Object Oriented Programming

ICT2122

Inheritance

P.H.P. Nuwan Laksiri
Department of ICT
Faculty of Technology
University of Ruhuna

Lesson 03 - OOP Concepts - Part 01

Recap

- *JAVA – this keyword*
  - methods
  - constructors
- *JAVA - Constructor Chaining*
- *JAVA - Anonymous objects*
- *JAVA – Garbage Collection*
- *Static in Java*
  - *Static Fields*
  - *Static Methods*
  - *Static Initializers*
- Preventing instantiating a class

Outline

Object Oriented Concepts

- Object Oriented Programming simplifies the software development and maintenance by providing some concepts,
  - Object
  - Class
  - Inheritance
  - Polymorphism
  - Abstraction
  - Encapsulation

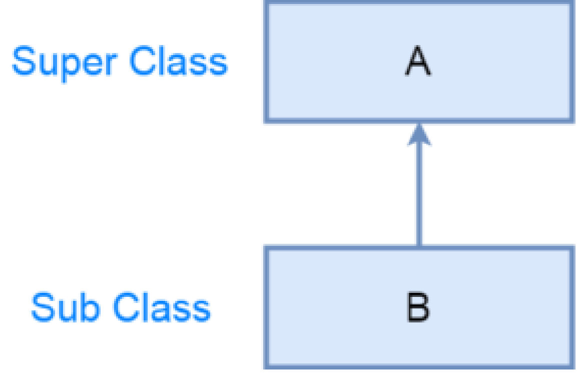
Classes and Objects

A class is like a cookie cutter; it defines the shape of objects

Objects are like cookies; they are instances of the class

Photograph courtesy of Guillaume Brialon on Flickr.

Inheritance



- When one object(*sub class*) **acquires all the properties and behaviors** of another object(*super class*).
  - Let you create classes that are derived from other classes.

- A class that is derived from another class is called a **sub class** (also a derived class, extended class, or child class).
- The class from which the subclass is derived is called a **super class** (also a base class or a parent class).

Inheritance

- A derived class automatically takes on all the behavior and attributes of its base class.
  - Subclass inherits all the members from its superclass
    - fields, methods, and nested classes
- A derived class can add features to the base class it inherits by defining its own methods and fields.
- A derived class can also change the behavior provided by the base class.

- Inheritance is best used to implement *is-a* type of relationships.

Inheritance

- The idea here is that you add what you want to the new class to give it more customized functionality than the original class.
- The subclass exhibits the behaviors of its superclass and can add behaviors that are specific to the subclass.
- This is why inheritance is sometimes referred to as **specialization**

