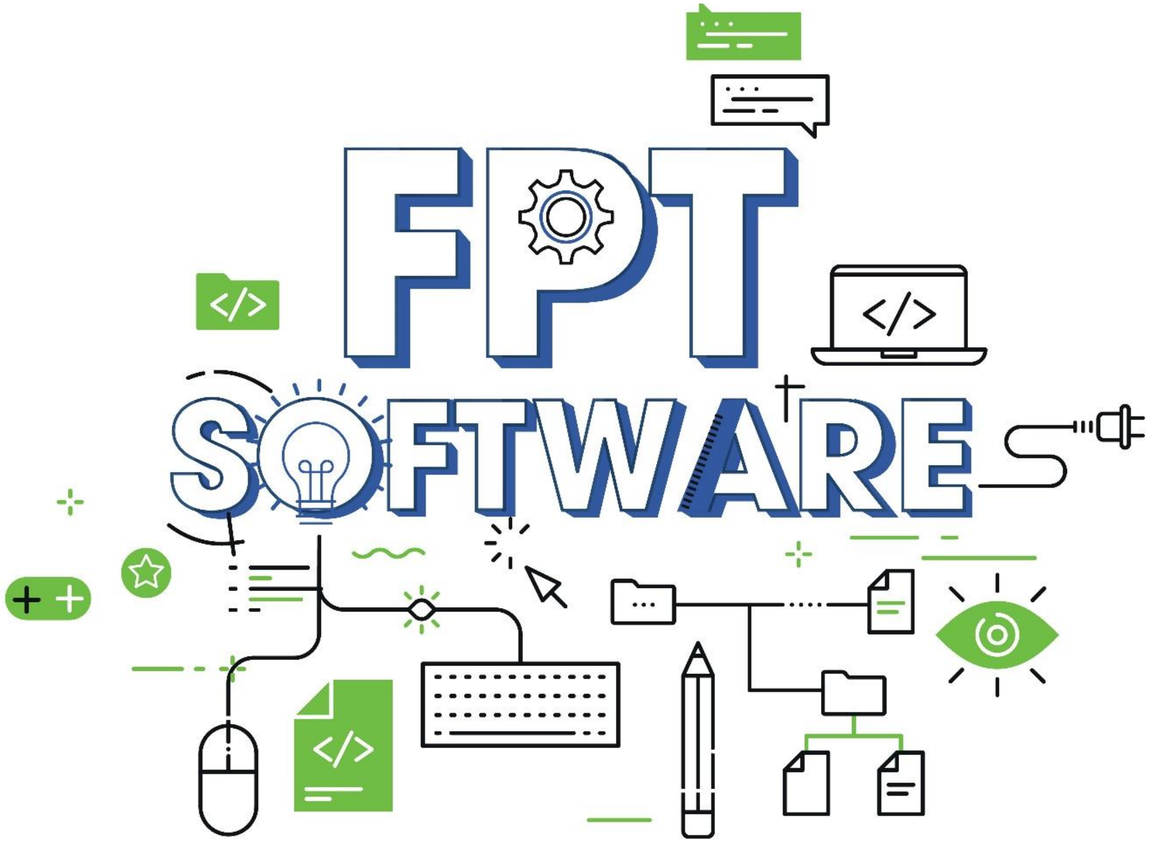
OOP IN JAVA

Instructor: DieuNT1



Agenda

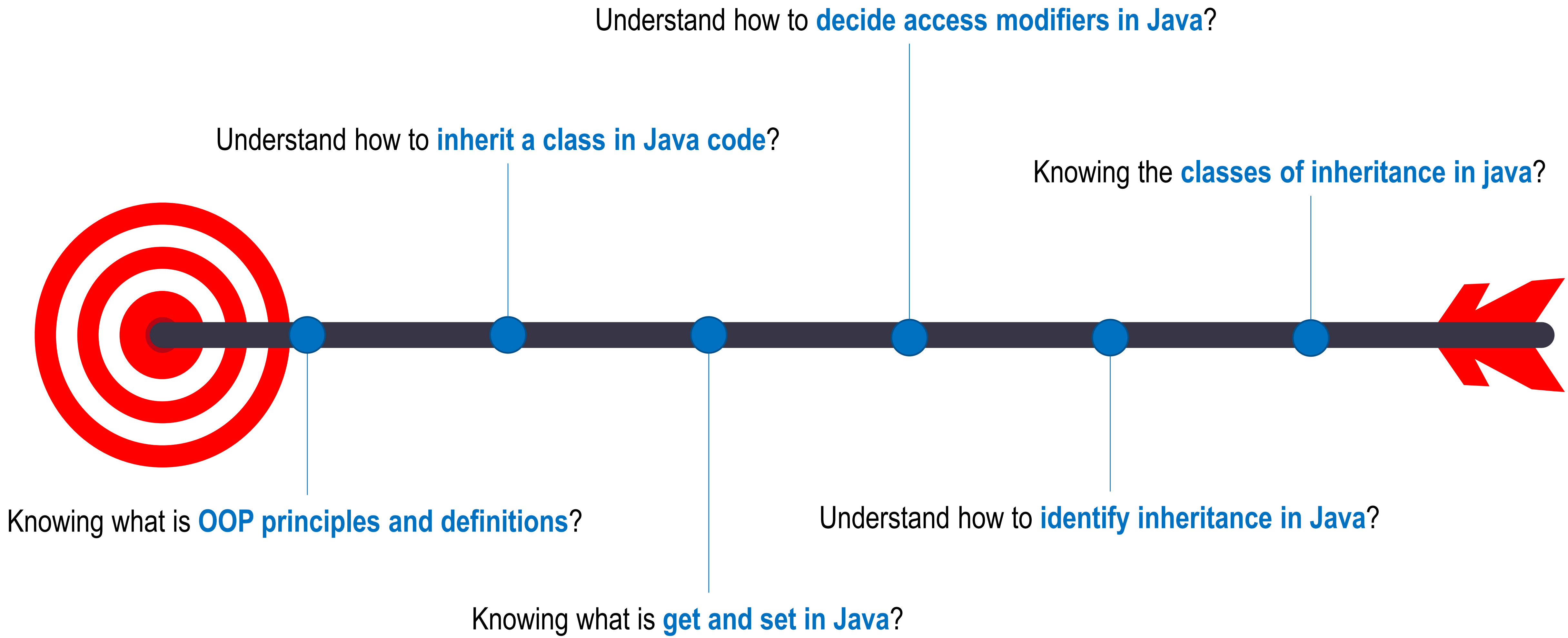
01. OOP Introduction

02. Principles of Object-Oriented Programming

- ✓ Encapsulation
- ✓ Inheritance
- ✓ Abstraction
- ✓ Polymophims

03. Q&A

Lesson Objectives



Knowing what is **OOP principles and definitions**?

Understand how to **inherit a class in Java code**?

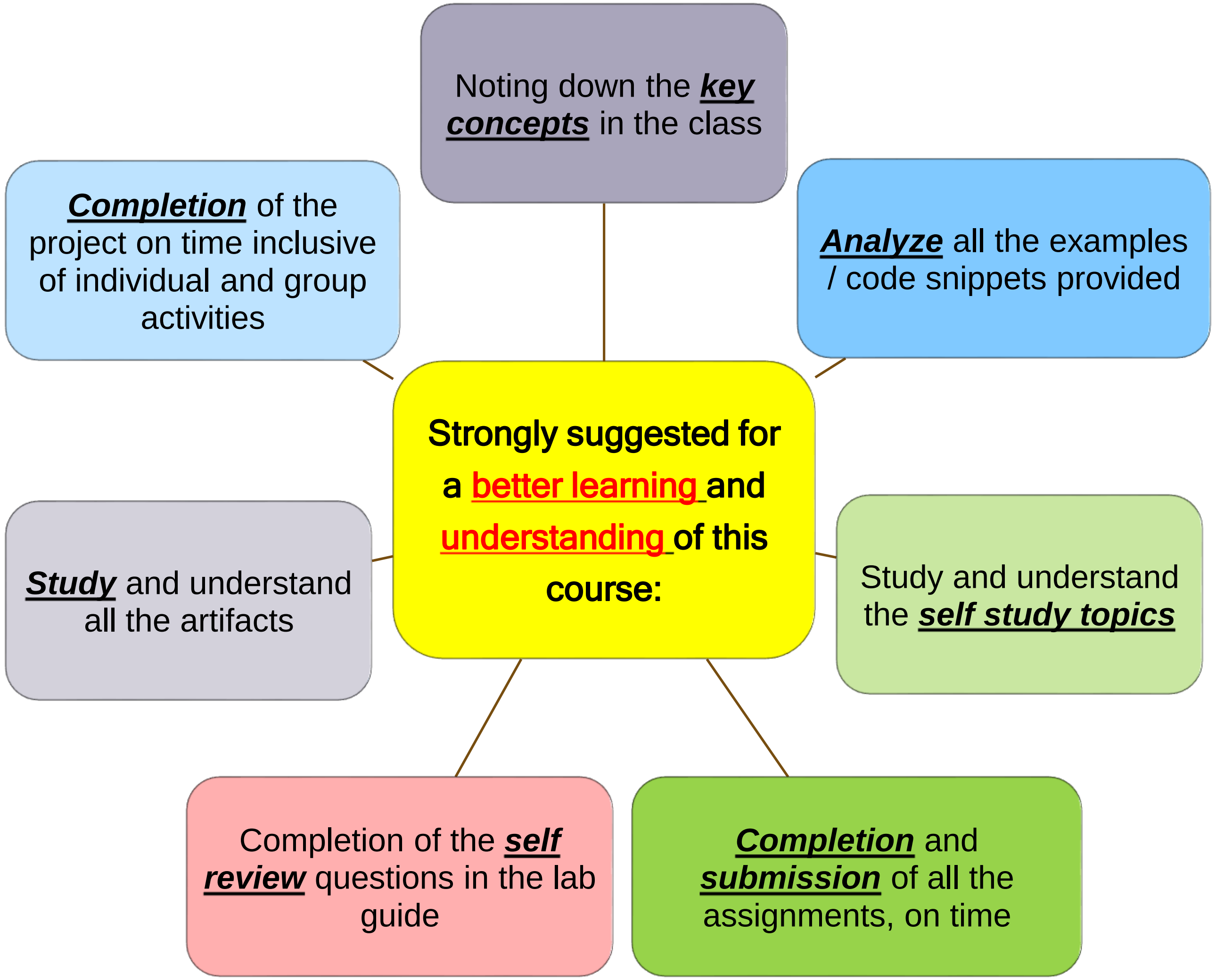
Knowing what is **get and set in Java**?

Understand how to **decide access modifiers in Java**?

Understand how to **identify inheritance in Java**?

Knowing the **classes of inheritance in java**?

Learning Approach



Strongly suggested for a <u>better learning</u> and <u>understanding</u> of this course:

Noting down the ***key concepts*** in the class

Analyze all the examples / code snippets provided

Study and understand the ***self study topics***

Completion and ***submission*** of all the assignments, on time

Completion of the ***self review*** questions in the lab guide

Study and understand all the artifacts

Completion of the project on time inclusive of individual and group activities

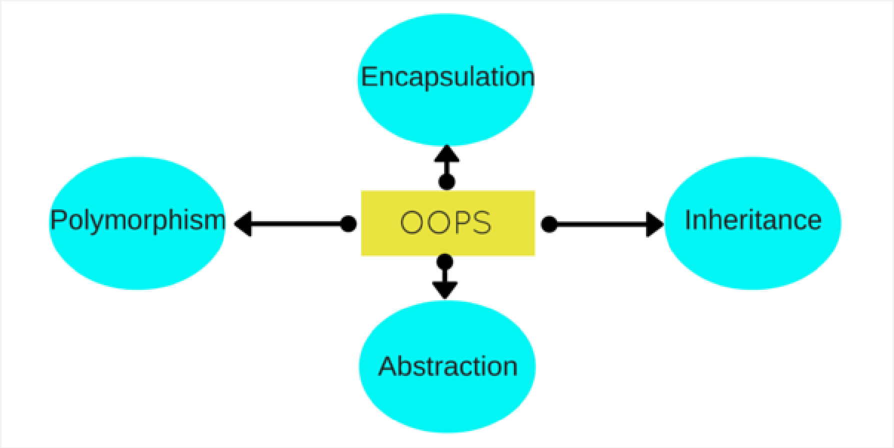
Section 1

Introduction

Introduction

Object - Oriented Programming system (OOPs) is a programming paradigm based on the concept of "objects" that contain data and methods.

