# OOP in Java

## Object Oriented Programming?

- Procedural programming is about **writing procedures or methods that perform operations on the data**, while object-oriented programming is about **creating objects that contain both data and methods**.

## Object Oriented Programming?

- is a methodology or paradigm to design a program using classes and objects. It **simplifies the software development and maintenance** by providing some concepts.
    - Object, Class
    - Inheritance
    - Polymorphism
    - Abstraction
    - Encapsulation

## Advantage of OOPs

- OOP provides **a clear structure** for the programs
- OOP helps to keep the Java code DRY "**Don't Repeat Yourself**", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full **reusable applications** with less code and shorter development time

## Advantage of OOPs

- Procedure-oriented programming language **it is not easy to manage if code grows** as project size grows.
- OOPs provides data hiding whereas **in Procedure-oriented programming language a global data can be accessed from anywhere**.
- OOPs provides ability to **simulate real-world event much more effectively**.

## Class, Object

- a class is a template for objects, and an object is an instance of a class
- **Any entity that has state and behavior** is known as an **object**. It can be physical and logical.
- **Collection of objects** is called class.

## Create a class, an object

```
public class MyClass {
  int x = 5;
  public static void main(String[] args) {
    MyClass myObj = new MyClass();
    System.out.println(myObj.x);
  }
}
```

## Create a class, an object

```
<modifiers> class <ClassName> {
    <modifiers> <DataType> <attributeName>;
    ...
    <modifiers> <ReturnType><methodName>(<parameters>) {
        ...
    }
}
```

## Attributes/Fields

```
public class Person {
  String fname = "John";
  String lname = "Doe";
  int age = 24;
  public static void main(String[] args) {
       Person myObj = new Person();
       System.out.println("Name:" + myObj.fname + "
       " + myObj.lname);
       System.out.println("Age:" + myObj.age);
  }
}
```

## Methods

- **Static vs. Non-Static**
    - You will often see Java programs that have either `static` or `public` attributes and methods.
    - We created a `static` method, which means that it can be accessed without creating an object of the class, unlike `public`, which can only be accessed by objects

```
public class MyClass {
  // Static method
  static void myStaticMethod() {
    System.out.println("Static methods can be called without creating objects");
  }
  // Public method
  public void myPublicMethod() {
    System.out.println("Public methods must be called by creating objects");
  }
```

```java
// Main method
  public static void main(String[] args) {
    myStaticMethod(); // Call the static method
    // myPublicMethod(); This would compile an
error

    MyClass myObj = new MyClass(); // Create an
object of MyClass
    myObj.myPublicMethod(); // Call the public
method on the object
  }
 }
```

## Constructor

- A constructor in Java is a special method that is used to initialize objects. The constructor is called when an object of a class is created

```
// Create a MyClass class
public class MyClass {
  int x;

  // Create a class constructor
  public MyClass() {
    x = 5;
  }

  public static void main(String[] args) {
    MyClass myObj = new MyClass();
    System.out.println(myObj.x);
  }
}
```

## Constructor

- A *special type of method* that is used to initialize the object.