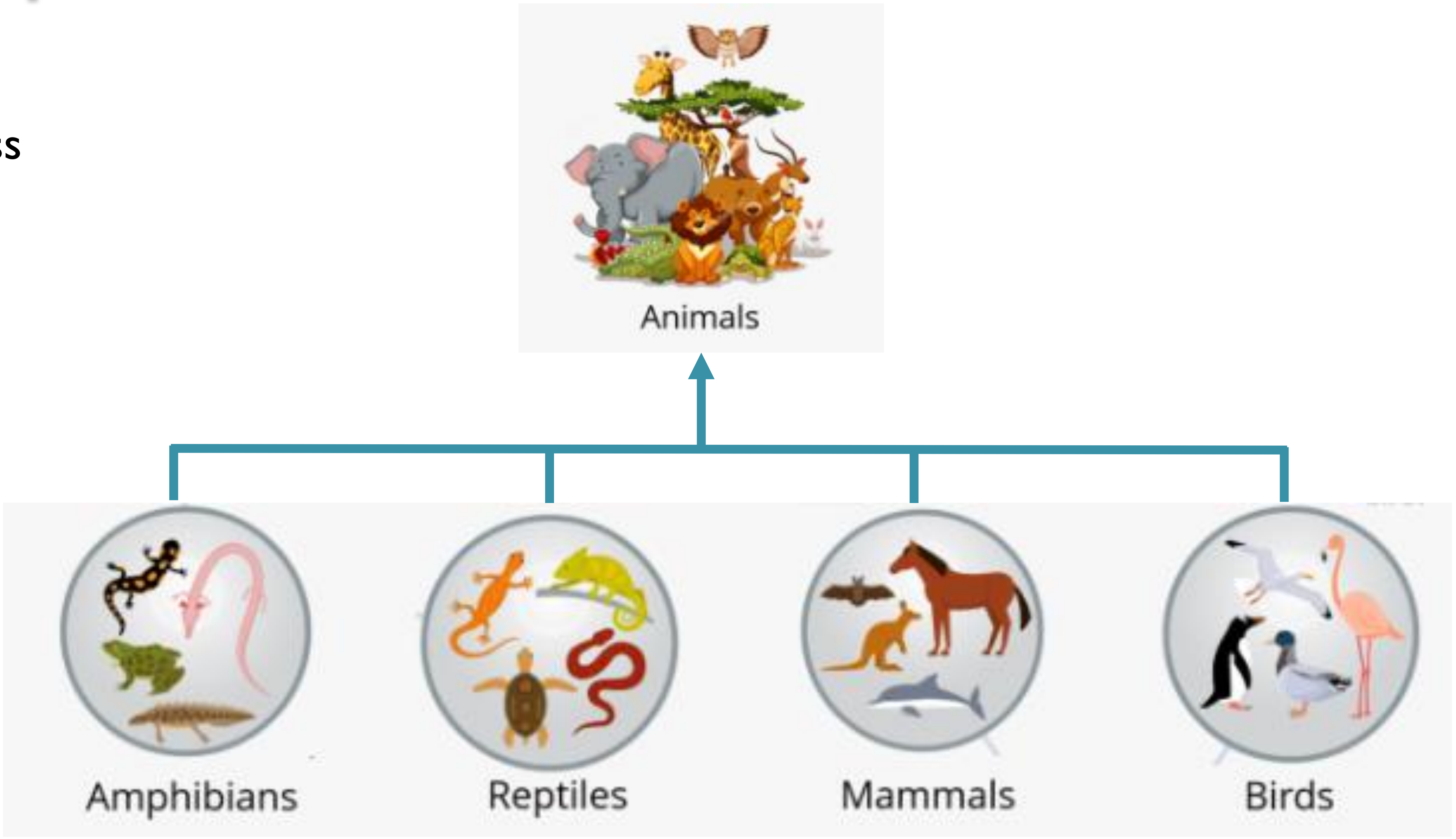
Inheritance

- The **direct superclass** is the superclass from which the **subclass explicitly inherits**.
- An **indirect superclass** is any class above the direct superclass in the class hierarchy.
- The Java **class hierarchy begins with class Object** (in package java.lang)
  - Every class in Java directly or indirectly extends (or "inherits from") Object.
- Java supports only **single inheritance**, in which each **class is derived from exactly one direct superclass**.

Example

Super Class

Sub Class



Hands – On

- Creating sub classes

```
Public class ClassName extends BaseClass{
        //class body
}
```

Creating Subclasses

- Suppose you have a class named “**Vehicle**” that has one method, “**start()**”, which you can use to start the vehicle and that prints out “**Starting**…..”.

```
public class Vehicle {
        public void start(){
                System.out.println(“Starting ….”);
        }
}
```

Creating Subclasses

- There are all kinds of vehicles, so if you want to specialize the "**Vehicle**" class into a "**Car**" class, you can use inheritance concept with the "**extends**" keyword.

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

- Car is derived from Vehicle class, and it will inherit "**start()**" method from "**Vehicle**"

Creating Subclasses

- You can also add your own data members and methods in subclasses.

```
public class Car extends Vehicle {
        public void drive() {
                System.out.println("Driving ….");
        }
}
```

- Now, can access the "**start()**" method and the "**drive()**" method in objects of the "**Car**" class, as shown below

Creating Subclasses

- Now, can access the “**start()**” method and the “**drive()**” method in objects of the “**Car**” class, as shown in this example:

```
class VehicleDemo {
        public static void main(String[] args) {
                System.out.println(“Creating a Car”);
                Car c = new Car();
                c.start();
                c.drive();
        }
}
```