- The primary purpose of object-oriented programming is to **increase the flexibility and maintainability** of programs.
- Java is an **object oriented language** because it provides the features to implement an object oriented model.
- These features includes **Abstraction**, **Encapsulation**, **Inheritance** and **Polymorphism**.



Classes and Objects - Recap

In Java, **classes** serve as blueprints for creating objects.
A class defines the properties (*attributes*) and behaviors (*methods*) that objects of that class will possess.
Objects are instances of a class, created using the new keyword.

- **Example:**

```java
public class Car {
    // Attributes
    private String brand;
    private String color;
    // Constructor
    public Car(String brand, String color) {
        this.brand = brand;
        this.color = color;
    }
    // Method
    public void drive() {
        System.out.println("Driving the " + brand + " car in " + color + " color.");
    }
    // getter and setter methods
}
```

Classes and Objects - Recap

- **Creating objects:**

```
//Creating objects
Car car1 = new Car("Toyota", "Red");
Car car2 = new Car("Honda", "Blue");

//Accessing attributes and invoking methods
System.out.println(car1.getBrand()); // Output: Toyota
car2.drive(); // Output: Driving the Honda car in Blue color.
```

- **Output:**

```
Toyota
Driving the Honda car in Blue color.
```

Encapsulation

Encapsulation is a mechanism that bundles data (attributes) and methods together within a class, hiding the internal implementation details from outside access.

- It helps achieve data abstraction and provides control over access to class members using access modifiers (*private*, *protected*, *public*, etc.).

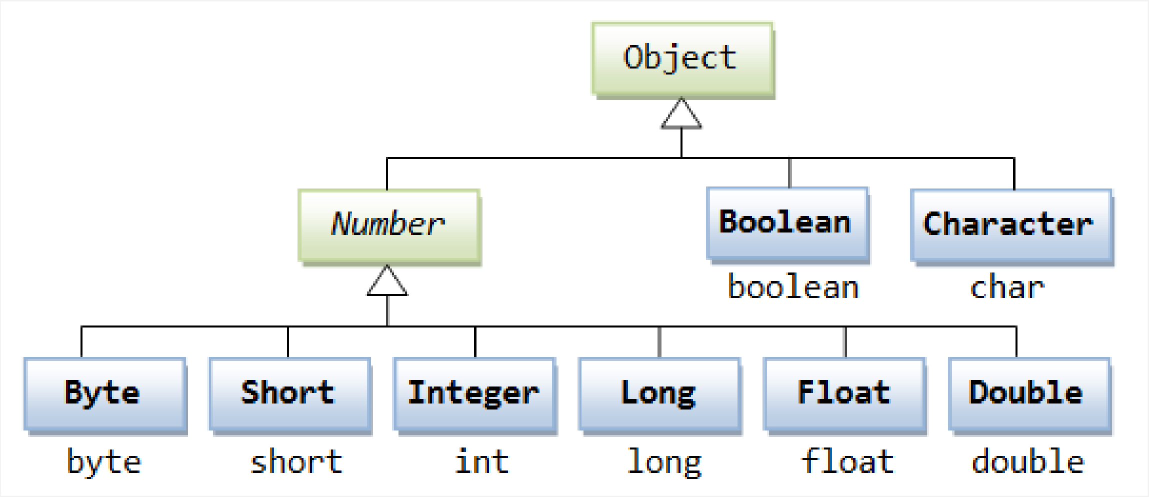
```
public class Car {
    // Attributes
    private String brand;
    private String color;
    // Constructor
    public Car(String brand, String color) {
    this.brand = brand;
    this.color = color;
    }
    // Method
    public void drive() {
        System.out.println("Driving the " + brand +
        " car in " + color + " color.");
    }
```

```
  // Getters and setters for encapsulated attributes
  public String getBrand() {
    return brand;
  }
  public void setBrand(String brand) {
     this.brand = brand;
  }

}
```

Inheritance

Inheritance allows the creation of new classes (derived or child classes) based on existing classes (base or parent classes).

- The **derived class** inherits the *attributes* and *methods* of the base class and can add its own unique characteristics.
- Java **supports** single inheritance (a class can inherit from only one class) but allows for multiple levels of inheritance.



Inheritance

- **Example:**

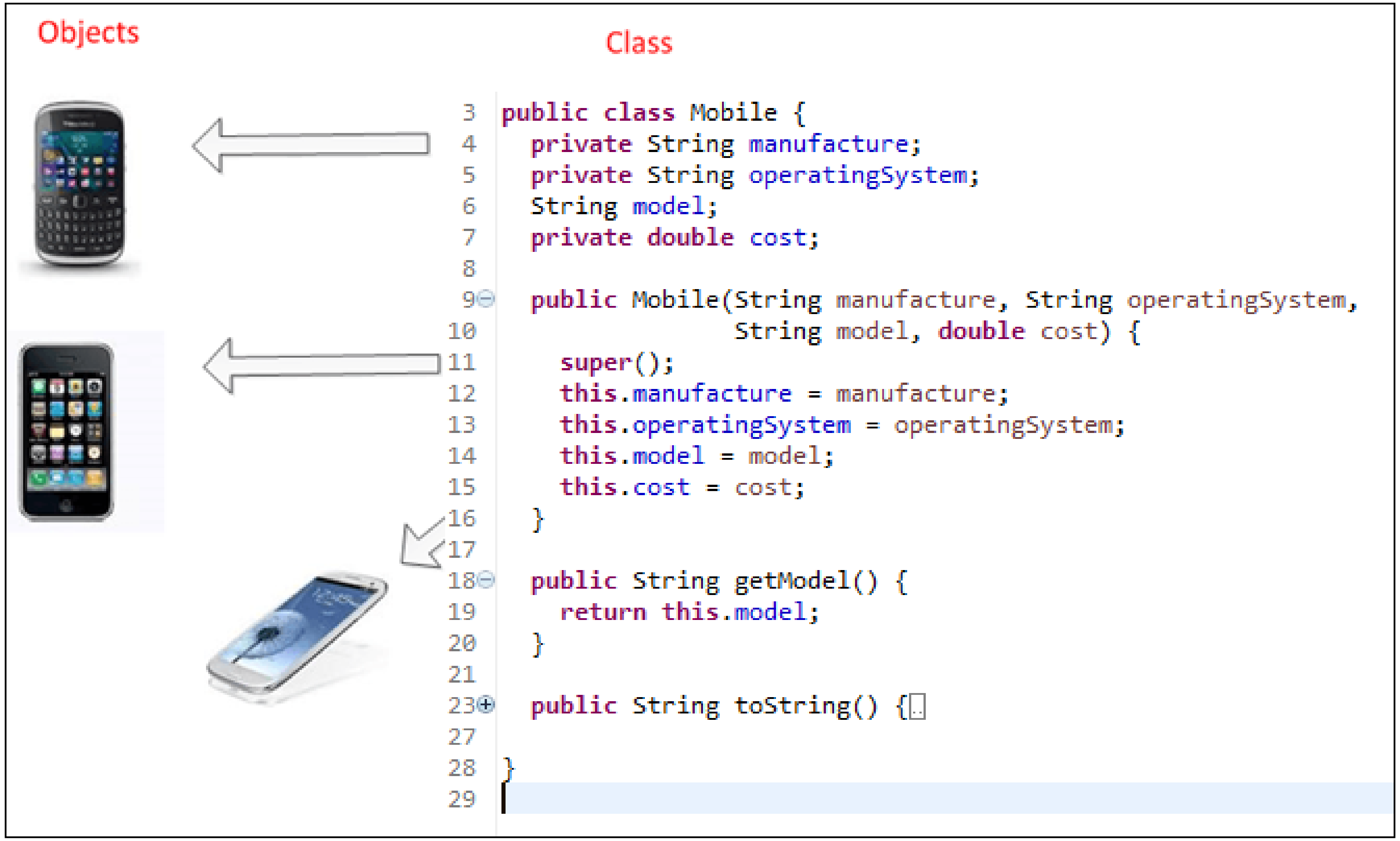
```
public class Car extends Vehicle {
     // Attributes
     private String brand;
     private String color;

     // Constructor
     public Car(String brand, String color) {
          super(brand);
          this.color = color;
     }

     // Method
     public void drive() {
          System.out.println("Driving the " +
                    brand + " car in " + color + " color.");
     }
}
```

```
// Creating objects and invoking inherited methods
Car car = new Car("Toyota", "Red");
car.start(); // Output: Starting the Toyota vehicle.
car.drive(); // Output: Driving the Toyota car in Red color.
```

Inheritance

- You can look into the following example for inheritance concept.

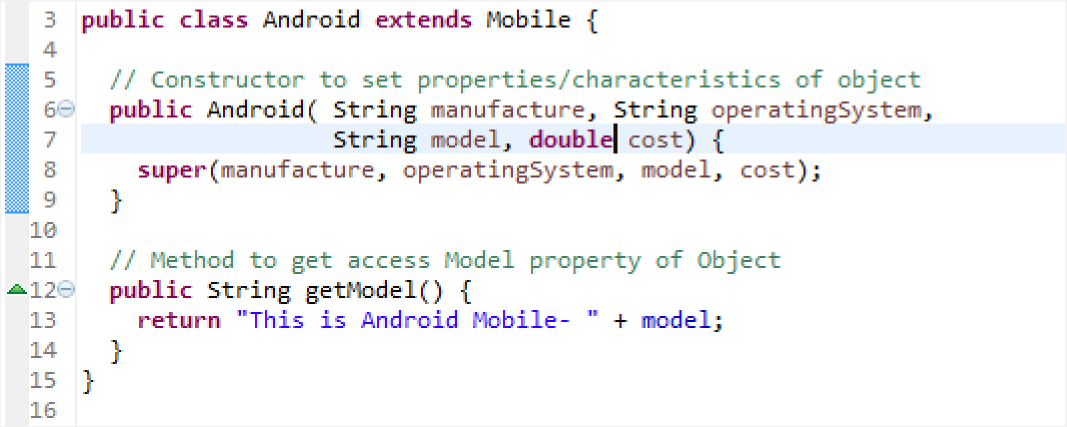
- **Mobile** class:



```java
public class Mobile {
  private String manufacture;
  private String operatingSystem;
  String model;
  private double cost;

  public Mobile(String manufacture, String operatingSystem,
                String model, double cost) {
    super();
    this.manufacture = manufacture;
    this.operatingSystem = operatingSystem;
    this.model = model;
    this.cost = cost;
  }

  public String getModel() {
    return this.model;
  }

  public String toString() {

}
```

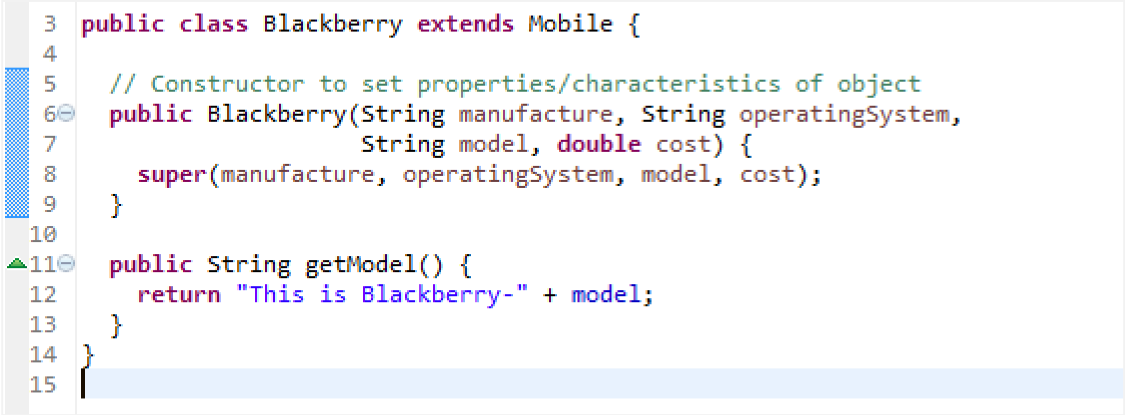
Inheritance

- The **Mobile** class extended by other specific class like **Android** and **Blackberry**.
- **Android** class:



```java
public class Android extends Mobile {

  // Constructor to set properties/characteristics of object
  public Android( String manufacture, String operatingSystem,
                  String model, double cost) {
    super(manufacture, operatingSystem, model, cost);
  }

  // Method to get access Model property of Object
  public String getModel() {
    return "This is Android Mobile- " + model;
  }
}

```

- **Blackberry** class:



```java
public class Blackberry extends Mobile {

  // Constructor to set properties/characteristics of object
  public Blackberry(String manufacture, String operatingSystem,
                    String model, double cost) {
    super(manufacture, operatingSystem, model, cost);
  }

  public String getModel() {
    return "This is Blackberry-" + model;
  }
}

```

Polymorphism

Polymorphism is the ability of objects of different classes to respond differently to the same method call.

