



Cambodian University of Specialties

Faculty of Science and Technology



Java OOP III



Introduction to Java OOP

1. Polymorphism
2. Abstraction Classes
3. Abstraction methods

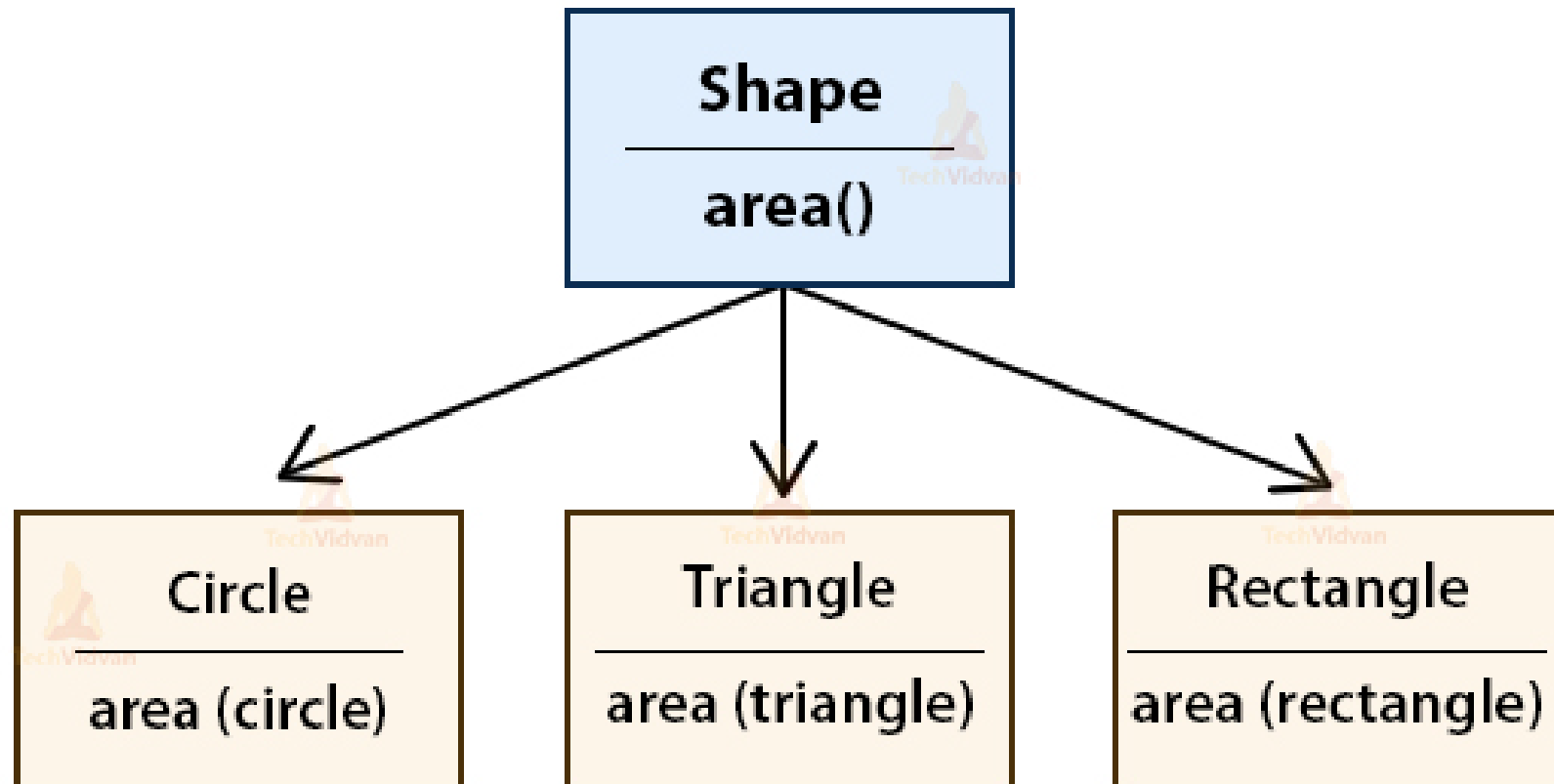


Java Polymorphism

- Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.
- Like we specified in the previous chapter; [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.
- For example, think of a superclass called Animal that has a method called animalSound(). Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

