JAVA PROGRAMMING 1

Summer 2018 - Christian Hur

# Unit 10 Lecture - Inheritance

Reading: Chapter 10

# Objectives:

* Learn about the concept of inheritance
* Extend classes
* Override superclass methods
* Call constructors during inheritance
* Access superclass methods
* Employ information hiding
* Learn which methods you cannot override

# Introduction

You may not know it, but you’ve already been using Inheritance since day one of this course. In Java, every object automatically is a child of a class named **Object**. Things like the Integer, String, JOptionPane, Scanner, etc. are all inheriting the Object class either directly or indirectly.

You’ve learned to create your own custom classes. In this unit, you’re going to learn how to create base class and derived classes.

# Inheritance (“is-a” relationship)

Object-oriented programming allows you to define new classes from existing classes. This is called **inheritance**. Object inheritance is where one class is derived from another, just as humans inherit qualities from their parents. Of course, the “qualities” in the object-oriented world are attributes (variables) and methods (functions). Inheritance enables you to define a general class (i.e., a superclass) and later extend it to more specialized classes (i.e., subclasses). This kind of inheritance means the two classes have an “**is-a**” relationship, in that a *ChildClass* is a type of *ParentClass*.

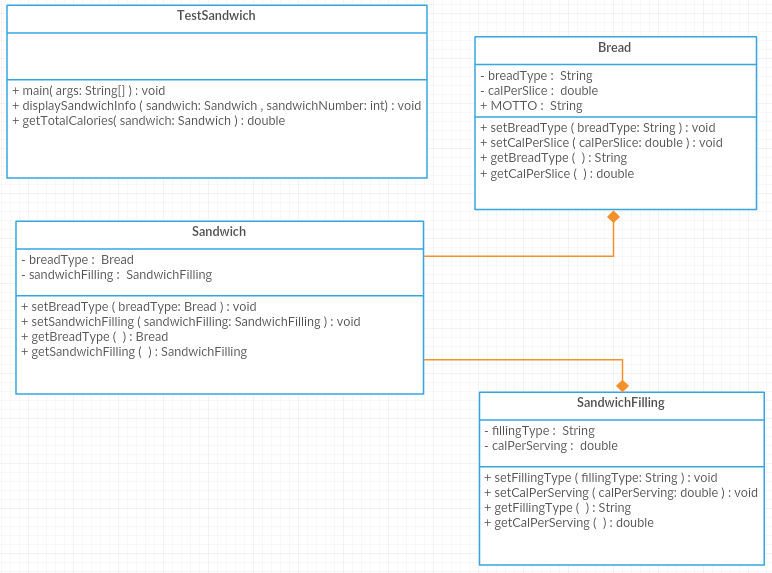
* Inheritance is a one-way proposition; a child inherits from a parent, not the other way around.
* Superclasses are smaller than Subclasses because a subclass contains (inherits) all properties and methods in the superclass and its own properties and methods.
* Subclasses are more specific than the superclass they extend.
* Two classes cannot inherit each other (forming a loop or circular inheritance). For example, a child cannot the parent of his own father????
* A subclass cannot inherit more than one superclass, but a superclass can be inherited by multiple superclasses. In other words, no such thing as a “hybrid” class in Java. For example, a sparrow cannot be both a bird and a raptile.

## Using Unified Modeling Langauge (UML) Diagrams

**Unified Modeling Language** or **UML** is consisted of many different types of diagrams that programmers use to help them construct classes and the different relationships between them (e.g. inheritance, composition, aggregation, dependency, etc.). Two of the most widely used UML diagrams in programming are the **class** and **object** diagrams. The ones you see here and in the textbook are mostly class diagrams. You can read more about UML here: <http://www.uml.org/>

### Composition & Aggregation (“**has a**” model)

If you recall from the Sandwich project we completed in Unit 4, I briefly introduced the UML diagrams like the following.