- If *one task is performed by different ways*, it is known as polymorphism.
- In Java, polymorphism is achieved through method overriding and method overloading.
- **Example:**

```java
//Base class
public class Animal {
    public void makeSound() {
        System.out.println("The animal makes a sound.");
    }
}
```

Polymorphism

- **Example:**

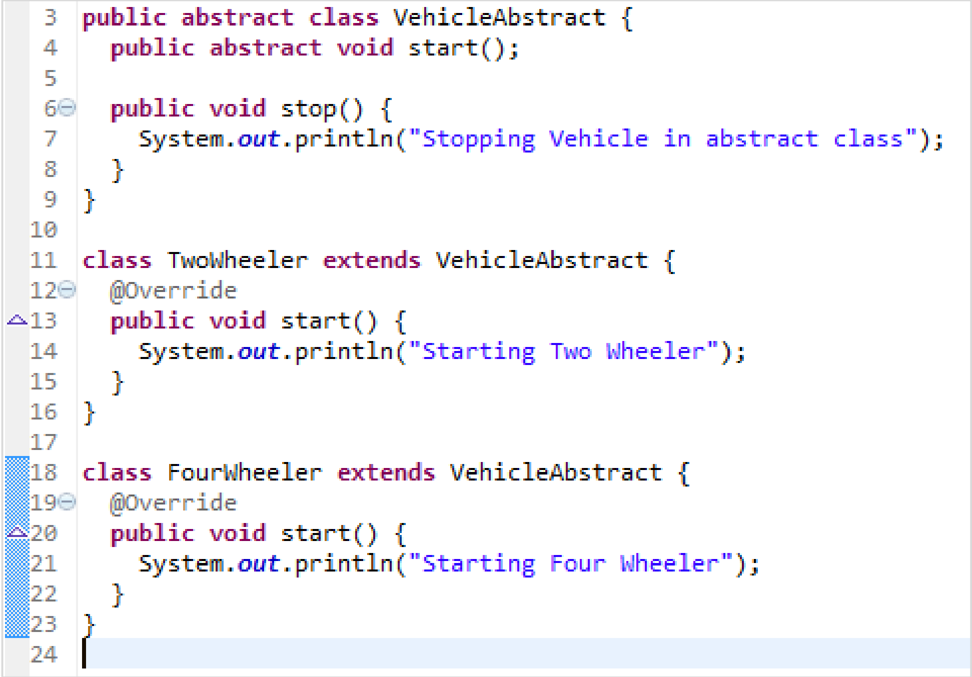
```
//Derived classes
public class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("The dog barks.");
    }
}
```

```
public class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("The cat meows.");
    }
}
```

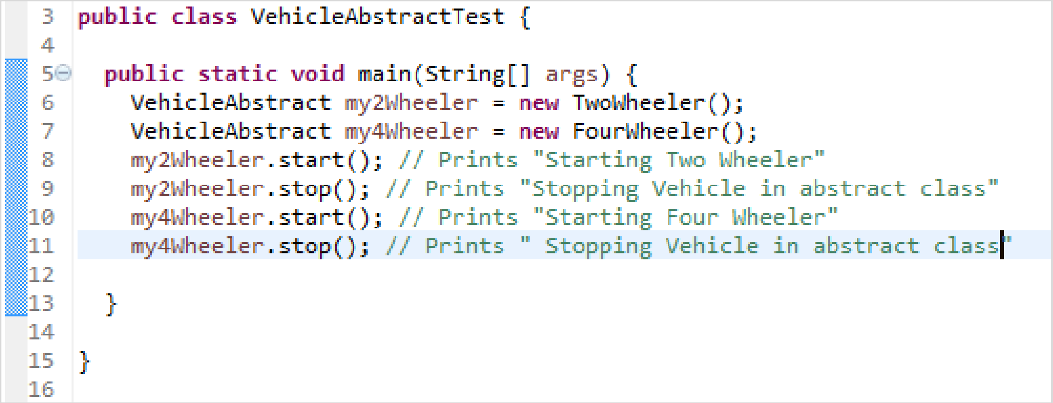
Abstraction

Abstraction is a process which *displays only the information needed* and *hides the unnecessary information*. We can say that the main purpose of abstraction is data hiding.

- **Abstraction** means selecting data from a large number of data to show the information needed, which helps in reducing programming complexity and efforts.
- Use **abstract class** and **interface** to achieve abstraction.



```java
public abstract class VehicleAbstract {
  public abstract void start();

  public void stop() {
    System.out.println("Stopping Vehicle in abstract class");
  }
}

class TwoWheeler extends VehicleAbstract {
  @Override
  public void start() {
    System.out.println("Starting Two Wheeler");
  }
}

class FourWheeler extends VehicleAbstract {
  @Override
  public void start() {
    System.out.println("Starting Four Wheeler");
  }
}
```



```java
public class VehicleAbstractTest {

  public static void main(String[] args) {
    VehicleAbstract my2Wheeler = new TwoWheeler();
    VehicleAbstract my4Wheeler = new FourWheeler();
    my2Wheeler.start(); // Prints "Starting Two Wheeler"
    my2Wheeler.stop(); // Prints "Stopping Vehicle in abstract class"
    my4Wheeler.start(); // Prints "Starting Four Wheeler"
    my4Wheeler.stop(); // Prints " Stopping Vehicle in abstract class"

  }

}
```

Section 2

OOP Principles in Java

OOP Principles in Java

- ✓**Encapsulation**
- ✓Inheritance
- ✓Abstraction
- ✓Polymophims

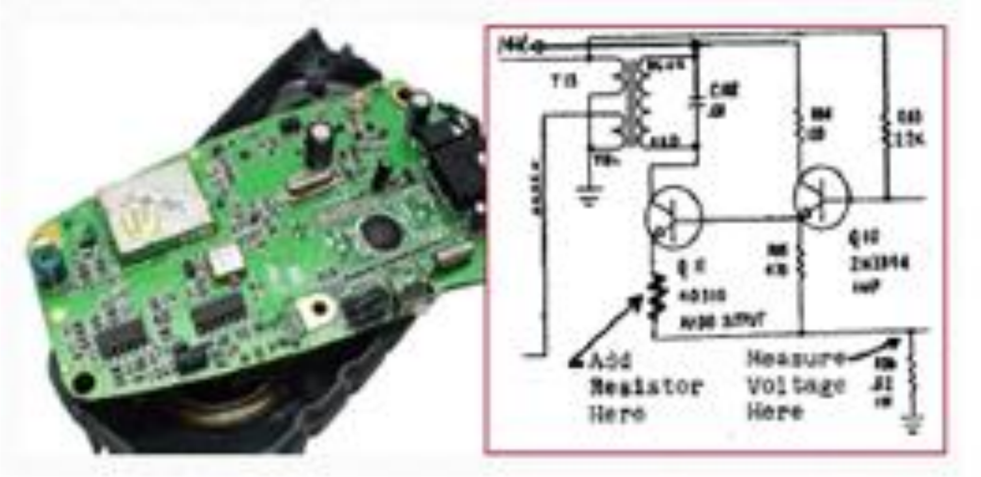
Encapsulation Overview

Encapsulation is a fundamental concept in object-oriented programming (OOP) that binds data (variables) and methods (functions) operating on that data into a single unit, known as a class in Java.

✓ Is the technique of making the fields in a class private

✓ Providing access to the fields via public methods.

- Prevents the *code* and *data* being randomly accessed by other code defined outside the class.
- The ability to *modify* our implemented code *without breaking* the code of others who use our code.



Implementing Encapsulation in Java

- **Two steps to implement encapsulation feature:**
  - ✓ Make the **instance variables private** so that they cannot be accessed directly from outside the class. You can only set and get values of these variables through the methods of the class.
  - ✓ Have **getter** and **setter methods** in the class to set and get the values of the fields.

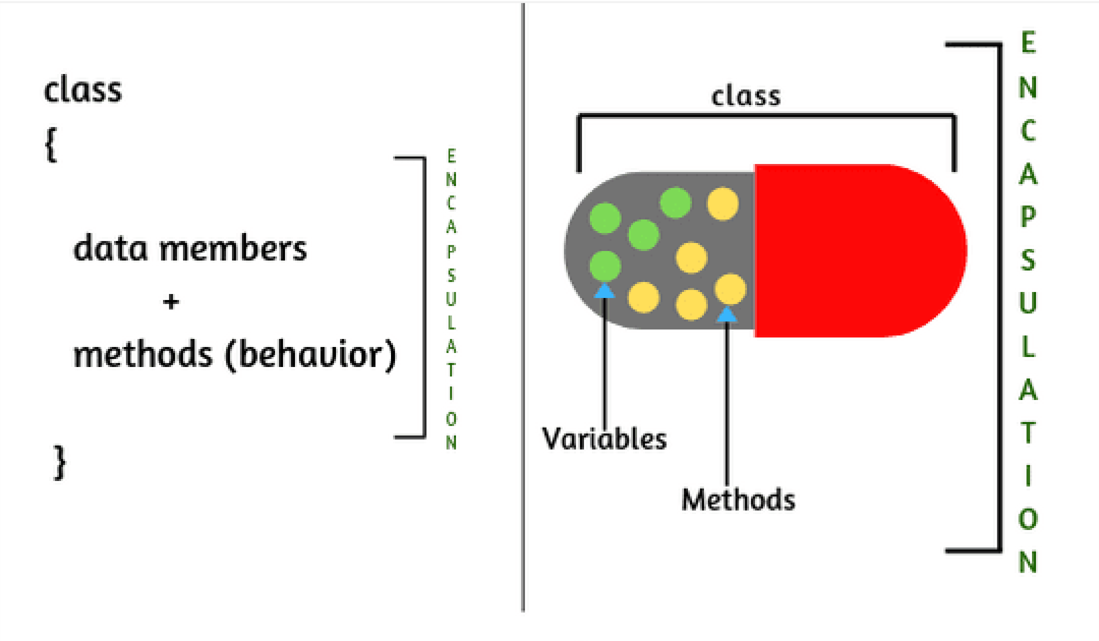
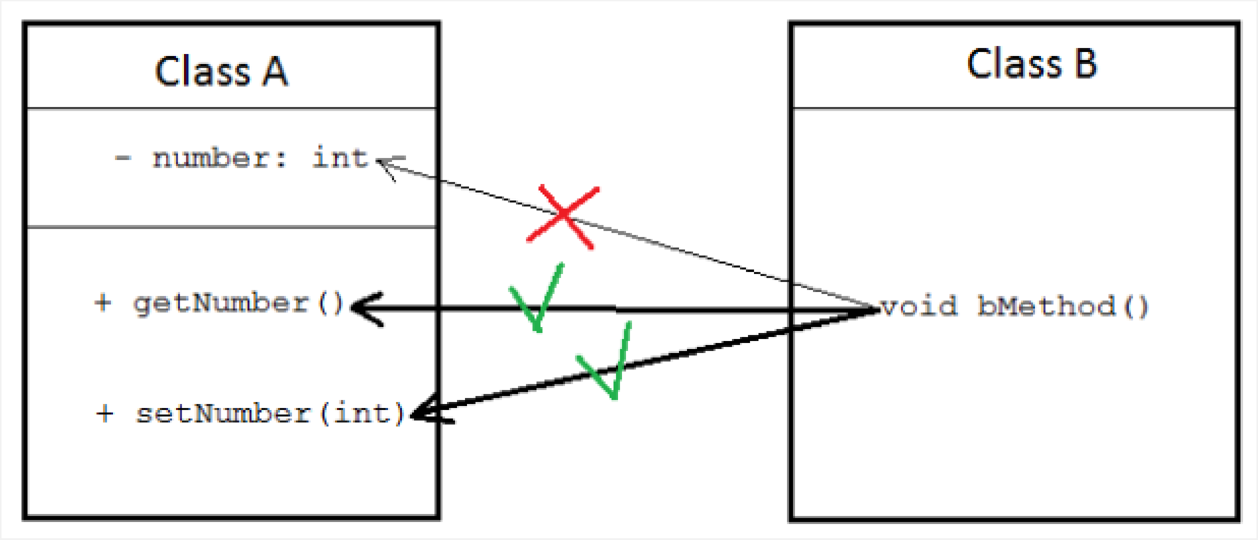