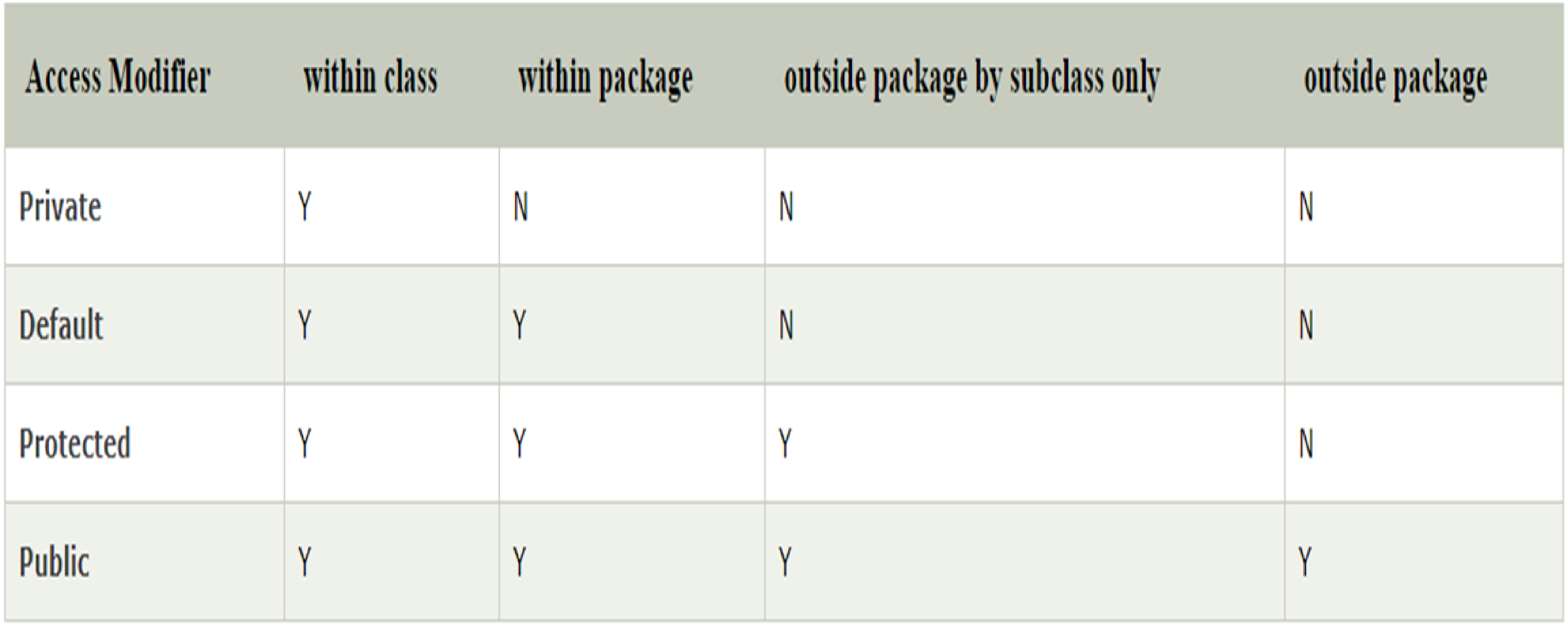
Homework

- Modify your "**Vehicle**" class as shown below

```
public class Vehicle {
        double speed = 40.0 ;
        public void start(){
                System.out.println("Starting ….");
        }
}
```

- Try to access the state "speed" through a *Car* object using dot(.) operator by changing the access modifier of speed *to public, private, protected* and *default* inside the VehicleDemo class

Behavior of Java Access Modifiers



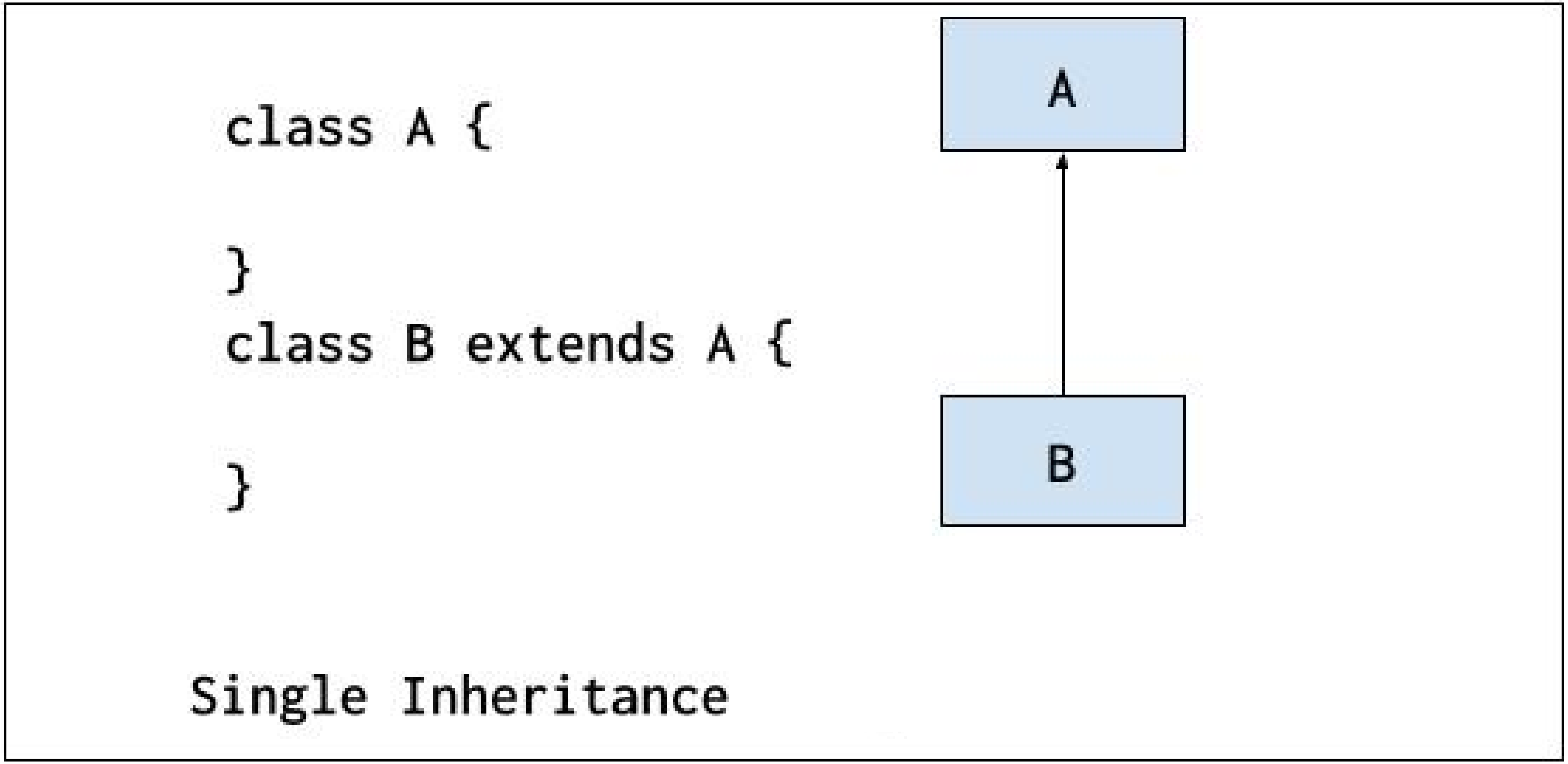
Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Types of inheritance in Java

