



HEXWARE

Object Oriented Programming [OOP]

Course Objectives

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



An abstract graphic of glowing blue circuit lines and nodes on a dark blue background, with the title text centered over it.

Introduction to OOP



Introduction to OOP

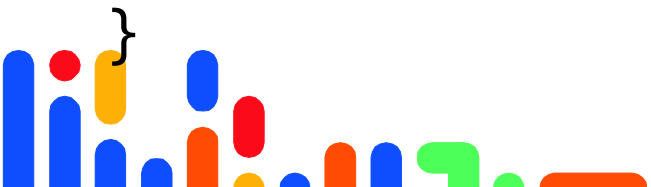
- The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language.
- **Smalltalk** is considered the first truly object-oriented programming language.
- The main aim of object-oriented programming is to implement real-world entities.
- Object-Oriented Programming is a paradigm that provides many concepts, such as **inheritance**, **data binding**, **polymorphism**, etc.
- The popular object-oriented languages are Java, C#, PHP, Python, C++, etc.



Class

- A class in **Java** is a blueprint which includes all your data. A class contains fields (variables) and methods to describe the behavior of an object.

```
class House
{
    String address;
    String color;
    double area;
    int doorNo;
    void openDoor()
        { //Write code here }
    void closeDoor()
        { //Write code here }
```



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 Primitives?

// Data of baby Alex

String **nameAlex**;
double **weightAlex**;

// Data of baby David

String **nameDavid**;
double **weightDavid**;

// Data of baby David

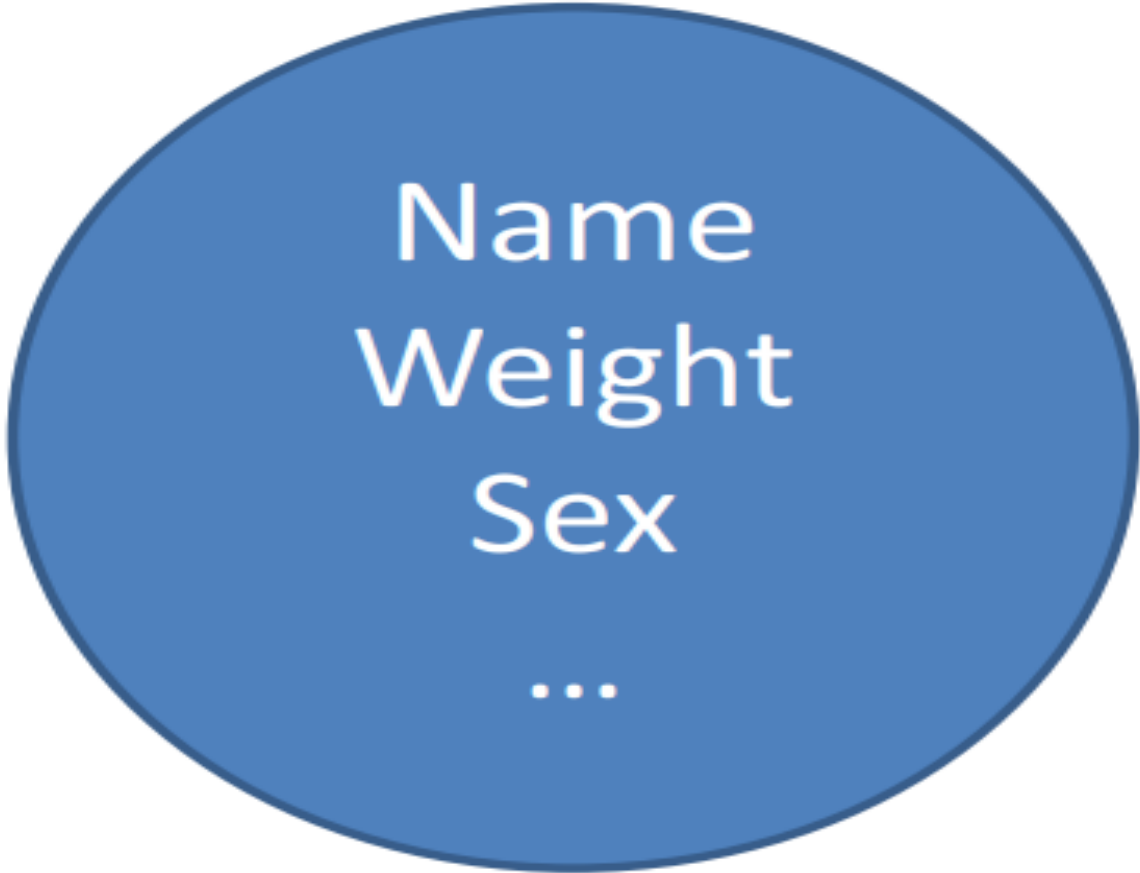
String **nameDavid2**;
double **weightDavid2**;



David2?
Terrible 😞



What if there are 500 babies !!!!

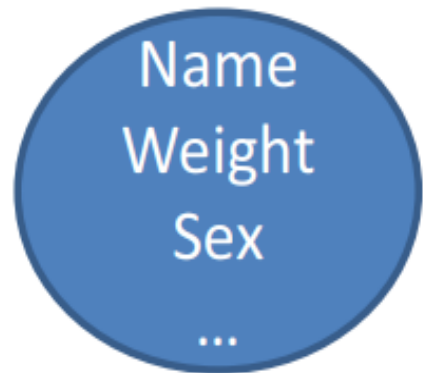


Name
Weight
Sex
...

Baby1



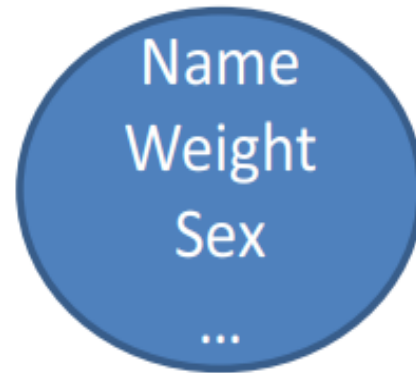
Why use Classes?



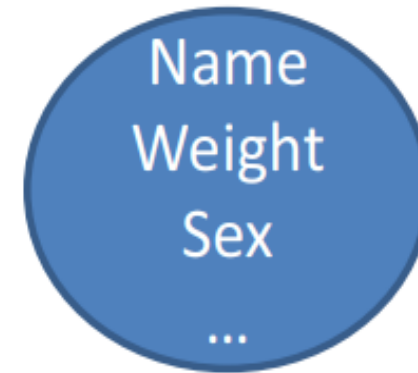
Baby1



Baby2



Baby3



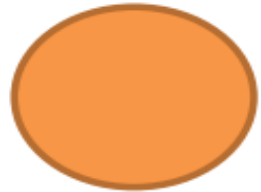
Baby4

496
more
Babies

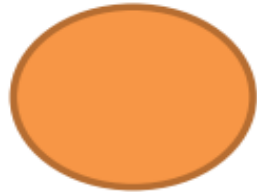
...



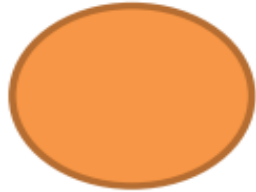
Why use Classes?



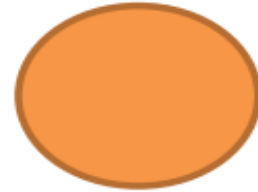
Nurse1



Nurse2



Nurse3



Nurse4

More nurses...



Baby1



Baby2



Baby3

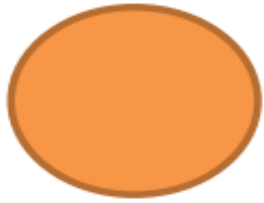


Baby4

496 more
Babies ...

Nursery

Why use Classes?