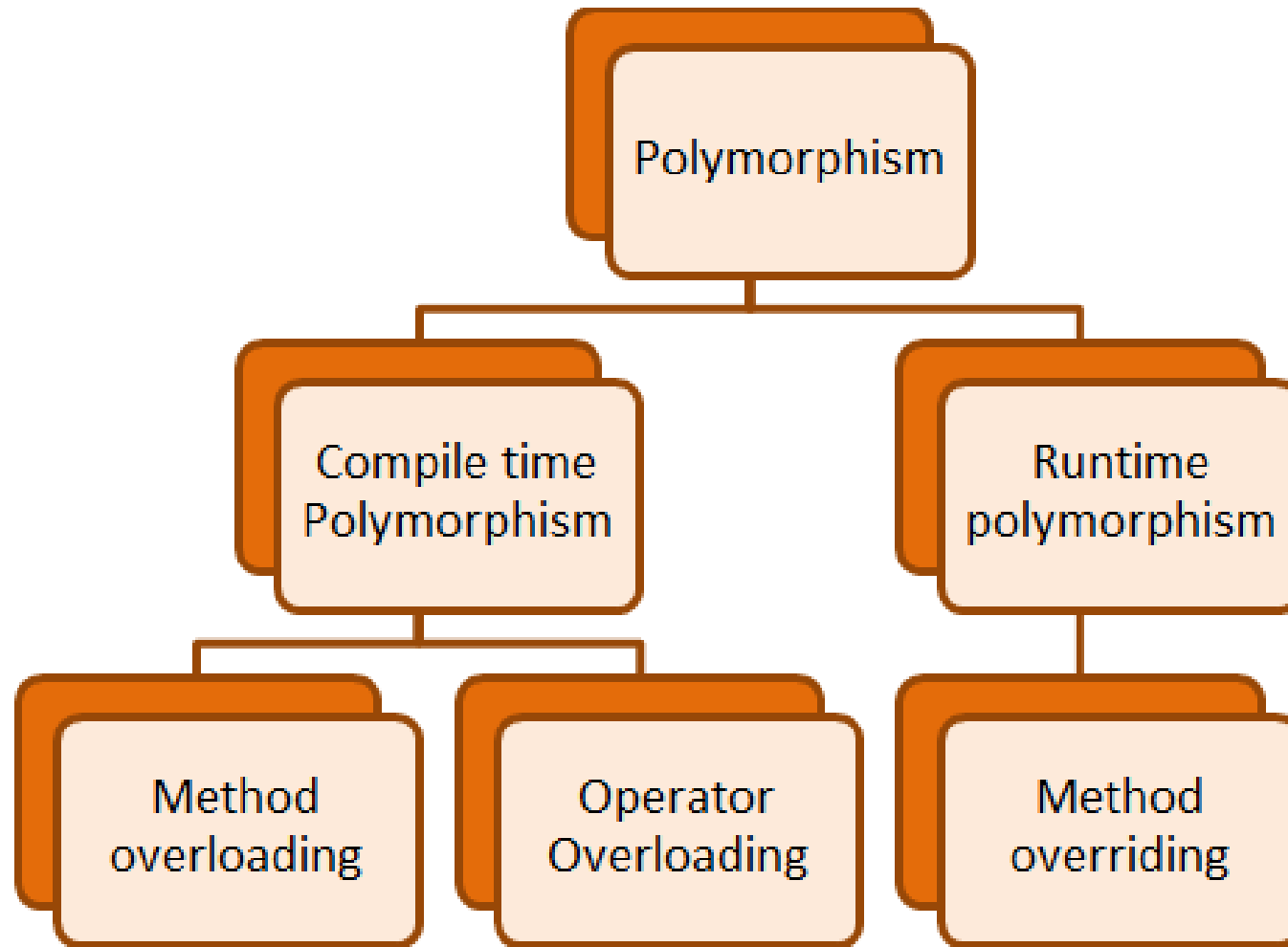


Java Polymorphism





Java Polymorphism





Java Polymorphism

Runtime Polymorphism in Java

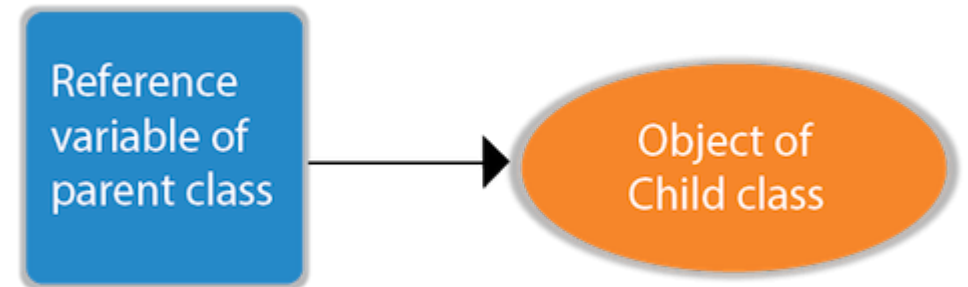
- **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.
- In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.



Java Polymorphism

Runtime Polymorphism in Java

- **Upcasting**
- If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:





Java Polymorphism

Runtime Polymorphism in Java

```
class A{}  
class B extends A{}
```

```
A a=new B();//upcasting
```

```
interface I{}  
class A{}  
class B extends A implements I{}
```

Here, the relationship of B class would be:

```
B IS-A A  
B IS-A I  
B IS-A Object
```




Java Polymorphism

Runtime Polymorphism in Java

- In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.
- Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.



Java Polymorphism

Runtime Polymorphism in Java

```
class Bike{  
    void run(){System.out.println("running");}  
}  
class Splendor extends Bike{  
    void run(){System.out.println("running safely with 60km");}  
  
    public static void main(String args[]){  
        Bike b = new Splendor();//upcasting  
        b.run();  
    }  
}
```