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

- Multiple Inheritance → Not supported

Single Inheritance

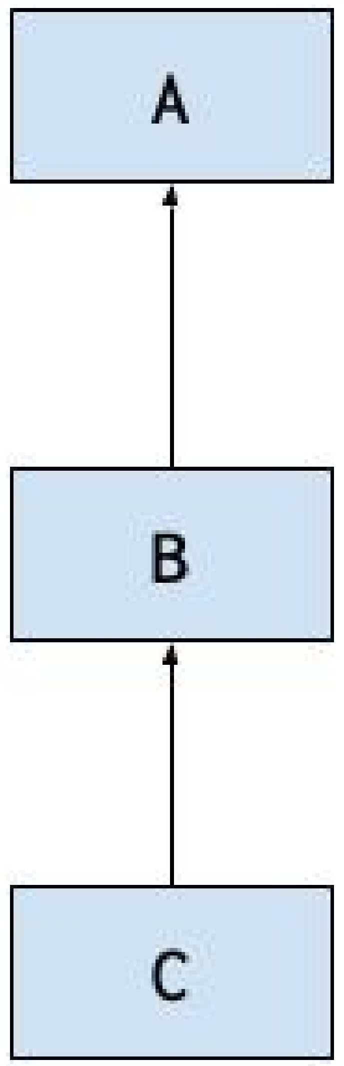


```
class A {

}
class B extends A {

}
```

Single Inheritance

Multilevel Inheritance

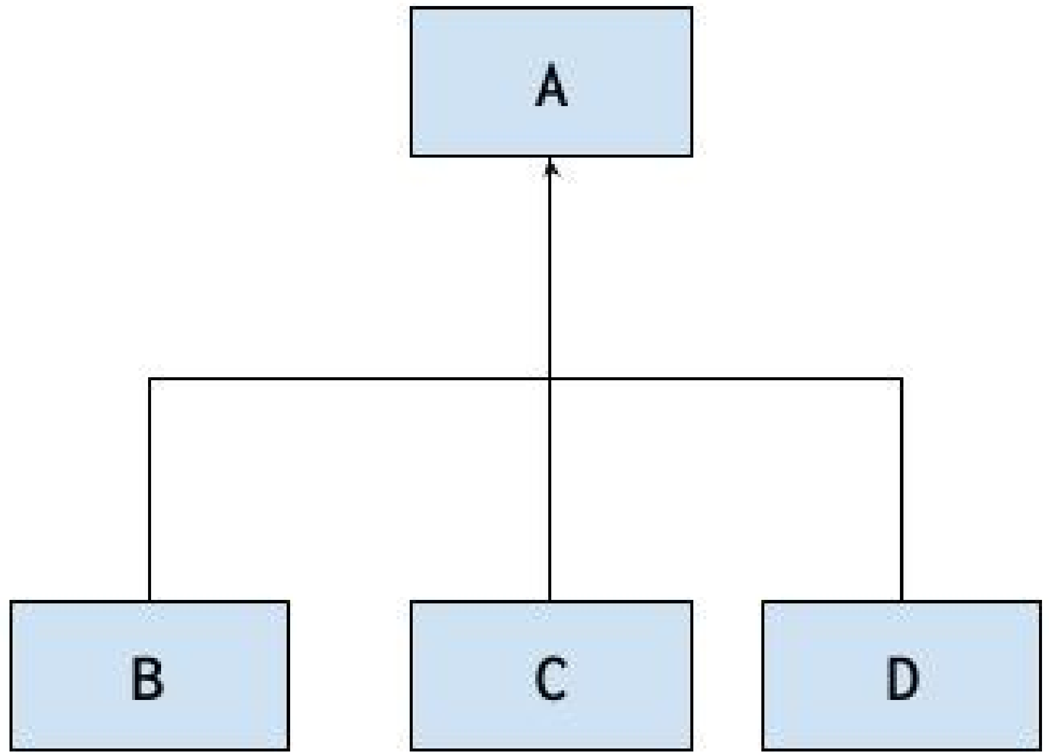
```
class A {

}
class B extends A {

}
class C extends B {

}
```