Fig: Encapsulation

Benefits of Encapsulation

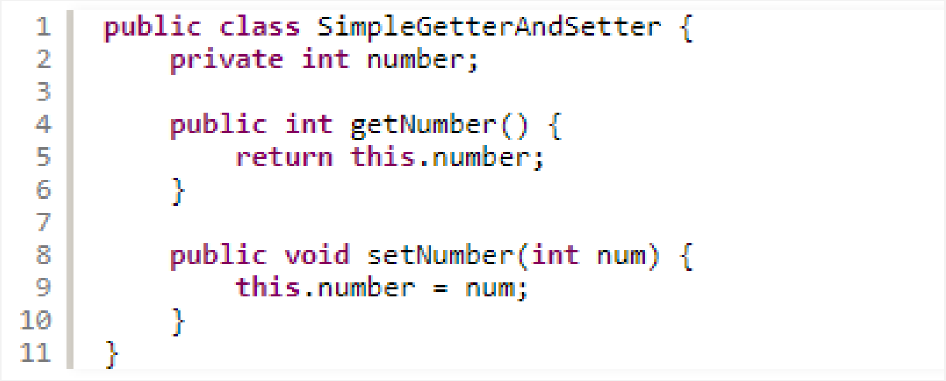
- **Prevents accidental modification**: Encapsulated data is protected from accidental changes by other classes.
- **Enforces data integrity**: Only authorized methods can manipulate the data, ensuring its consistency and validity.
- **Improves security**: Encapsulation protects sensitive data from unauthorized access and modification.



***Getter** and **setter** are two conventional methods that are used for **retrieving** and **updating** value of a variable.*

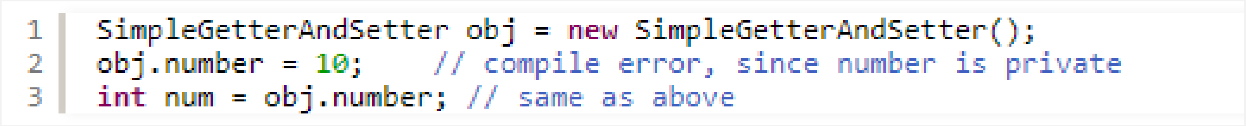
Getter and Setter methods

- The following code is an example of simple class with a private variable and a couple of getter/setter methods:



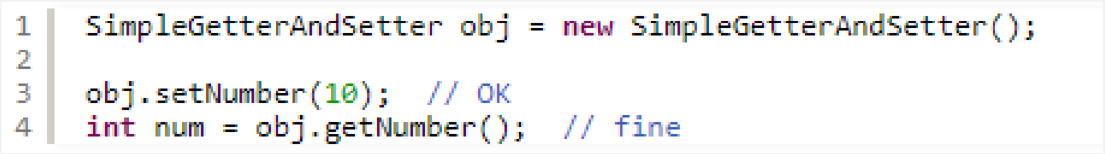
```java
public class SimpleGetterAndSetter {
    private int number;

    public int getNumber() {
        return this.number;
    }

    public void setNumber(int num) {
        this.number = num;
    }
}
```

✓ “**number**” is a private variable: code from outside this class cannot access the variable directly:



```java
SimpleGetterAndSetter obj = new SimpleGetterAndSetter();
obj.number = 10;    // compile error, since number is private
int num = obj.number; // same as above
```

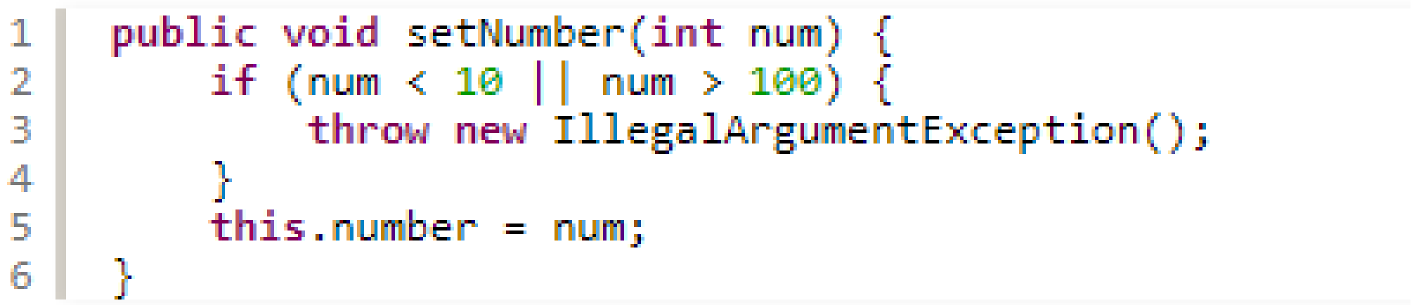
✓ Instead, the outside code have to invoke the getNumber() and the setNumber() in order to read or update the variable, for example:



```java
SimpleGetterAndSetter obj = new SimpleGetterAndSetter();

obj.setNumber(10);  // OK
int num = obj.getNumber();  // fine
```

Why getter and setter?

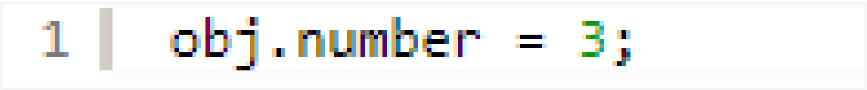
- By using **getter** and **setter**, the programmer can control how to variables are accessed and updated in a **correct** manner.
- **Example**:



```java
public void setNumber(int num) {
    if (num < 10 || num > 100) {
        throw new IllegalArgumentException();
    }
    this.number = num;
}
```

✓That ensures the value of *number is always set between 10 and 100*.

✓Suppose the variable number can be updated directly, the caller can set any arbitrary value to it:



```java
obj.number = 3;
```

Naming Convention for Getter and Setter

▪ JavaBeans Convention:

- **Getters**:
  - Prefix the method name with `get`.
  - Capitalize the first letter of the property name.
  - Example: `public String getName() { return name; }`
- **Setters:**
  - Prefix the method name with `set.`
  - Capitalize the first letter of the property name.
  - Include a single parameter of the property type.
  - Example: `public void setName(String name) { this.name = name; }`

Naming convention for getter and setter

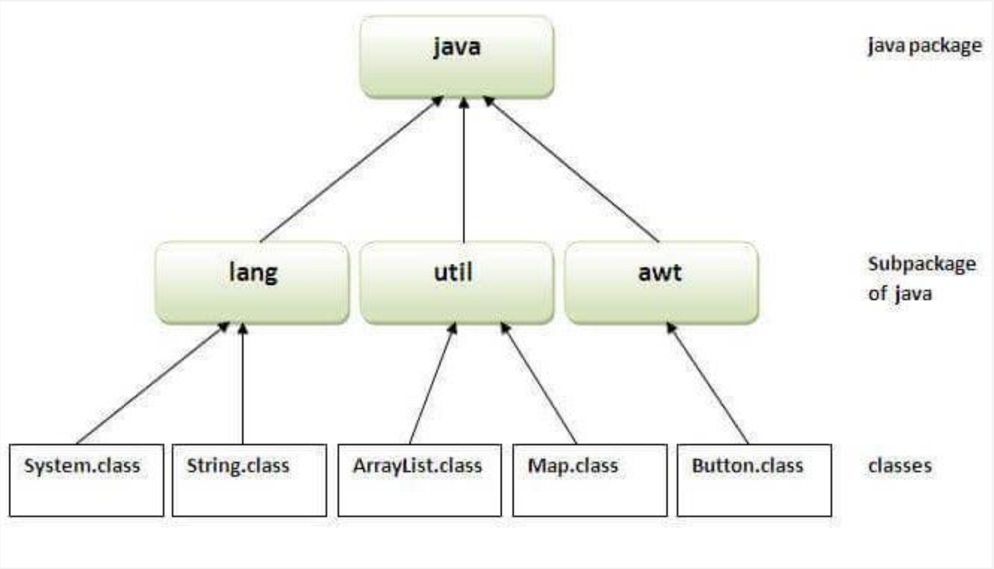
- **Boolean Properties:** For boolean properties, the is prefix is commonly used instead of `get`.

- **Getter:**
    - `public boolean isVisible() { return visible; }`
- **Setter:**
    - `public void setVisible(boolean visible) { this.visible = visible; }`

- **Additional Considerations:**
    - ✓ Use consistent naming conventions for all getters and setters within your project.
    - ✓ Avoid overly long or descriptive names.
    - ✓ Use descriptive names when necessary for clarity.
    - ✓ Consider using annotations to provide additional information about getters and setters.

Java Package

- A **java package** is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two form: **built-in package** and **user-defined package**.
  - ✓ There are many built-in packages such as *java*, *lang*, *awt*, *javax*, *swing*, *net*, *io*, *util*, *sql* etc.
  - ✓ We will have the detailed learning of creating and using user-defined packages.
- **Advantage of Java Package:**
  - ✓ Java package is used to categorize the classes and interfaces so that they can be easily maintained.
  - ✓ Java package provides access protection.
  - ✓ Java package removes naming collision.
- There are **three ways** to access the package from outside the package:
  - ✓ import package.*;
  - ✓ import package.classname;
  - ✓ fully qualified name.



Access Modifiers in Java

Access modifiers are keywords used in Java to control the visibility of classes, members (fields and methods), and constructors.

- **There are four types of Java access modifiers:**
  - ✓ **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
  - ✓ **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
  - ✓ **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
  - ✓ **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

➢ **Non-access modifiers**: static, abstract, synchronized, native, volatile, transient, etc.

Access Modifiers

Access Modifier	Within class	Within package	Outside package by subclass only	Outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

this Keyword

The **this** keyword is a fundamental concept in object-oriented programming (OOP) languages, including Java. It refers to the **current object** within a method or constructor.