Nurse

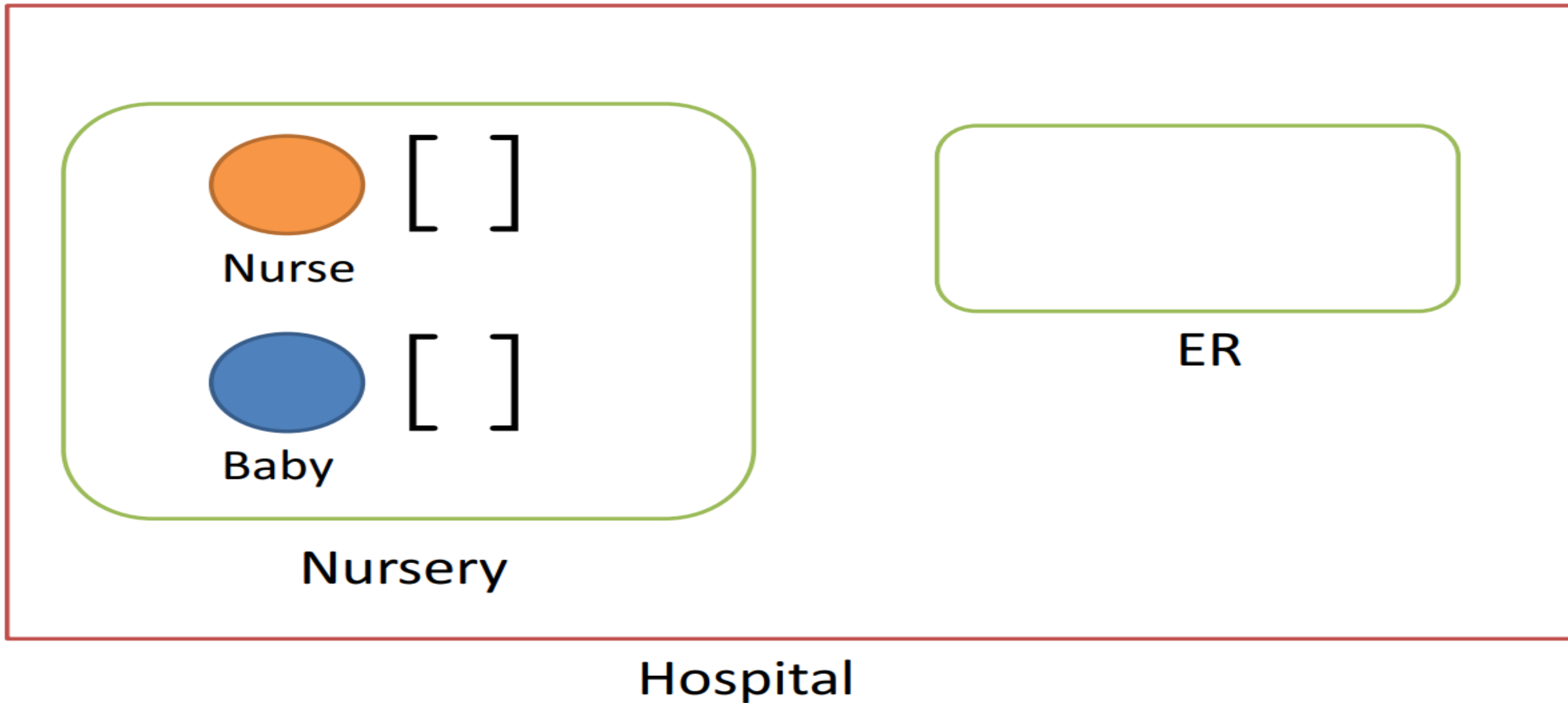


Baby



Nursery

Why use Classes?

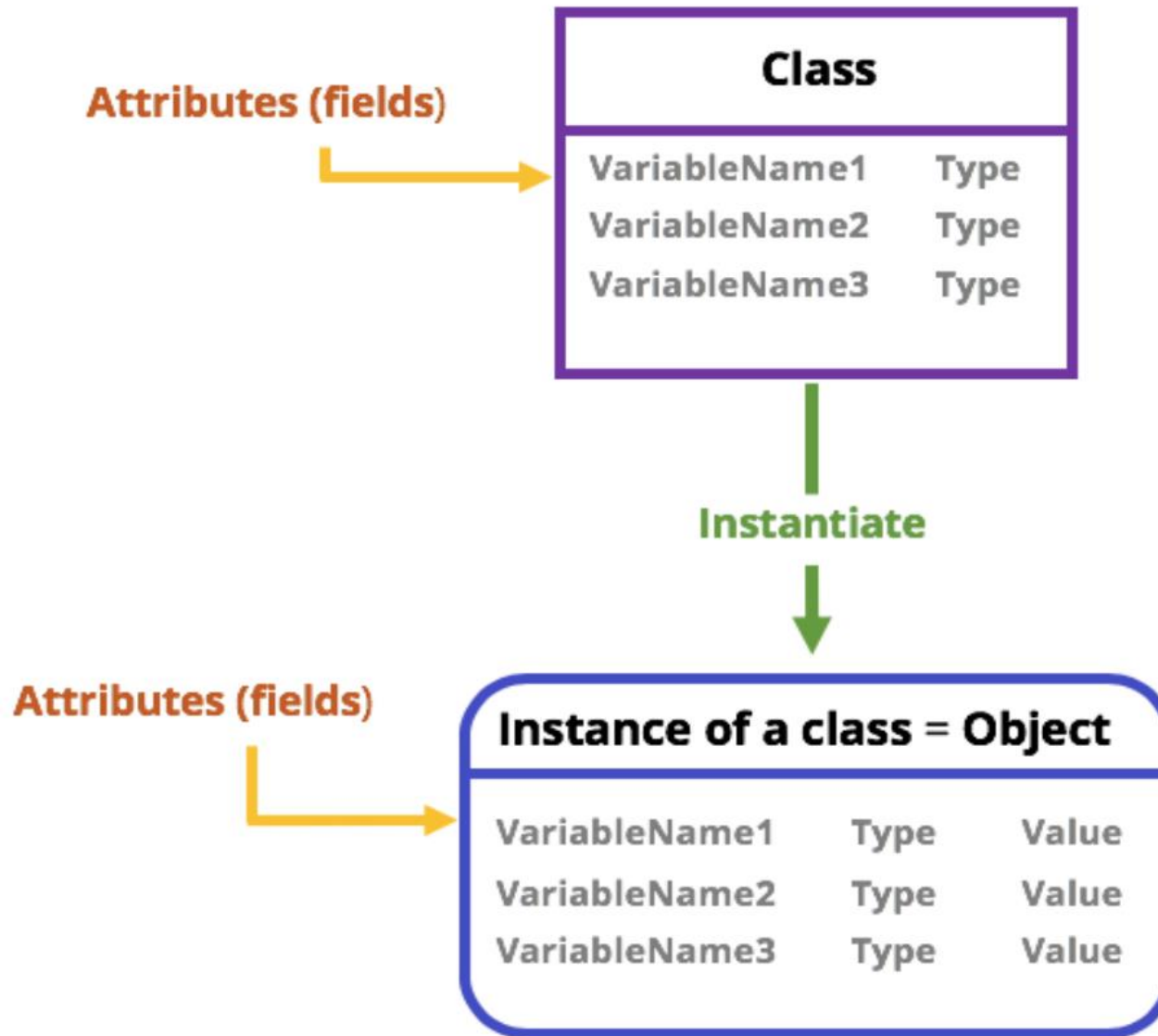


An abstract graphic of glowing blue circuit lines and nodes on a dark blue background, extending from the bottom left towards the center. The lines are of varying thickness and form a complex, branching pattern. Small, bright white dots are placed at various points along these lines, resembling nodes or data points. The overall effect is a sense of digital connectivity and flow.

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 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

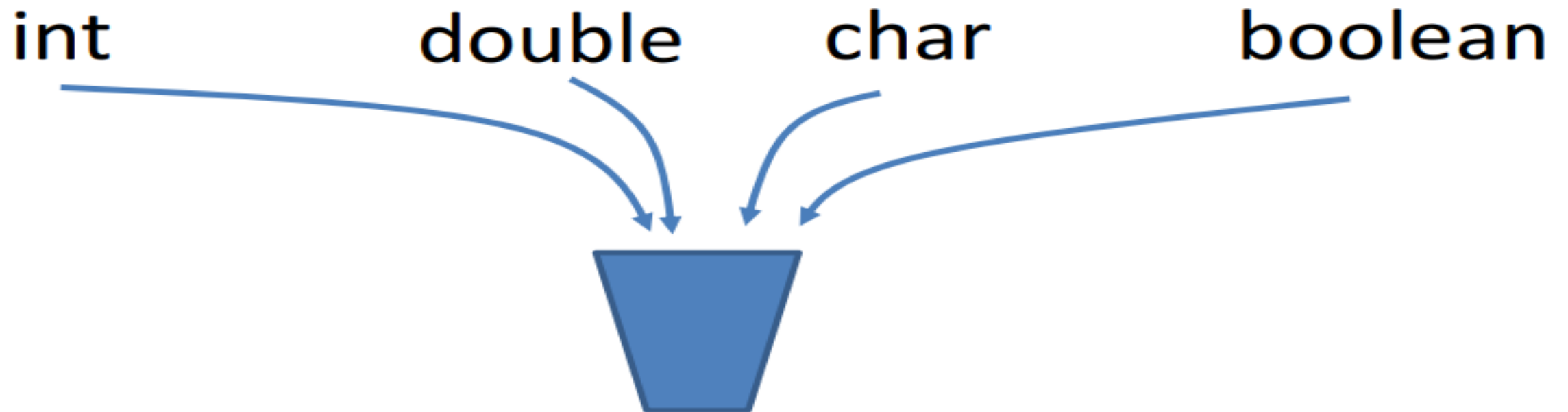
Reference types are arrays and objects – **String, int[], Baby, ...**



