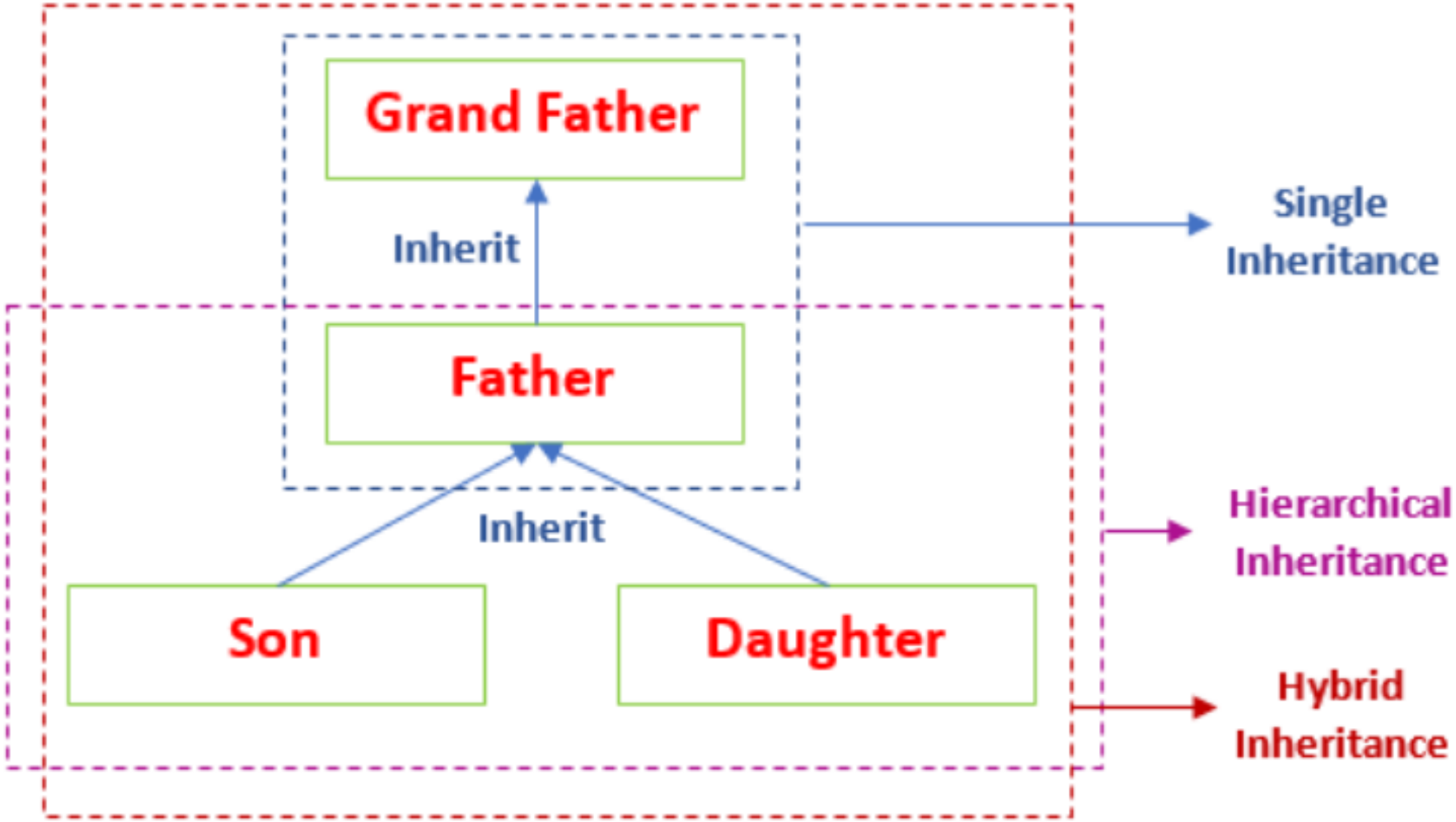
- Class A is inherited by classes B and C.
- Classes B and C are inherited by class D. This is where the actual ambiguity arises.
- Assume that classes B and C both contain the same method(say Perform()) with the same signature. (Perform() can be a method in A that B and C have overridden). If an object is instantiated for class D, and this object is used to call the method which is present in both class B and C, take a moment to understand the scenario.
- Compiler would be confused because the compiler does not know which class it should call to execute the method(Perform()) as it is present in both classes.
- This is the actual reason why Multiple Inheritance in Java is not supported.