- **Here are some examples of how to use the this keyword:**
  - ✓ Use `this.<member_name>` to access private members of the **current object**. This avoids confusion between *local variables* and *members* with the same name.

```java
public class Point {
    private int x;
    private int y;
    public void setX(int x) {
        this.x = x; // Avoiding confusion with parameter
    }
    public int getX() {
        return this.x; // Accessing object member
    }
}
```

this Keyword

- Use `this` to call another method within the *same object*, creating a chain of method calls.

```java
public class User {
    private String name;
    private String email;

    public User setName(String name) {
        this.name = name;
        return this; // Chain to another method
    }

    public User setEmail(String email) {
        this.email = email;
        return this; // Chain to another method
    }
}
```

```java
User user = new User().setName("Jonh").setEmail("john@example.com");
```

this Keyword

- Use this to call another constructor within the same class.

```
public class Account {
    private String name;
    private double balance;

    public Account(String name) {
        this(name, 0.0); // Calling another constructor
    }

    public Account(String name, double balance) {
        this.name = name;
        this.balance = balance;
    }
}
```

Section 3

Inheritance

OOP Principles in Java

- ✓**Encapsulation**
- ✓Inheritance
- ✓Abstraction
- ✓Polymophims

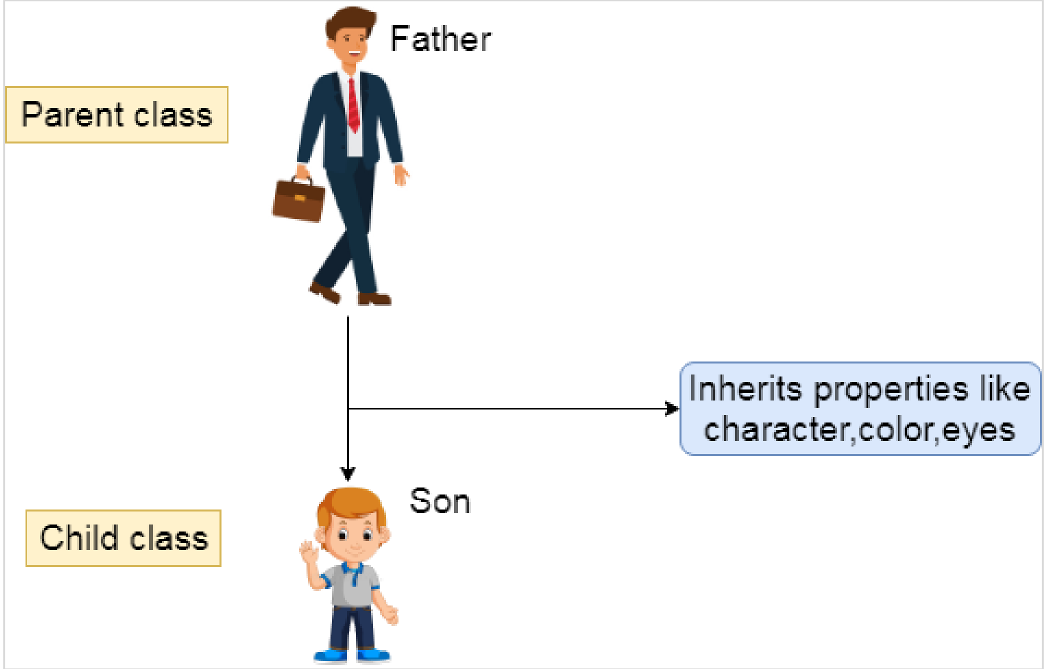
Inheritance Overview

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows you to build **new classes based on existing ones**.
It promotes code *reuse* and *helps to organize your code into a hierarchy*.

- **Inheritance** allows you to define a new class by specifying only the ways in which it differs from an existing class.
- Inheritance promotes software reusability:
  - ✓ Absorb existing class's data and behaviors
  - ✓ Enhance with new capabilities
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

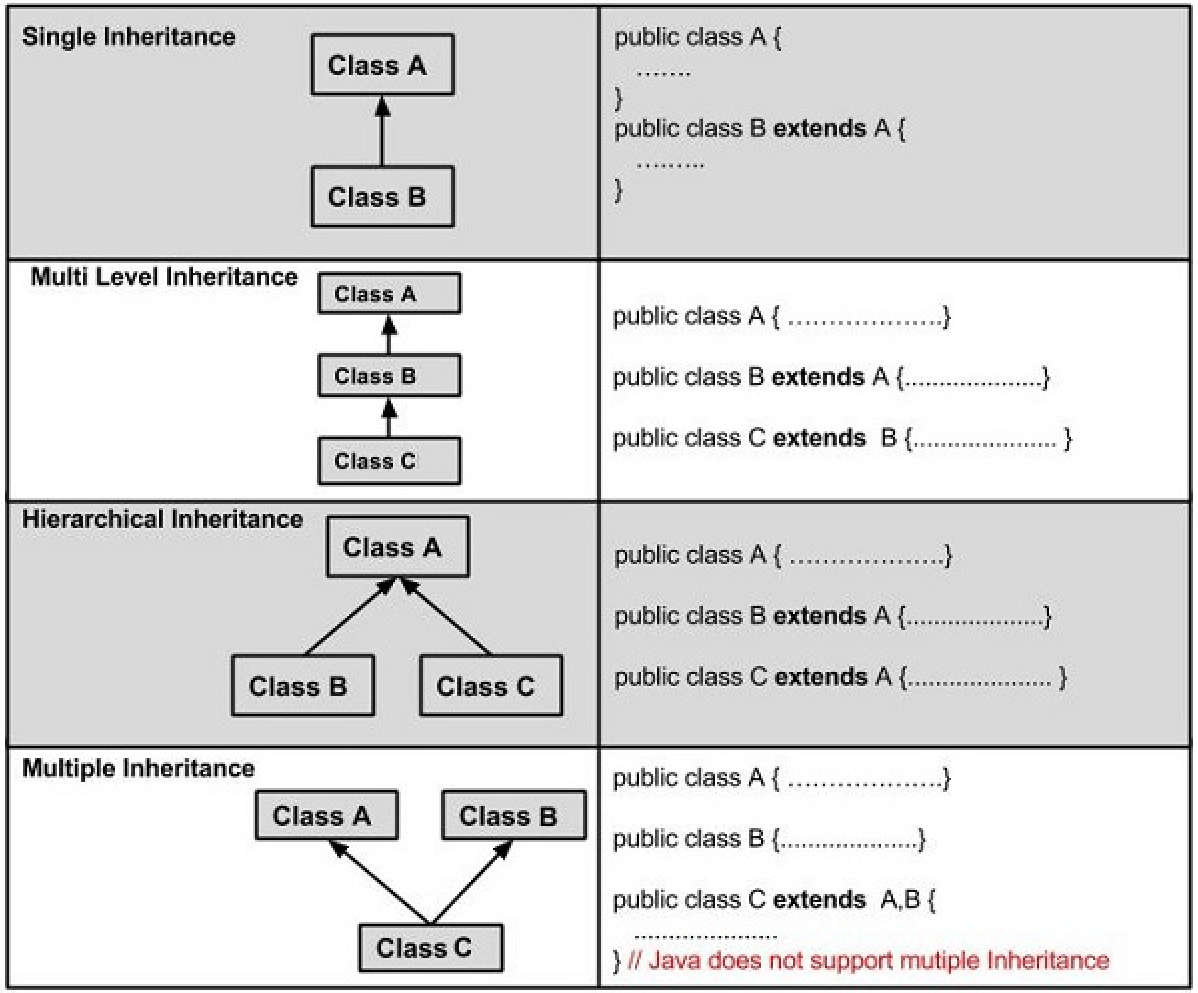
Key Concepts

- **Superclass (Parent Class):** The existing class from which new classes inherit.
- **Subclass (Child Class):** The new class that inherits properties and behavior from the superclass.
- **Inheritance Hierarchy:** The relationship between superclasses and subclasses, forming a tree-like structure.



Type of Inheritance

- Three types of inheritance in java: **single**, **multilevel** and **hierarchical**.



Single Inheritance

```
public class A {
   .......
}
public class B extends A {
   .........
}
```

Multi Level Inheritance

```
public class A { ...................}

public class B extends A {....................}

public class C extends  B {..................... }
```

Hierarchical Inheritance

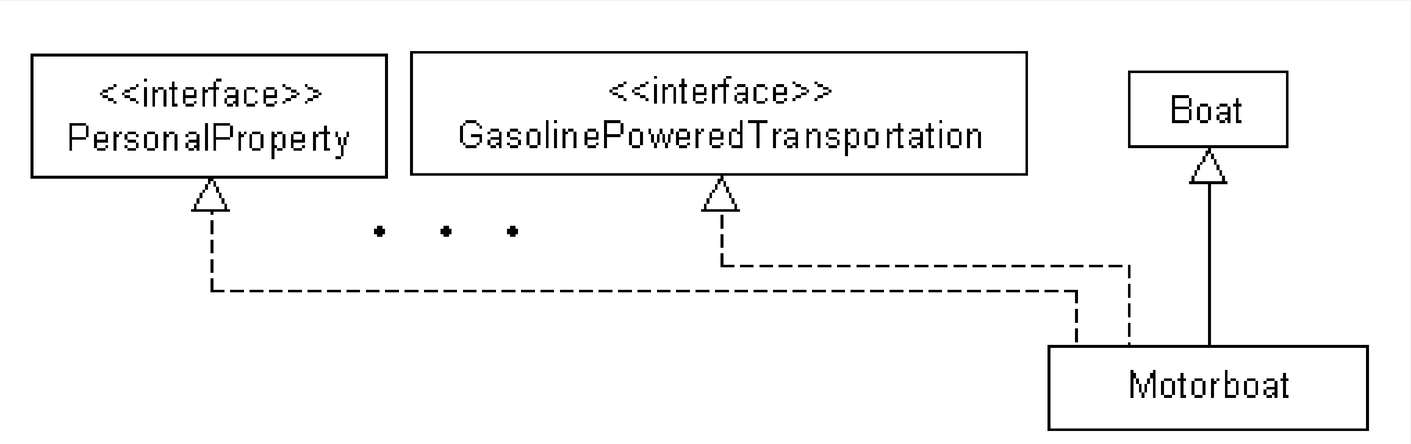
```
public class A { ...................}

public class B extends A {....................}

public class C extends A {..................... }
```

Multiple Inheritance

```
public class A { ...................}

public class B {....................}

public class C extends  A,B {
   ......................
} // Java does not support mutiple Inheritance
```

Inheritance

- **Two kinds:**
  - ✓ implementation: the code that defines methods.
  - ✓ interface: the method prototypes only.
- You can't extend more than one class!
  - ✓ the derived class can't have more than one base class.
- You can do multiple inheritance with *interface* inheritance.



extends Keyword

- **extends** is the keyword used to inherit the properties of a class.
- **Syntax:**

```
class Super {
   .....
   .....
}

class Sub extends Super {
   .....
   .....
}
```

extends Keyword - sample

- Create a super class:

```java
public class Calculation {
   protected int z;

   public void addition(int x, int y) {
      z = x + y;
      System.out.println("The sum of the given numbers:" + z);
   }

   public void substraction(int x, int y) {
      z = x - y;
      System.out.println("The difference
                   between the given numbers:" + z);
   }
}
```

```java
public class MyCalculation extends Calculation {

   public void multiplication(int x, int y) {
     z = x * y;
     System.out.println("The product of the given numbers:" + z);
   }
   public static void main(String args[]) {
      int a = 20, b = 10;

   MyCalculation demo = new MyCalculation();
      demo.addition(a, b);
      demo.substraction(a, b);
      demo.multiplication(a, b);
   }
}
```

- Output:

```
The sum of the given numbers:30
The difference between the given numbers:10
The product of the given numbers:200
```

Inheritance Vocabulary

- **“IS-A”**

✓“IS-A” relationship – this thing **is a** type of that thing

  - Inheritance
  - Subclass object treated as superclass object

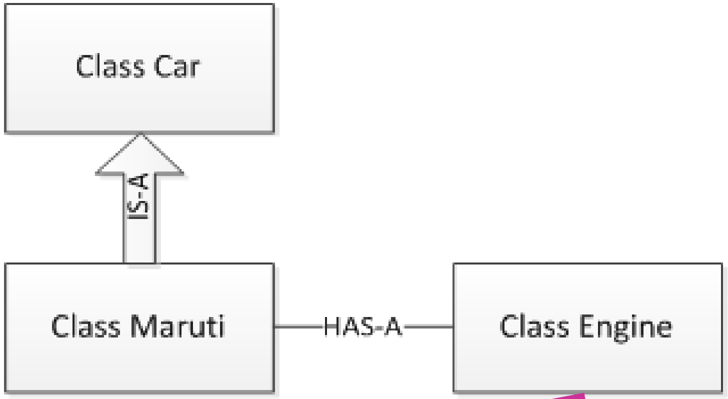
- **“HAS-A”**

✓ “HAS-A” relationship - class A **HAS-A** B if code in class A has a reference to an instance of class B.