  **<class_name>(){}**

  1) Default constructor (no-arg constructor)
  2) Parameterized constructor

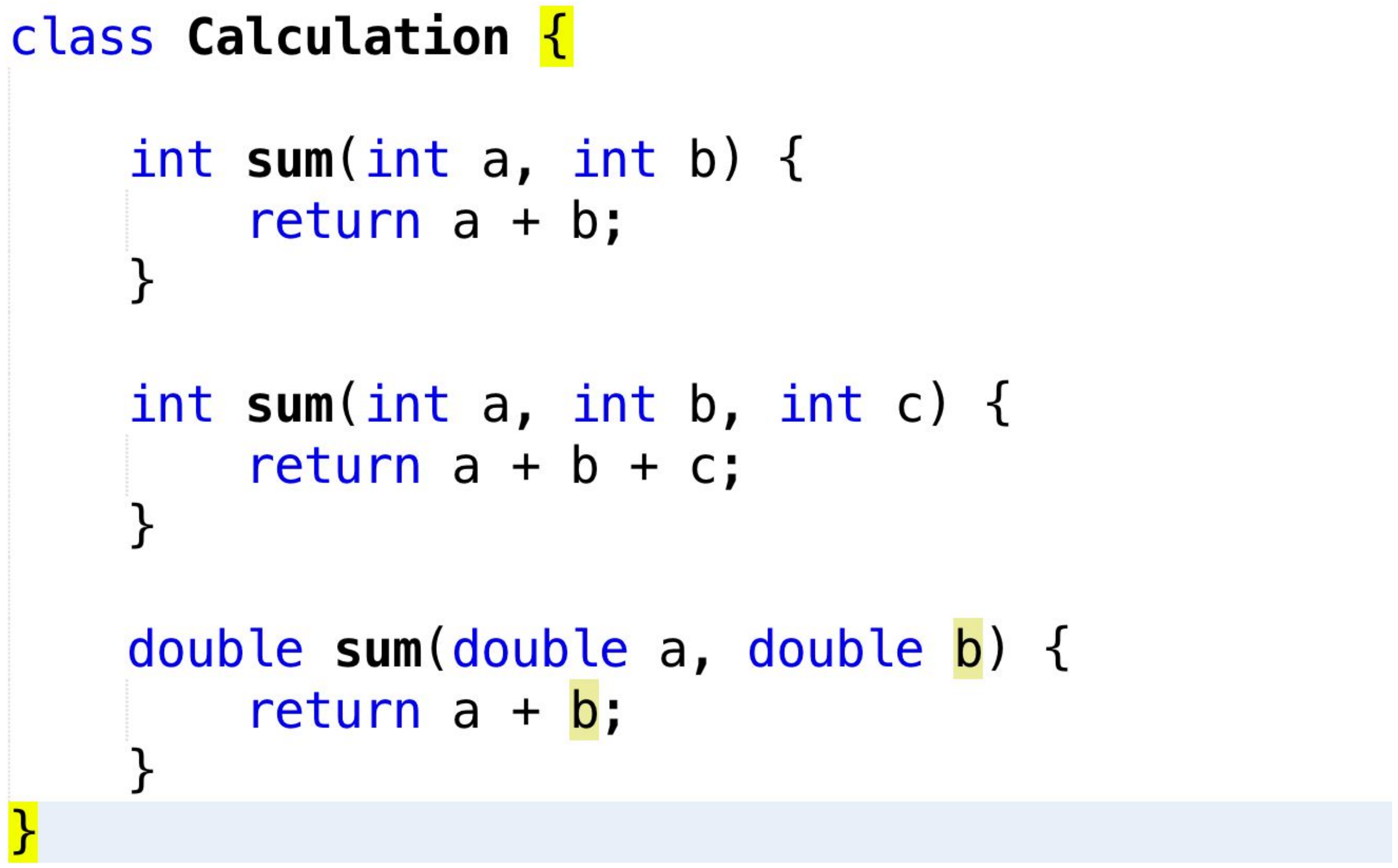
- **Rule: If there is no constructor in a class, compiler automatically creates a default constructor.**

## Demo - Java Constructor

https://www.w3schools.com/java/java_constructors.asp

## Method Overloading

- If a class have **multiple methods by same name but different parameters**, it is known as **Method Overloading**.
- Method overloading **increases the readability of the program**.
- Changing number of arguments/changing the data type.
- Not possible by changing the return type of the method because there may occur ambiguity.



```
class Calculation {

    int sum(int a, int b) {
        return a + b;
    }

    int sum(int a, int b, int c) {
        return a + b + c;
    }

    double sum(double a, double b) {
        return a + b;
    }
}
```

## Modifiers

- **Access Modifiers** (*ex: public, protected, private, default*) - controls the access level.
- **Non-Access Modifiers** (*ex: final, abstract, static, synchronized*) - do not control access level, but provides other functionality.

## Access Modifiers - *Class*

- **public**: The class is accessible by any other class
- *default*: The class is only accessible by classes in the same package

## Access Modifiers -
## Attributes, Methods, Constructor

- **public**: The class is accessible by any other class
- **protected**: The code is accessible in the same package and **subclasses**
- **private**: The code is only accessible within the declared class
- *default*: The class is only accessible by classes in the same package

## Non Access Modifiers -
### For Class

- **final**: The class cannot be inherited by other classes
- **abstract**: The class cannot be used to create objects

## Non Access Modifiers -
## For Attributes, Methods

- final: Attributes and methods cannot be overridden/modified
- static: Attributes and methods belongs to the class, rather than an object
- abstract: Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example **abstract void run();**. The body is provided by the subclass
- Synchronized: Methods can only be accessed by one thread at a time

## Demo - Java Modifiers

https://www.w3schools.com/java/java_modifiers.asp

## Java Packages

- A package in Java is used to **group related classes**. Think of it as **a folder in a file directory**. We use packages to **avoid name conflicts**, and to write a better maintainable code. Packages are divided into two categories:
    - **Built-in Packages** (packages from the Java API)
    - **User-defined Packages** (create your own packages)