Output:

```
running safely with 60km.
```



Java Polymorphism

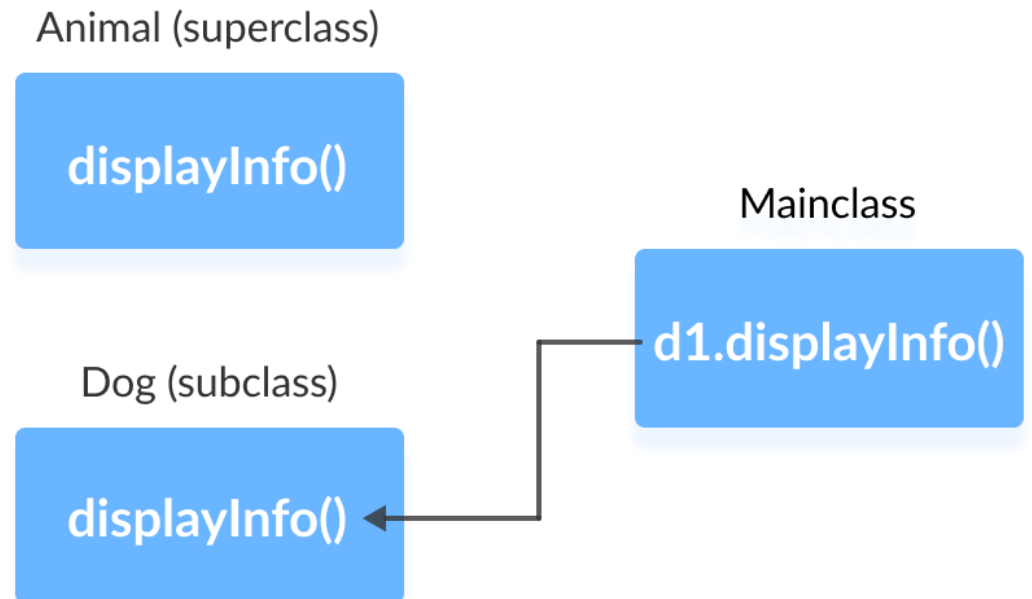
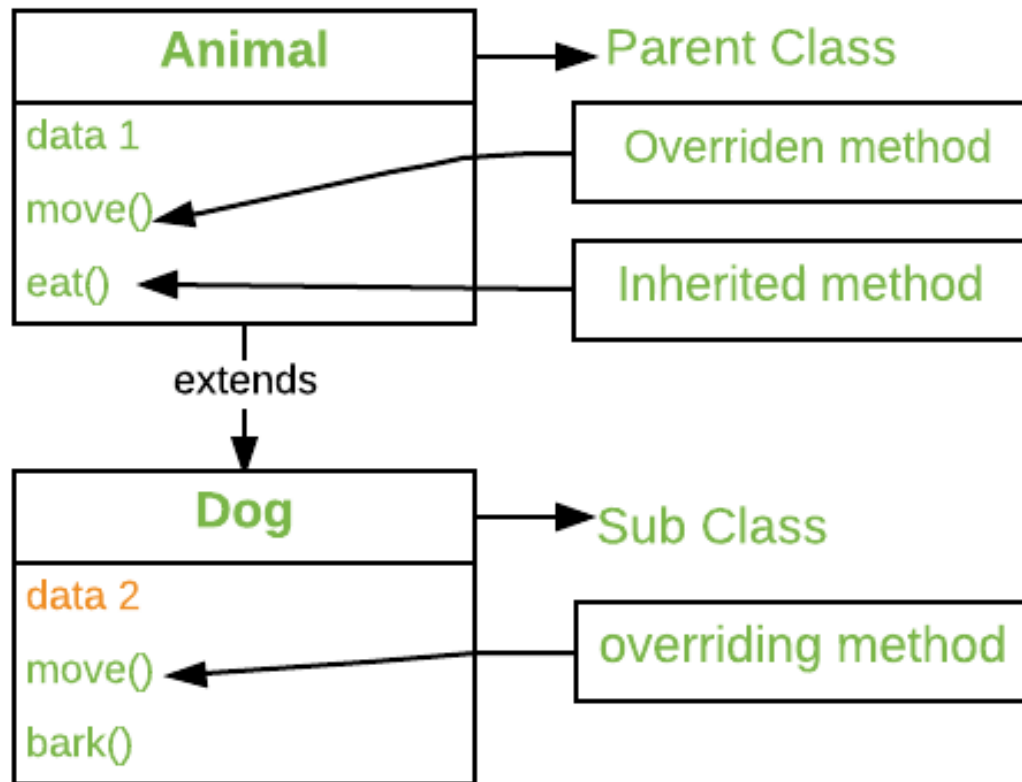
- **Java Method Overriding**

- During [inheritance in Java](#), if the same method is present in both the superclass and the subclass. Then, the method in the subclass overrides the same method in the superclass. This is called method overriding.
- In this case, the same method will perform one operation in the superclass and another operation in the subclass.



Java Polymorphism

- Java Method Overriding





Java Polymorphism

- Java Method Overriding

```
class Animal {  
    public void displayInfo() {  
        System.out.println("I am an animal.");  
    }  
}  
  
class Dog extends Animal {  
    @Override  
    public void displayInfo() {  
        System.out.println("I am a dog.");  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        Dog d1 = new Dog();  
        d1.displayInfo();  
    }  
}
```

Output:

I am a dog.



Java Polymorphism

- **super Keyword in Java Overriding**
- A common question that arises while performing overriding in Java is:
- Can we access the method of the superclass after overriding?
- Well, the answer is Yes. To access the method of the superclass from the subclass, we use the super keyword.



Java Polymorphism

- **super** Keyword in Java Overriding

```
class Animal {  
    public void displayInfo() {  
        System.out.println("I am an animal.");  
    }  
}  
  
class Dog extends Animal {  
    public void displayInfo() {  
        super.displayInfo();  
        System.out.println("I am a dog.");  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        Dog d1 = new Dog();  
        d1.displayInfo();  
    }  
}
```

Output:

```
I am an animal.  
I am a dog.
```



Java Polymorphism

- **Access Specifiers in Method Overriding**
- The same method declared in the superclass and its subclasses can have different access specifiers. However, there is a restriction.
- We can only use those access specifiers in subclasses that provide larger access than the access specifier of the superclass. For example,
- Suppose, a method `myClass()` in the superclass is declared protected. Then, the same method `myClass()` in the subclass can be either public or protected, but not private.



Java Polymorphism

- Access Specifiers in Method Overriding

```
class Animal {  
    protected void displayInfo() {  
        System.out.println("I am an animal.");  
    }  
}  
  
class Dog extends Animal {  
    public void displayInfo() {  
        System.out.println("I am a dog.");  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        Dog d1 = new Dog();  
        d1.displayInfo();  
    }  
}
```

Output:

I am a dog.



Java Polymorphism

- **Java Method Overloading**

- In a Java class, we can create methods with the same name if they differ in parameters. This is known as method overloading in Java. Here, the same method will perform different operations based on the parameter.