  - Aggregation
  - Object contains one or more objects of other classes as members

Example: Maruti *is a* Car

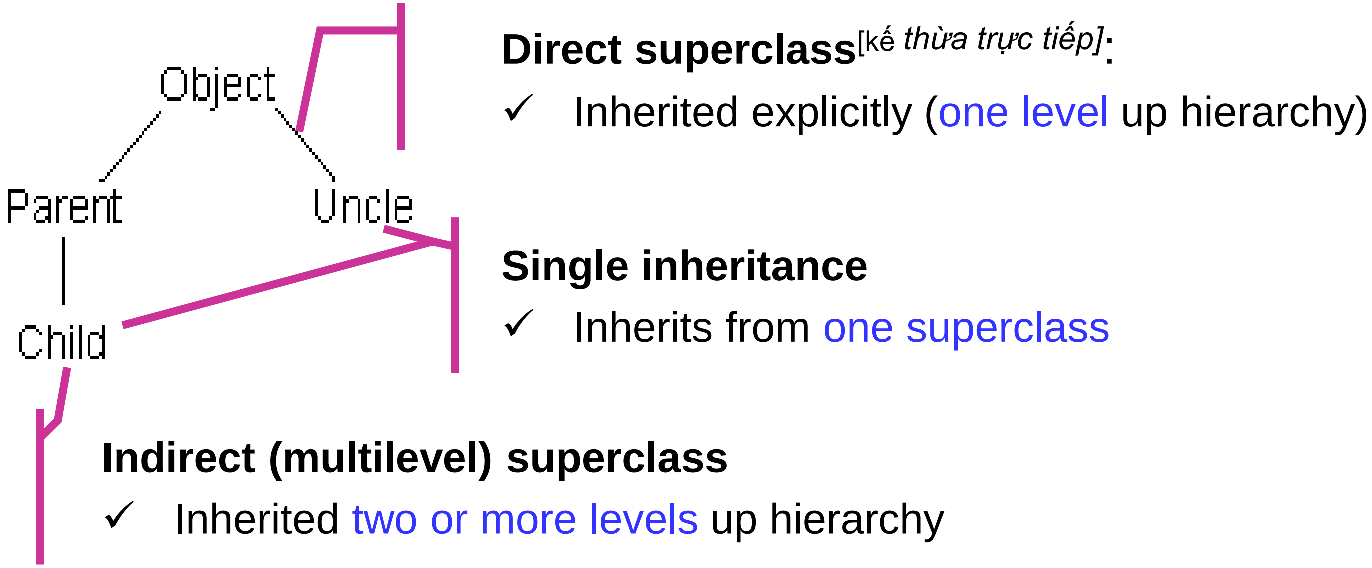
✓ *Car* properties/behaviors also *Maruti* properties/behaviors



Example: Maruti *has a* Engine

Inheritance Vocabulary

- **Class hierarchy**



Direct superclass[kế *thừa trực tiếp]*:

✓ Inherited explicitly (one level up hierarchy)

Single inheritance

✓ Inherits from one superclass

Indirect (multilevel) superclass

✓ Inherited two or more levels up hierarchy

- **Multiple inheritance:**

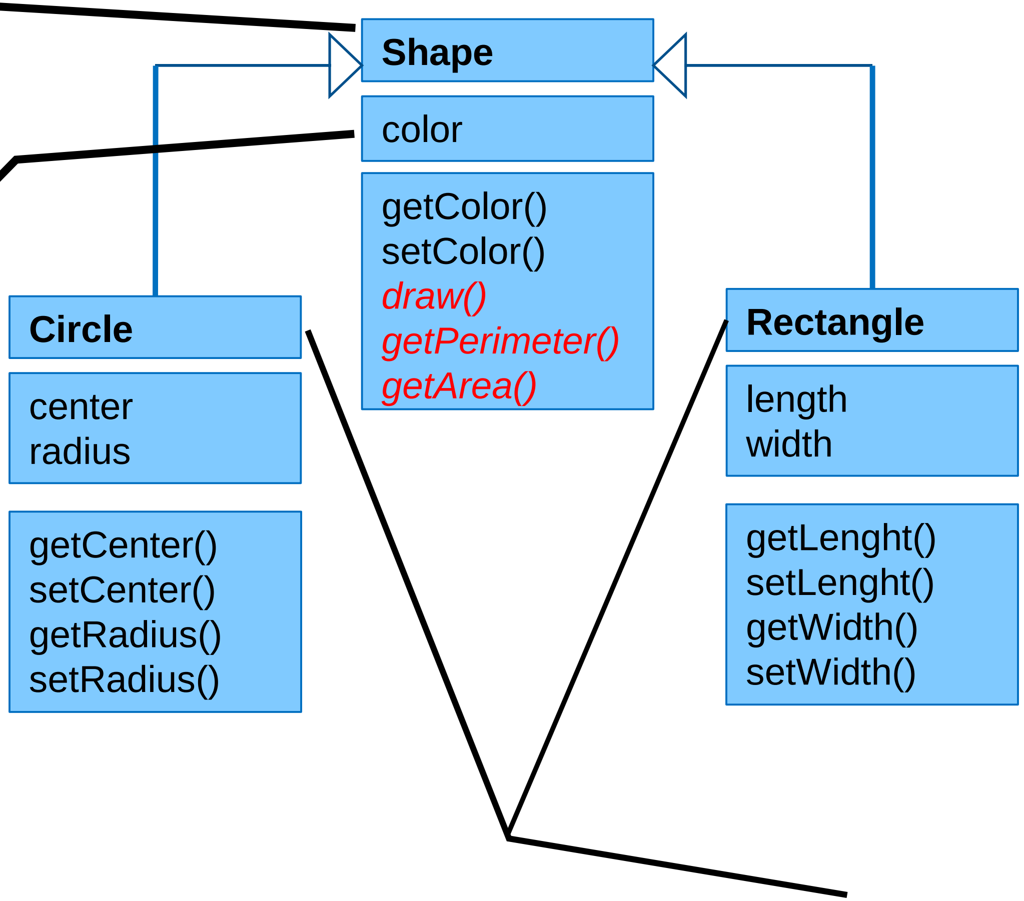
  ✓ Inherits from multiple superclasses

  - **Java does not support multiple inheritance in *classes***

Inheritance Example

Shape is superclass

Circle and **Rectangle** has color property



Shape

color

getColor()
setColor()
draw()
getPerimeter()
getArea()

Circle

center
radius

getCenter()
setCenter()
getRadius()
setRadius()

Rectangle

length
width

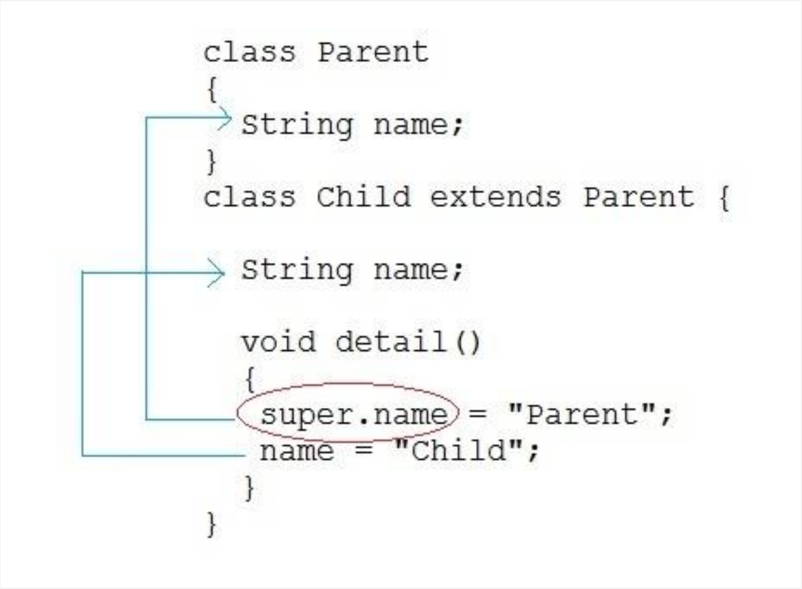
getLenght()
setLenght()
getWidth()
setWidth()

- ✓ Circle isa Shape, but Shape is not a Circle.
- ✓ Method draw(), getPerimeter(), getArea() in Circle overriding method draw() , getPerimeter(), getArea() in Shape.
- ✓ If we add/remove property to/from Shape, then it’s affected to Circle and Rectangle.

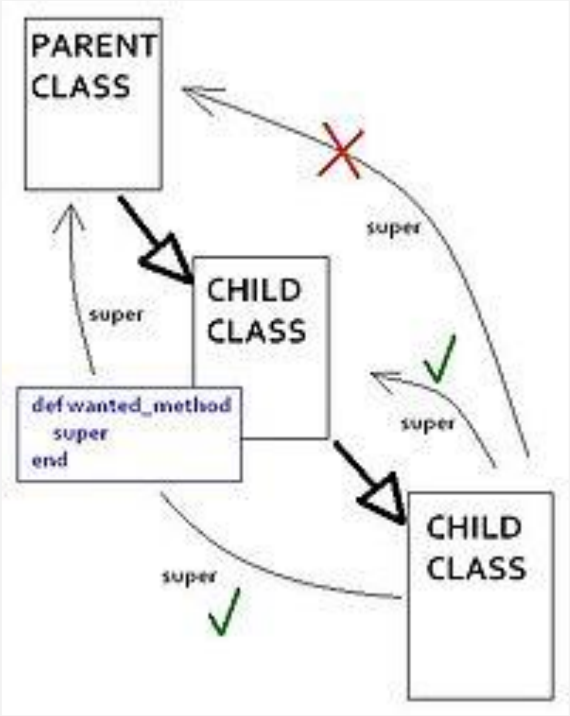
Circle and **Rectangle** are subclasses

super keyword

- Can use **super** keyword to access all (non-private) superclass methods.
  - ✓ even those replaced with new versions in the derived class.
- Can use **super()** to call base class constructor.