How java stores primitives

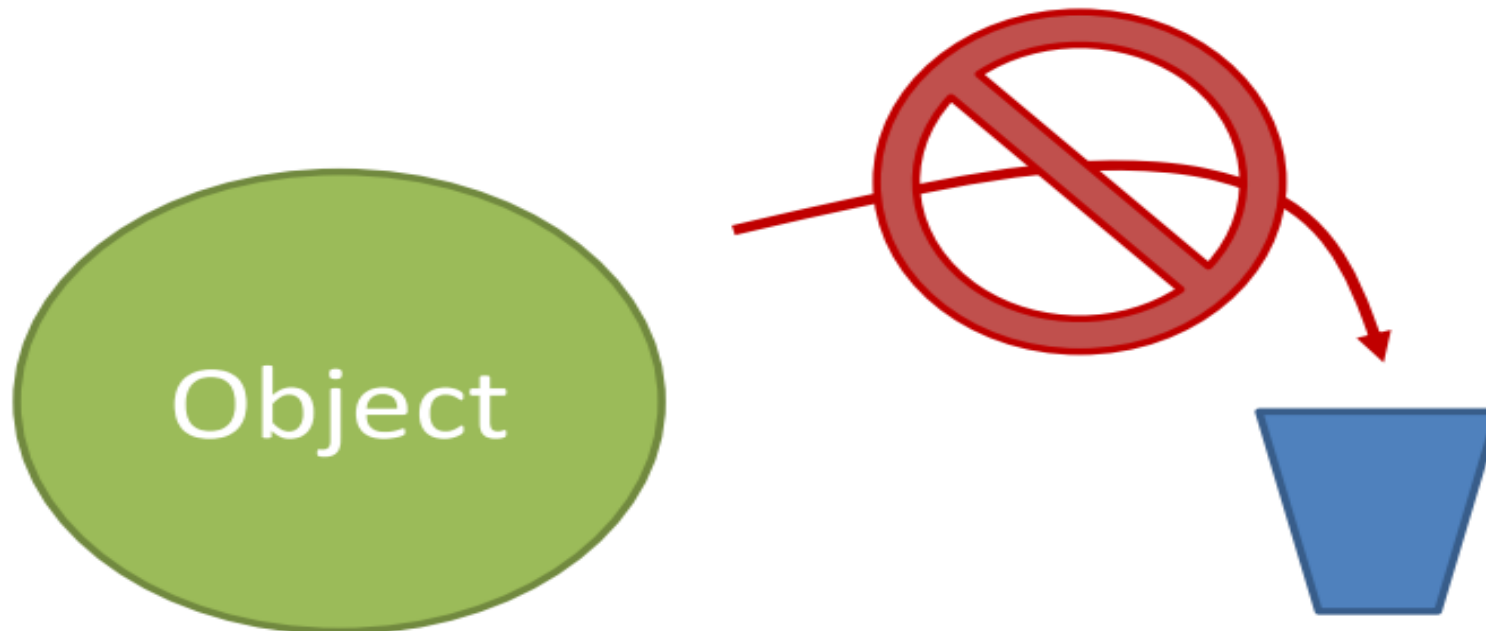
Variables are like fixed size cups

Primitives are small enough that they just fit into the cup



How java stores objects

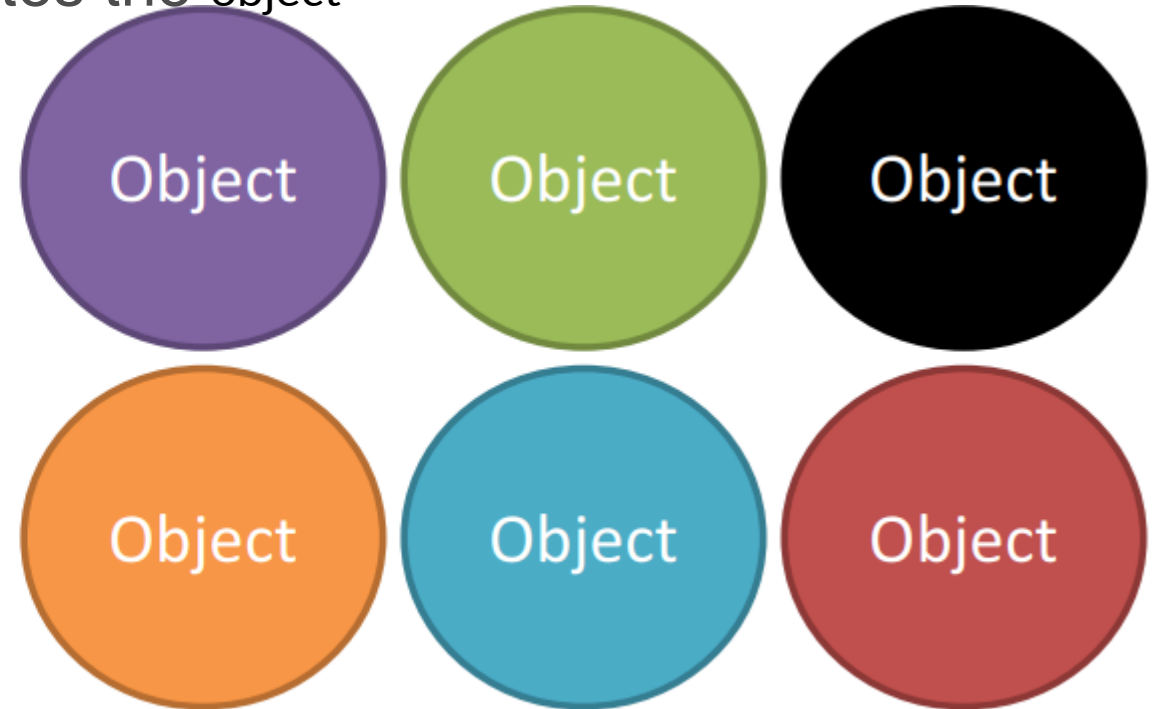
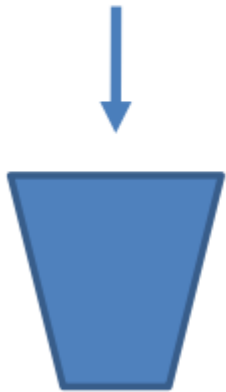
- Objects are too big to fit in a variable
- Stored somewhere else
- Variable stores a number that locates the object