Java Polymorphism

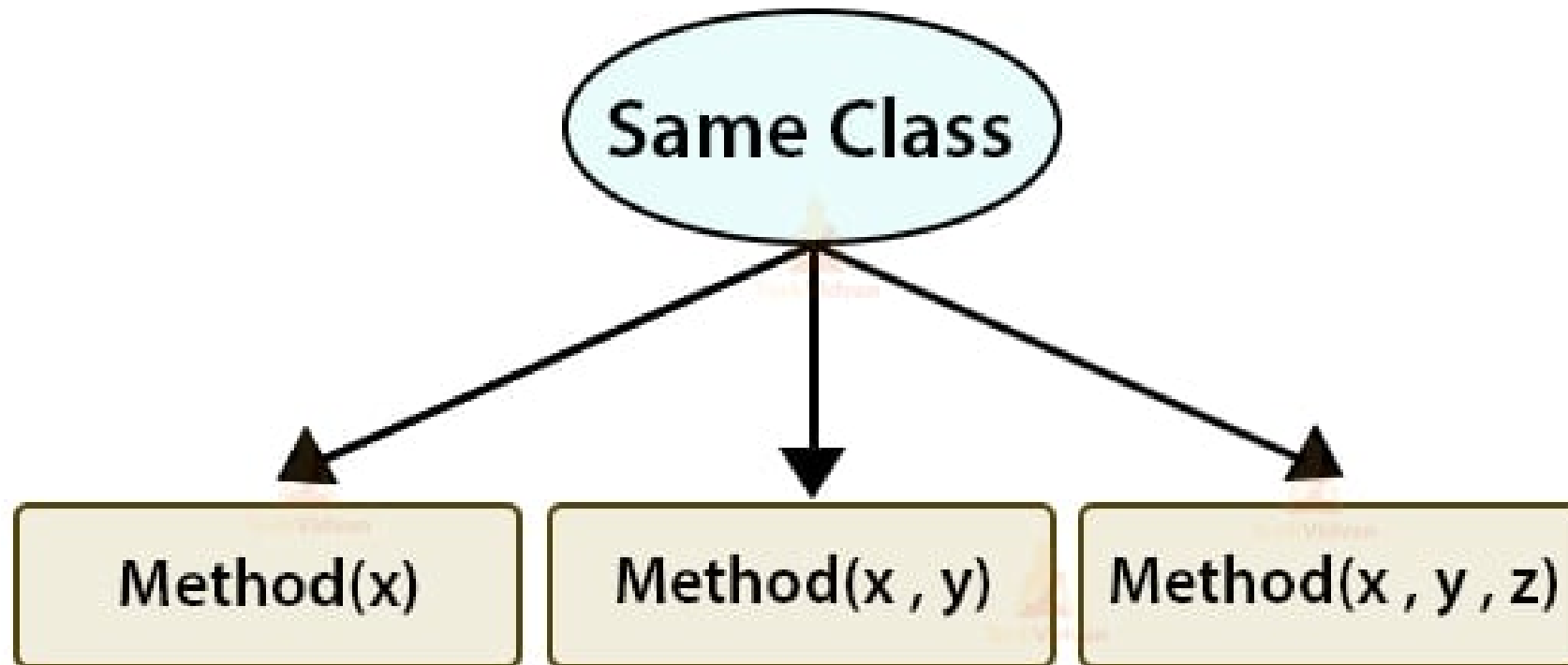
- Java Method Overloading

```
void func() { ... }  
void func(int a) { ... }  
float func(double a) { ... }  
float func(int a, float b) { ... }
```



Java Polymorphism

- Java Method Overloading





Java Polymorphism

• Java Method Overloading

```
class Pattern {  
  
    // method without parameter  
    public void display() {  
        for (int i = 0; i < 10; i++) {  
            System.out.print("*");  
        }  
    }  
  
    // method with single parameter  
    public void display(char symbol) {  
        for (int i = 0; i < 10; i++) {  
            System.out.print(symbol);  
        }  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        Pattern d1 = new Pattern();  
  
        // call method without any argument  
        d1.display();  
        System.out.println("\n");  
  
        // call method with a single argument  
        d1.display('#');  
    }  
}
```

Output:

```
*****
```

```
#####
```



Java Abstract Class

- **Abstraction Classes**

- A class which is declared with the **abstract** keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).
- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.