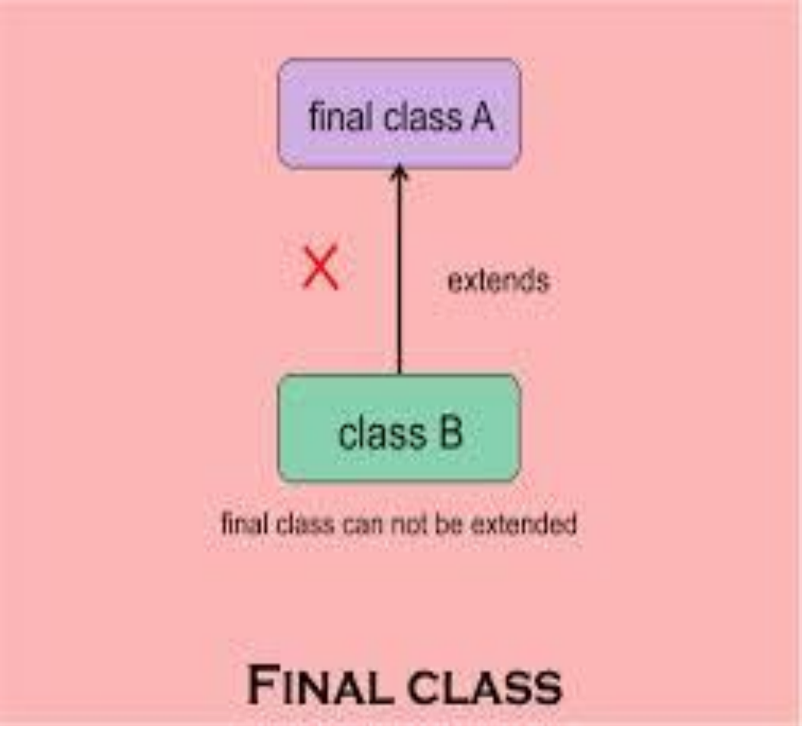
```
class Parent
{
  String name;
}
class Child extends Parent {

  String name;

  void detail()
  {
   super.name = "Parent";
   name = "Child";
  }
}
```



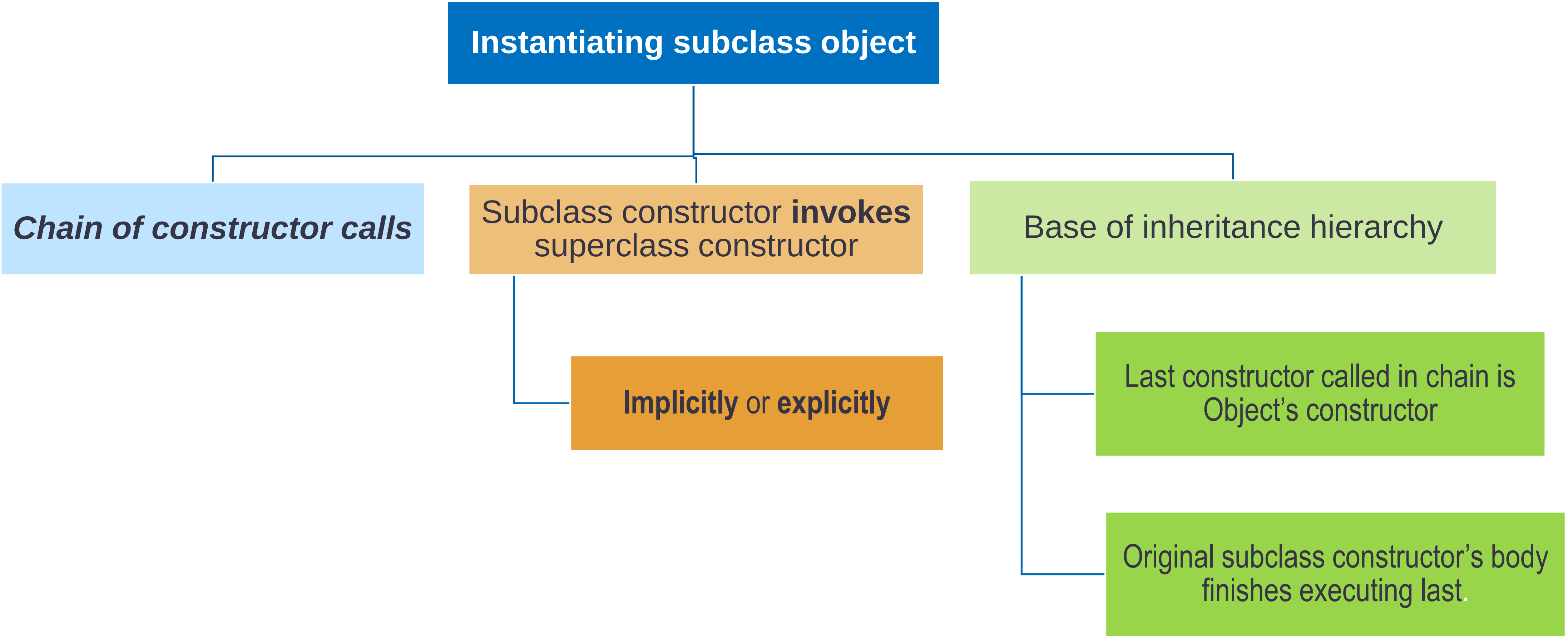
- **Subclass methods are not superclass methods**

Final class - recap

- You can declare an class is `final` - this prevents the class from being subclassed.
- Of course, an `abstract` class cannot be a `final` class.



Constructor and Finalizers



Instantiating subclass object

Chain of constructor calls

Subclass constructor **invokes** superclass constructor

Implicitly or **explicitly**

Base of inheritance hierarchy

Last constructor called in chain is Object's constructor

Original subclass constructor's body finishes executing last.

Constructor and Finalizers

- **Examples:**

```
class Building {
    Building() {
        System.out.print("b ");
    }

    Building(String name) {
        this();
        System.out.print("bn " + name);
    }
}
```

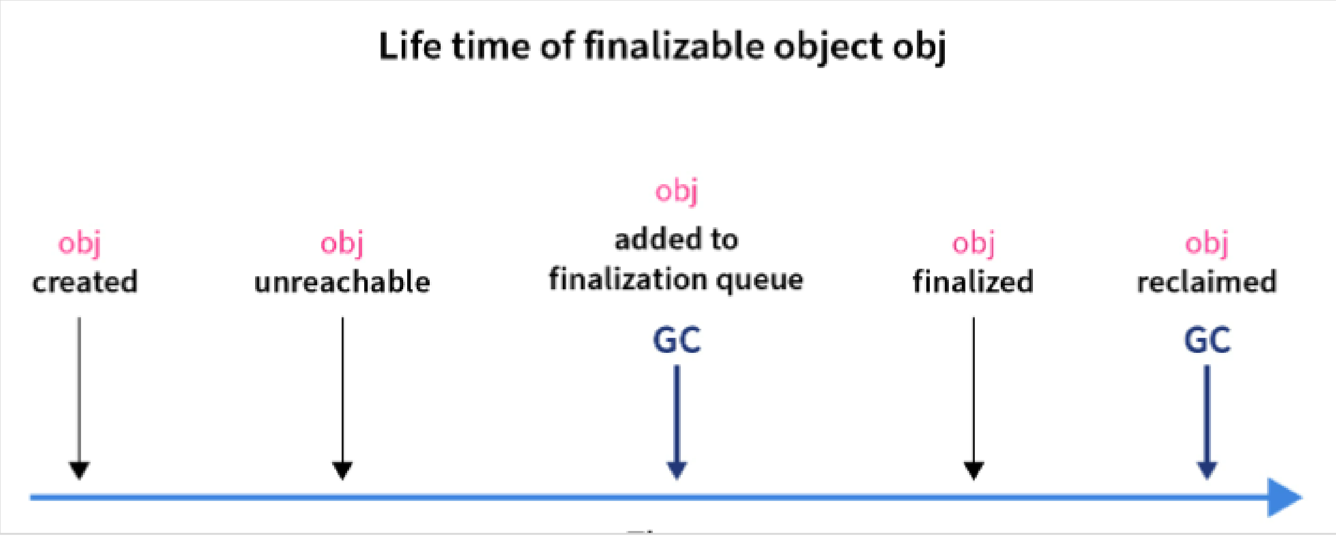
```
public class House extends Building {
    House() {
        System.out.print("h ");
    }

    House(String name) {
        this();
        System.out.print("hn " + name);
    }

    public static void main(String[] args) {
        new House("x ");
    }
}
```

- **Output:**

```
h hn x
```

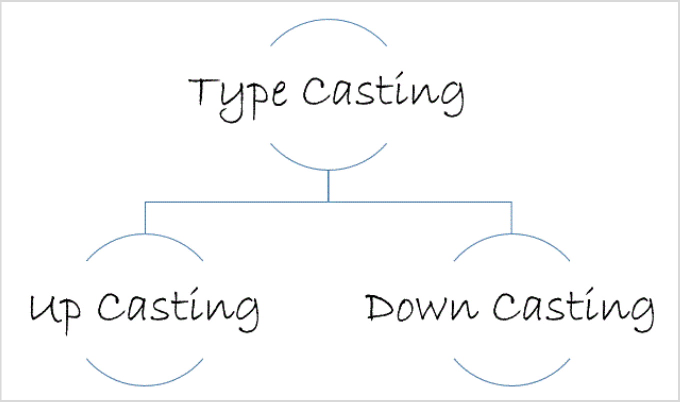
Constructor and Finalizers

- **Garbage collecting subclass object: Chain of `finalize` method calls**
  - ✓ **Reverse** order of constructor chain
  - ✓ Finalizer of **subclass called first**
  - ✓ Finalizer of **next superclass** up hierarchy next
  - ✓ Continue up hierarchy until final superreached
  - ✓ After final superclass (`Object`) finalizer, object removed from memory



Casting Objects

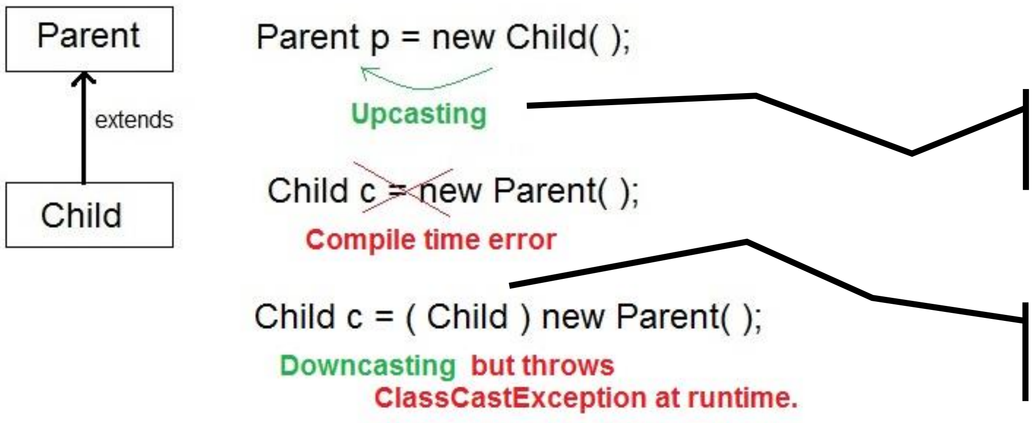
- Java permits an object of a subclass type to be treated as an object of any superclass type. *This is called upcasting.*
- Upcasting and downcasting are NOT like casting primitives from one to other.



Type Casting

Up Casting

Down Casting



```
Parent p = new Child( );
```

Upcasting

```
Child c ~~=~~ new Parent( );
```

Compile time error

```
Child c = ( Child ) new Parent( );
```

Downcasting but throws ClassCastException at runtime.

Upcasting is done automatically.