Hierarchical Inheritance



Demo on Hierarchical Inheritance

Hybrid Inheritance



Demo on Hybrid Inheritance

Polymorphism

Polymorphism

Polymorphism in OOP is the ability of an entity to take several forms. In other words, it refers to the ability of an object (or a reference to an object) to take different forms of objects.

Poly means many. Morphism means forms.

Polymorphism

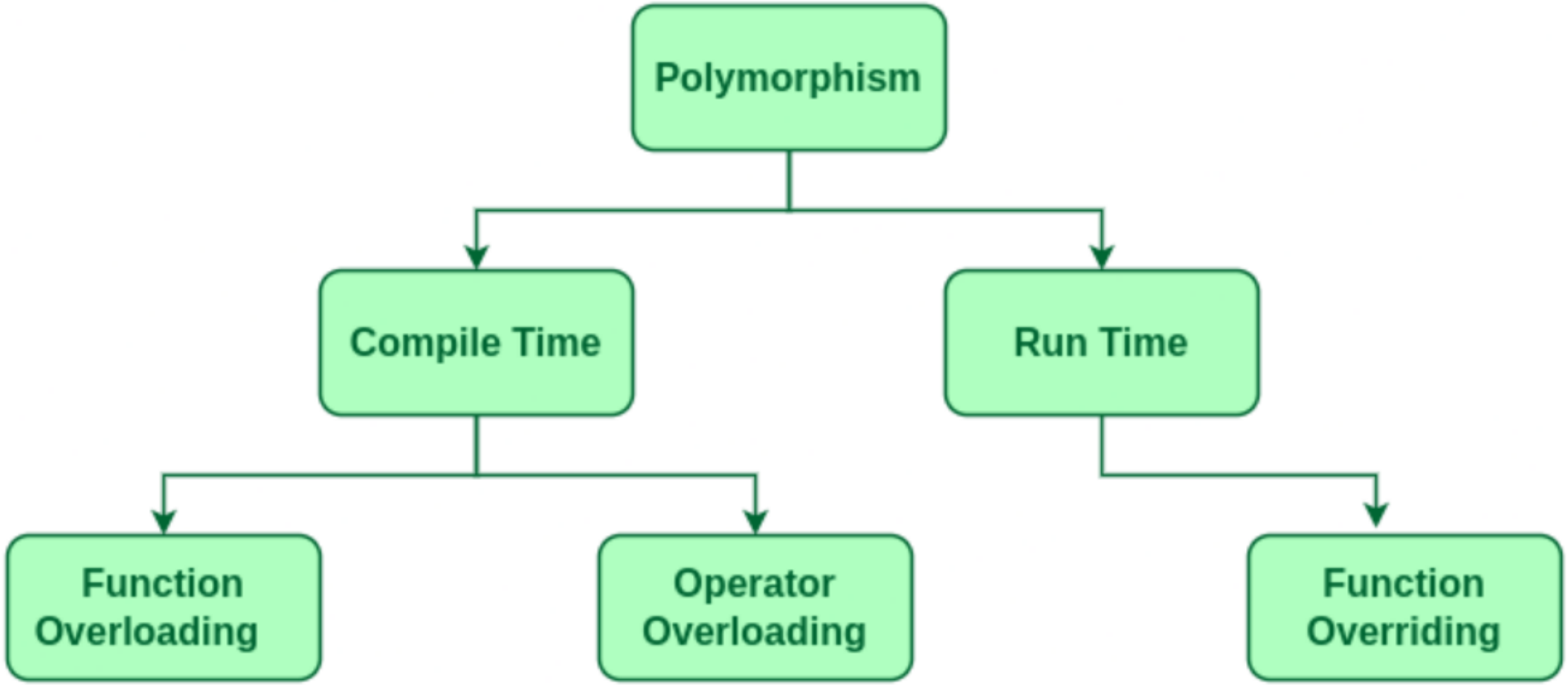
In Shopping malls behave like Customer

In Bus behave like Passenger

In School behave like Student

At Home behave like Son Sitesbay.com

Polymorphism



Compiletime/Static Polymorphism
Early Binding

Method Overloading is a feature that allows a class to have more than one method having the same name, but different signature.

Number of parameters.	Data type of parameters.	Sequence of parameters.
• add(int, int) • add(int, int, int)	• add(int, int) • add(int, float)	• add(int, float) • add(float, int)

Invalid Case of Method Overloading

Method Overloading is a feature that allows a class to have more than one method having the same name, but different signature.

int add(int, int)
float add(int, int)

Method Overloading and Type Promotion