Source: [https://www.w3schools.com/java/java_packages.asp](https://www.w3schools.com/java/java_packages.asp)

## Encapsulation

is to **make sure that "sensitive" data is hidden from users**. To achieve this, you must:

- declare class **variables/attributes** as `private`
- provide public **get** and **set methods to access and update the value** of a `private` variable

```
public class Person {
    private String name;

    // Getter
    public String getName() {
        return name;
    }

    // Setter
    public void setName
    (String newName) {
        this.name = newName;
    }
}
```

## Encapsulation - Demo

https://www.w3schools.com/java/showjava_classes.asp?filename=demo_getset_error

https://www.w3schools.com/java/showjava_classes.asp?filename=demo_getset

# Inheritance
# (Subclass & Superclass)

it is possible to inherit **attributes** and **methods** from one class to another

- **subclass (child)** - the class that inherits from another class
- **superclass (parent)** - the class being inherited from

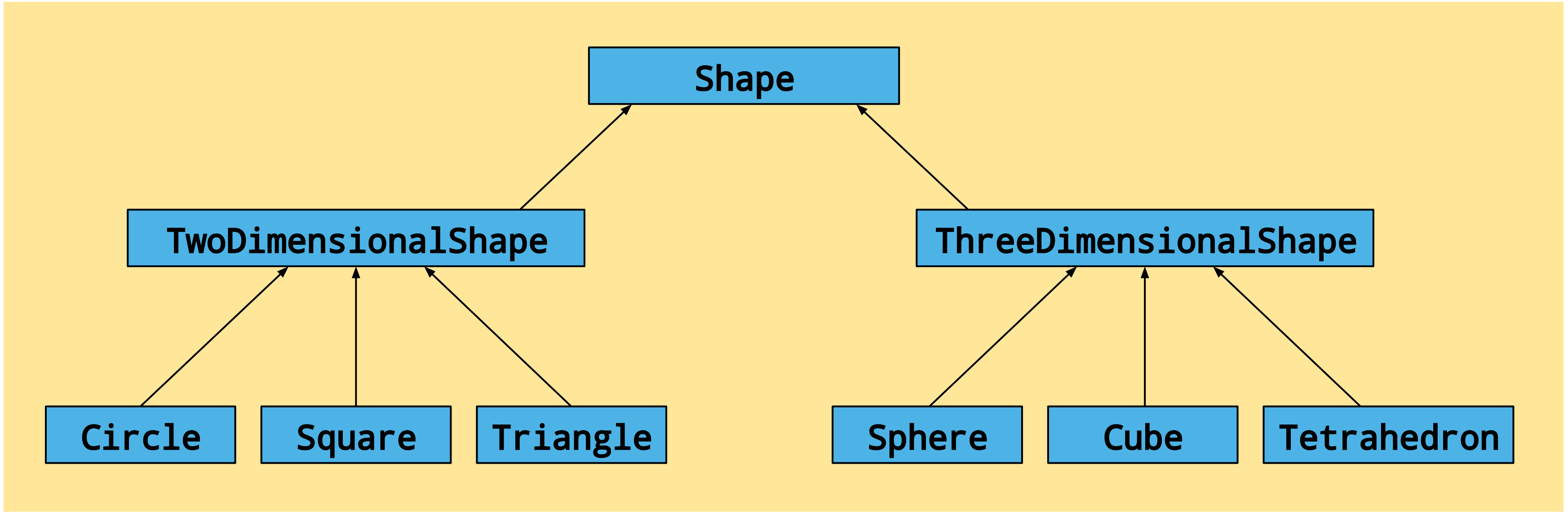
```
class Car extends Vehicle {
    // ....
}
```

```java
class Vehicle {
  protected String brand = "Ford";        // Vehicle attribute
  public void honk() {                    // Vehicle method
    System.out.println("Tuut, tuut!");
  }
}

class Car extends Vehicle {
  private String modelName = "Mustang";    // Car attribute
  public static void main(String[] args) {

    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (from the Vehicle class) on the myCar object
    myCar.honk();

    // Display the value of the brand attribute (from the Vehicle class) and the
value of the modelName from the Car class
    System.out.println(myCar.brand + " " + myCar.modelName);
  }
}
```