How Java stores objects

Objects are too big to fit in a variable

- Stored somewhere else
- Variable stores a number that locates the object



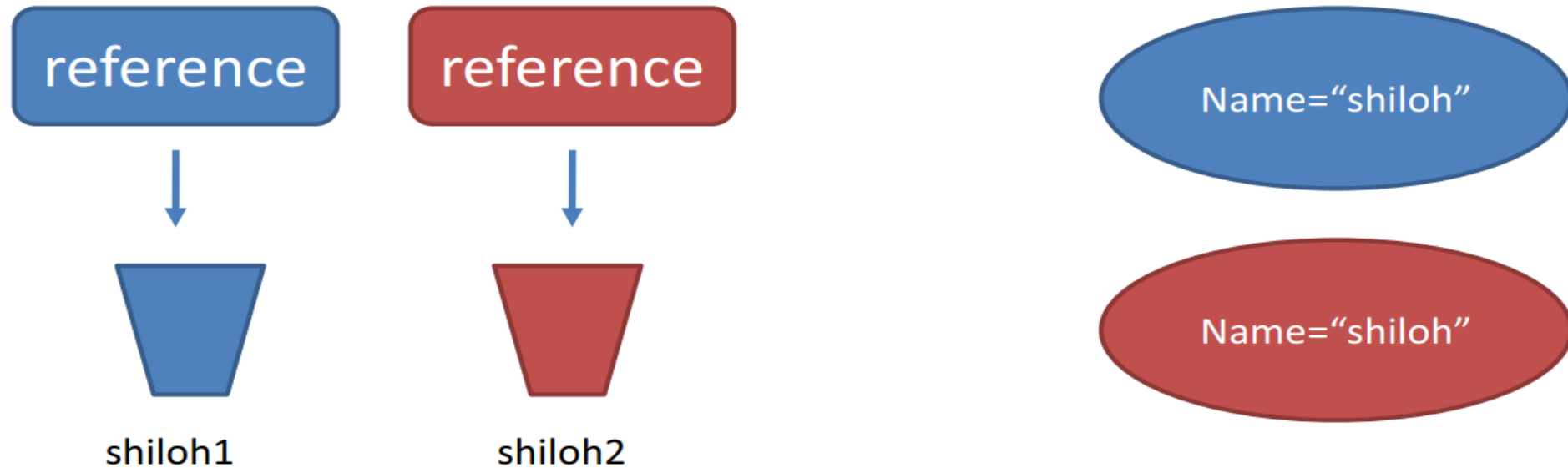
References

- 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

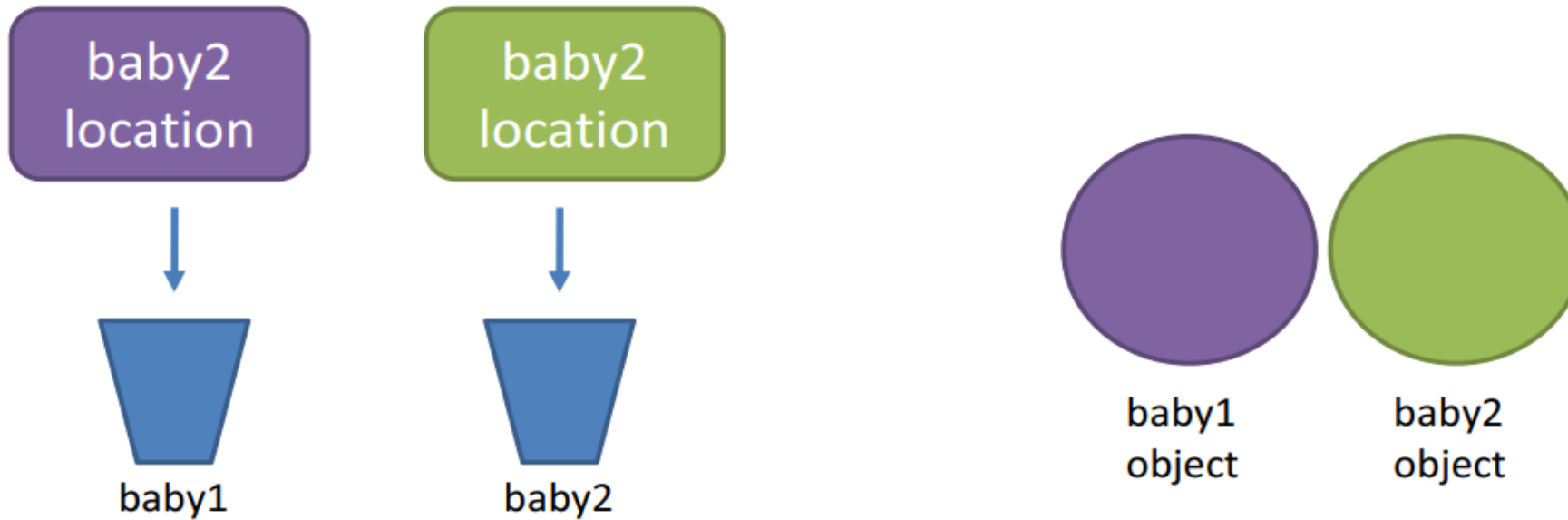
```
Baby shiloh1 = new Baby("shiloh");
Baby shiloh2 = new Baby("shiloh");
```



# References

- Using = updates the reference.

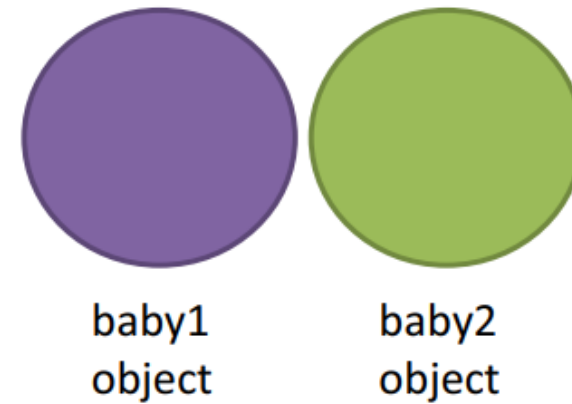
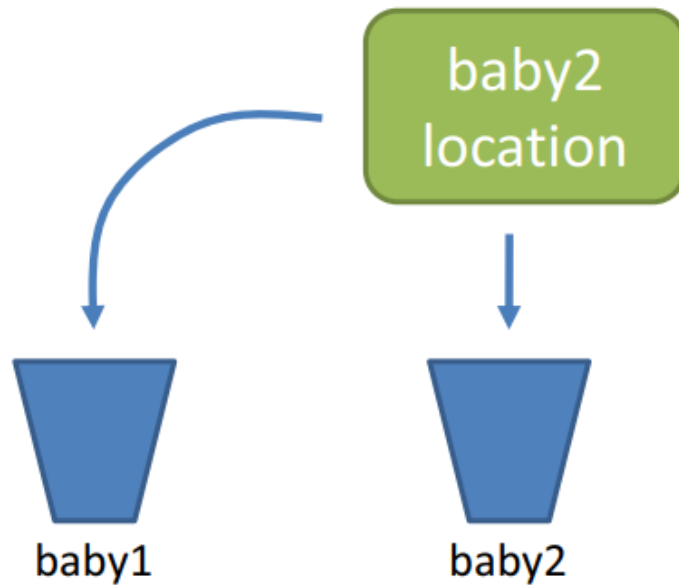
```
baby1 = baby2
```



# 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



An abstract graphic of glowing blue circuit lines and nodes on a dark blue background, extending from the bottom left towards the top right. The lines are of varying thickness and form a complex, branching pattern. Small white dots are placed at various points along these lines, resembling nodes or data points. The overall effect is a sense of digital connectivity and flow.

# 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.

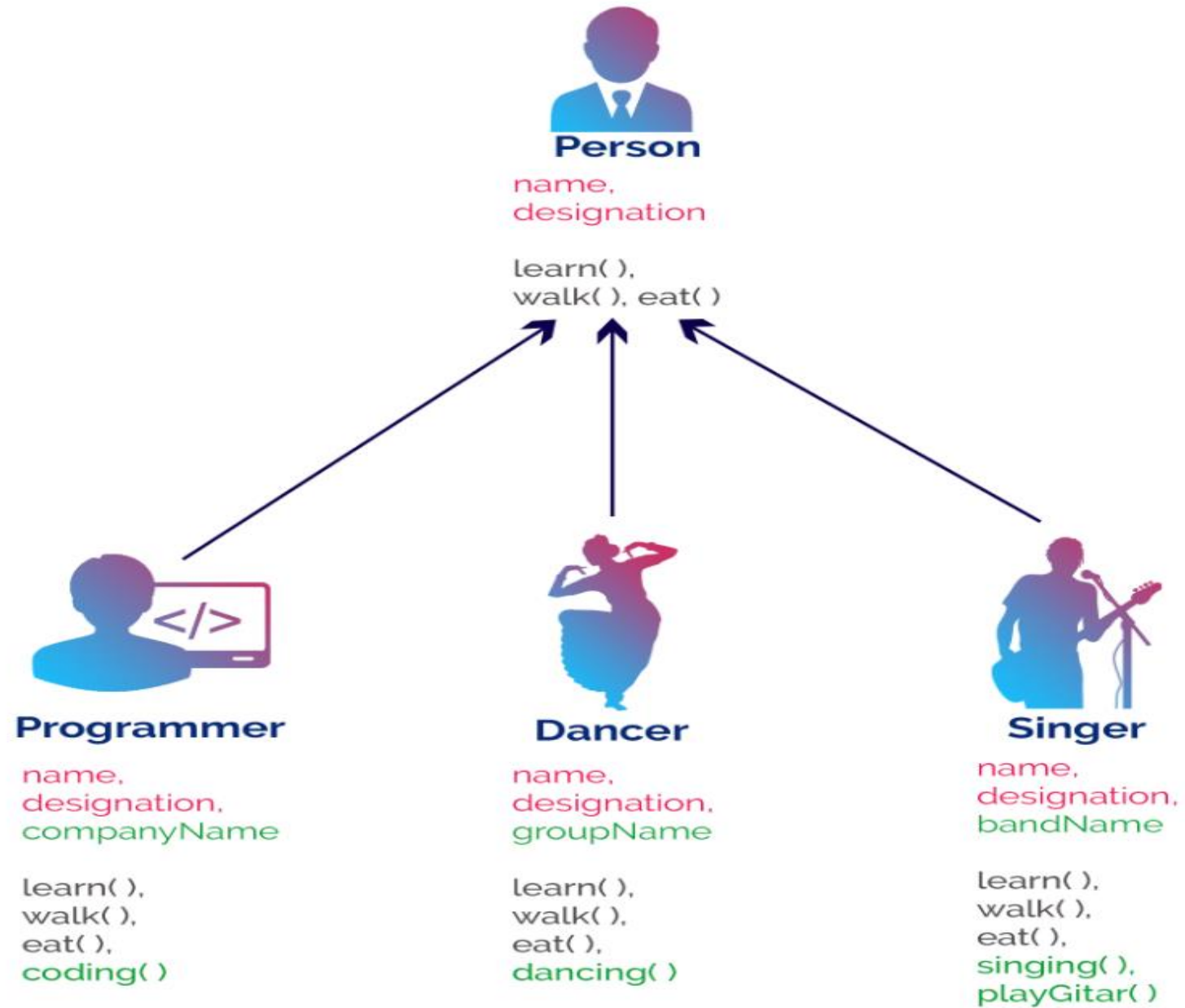
The class which acquires the properties is termed **as sub class/child class/derived class**.