When a data type of smaller size is promoted to the data type of bigger size than this is called type promotion

```
class Demo{
   void disp(int a, double b){
            System.out.println("Method A");//Output
   }
   void disp(int a, double b, double c){
            System.out.println("Method B");
   }
   public static void main(String args[]){
            Demo obj = new Demo();
            /* I am passing float value as a second argument but
             * it got promoted to the type double, because there
             * wasn't any method having arg list as (int, float)
             */
            obj.disp(100, 20.67f);
   }
}
```

Type Promotion table:

byte → short → int → long
short → int → long
int → long → float → double
float → double
long → float → double

Valid / Invalid cases of method overloading

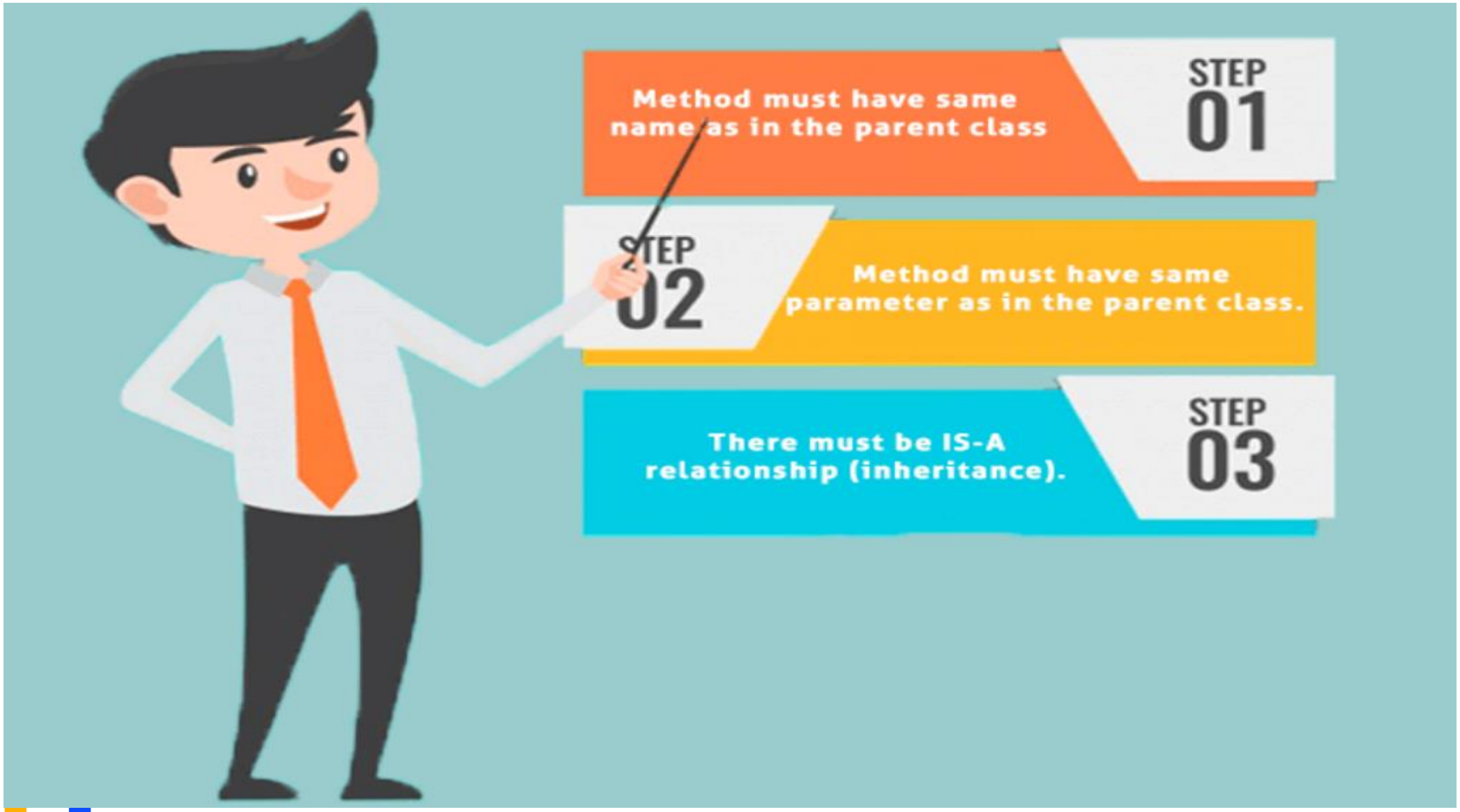
Case 1:	**int mymethod(int a, int b, float c)** **int mymethod(int var1, int var2, float var3)**
Case 2:	int mymethod(int a, int b) int mymethod(float var1, float var2)
Case 3:	int mymethod(int a, int b) int mymethod(int num)
Case 4:	float mymethod(int a, float b) float mymethod(float var1, int var2)
Case 5:	int mymethod(int a, int b) float mymethod(int var1, int var2)