Downcasting must be manually done by the programmer

Casting Objects Examples

```java
class Animal {
    public void eat() {
        System.out.println("Generic Animal Eating Generically");
    }
}

class Horse extends Animal {
    public void eat() {
        System.out.println("Horse eating hay, oats, " + "and horse treats");
    }

    public void buck() {
        System.out.println("This is buck");
    }
}
```

```java
public class TestAnimals {
    public static void main(String[] args) {
        Animal a = new Animal();
        Animal b = new Horse(); // Animal ref, but a Horse object – upcasting
        Horse h = new Horse();
        a.eat(); // Runs what?
        b.eat(); // Runs what?

        // What is the result?
        Animal c = new Horse();
        c1.buck();    ➔ Cannot invoke subclass-only (Horse) methods on subclass
    }                 object through superclass (Animal) reference
}
```

instanceof Operator

- The **Java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).
- The **instanceof** is also known as type *comparison operator* because it compares the instance with type.
  - ✓ It returns either **true** or **false**.
  - ✓ If we apply the instanceof operator with any variable that has null value, it returns false.
- **Example**:

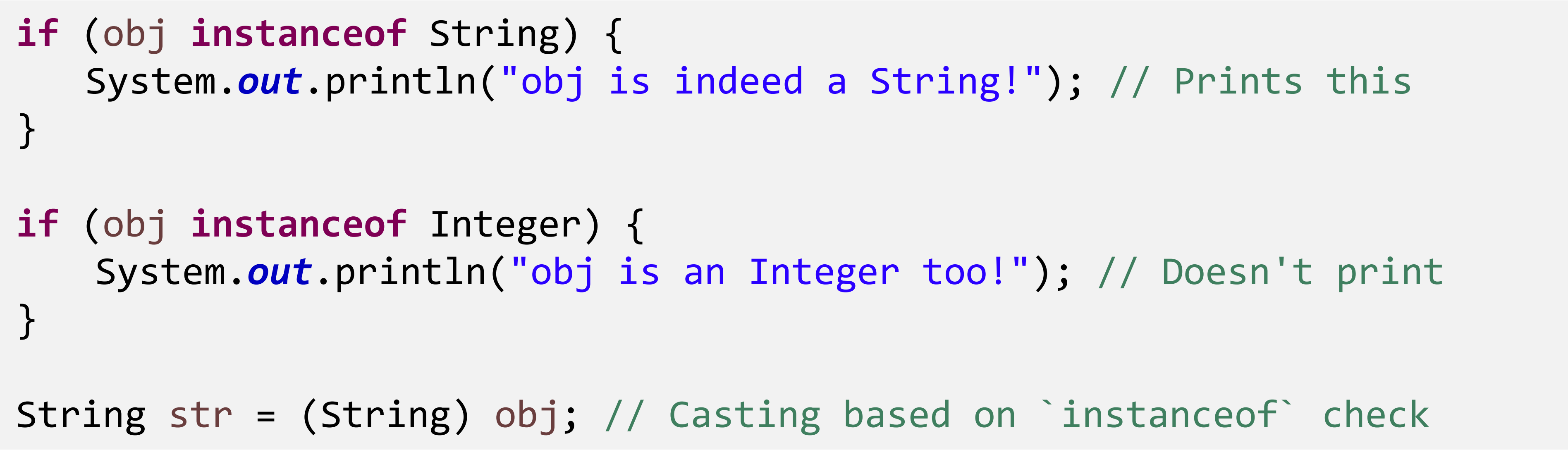
```java
class Animal{}
class Dog extends Animal{//Dog inherits Animal
 public static void main(String args[]){
      Animal a = new Animal();
      Dog d = new Dog();
      System.out.println(a instanceof Animal);//true
      System.out.println(d instanceof Animal);//?
 }
}
```

Output:

```
true
true
```

instanceof Operator

- **Examples:**



```
if (obj instanceof String) {
    System.out.println("obj is indeed a String!"); // Prints this
}

if (obj instanceof Integer) {
    System.out.println("obj is an Integer too!"); // Doesn't print
}

String str = (String) obj; // Casting based on `instanceof` check
```

instanceof operator

- When **Subclass type refers to the object of Parent class**, it is known as downcasting.
- Let's see the example, downcasting be performed without the use of instanceof operator:
  - ✓ The actual object that is referred by '*animal*' is an object of Dog class. **So if we downcast it, it is fine.**
- **Example:**

```
class Animal {
}

class Dog extends Animal {

}

public class Main {
  public static void main(String[] args) {
    Animal animal = new Dog();
     Dog d = (Dog) animal;
    System.out.println("Downcasting performed");
  }
}
```

- **Output:**

```
Downcasting performed
```

instanceof operator

- **But what will happen if we write:**

```
Animal animal = new Animal();
Dog d = (Dog) animal;
```

- **You faced with an exception:**

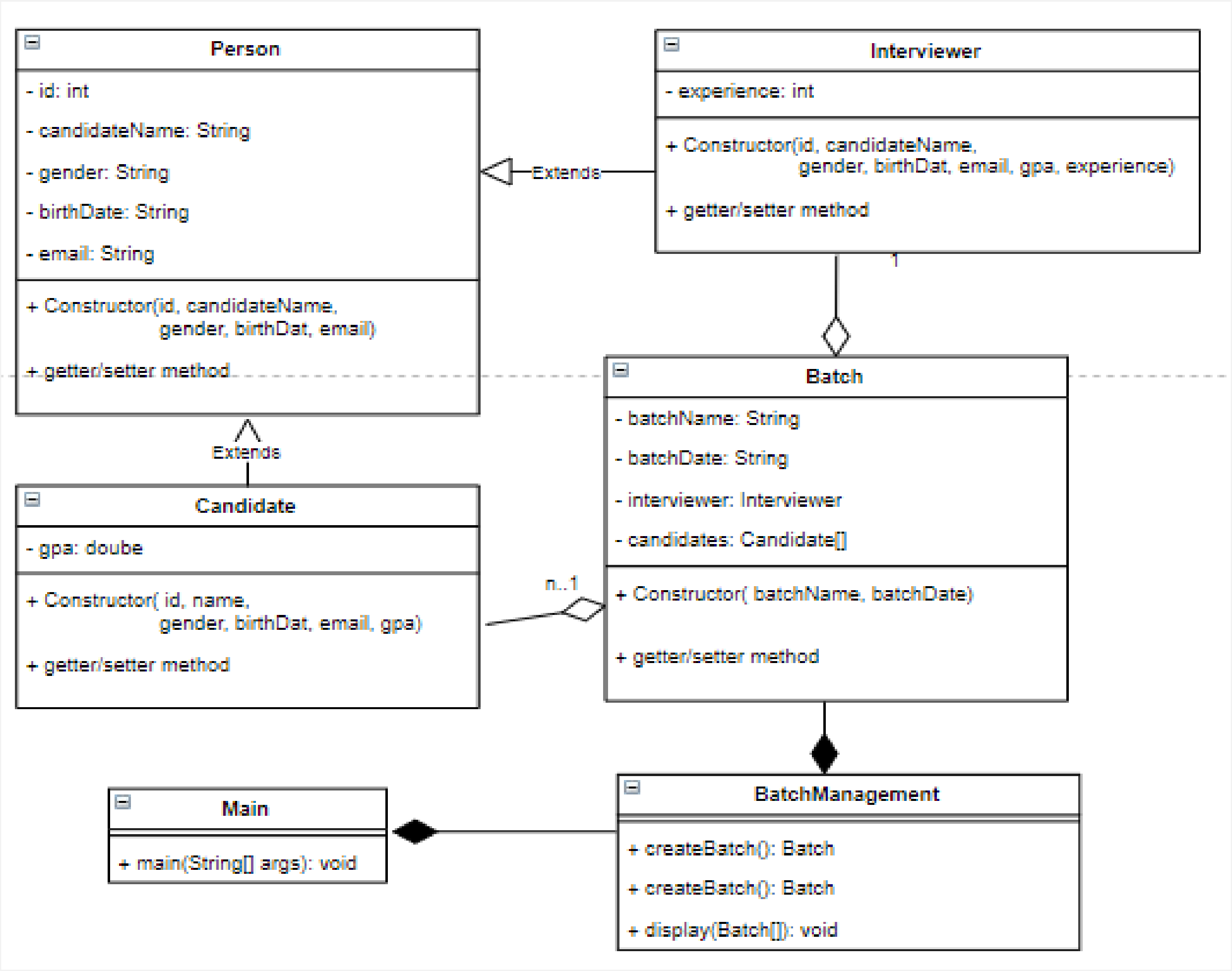
```
Exception in thread "main" java.lang.ClassCastException: fa.training.jpe.Animal cannot be cast to Dog
```

- **To resolve this problem,** should perform downcasting with java **instanceof** operator. Re-write code:

```
class Animal {}

class Dog extends Animal { }

public class Main {
  public static void main(String[] args) {
    Animal animal = new Animal();
    if (animal instanceof Dog) {
          Dog d = (Dog) animal;
    }
  }
}
```

Practice time

- In class diagrams, as shown in following Figure. Let’s implement it using Java:



Person

- id: int
- candidateName: String
- gender: String
- birthDate: String
- email: String

+ Constructor(id, candidateName, gender, birthDat, email)
+ getter/setter method

Interviewer

- experience: int

+ Constructor(id, candidateName, gender, birthDat, email, gpa, experience)
+ getter/setter method

Candidate

- gpa: doube

+ Constructor(id, name, gender, birthDat, email, gpa)
+ getter/setter method

Batch

- batchName: String
- batchDate: String
- interviewer: Interviewer
- candidates: Candidate[]

+ Constructor(batchName, batchDate)
+ getter/setter method

Main

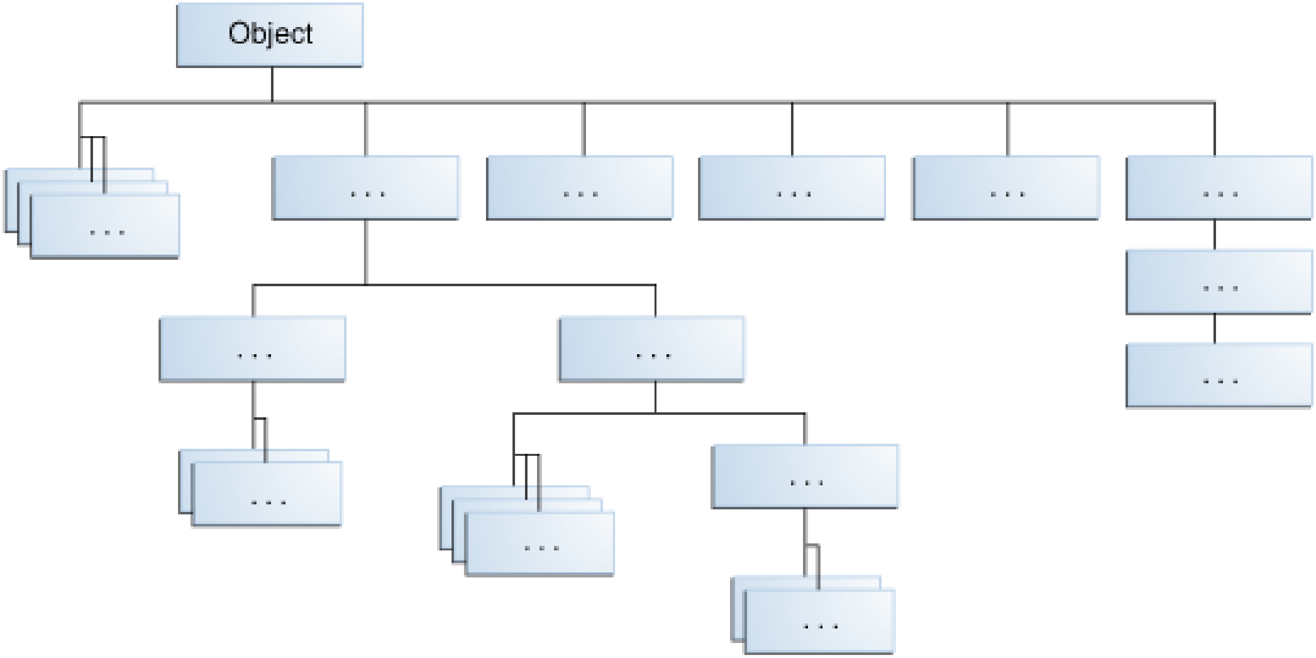
+ main(String[] args): void

BatchManagement

+ createBatch(): Batch
+ createBatch(): Batch
+ display(Batch[]): void

The class Object

- Granddaddy of all Java classes.
- All methods defined in the class Object are available in every class.
- Any object can be cast as an `Object`.



Summary

- Inheritance is a mechanism that allows one class to **reuse** the implementation provided by another.
- A class always **extends** exactly one superclass.
  - ✓ If a class does not explicitly extend another, it implicitly extends the class Object.
- A superclass method or field can be accessed using a **super**. keyword.
- Subclass objects can not access superclass's private data unless they change into **protected** access level.
- If a constructor does not explicitly invoke another (this() or super()) constructor, it implicitly Invokes the superclass's no-args constructor.
- Encapsulation:
  - ✓ Hiding implementation details from clients.

References

- *https://docs.oracle.com/javase/tutorial/java/concepts/*
- *https://www.oracle.com/java/technologies/oop.html*
- *https://www.geeksforgeeks.org/object-oriented-programming-oops-concept-in-java/*
- *https://www.freecodecamp.org/news/object-oriented-programming-concepts-java/*

Questions

THANK YOU!