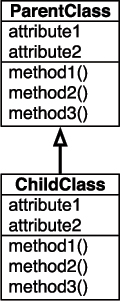
The diagram above shows two solid orange lines connecting the Bread and SandwichFilling classes to the Sandwich class. The lines also have a solid diamond shape. This indicates a “**composition**” which is a relationship in which a class contains one or more members of another class, when those members would not continue to exist without the object that contains them. For example the Sandwich class is composed of (contains, **has**) objects of both the SandwichFilling class and Bread class. Composition is known as a “***has a***” model.

Theoretically, for example, a Sandwich “*has*” (is composed of) Bread and SandwichFilling, but if there is no sandwich then there cannot be Bread and SandwichFilling. A better example is a Business and Departments. A Business has many Departments, but if there is no Business then the Deparments **cannot exist**.

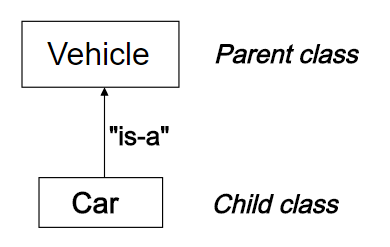
**Aggregation** is a specific type of composition in which a class contains one or more members of another class, but when the object that contains them no longer exists, those members would continue to exist. For example, if a business or department closed, the employees would continue to exist.

### Inheritance (“is a” model)

In a UML class diagram for Inheritance, it is represented using an arrow between the two classes. The following diagrams show **Inheritance**. The arrow should go from the subclass (ChildClass) to the base class (ParentClass). The arrow points to the parent. Conventionally, the parent class is placed above the child classes.



Inheritance is known as a “**is a**” model. For example, the ChildClass “is a” subclass of the ParentClass. A clearer example is the relationship between a Vehicle class and a Car class. The Vehicle is a superclass and the Car is a subclass because a Car “***is a***” type of Vehicle, but a Vehicle may not be a type of Car.



With inheritance you have a parent class and a child class: the latter is inherited from the former. The **parent class** is also called a **base class** or **superclass**. The **child class** is also called a **derived class** or **subclass**.

In Java, it’s convenient to prefer the terms superclass and subclass because the term “***super***” in superclass matches the Java call ***super()***, a special command that calls the constructor of the superclass.

## 

## 

## Extending Classes (Creating an Inheritance)

To create inheritance, use the keyword “**extends**” in the class header. For example, the following Car class header creates a superclass-subclass relationship between **Vehicle** and **Car**:

//Superclass

public class Vehicle {

private String make, String model;

private int year;

public void setName(String make){

this.make = make;

}

}

//Subclass

public class Car **extends** Vehicle{

private String VIN

}

Each Car automatically receives the data fields and methods of the superclass Vehicle; you then add new fields and methods to the newly created subclass.

public class Program {

public static void main(String[] args) {

Car car = new car();

car.setMake("Ford");

car.model = “Mustang”;

print(car.getMake());

print(car.model);

}

public static void print(Object obj){

System.out.println(obj.toString());

}

}

The car object has access to all the parent Vehicle class methods and properties, as well as its own class’s new methods and properties.

## Instanceof Operator

You can use the **instanceof** operator to determine whether an object is a member or descendant of a class. For example, using the code in the example above:

Car car = new car();

car.setMake("Ford");

car.model = “Mustang”;

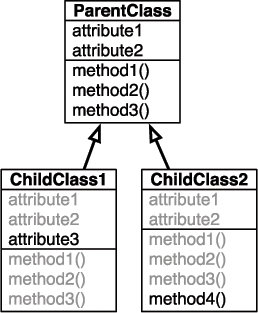
print(car instanceof Car); // TRUE

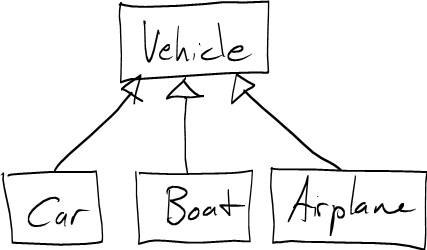
print(car instanceof Vehicle); // TRUE - because superclass