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

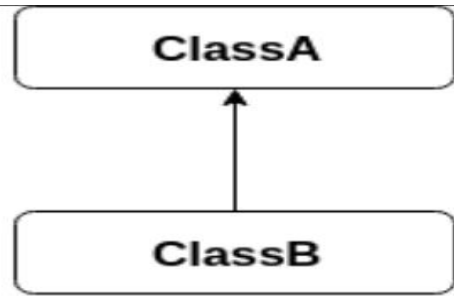
It reduces the line of codes and helps in code reusability.



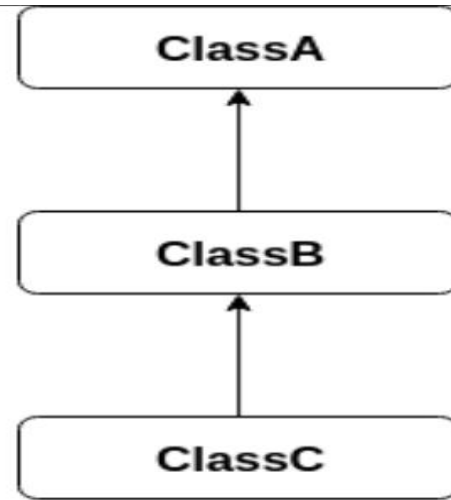
# Inheritance



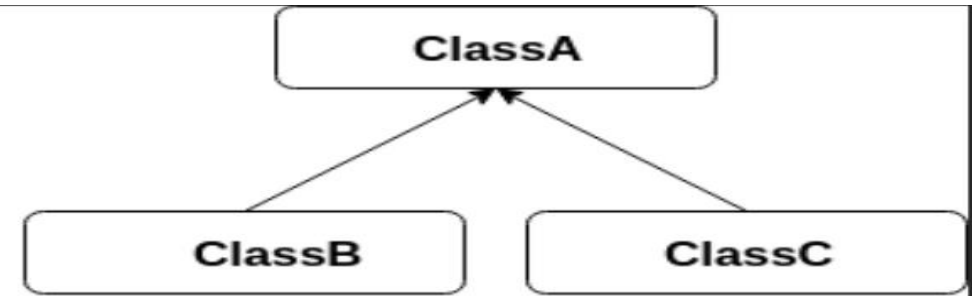
# Types of inheritance



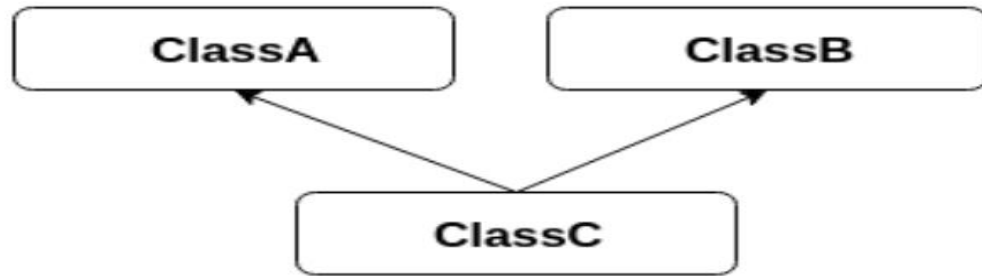
Single Inheritance



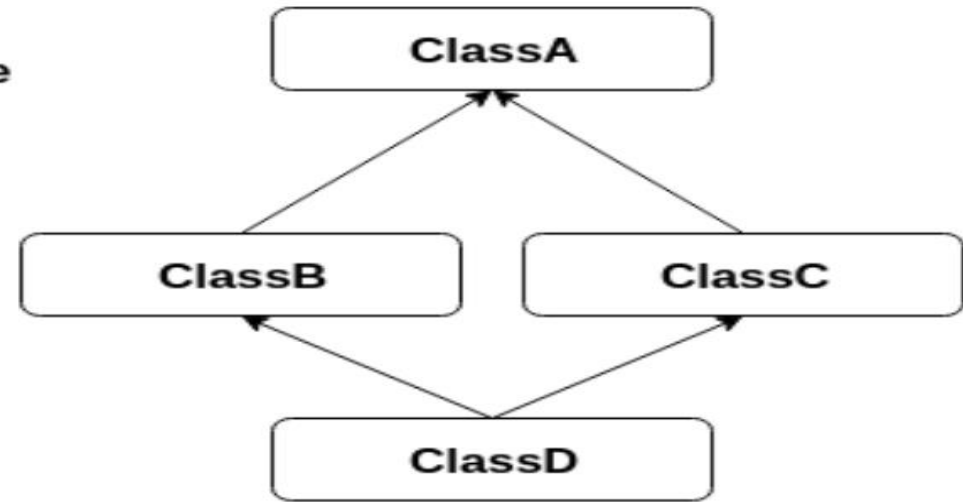
Multilevel Inheritance



Hierarchical Inheritance



Multiple Inheritance



Hybrid Inheritance

# Single Inheritance



HEXAWARE

```
1 package com.hexaware.entities;
2 //Class Object is the root of the class hierarchy. Every class has Object as a superclass.
3 public class Person extends Object {
4 // instance/member variables
5 private int uId;
6 private String name;
7 private int age;
8 private long mobileNumber;
9 // class level variable
10 public static String collegeName = "SSN";
11 // default constructor
12 public Person() {
13 super();
14 }
15 // parameterised
16 public Person(int uId, String name, int age, long mobileNumber) {
17 super();
18 this.uId = uId;
19 this.name = name;
20 this.age = age;
21 this.mobileNumber = mobileNumber;
22 }
23 }
```



# Single Inheritance

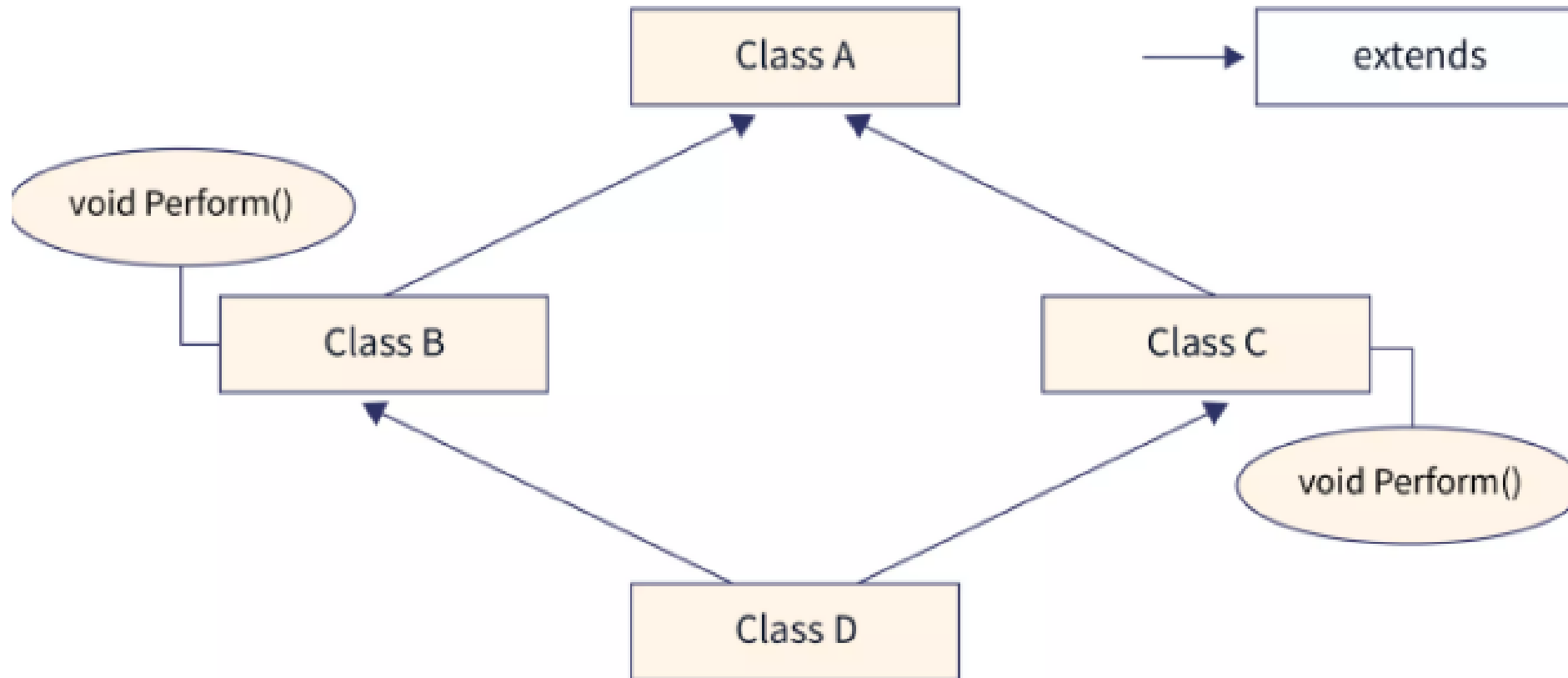
```
1 package com.hexaware.entities;
2 public class Employee extends Person {
3 private int employeeId;
4 private int cabinNumber;
5 private String designation;
6 private double salary;
7 // all 8 variables and one static variable gets allocated with the memory
8 public Employee() {
9 super();
10 // TODO Auto-generated constructor stub
11 }
12
13 public Employee(int employeeId, int cabinNumber, String designation, double salary,
14 int uId, String name, int age, long mobileNumber) {
15 // call to superclass parameterised constructor. static binding/compiletime
16 // polymorphism
17 super(uId, name, age, mobileNumber);
18 this.employeeId = employeeId;
19 this.cabinNumber = cabinNumber;
20 this.designation = designation;
21 this.salary = salary;
22 }
```