Java Abstract Class

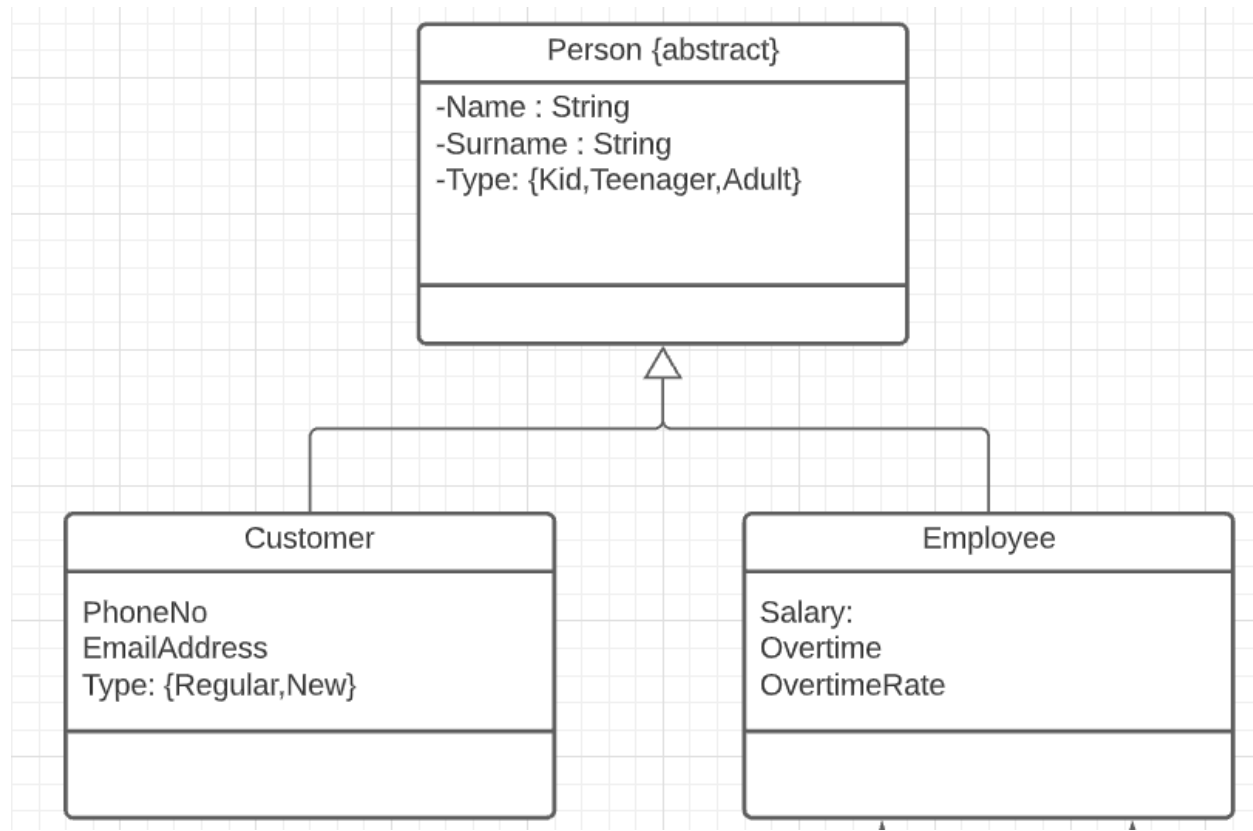
- **Abstraction Classes**

1. An abstract class must be declared with an abstract keyword.
2. It can have abstract and non-abstract methods.
3. It cannot be instantiated.
4. It can have [constructors](#) and static methods also.
5. It can have final methods which will force the subclass not to change the body of the method.



Java Abstract Class

- Abstraction Classes





Java Abstract Class

- Abstraction Classes

```
// create an abstract class
abstract class Language {
    // fields and methods
}

...

// try to create an object Language
// throws an error
Language obj = new Language();
```



Java Abstract Class

- **Abstraction Classes**

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



Java Abstract Class

- Abstraction Classes

```
// Abstract class
abstract class Animal {
    // Abstract method (does not have a body)
    public abstract void animalSound();
    // Regular method
    public void sleep() {
        System.out.println("Zzz");
    }
}
```

```
class Main {
    public static void main(String[] args) {
        Pig myPig = new Pig(); // Create a Pig object
        myPig.animalSound();
        myPig.sleep();
    }
}
```

```
// Subclass (inherit from Animal)
class Pig extends Animal {
    public void animalSound() {
        // The body of animalSound() is provided here
        System.out.println("The pig says: wee wee");
    }
}
```



Java Abstract Methods

- **Abstraction Methods**

A method that doesn't have its body is known as an abstract method.

We use the same **abstract** keyword to create abstract methods. For example,

```
abstract void display();
```



Java Abstract Methods

- **Abstraction Methods**

- Here, display() is an abstract method. The body of **display()** is replaced by ;.
- If a class contains an abstract method, then the class should be declared **abstract**. Otherwise, it will generate an error. For example,

```
// error
// class should be abstract
class Language {

    // abstract method
    abstract void method1();
}
```



Java Abstract Methods

- **Java Abstract Class and Method**
- Though abstract classes cannot be instantiated, we can create subclasses from it. We can then access members of the abstract class using the object of the subclass. For example,

Output

```
This is Java programming
```

```
abstract class Language {  
  
    // method of abstract class  
    public void display() {  
        System.out.println("This is Java Programming");  
    }  
}  
  
class Main extends Language {  
  
    public static void main(String[] args) {  
  
        // create an object of Main  
        Main obj = new Main();  
  
        // access method of abstract class  
        // using object of Main class  
        obj.display();  
    }  
}
```



Java Abstract Methods

- **Implementing Abstract Methods**
- If the abstract class includes any abstract method, then all the child classes inherited from the abstract superclass must provide the implementation of the abstract method. For example,

Output

```
Bark bark  
I can eat.
```

```
abstract class Animal {  
    abstract void makeSound();  
  
    public void eat() {  
        System.out.println("I can eat.");  
    }  
}  
  
class Dog extends Animal {  
  
    // provide implementation of abstract method  
    public void makeSound() {  
        System.out.println("Bark bark");  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
  
        // create an object of Dog class  
        Dog d1 = new Dog();  
  
        d1.makeSound();  
        d1.eat();  
    }  
}
```



Java Abstract Methods

- **Accesses Constructor of Abstract Classes**
- An abstract class can have constructors like the regular class. And, we can access the constructor of an abstract class from the subclass using the **super** keyword. For example,

```
abstract class Animal {  
    Animal() {  
        ...  
    }  
}  
  
class Dog extends Animal {  
    Dog() {  
        super();  
        ...  
    }  
}
```




Java Abstract Methods

```
abstract class MotorBike {  
    abstract void brake();  
}  
  
class SportsBike extends MotorBike {  
  
    // implementation of abstract method  
    public void brake() {  
        System.out.println("SportsBike Brake");  
    }  
}  
  
class MountainBike extends MotorBike {  
  
    // implementation of abstract method  
    public void brake() {  
        System.out.println("MountainBike Brake");  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        MountainBike m1 = new MountainBike();  
        m1.brake();  
        SportsBike s1 = new SportsBike();  
        s1.brake();  
    }  
}
```

Output:

```
MountainBike Brake  
SportsBike Brake
```



Java Abstract

- We use the **abstract** keyword to create abstract classes and methods.
- An **abstract** method **doesn't** have any implementation (method body).
- A class containing **abstract** methods should also be **abstract**.
- We cannot create objects of an abstract class.
- To implement features of an abstract class, we **inherit subclasses from it and create objects of the subclass**.
- **A subclass must override all abstract methods of an abstract class.** However, if the subclass is declared abstract, it's not mandatory to override abstract methods.