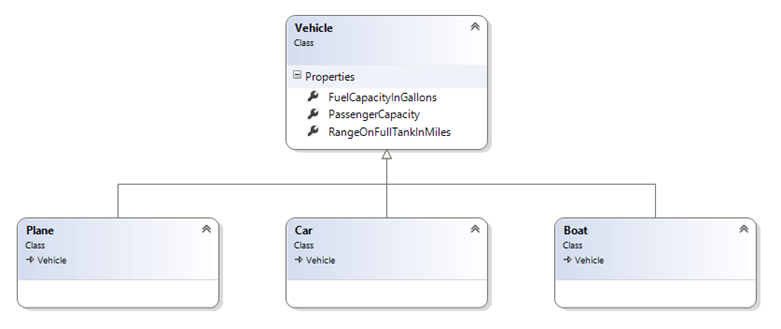
print(car instanceof Boat); // FALSE

## Multiple Inheritance

A subclass cannot inherit more than one superclass, but a superclass can be inherited by multiple subclasses.

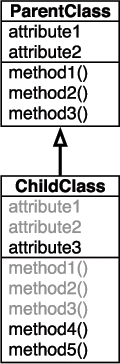






## Overriding & Polymorphism

The inherited subclass can even have its own unique qualities that the superclass doesn’t have. If you wanted to change the behavior of the superclass’s methods, you can ***override*** them in the subclass by redefining the method and its own implementation. This is ***polymorphism***, where calling the same method can have different results, depending on the object type. In other words, polyporphism is using the same method name to indicate different implementations.



The combination of a function’s name and its arguments (the number of arguments, specifically) is referred to as the function’s ***signature***.

**Override** is when a method in the subclass that has the same name and same signature as the parent class.

**Overload** is when a method that has the same name but different signature.

## 

## 

## Overriding a Superclass Method

public class Vehicle {

public int calculateSpeed(int i){

return 5 \* i;

}

}

public class Car extends Vehicle{

public int calculateSpeed(int i){

return 10 \* i;

}

}

public class Boat extends Vehicle{

@Override

public int calculateSpeed(int i){

return 2 \* i;

}

}

Use the **@Override** tag (decorator) is optional but helpful in preventing errors. It’s also used as a form of documentation.

Main Program:

Car car = new Car();

print(car.calculateSpeed(5) + “ MPH”); // 50 MPH

Boat boat = new Boat();

print(boat.calculateSpeed(5) + “ KNOTS”); // 10 KNOTS

Three types of methods that you cannot override in a subclass are:

* **static** methods
* **final** methods
* Methods within **final** classes

## Calling Default Constructors During Inheritance

Constructors are called from the Superclass down to Subclasses. When you create any subclass object, the superclass constructor **must** execute first, and then the subclass constructor executes. For example:

public class Vehicle {

public Vehicle() {

System.out.println("Vehicle superclass");

}

}

public class Car extends Vehicle{

public Car(){

System.out.println("Car subclass");

}

}

public class Boat extends Vehicle{

public ClassicCar() {

System.out.println("CalssicCar subclass");

}

}

Main Program:

ClassicCar car = new ClassicCar();

Output:

Vehicle superclass

Car subclass

CalssicCar subclass

Because every object automatically is a child of a class named **Object**, when you create parent and child classes of your own, the child classes actually use three constructors.

## Using Superclass Constructors that Require Arguments

When a superclass has a default constructor, you can create a subclass with or without its own constructor.

When you use a class as a superclass and the class has only constructors that require arguments, you must be certain that any subclasses provide the superclass constructor

with the arguments it needs.

When a superclass contains only constructors that require arguments, you must include at least one constructor for each subclass you create.

If all superclass constructors require arguments, the first statement within each subclass constructor **must** call one of the superclass constructors.