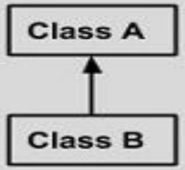
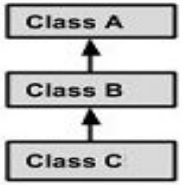
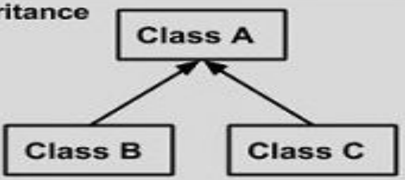
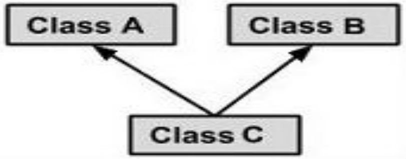
## Inheritance - demo

https://www.w3schools.com/java/showjava.asp?filename=demo_inherit

# Inheritance

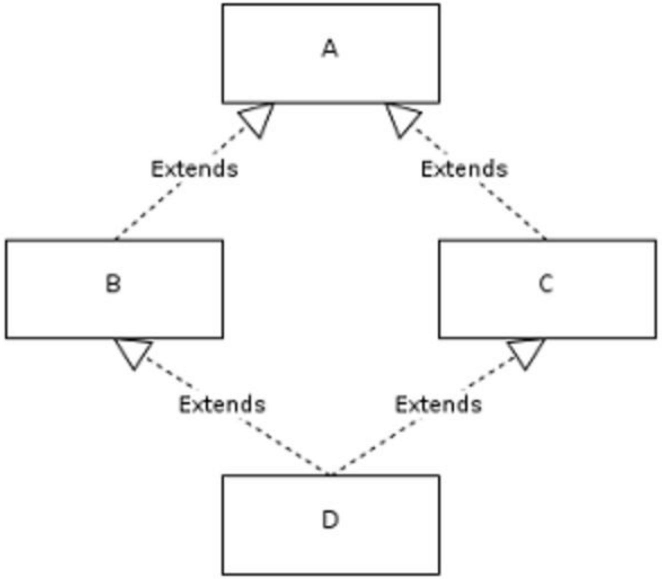


# Inheritance

| | |
|---|---|
| **Single Inheritance**  | public class A {<br>.......<br>}<br>public class B **extends** A {<br>.........<br>} |
| **Multi Level Inheritance**  | public class A { ...................}<br>public class B **extends** A {....................}<br>public class C **extends** B {..................... } |
| **Hierarchical Inheritance**  | public class A { ...................}<br>public class B **extends** A {....................}<br>public class C **extends** A {..................... } |
| **Multiple Inheritance**  | public class A { ...................}<br>public class B {....................}<br>public class C **extends** A,B {<br>........................<br>} // Java does not support mutiple Inheritance |

# Not support multiple inheritance

- Ambiguity around Diamond problem.
- Multiple inheritances **does complicate the design.** it's not because of technical difficulty but more to maintainable and clearer design.
- Rarely Used.



## Inheritance

- **Final**: a class has no subclass if it is described “final”.
- **Abstract:** an abstract class could not defined with “final”.

# Inheritance

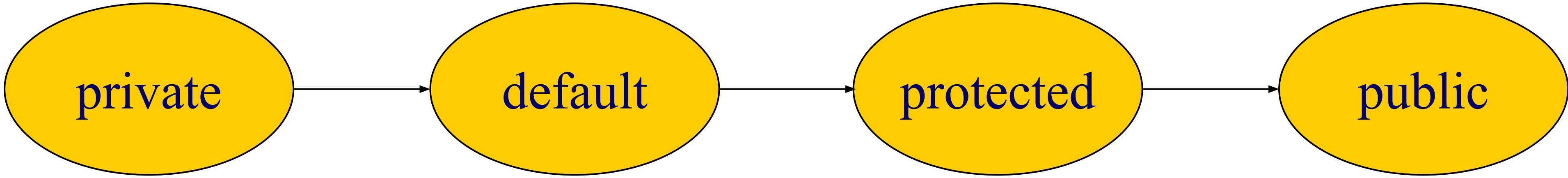
## Overriding Methods

- The ability of a subclass to override a method allows a class to inherit from a superclass whose behavior is "close enough" and then to modify behavior as needed. The overriding method has the same name, number and type of parameters, and return type as the method that it overrides

# Inheritance

## Overriding Methods

- **The access specifier for an overriding method can allow more, but not less, access than the overridden method**. For example, a protected instance method in the superclass can be made public, but not private, in the subclass.