Multilevel Inheritance

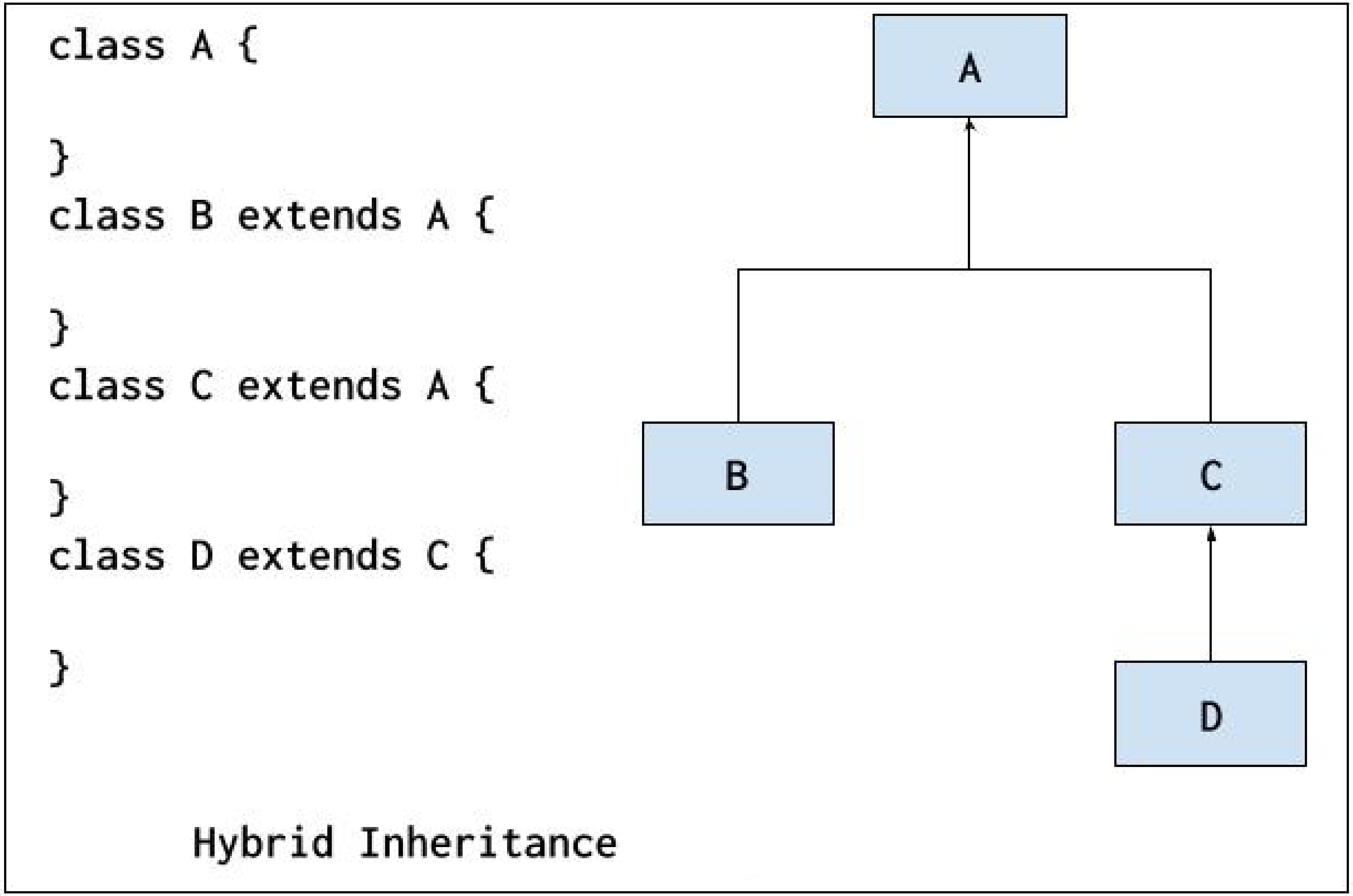
Hierarchical Inheritance

```
class A {

}
class B extends A {

}
class C extends A {

}
class D extends A {

}
```