The format of the statement that calls a superclass constructor from the subclass constructor is:

super(arguments);

The keyword **super** always refers to the superclass of the class in which you use it.

public class Vehicle {

private int year;

public Vehicle(int year) {

this.year = year;

}

}

public class Car extends Vehicle{

public Car(){

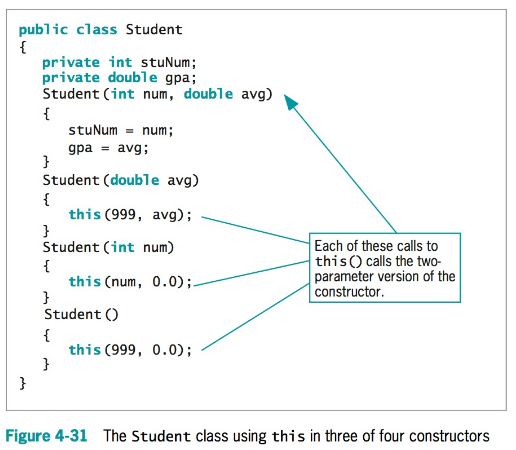
super(1990); //Must be the first statement

}

}

## super() vs this()

* **super()** is used to call the superclass constructor.
* **this()** is used to call another constructor within the same class.



## 

## 

## Accessing Superclass Methods

If a method has been overridden but you want to use the superclass version within the subclass, you can use the keyword **super** to access the parent class method.

## 

## 

## 

## 

## Protected Access Modifier (Information Hiding)

The concept of keeping data private is known as information hiding.

**Private** members of the parent class are not accessible within a child class’s methods.

When a method or a variable is marked as **protected**, it can be accessed from:

* Within the enclosing class.
* Other classes in the same package as the enclosing class.
* Sub classes, regardless of packages.

## Calling Superclass Methods

The keyword *super* can also be used to reference a method other than the constructor in the superclass. The syntax is:

public void [[arrow]]printCircle() {  
 System.out.println("The circle is created " +   
 super.getDateCreated() + " and the radius is " + radius);  
}

*Overriding Methods*

*To override a method, the method must be defined in the subclass using the same signature and the same return type as in its superclass.*

*A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as method overriding.*

1 public class CircleFromSimpleGeometricObject   
2 extends SimpleGeometricObject {  
3 // Other methods are omitted  
4   
5 // Override the toString method defined in the superclass

6 public String toString() {  
7 return super.toString() + "\nradius is " + radius;  
8 }  
9 }

## 

## 

## Overriding vs. Overloading

* *Overloading means to define multiple methods with the same name but different signatures.*
* *Overriding means to provide a new implementation for a method in the subclass.*

**

*Diagram obtained from: https://www.programcreek.com/2009/02/overriding-and-overloading-in-java-with-examples/*

*Overriding: (must have same signature)*

public class Test {  
 public static void main(String[] args) {  
 A a = new A();  
 a.p(10);   
 a.p(10.0);   
 }  
}  
   
class B {  
 public void **print**(double i) {  
 System.out.println(i \* 2);   
 }  
}  
   
class A extends B {  
 // This method overrides the method in B  
 public void **print**(double i) {  
 System.out.println(i);   
 }  
}

*Overloading: (must have different signature)*

public class Test {  
 public static void main(String[] args) {  
 A a = new A();  
 a.p(10); //calls A’s method **print(int i)**  
 a.p(10.0); //calls B’s method **print(double i)**

A.total = 12; //calls B’s data field **total**

a.p(“Hi”); //calls A’s **method** which calls B’s **method**  
 }  
}  
   
public B {

double **total**;  
 public void **print(double i)** {

total += i;  
 System.out.println(i \* 2);   
 }

public void **print(String s)** {  
 System.out.println(s);   
 }

}  
   
class A extends B {  
 // This method overloads the method in B  
 public void **print(int i)** {  
 System.out.println(i);   
 }

@Override

public void **print(String s)** {  
 System.out.println(super.p(s));   
 }  
  
}