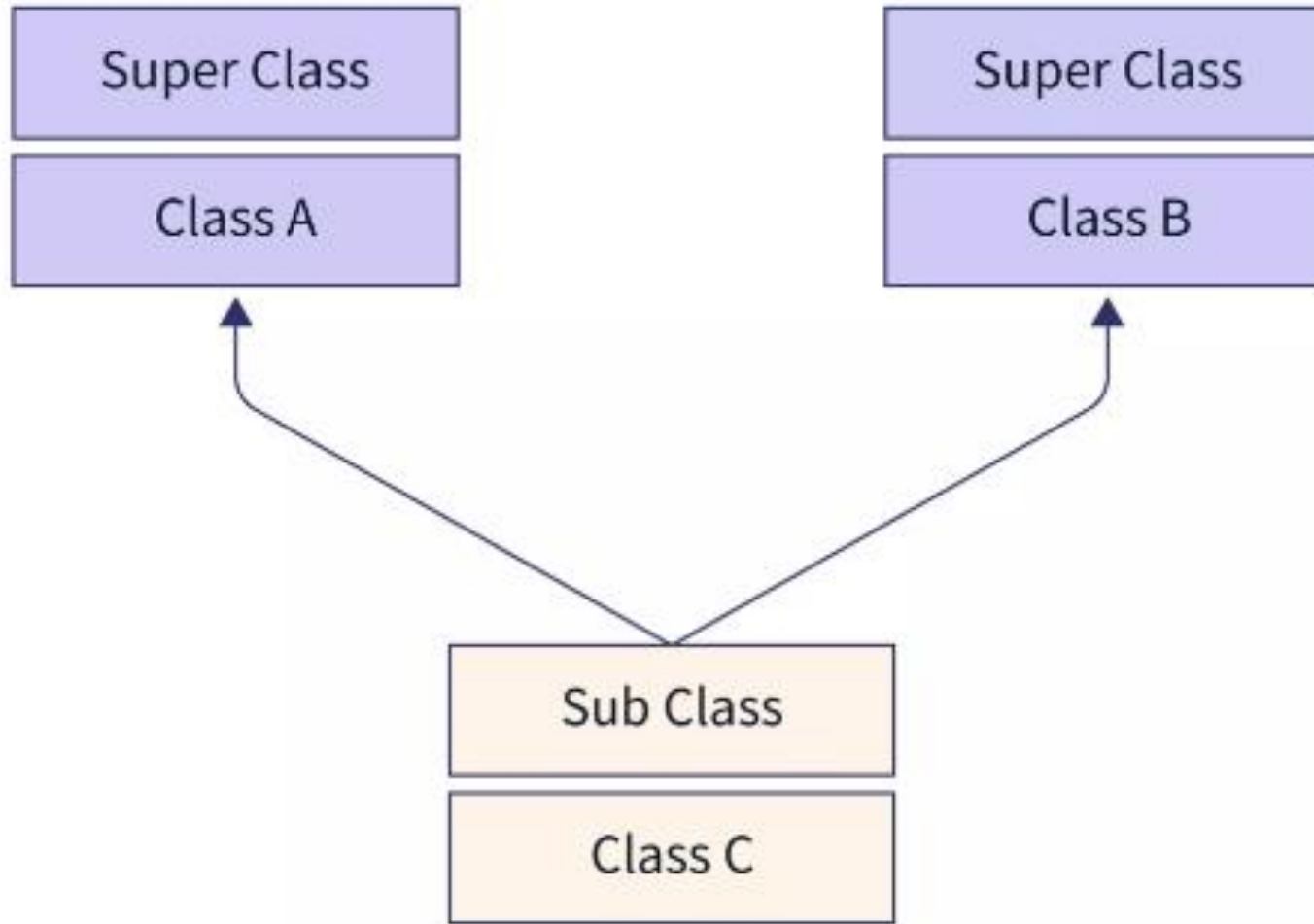
# Demo on Single Inheritance



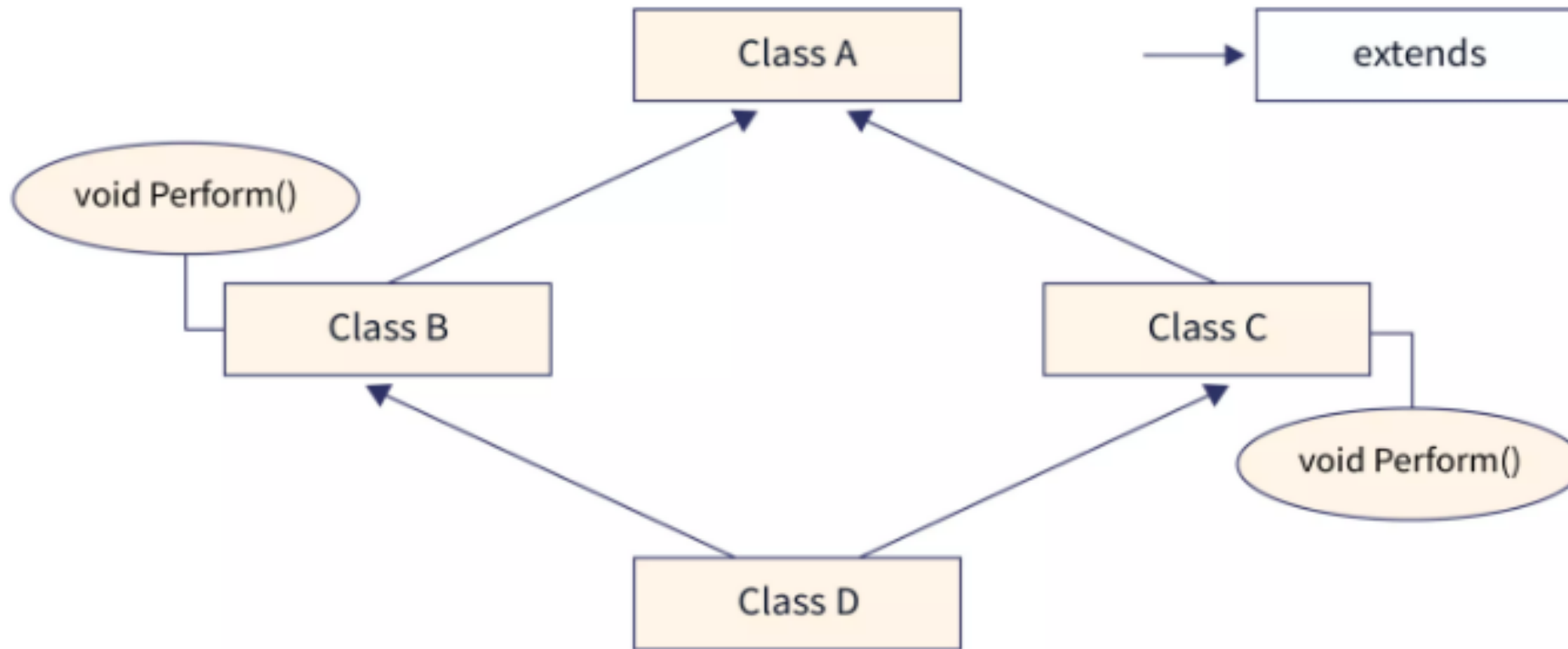
# Multiple Inheritance



# Multiple Inheritance



# Multiple Inheritance

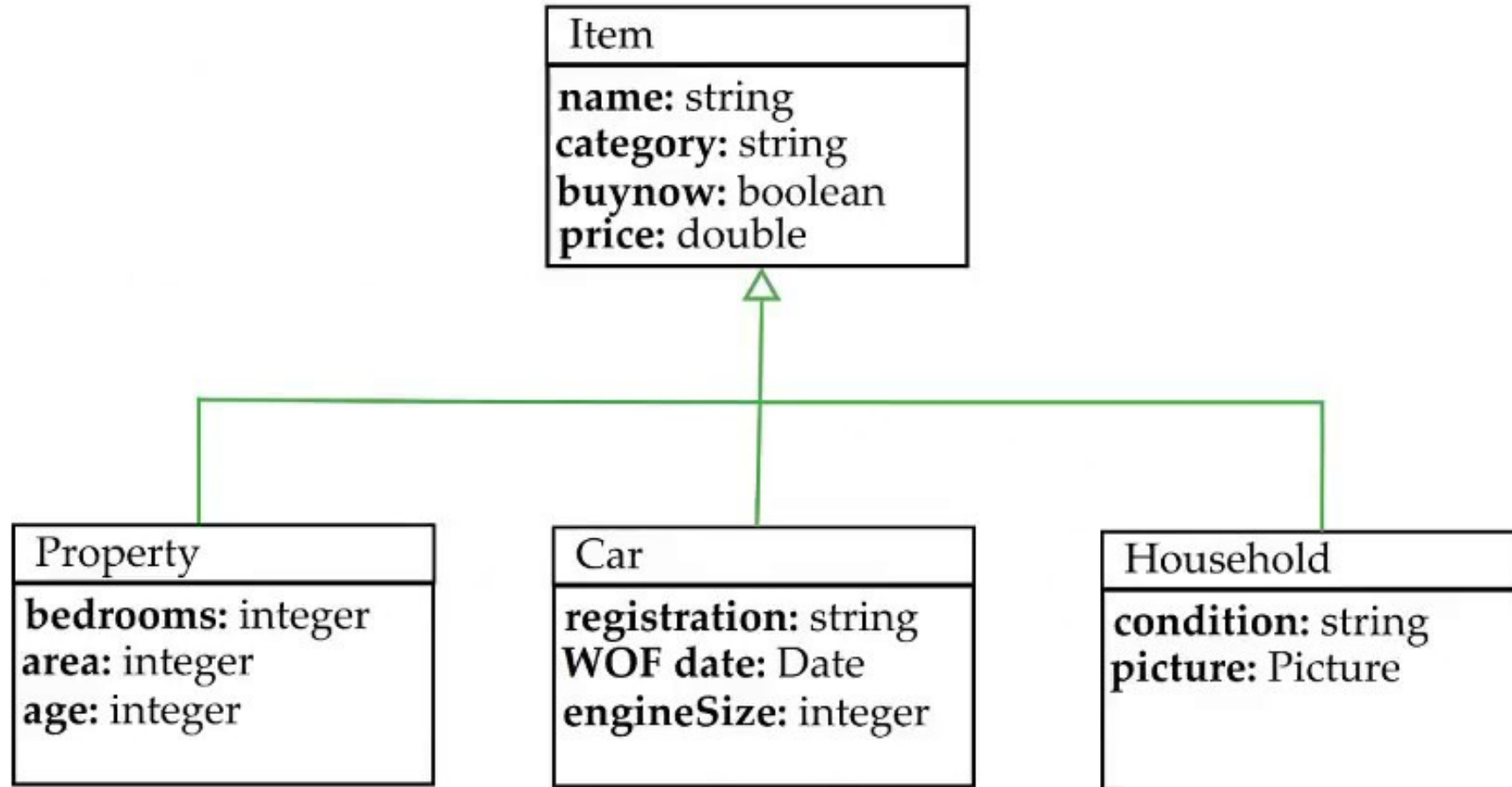


# Multiple Inheritance

- 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



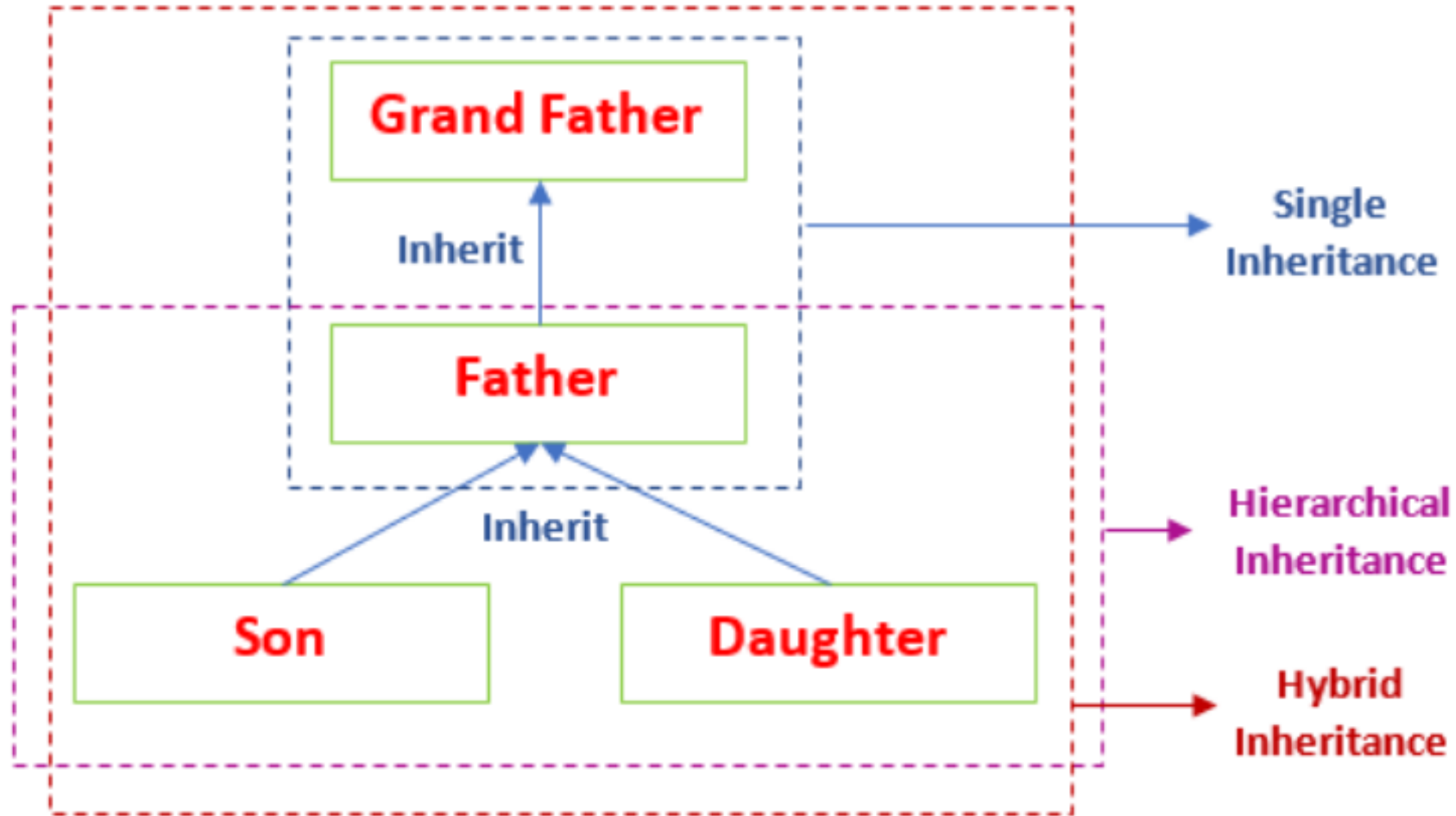


# Demo on Hierarchical Inheritance





# Hybrid Inheritance





# Demo on Hybrid Inheritance



An abstract graphic featuring a complex, glowing blue circuit pattern that flows from the bottom left towards the top right. The pattern consists of numerous thin, interconnected lines and small, bright white dots, creating a sense of dynamic energy and technological sophistication. The background is a solid, deep blue. The word "Polymorphism" is centered in the middle of the image, overlaid on the circuit pattern.