Hierarchical Inheritance

Hybrid Inheritance



```
class A {

}
class B extends A {

}
class C extends A {

}
class D extends C {

}
```

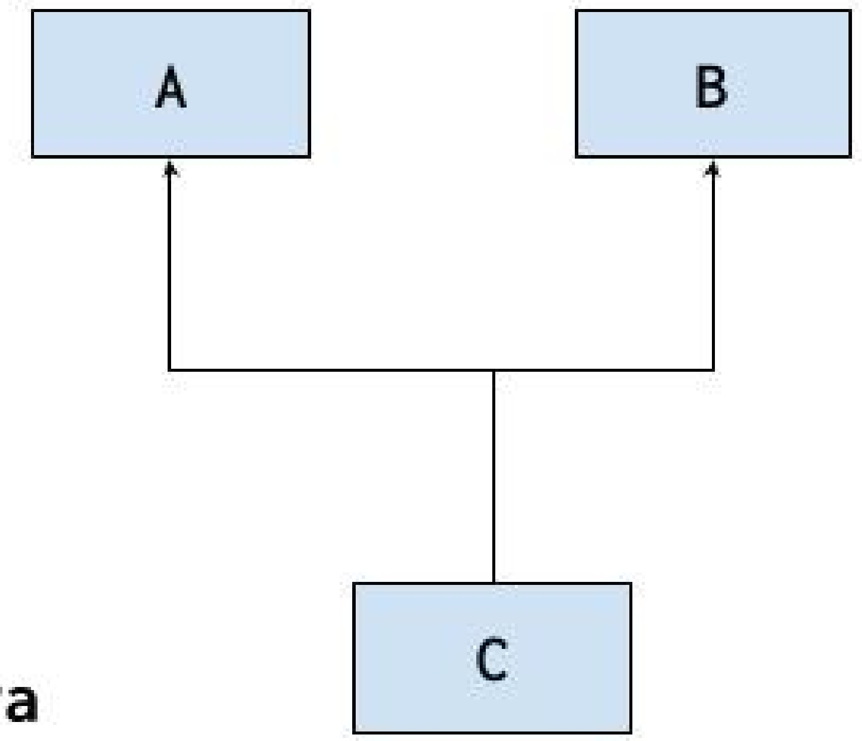
Hybrid Inheritance

$$\begin{tikzcd}
 & A & \\
B \arrow[ur] & & C \arrow[ul] \\
 & & D \arrow[u]
\end{tikzcd}$$

Multiple Inheritance

```
class A {

}
class B {

}
class C extends A, B {

}
```