Java OOP II



1. Encapsulation

2. Interfaces

3. Packages



Java Encapsulation

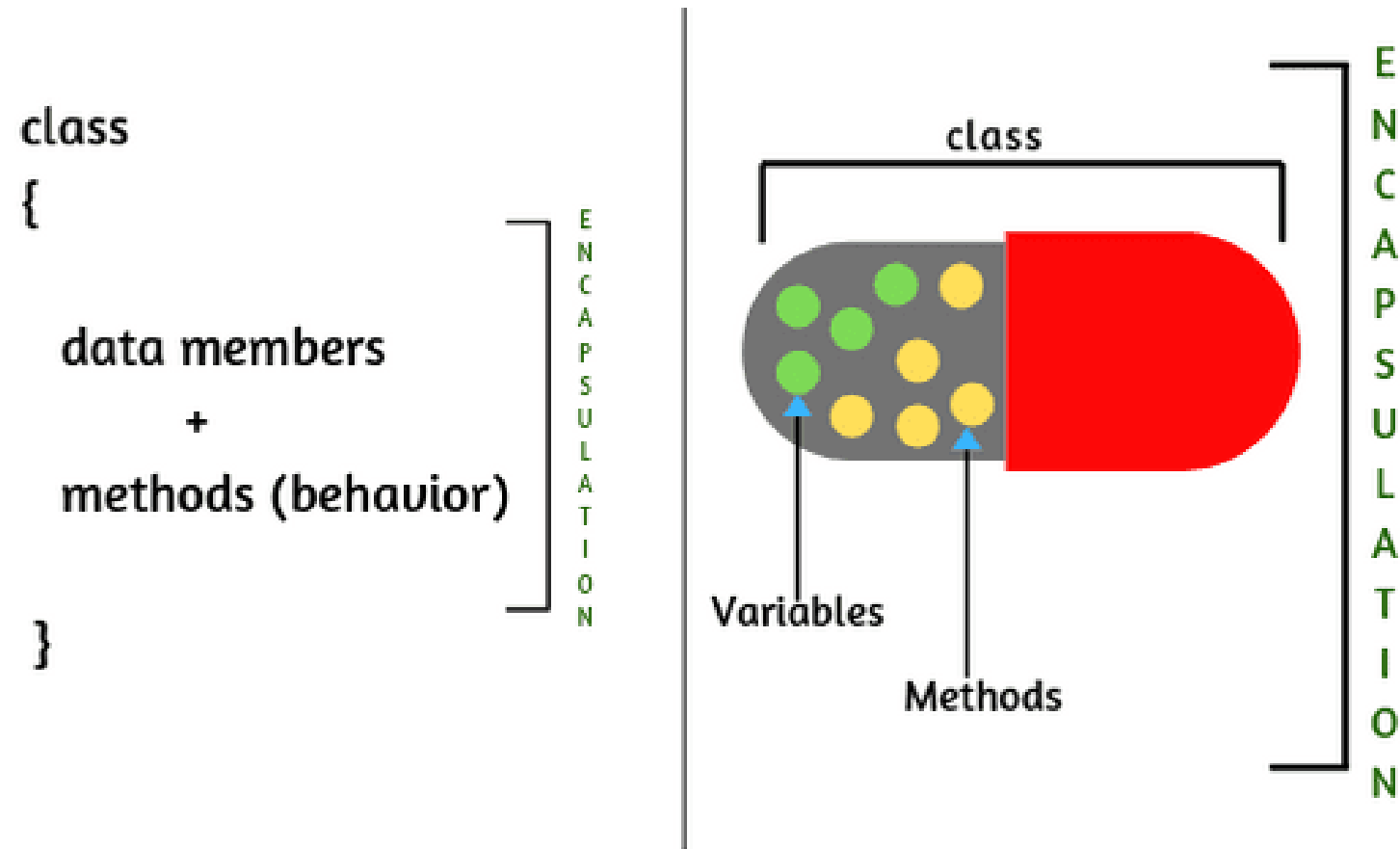


Fig: Encapsulation



Java Encapsulation

- **Encapsulation** is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.
- Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.
- To achieve encapsulation in Java –
- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.



Java Encapsulation

```
class Person {  
    private String name;  
    private int age;  
    public String getName() { return name; }  
    public void setName(String name) { this.name = name; }  
    public int getAge() { return age; }  
    public void setAge(int age) { this.age = age; }  
}  
  
public class Main {  
    public static void main(String[] args)  
    {  
        Person person = new Person();  
        person.setName("John");  
        person.setAge(30);  
        System.out.println("Name: " + person.getName());  
        System.out.println("Age: " + person.getAge());  
    }  
}
```

Output

Name: John

Age: 30



Java Encapsulation Read-Only class

//A Java class which has only getter methods.

```
public class Student{  
    //private data member  
    private String college="AKG";  
    //getter method for college  
    public String getCollege(){  
        return college;  
    }  
}
```

```
s.setCollege("KITE");//will render compile time error
```



Java Encapsulation Write-Only class

//A Java class which has only setter methods.

```
public class Student{  
    //private data member  
    private String college;  
    //getter method for college  
    public void setCollege(String college){  
        this.college=college;  
    }  
}
```

System.out.println(s.getCollege());//Compile Time Error, because there is no such method
System.out.println(s.college);//Compile Time Error, because the college data member is private.
//So, it can't be accessed from outside the class

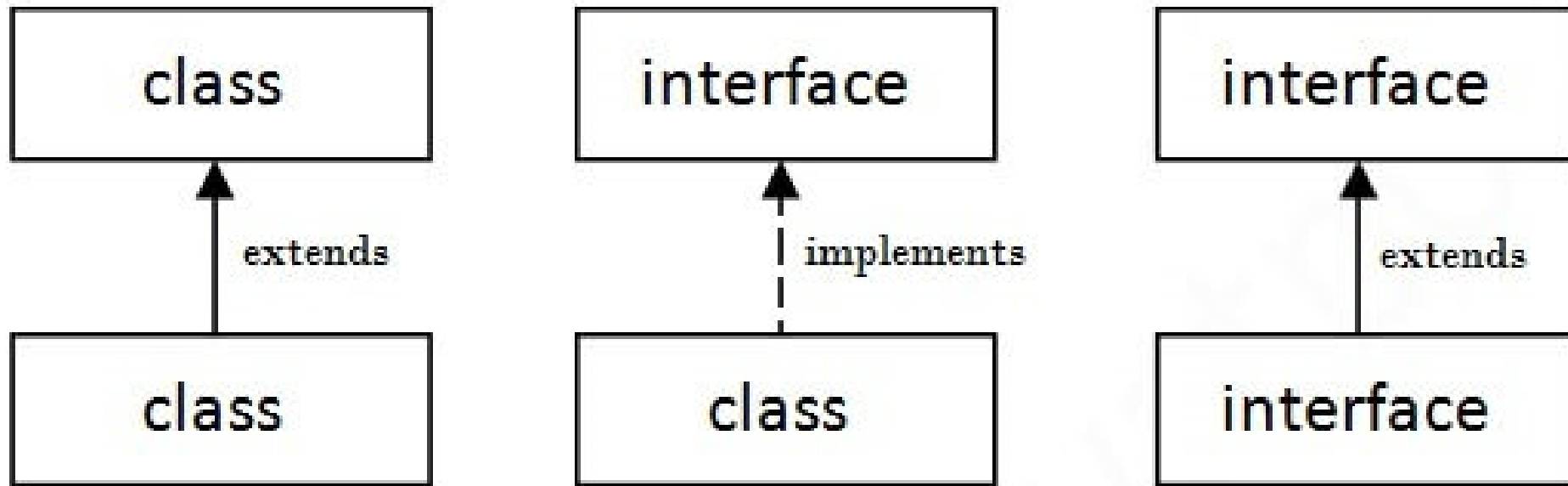


Java Interface

- An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.
- The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.
- In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.