Valid / Invalid cases of method overloading

Case 1: **int mymethod(int a, int b, float c)** **int mymethod(int var1, int var2, float var3)**	**Compile time error. Argument lists are exactly same. Both methods are having same number, data types and same sequence of data types.**
Case 2: int mymethod(int a, int b) int mymethod(float var1, float var2)	**Perfectly fine.** Valid case of overloading. Here data types of arguments are different.
Case 3: int mymethod(int a, int b) int mymethod(int num)	**Perfectly fine.** Valid case of overloading. Here number of arguments are different.
Case 4: float mymethod(int a, float b) float mymethod(float var1, int var2)	**Perfectly fine.** Valid case of overloading. Sequence of the data types of parameters are different, first method is having (int, float) and second is having (float, int).
Case 5: int mymethod(int a, int b) float mymethod(int var1, int var2)	**Compile time error.** Argument lists are exactly same. Even though return type of methods are different, it is not a valid case. Since return type of method doesn't matter while overloading a method.

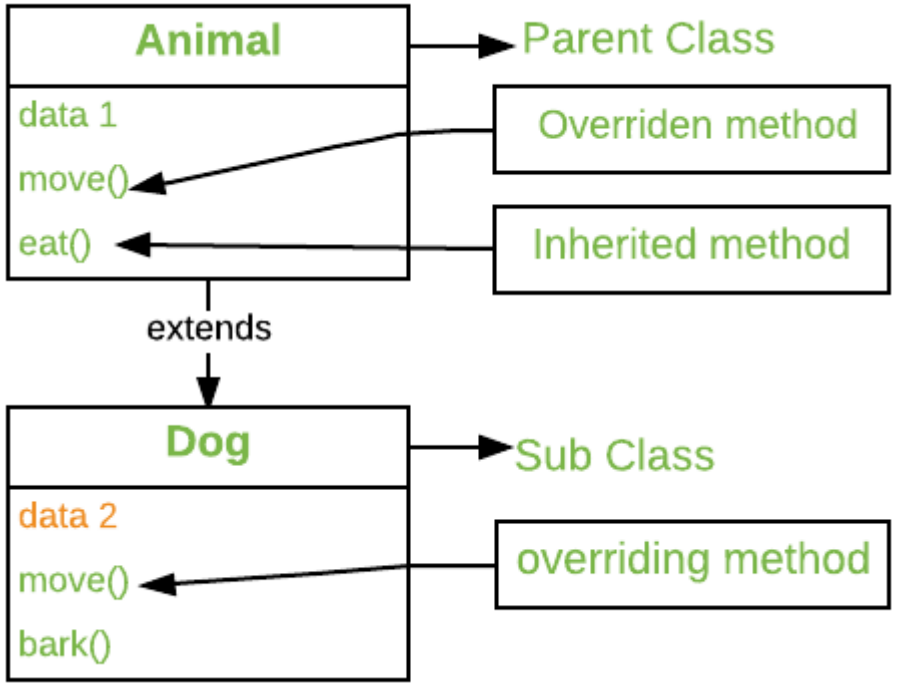
Method Overloading Demo

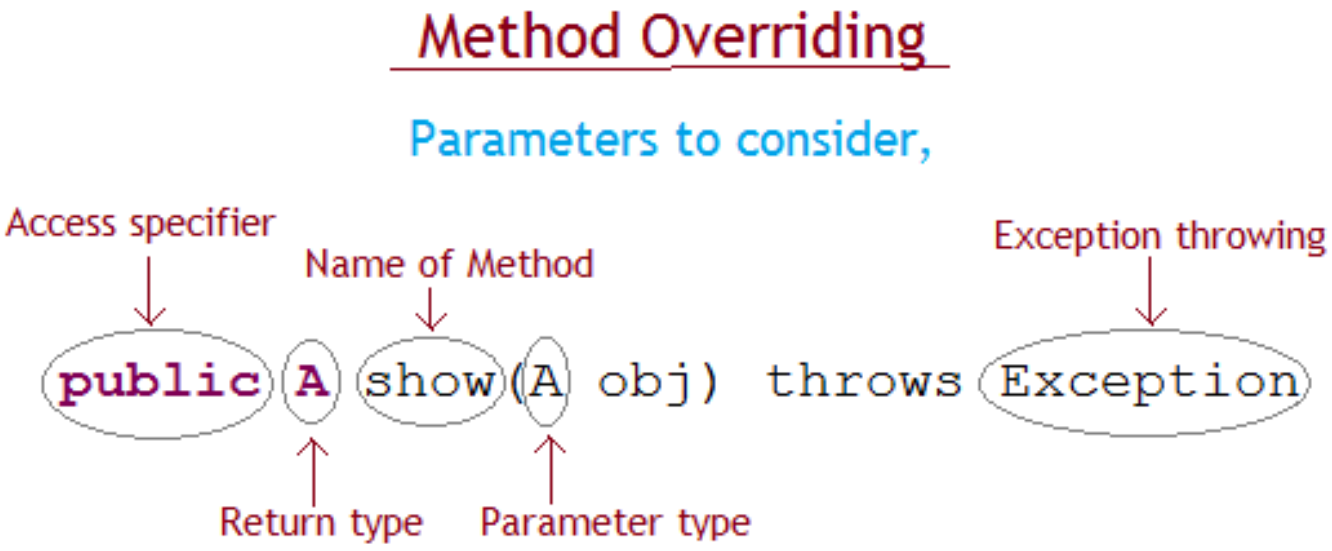
Method Overriding



Method Overriding

Declaring a method in **sub class** which is already present in **parent class** is known as method overriding.
method in parent class is called overridden method.
the method in child class is called overriding method.





Method Overriding/ Runtime Polymorphism / Dynamic Binding

Dynamic Method Dispatch

- When a parent class reference points to the child class object then the call to the overridden method is determined at runtime, because during method call which method(parent class or child class) is to be executed is determined by the type of object.
- This process in which call to the overridden method is resolved at runtime is known as dynamic method dispatch.