This is not allowed in Java



Multiple Inheritance

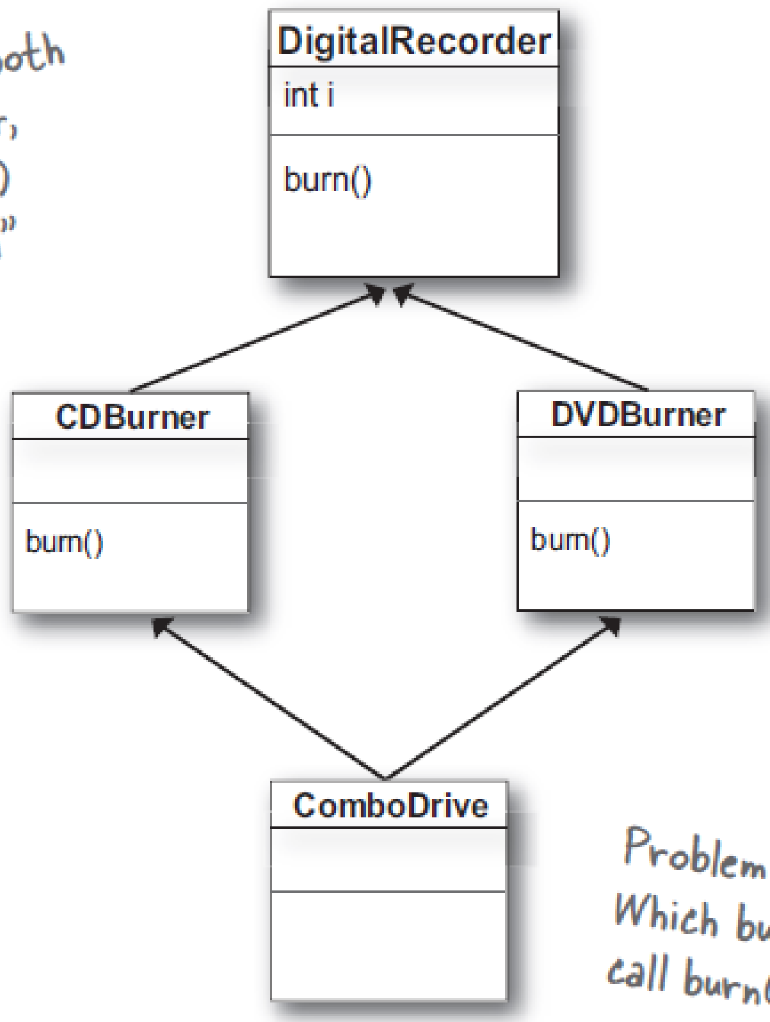
Multiple Inheritance

- Why it’s not allowed
  - Diamond problem

- What if we needed to implement such scenario?
  - Interfaces

Diamond problem

CDBurner and DVDBurner both inherit from DigitalRecorder, and both override the burn() method. Both inherit the "i" instance variable.



Imagine that the "i" instance variable is used by both CDBurner and DVDBurner, with different values. What happens if ComboDrive needs to use both values of "i"?

Problem with multiple inheritance. Which burn() method runs when you call burn() on the ComboDrive?

Overriding Methods (Instance Methods)

- An instance method in a subclass
  - **with the same signature** (name, plus the number and the type of its parameters) and
  - **return type**
  - as an instance method in the superclass

overrides the superclass's method.

- Use **@Override** annotation

- Method Signature ??

Overriding Methods (Instance Methods)

```
public class Car extends Vehicle {
        @Override
        public void start(){
                System.out.println(“Car is Starting ….”);
        }

        public void drive() {
                System.out.println(“Driving ….”);
        }
}
```

Overriding Methods (Instance Methods)

Special things

- Return Type
  - Can return a **covariant return type**

- Access Modifier
  - The access specifier for an overriding method can allow more, but not less, access than the overridden method.
    - For example,

      protected instance method in the superclass

      can be made public in the subclass √

      but not private in the subclass X

Hiding Methods (Static/Class Methods)

- If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass hides the one in the superclass.

In Vehicle

```
public static void printTopSpeed(){
        System.out.println("Top speed of Vehicle is 50");
}
```

In Car

```
public static void printTopSpeed(){
        System.out.println("Top speed of Car is 300");
}
```

Summary
Overriding Methods and Hiding Methods

	Superclass Instance Method	**Superclass Static Method**
Subclass Instance Method	Overrides	Generates a compile-time error
Subclass Static Method	Generates a compile-time error	Hides

Hiding Fields

- Within a class, a field that has the same name as a field in the superclass hides the superclass's field
  - even if their types are different.
- Field in the superclass must access through ***super***

- Generally, it's not recommend hiding fields as it makes code difficult to read.

Usage of *this* and *super* in Subclasses

- *this*
  - Refer to the current object instance.
  - Used to distinguished between local variable or a parameter with a class field.

- *super*
  - keyword is used when a subclass needs to access the members of its superclass.
- Usage of super keyword,
  - *super.fieldName* can be used to refer immediate parent class instance variable.
  - *super.methodName()* can be used to invoke immediate parent class method.
  - *super()* can be used to invoke immediate parent class constructor.

Usage of *super* in Subclass

Refer parent class method

- In Car

```
@Override
public void start(){
        System.out.println("Car is Starting ….");
        super.start();
}
```

- Use inside ***overriding(subclass)*** method to invoke ***overridden(super class)*** method

Usage of *super* in Subclass

Refer parent class instance variable

- In Vehicle

  add public String color = “White”;

- In Car ???

```
public void printColor(){
        color = “Red”;
        System.out.println(“Car color :  ” + color); //this.color
        System.out.println(“Vehicle color :  ” + super.color);
}
```

Constructors in Subclasses

- Instantiating a subclass object begins a chain of constructor calls
  - The subclass constructor, before performing its own tasks, explicitly uses super to call one of the constructors in its direct superclass or implicitly calls the superclass's default or no-argument constructor
- If the superclass is derived from another class, the superclass constructor invokes the constructor of the next class up the hierarchy, and so on.
- The last constructor called in the chain is always Object's constructor.
- Original subclass constructor's body finishes executing last.
- Each superclass's constructor manipulates the superclass instance variables that the subclass object inherits.