Java Interface





Java Interface



Multiple Inheritance in Java



Java Interface

```
interface Animals{  
    void makeSound();  
}
```

interface
declaration in
java

Abstract
method


```
class Cow implements  
Animals{  
    void makeSound(){  
        System.out.print("The sound  
a cow makes is moo");  
    }  
}
```

```
class Dog implements Animals{  
    void makeSound() {  
        System.out.println("The sound a  
dog makes is woof woof!! ");  
    }  
}
```

```
class Cat implements Animals {  
    void makeSound() {  
        System.out.print("The sound a  
cat makes is meow");  
    }  
}
```




Java Interface



Difference between Class & Interface in Java

Class

- Class can instantiate variable & create object
- Class can contain concrete methods
- The access specifiers used with classes are private, protected and public



10460	Benefits	
35246	Payroll	
76745	Solaries	
76023	Commissions and bonuses	12.44
23674	Personnel Total	
10460	Benefits	
35246	Payroll	
76745	Solaries	
76023	Commissions	
23674	Personnel Total	12.32
Stocks Exchange Company (As)		
Worminnud		0.83745x7
776		
0000.09 - 02.70583		

Interface

- Interface can't instantiate variable & create an object
- The interface can't contain concrete methods
- In interface only one public specifier is used



Java Interface

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.



Java Interface

Syntax:

```
interface <interface_name>{  
  
    // declare constant fields  
    // declare methods that abstract  
    // by default.  
}
```



Java Interface

Java Interface Example

- In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

```
interface printable{  
    void print();  
}  
  
class A6 implements printable{  
    public void print(){System.out.println("Hello");}  
  
    public static void main(String args[]){  
        A6 obj = new A6();  
        obj.print();  
    }  
}
```




Java Interface

Java Interface Example: Drawable

- In this example, the Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

```
//Interface declaration: by first user
interface Drawable{
    void draw();
}

//Implementation: by second user
class Rectangle implements Drawable{
    public void draw(){System.out.println("drawing rectangle");}
}

class Circle implements Drawable{
    public void draw(){System.out.println("drawing circle");}
}

//Using interface: by third user
class TestInterface1{
    public static void main(String args[]){
        Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
        d.draw();
    }
}
```



Java Interface

- Java Interface Example: Bank
- Let's see another example of java interface which provides the implementation of Bank interface.

Output:

```
ROI: 9.15
```

```
interface Bank{  
    float rateOfInterest();  
}  
  
class SBI implements Bank{  
    public float rateOfInterest(){return 9.15f;}  
}  
  
class PNB implements Bank{  
    public float rateOfInterest(){return 9.7f;}  
}  
  
class TestInterface2{  
    public static void main(String[] args){  
        Bank b=new SBI();  
        System.out.println("ROI: "+b.rateOfInterest());  
    }  
}
```



Java Interface

Multiple inheritance in Java by interface

- If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.

```
interface Printable{  
    void print();  
}  
  
interface Showable{  
    void show();  
}  
  
class A7 implements Printable,Showable{  
    public void print(){System.out.println("Hello");}  
    public void show(){System.out.println("Welcome");}  
  
    public static void main(String args[]){  
        A7 obj = new A7();  
        obj.print();  
        obj.show();  
    }  
}
```



Java Interface

- Interface inheritance
- A class implements an interface, but one interface extends another interface.

Output:

```
Hello  
Welcome
```

```
interface Printable{  
    void print();  
}  
  
interface Showable extends Printable{  
    void show();  
}  
  
class TestInterface4 implements Showable{  
    public void print(){System.out.println("Hello");}  
    public void show(){System.out.println("Welcome");}  
  
    public static void main(String args[]){  
        TestInterface4 obj = new TestInterface4();  
        obj.print();  
        obj.show();  
    }  
}
```