## Polymorphism

- **Polymorphism** means **"many forms"**, and it occurs when we have many classes that are related to each other by inheritance.
- Inheritance lets us inherit **attributes** and **methods** from another class. **Polymorphism uses those methods to perform different tasks**. This allows us to **perform a single action in different ways**.

```java
class MyMainClass {
  public static void main(String[] args) {
    // Create a Animal object
    Animal myAnimal = new Animal();
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object
    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
  }
}
```

## Polymorphism - Demo

https://www.w3schools.com/java/showjava.asp?filename=demo_polymorphism

## Java Abstraction

Data abstraction is the process of **hiding certain details and showing only essential information to the user.**

Abstraction can be achieved with either

- **abstract classes** or
- **interfaces**

# Abstract class, method

- Abstract class is a restricted class that cannot be used to create objects.
- Abstract method can only be used in an abstract class

```
abstract class Animal {
  public abstract void animalSound();
  public void sleep() {
    System.out.println("Zzz");
  }
}
```

```
Animal myObj = new Animal();
// will generate an error
```

## Demo

https://www.w3schools.com/java/java_abstract.asp

## Java Interface

- An interface is a **completely "abstract class"** that is used to group related methods with empty bodies:

```
// interface
interface Animal {
  // interface method (does not have a body)
  public void animalSound();
  public void run();

}
```

## Java Interface

- An interface is a **reference type** in Java. It is similar to class. It is **a collection of abstract methods**. A class implements an interface, thereby inheriting the abstract methods of the interface.
- Along with abstract methods, an interface may also contain **constants**, **default methods, static methods**, and **nested types**. Method bodies exist only for default methods and static methods.

## Java Interface

- You **cannot instantiate an interface**.
- An interface **does not contain any constructors**.
- All of the methods in an interface are abstract.
- An interface **cannot contain instance fields**. The only **fields** that can appear in an interface **must be declared both static and final**.
- An interface **is not extended by a class**; it is implemented by a class.
- An interface **can extend multiple interfaces**.

```java
// Pig "implements" the Animal interface
class Pig implements Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
  public void sleep() {
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}
```

## Java Interface - Multiple interface

```
interface FirstInterface {
  public void myMethod(); // interface method
}
interface SecondInterface {
  public void myOtherMethod(); // interface method
}
class DemoClass implements FirstInterface,
SecondInterface {
   // ...
}
```

Source: https://www.w3schools.com/java/java_interface.asp

## Java Interface

### Static and Default Methods in interfaces in Java

- Like regular interface methods, **default methods are implicitly public** — there's no need to specify the *public* modifier.
- Unlike regular interface methods, they are **declared with the *default* keyword at the beginning of the method signature**, and they **provide an implementation**

Source: https://www.baeldung.com/java-static-default-methods

# Java Interface

## Static and Default Methods in interfaces in Java

```
public interface MyInterface {
    // regular interface methods
    default void defaultMethod() {
        // default method implementation
    }
}
```

```java
public interface Vehicle {

    String getBrand();

    String speedUp();

    String slowDown();

    default String turnAlarmOn() {
        return "Turning the vehicle alarm on.";
    }

    default String turnAlarmOff() {
        return "Turning the vehicle alarm off.";
    }
}
```

```java
public class Car implements Vehicle {

    private String brand;

    // constructors/getters

    @Override
    public String getBrand() {
        return brand;
    }

    @Override
    public String speedUp() {
        return "The car is speeding up.";
    }

    @Override
    public String slowDown() {
        return "The car is slowing down.";
    }
}
```

```java
public static void main(String[] args) {
    Vehicle car = new Car("BMW");
    System.out.println(car.getBrand());
    System.out.println(car.speedUp());
    System.out.println(car.slowDown());
    System.out.println(car.turnAlarmOn());
    System.out.println(car.turnAlarmOff());
}
```

## Inner Class

```
class OuterClass {
  int x = 10;
  class InnerClass {
    int y = 5;
  }
}
public class MyMainClass {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}
// Outputs 15 (5 + 10)
```

## Inner Class

- The purpose of nested classes is to **group classes that belong together**, which makes your code **more readable and maintainable**.

## Private Inner Class

- **Unlike a "regular" class**, an inner class can be **private** or **protected**. If you don't want outside objects to access the inner class, declare the class as private:

```
class OuterClass {
  int x = 10;

  private class InnerClass {
    int y = 5;
  }
}
```

## Static inner class

An inner class can also be static, which means that you can access it without creating an object of the outer class:

```
class OuterClass {
  int x = 10;

  static class InnerClass {
    int y = 5;
  }
}
```

## Inner class - Demo

https://www.w3schools.com/java/java_inner_classes.asp