Calling supper class Constructors

- Suppose you have, a class named "A" that has a Constructor with no parameters:

```
class A
{
        A()
        {
                System.out.println("Inside A\'s Constructor");
        }
}
```

- Then you derive a subclass, "B" from "A":

```
class B extends A
{

}
```

Calling supper class Constructors

- When you create an object of class “B”, the Constructor in class “A” is called automatically

```
public class App
{
        public static void main(String[] args)
        {
                B obj = new B();
        }
}
```

- Compile above two programs and get the result:
  - Output :

```
Inside A’s Constructor
```

Calling supper class Constructors

- Suppose you add a constructor to class “B” that takes no parameters:

```
class B extends A
{
        B()
        {
         System.out.println(“Inside B\’s Constructor”);
         }
}
```

- In this case, when you create an object of class “B”, the constructors from both “A” and “B” are called:

```
Output :        Inside A’s Constructor
                Inside B’s Constructor
```

Calling supper class Constructors

- Now suppose you change B’s constructor so that it takes one parameter:

```
class B extends A
{
        B(String s)
        {
                System.out.println(“Inside B\’s Parameterized Constructor”);
                System.out.println(s);
         }
}
```

Calling supper class Constructors

```
class App
{
        public static void main(String arg[])
        {
                B obj = new B(“Hello”);
        }
}
```

- What will be the output?

Calling supper class Constructors

Things to remember

- super() call must be the very first statement in the constructor.
- If you don't explicitly call super, the compiler inserts a call to the default constructor of the base class.
  - In that case, the base class must have a default constructor.
  - If the base class doesn't have a default constructor, the compiler refuses to compile the program.
- If the superclass is itself a subclass, the constructor for its superclass is called in the same way.
  - This continues all the way up the inheritance hierarchy until you get to the Object class, which has no superclass.

Homework

```
class Box {
        double width;
        double height;
        double depth;
        Box(double w, double h, double d)
        {
                width=w;
                height=h;
                depth=d;
        }
}
```

Homework

- **Without using super keyword**

```
public class BoxWeight extends Box{
    double weight;

 BoxWeight(double w, double h, double d, double m)
 {
        width=w;
        height=h;
        depth=d;
        weight=m;
   }
}
```

Homework

- **Same code using super keyword**

```
public class BoxWeight extends Box{
      double weight;


       BoxWeight(double w, double h, double d, double m){
       super(w, h, d);
        weight=m;
    }
}
```

Usage of *final* keyword

- ***final*** with a variable
  - Create a constant whose value can't be changed after it has been initialized.

- ***final*** with a method
  - A final method is a method that can't be overridden by a subclass.

  Ex : public ***final*** void eat() { }

- ***final*** with a class
  - A final class is a class that can't be used as a base class.

  Ex : public ***final*** class MyConstants { }
  - In a final class all of its methods are considered to be final as well.

Casting Objects

- Casting means taking an **Object of one particular type and turning it into another Object type**.
- Casting can be done in following ways,
  - **Implicit casting**.
    - This is known as up casting (subclass class to super).
  - **Explicit casting.**
    - This is known as down casting (super class to subclass).

Casting Objects

Implicit casting.

- Casting shows the use of an object of one type in place of another type, among the objects permitted by inheritance and implementations.
  - **Object obj = new Car();** // object is indirect super class of car class
  - **Vehicle v = new Car();** //Vehicle is the direct super class of car class

Casting Objects

Explicit casting.

- Java does not permit to assign a super class object to a subclass object (implicitly) and still to do so, we need explicit casting.
  - Down casting requires explicit conversion.
    - Car myCar = obj;   // compile time error
    - Car myCar = (Car)obj; // explicit casting

Determining Object’s Type

- Use the ***instanceof*** operator

```
Vehicle v = new Vehicle();
Car c = new Car();
Vehicle v2 = newCar();

c instanceof  Car   // true
c instanceof  Vehicle   // true
c instanceof  Object   // true

v instanceof  Car   // false
```

- The object assigned to v2 is what type???

Summary

- Inheritance
  - Examples
  - Handson
- Creating Sub Classes
- Behavior of Java Access Modifiers
- Types of inheritance in Java
  - Single Inheritance
  - Multilevel Inheritance
  - Hierarchical Inheritance
  - Hybrid Inheritance
  - Multiple Inheritance
- Overriding Methods
- Hiding Methods
- Hiding Fields
- Usage of this and super in Subclasses
- Constructors in Subclasses
- Usage of final keyword
- Casting Objects
- Determining Object’s Type

References

- https://docs.oracle.com/javase/tutorial/java/landI/subclasses.html
- https://docs.oracle.com/javase/tutorial/java/concepts/inheritance.html
- How To Program (Early Objects)
  - By H .Deitel and P. Deitel
- Headfirst Java
  - By Kathy Sierra and Bert Bates

Questions ???

Thank You