Rules of method overriding

- Overriding method can only be written in Subclass, not in same class.
- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the super class.
- The access level cannot be more restrictive than the overridden method's access level.
- For example: if the super class method is declared public then the over-ridding method in the sub class cannot be either private or protected.

Rules of method overriding

- An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not.
- However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.

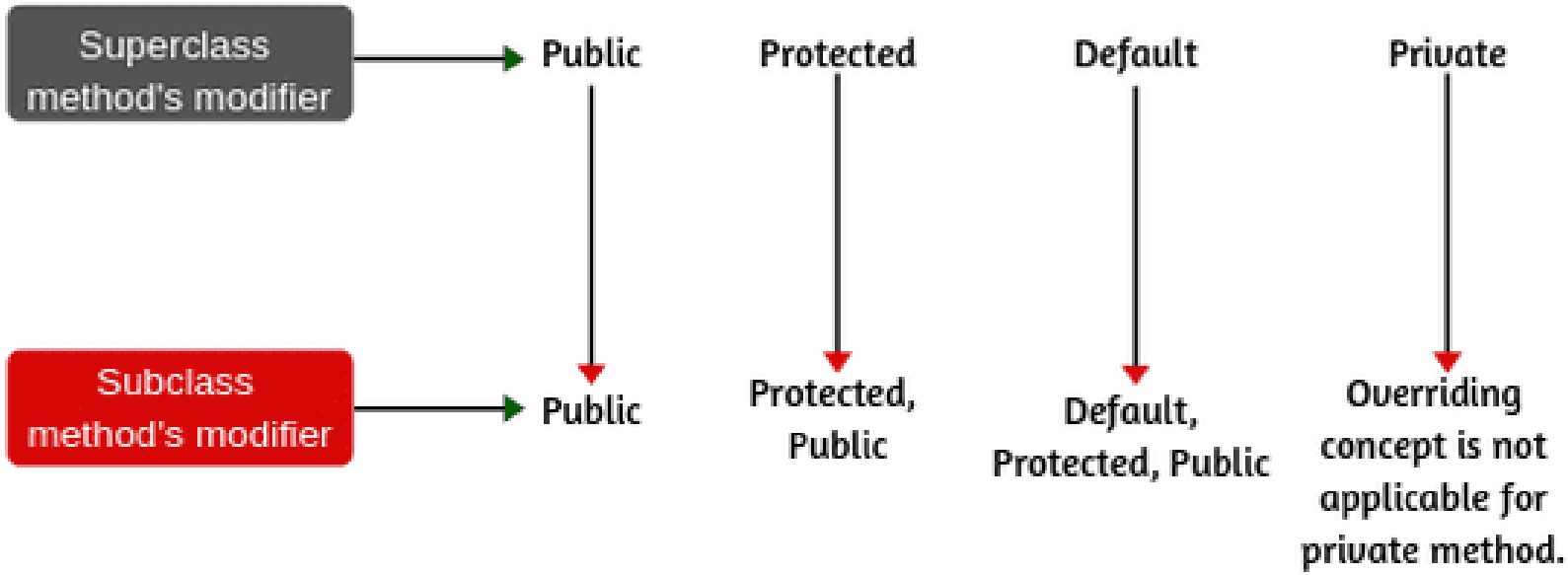
Constructors cannot be overridden.

Rules of method overriding

- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.

Method Overriding Demo

Access Modifiers in Java



The access modifier of the overriding method cannot be more restrictive than overridden method of superclass.

Private > Default > Protected > Public

More restrictive → Less restrictive

Fig: Applicable access modifiers to the overriding method

Access Specifiers Demo

Questions

Learning material references

Website

Java OOPs Concepts - Javatpoint

Object Oriented Programming (OOPs) Concept in Java - GeeksforGeeks

OOPs Concepts in Java (Updated 2023) | Great Learning (mygreatlearning.com)

HEXAWARE

Passionate Employees

Delighted Customers

Thank you

www.hexaware.com