# 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

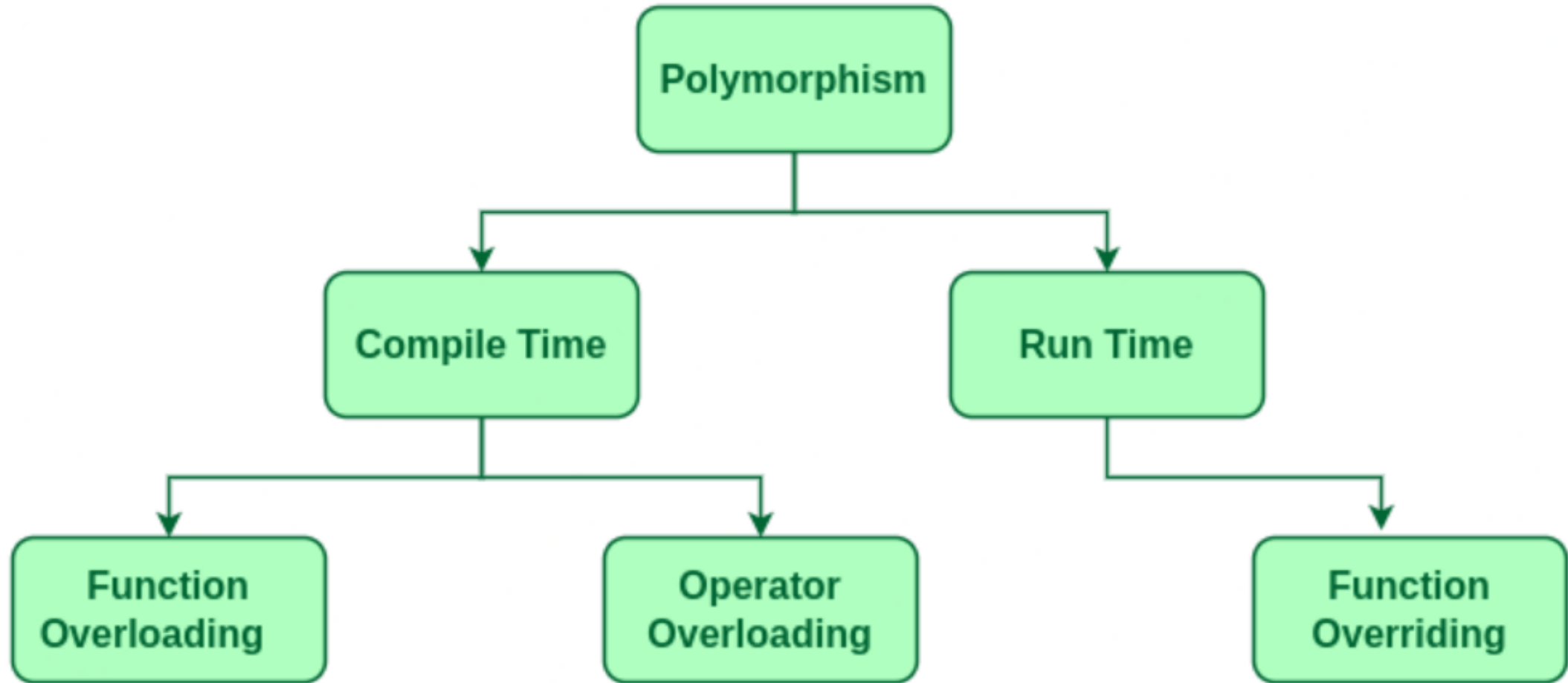
In Bus behave like Passenger

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.

- `add(int, int)`
- `add(int, int, int)`

### Data type of parameters.

- `add(int, int)`
- `add(int, float)`

### Sequence of parameters.

- `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

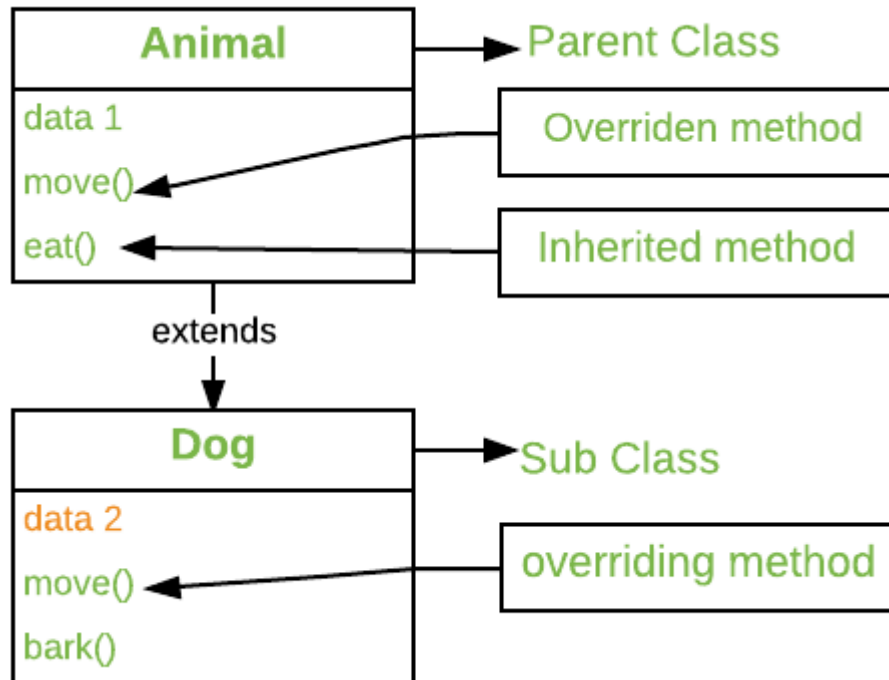


# Method Overriding

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

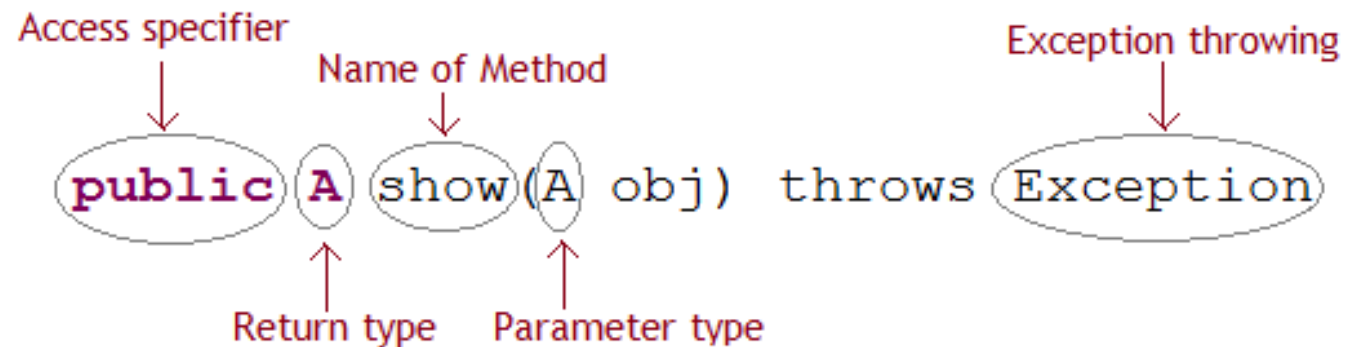
method in **parent class** is called **overridden method**.

the method in **child class** is called **overriding method**.



## Method Overriding

Parameters to consider,





An abstract graphic of glowing blue circuit lines and nodes on a dark blue background, with the text overlaid in the center-left.

# 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-riding method in the **sub class cannot be either private or protected**.



# Rules of method overriding

- An **overriding** method can throw any **unchecked 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.

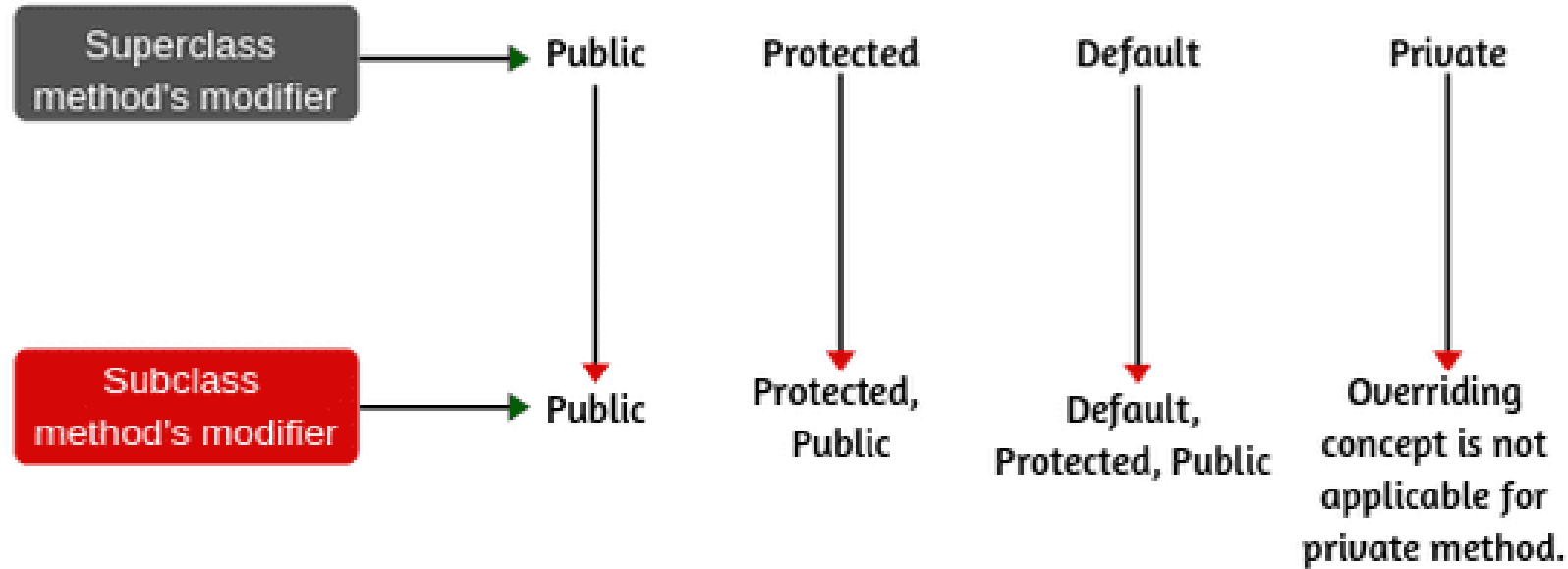




# Method Overriding Demo



# 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 \(mygreatlearning.com\)](#)







*Passionate Employees*

*Delighted Customers*

*Thank you*

---

[www.hexaware.com](http://www.hexaware.com)