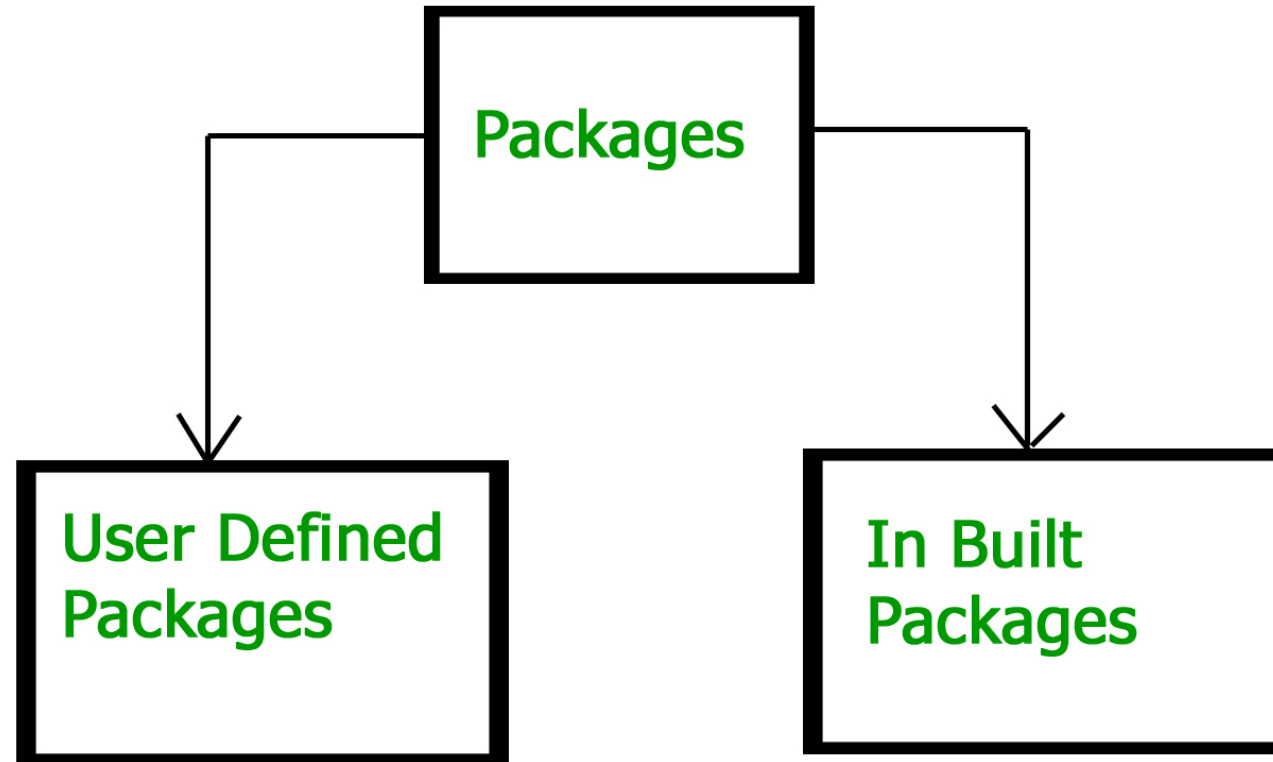


Java Package

- 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.

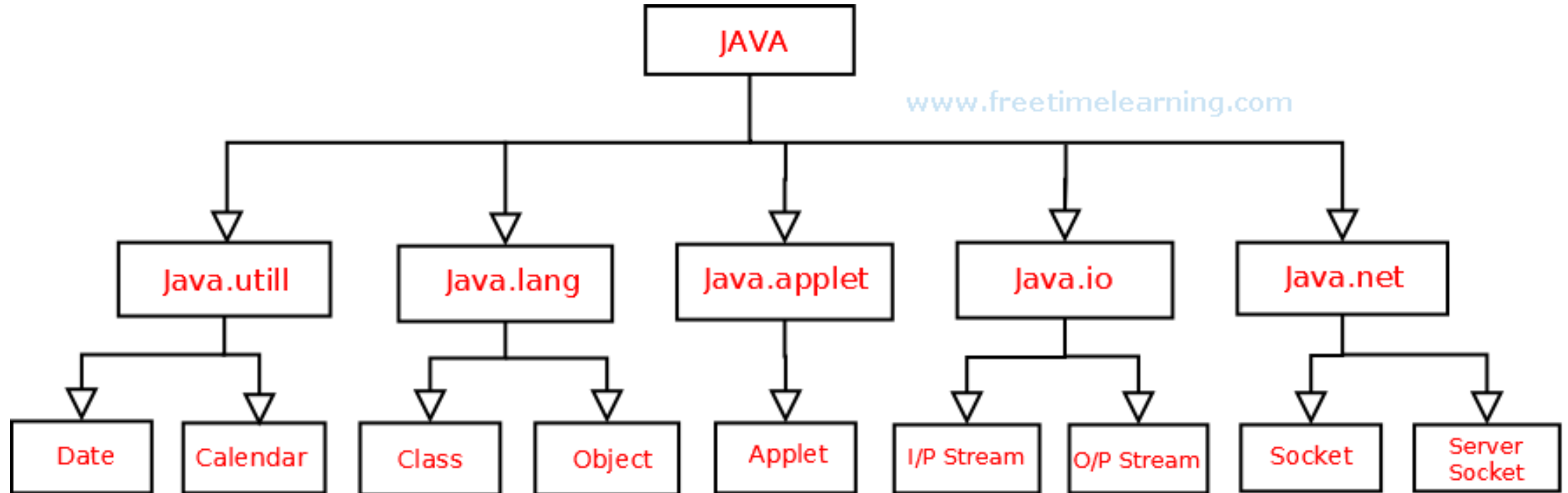


Java Package





Java Package





Java Package

MyPackageClass.java

```
package mypack;  
class MyPackageClass {  
    public static void main(String[] args) {  
        System.out.println("This is my package!");  
    }  
}
```




Java Package

Save the file as **MyPackageClass.java**, and compile it:

```
C:\Users\Your Name>javac MyPackageClass.java
```

Then compile the package:

```
C:\Users\Your Name>javac -d . MyPackageClass.java
```



Java Package

To Compile: `javac -d . Simple.java`

To Run: `java mypack.Simple`

Output:Welcome to package



Java Package

```
import package_one.ClassTwo;  
import package_name.ClassOne;  
public class Testing {  
    public static void main(String[] args){  
        ClassTwo a = new ClassTwo();  
        ClassOne b = new ClassOne();  
        a.methodClassTwo();  
        b.methodClassOne();  
    }  
}
```

Output:

```
Hello there i am ClassTwo  
Hello there its ClassOne
```



Java Package

- Java has an import statement that allows you to import an entire package (as in earlier examples), or use only certain classes and interfaces defined in the package.
- The general form of import statement is:

```
import package.name.ClassName;    // To import a certain class only  
import package.name.*            // To import the whole package
```

```
import java.util.Date; // imports only Date class  
import java.io.*;      // imports everything inside java.io package
```



Java Package



```
package com.programiz;

public class Helper {
    public static String getFormattedDollar (double value){
        return String.format("%.2f", value);
    }
}
```

```
import com.programiz.Helper;

class UseHelper {
    public static void main(String[] args) {

        double value = 99.5;
        String formattedValue = Helper.getFormattedDollar(value);
        System.out.println("formattedValue = " + formattedValue);
    }
}
```