GC01 Introductory Programming

Week 3 – Lecture

Dr Ghita Kouadri Mostefaoui
Department of Computer Science

Today's agenda

- Java inheritance
- Method overriding
- Method overloading
- Types of inheritance
- Interfaces
- The *Object* class
- Java non-access modifiers
- Class naming conventions

Inheritance

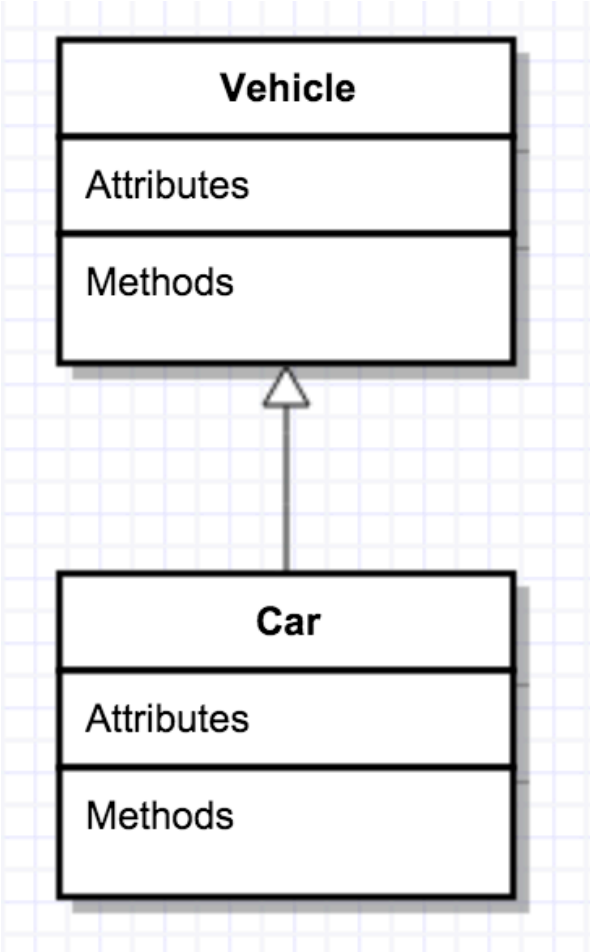
Inheritance

- Another fundamental object-oriented technique is inheritance, used to organize and create reusable classes
- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class,* or *superclass*, or *base class*
- The derived class is called the *child* class or *subclass*.
- That is, the child class *inherits the methods and data* defined for the parent class
- To tailor a derived class, the programmer can *add new variables or methods*, or *can modify the inherited ones*

Deriving Subclasses

- In Java, we use the reserved word **`extends`** to establish an inheritance relationship

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



What Member Variables Does a Subclass Inherit?

YES

- Member variables declared as `public` or `protected`.
- Member variables declared with no access modifier (`default`) as long as the subclass is in the *same package* as the superclass.

NO

- Subclasses don't inherit superclass's `private` member variables
- Subclasses don't inherit a superclass's member variable if the subclass declares a member *variable using the same name*. The subclass's member variable is said to hide the member variable in the superclass.

What Methods Does a Subclass Inherit?

YES

- Methods declared as `public` or `protected`.
- Methods declared with no access specifier (`default`) as long as the subclass is in the *same package* as the superclass.

NO

- Subclasses don't inherit the superclass's `private` methods.
- Subclasses don't inherit a superclass's method if the subclass declares a *method using the same name*. The method in the subclass is said to <u>override</u> the one in the superclass.

Private Members in a Superclass

- A subclass does not inherit the private members of its parent class. *However*, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.
- We use *setter* (mutator) and *getter* (accessor) methods
- Setters and getters enforce *encapsulation*. - this allows additional functionality (like validation) to be added more easily later.

```java
public class Bicycle {

    // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed;
    // declaring a private member
    private String model;


    // Constructors
    // methods
```

...

```java
    /**
     * @return the model
     */
    public String getModel() {
            return model;
    }

    /**
     * @param model the model to set
     */
    public void setModel(String model) {
            this.model = model;
    }

}
```

Does a subclass inherit its parent constructors?

- Answer is: No
- Constructors are not inherited, even though they have public visibility
- Yet we often want to use the parent's constructor to set up the "parent's part" of the object
- The `super` reference can be used to refer to the parent class, and is often used to invoke the parent's constructor

The `super` reference

```java
public class Bicycle {

    // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int
startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }

}
```

```java
public class MountainBike extends Bicycle {

    // the MountainBike subclass adds one field
    public int seatHeight;

    // the MountainBike subclass has one
constructor
    public MountainBike(int startHeight,
                        int startCadence,
                        int startSpeed,
                        int startGear) {
        super(startCadence, startSpeed,
              startGear);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

The `super` reference

- `super` is used to invoke immediate parent class constructor.

- `super` is used to refer immediate parent class instance variable.

- `super` is used to invoke immediate parent class method.

The `super` reference

```java
public class Bicycle {

    // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int
startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }

}
```

```java
public class MountainBike extends Bicycle {

    // the MountainBike subclass adds one field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight,
                        int startCadence,
                        int startSpeed,
                        int startGear) {
        super(startCadence, startSpeed,
              startGear);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }

   // referring to immediate parent
   // class instance variable
    public void display() {
        System.out.println(super.speed);
    }

}
```

The `super` reference

- `super` is used to invoke immediate parent class constructor.

- `super` is used to refer immediate parent class instance variable.

- `super` is used to invoke immediate parent class method.

The `super` reference

```java
public class Bicycle {

    // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int
startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }

}
```

```java
public class MountainBike extends Bicycle {

    // the MountainBike subclass adds one field
    public int seatHeight;

    // the MountainBike subclass has one
constructor
    public MountainBike(int startHeight,
                        int startCadence,
                        int startSpeed,
                        int startGear) {
        super(startCadence, startSpeed,
              startGear);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }

    // invoking immediate parent class method
    public void increaseSpeed(int increment){
      super.speedUp(increment*2);
    }
}
```

Controlling access to members of a class: REMINDER

An access modifier determines whether other classes can use a particular *member variable* or call a particular *method.*

- **public**
  - Accessible to everyone, both designer and users of the class
- **private**
  - Accessible only within class implementation, only class designer
- **protected**
  - Accessible to the package and all subclasses (classes that are designed based on this one)

- default (when you list no keyword)
  - Accessible to the package.

Java access modifiers: REMINDER

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N

The `protected` Modifier

- Visibility modifiers determine which class members are inherited and which are not
- Variables and methods declared with `public` visibility are inherited; those with `private` visibility are not
- There is a third visibility modifier that helps in inheritance situations: `protected`

Method Overriding

Method Overriding

What is overriding?

- The child class provides alternative implementation for parent class method.
- The key benefit of overriding is the ability to *define behavior that's specific to a particular subclass type*.
- Overridden method: In the superclass.
- Overriding method: In the subclass.

Method Overriding

```
public class Plane
{
   public void maxSpeed()
   {
       System.out.println("Max speed is 570 mph");
   }
}


Class SRBlackbird extends Plane
{
   public void maxSpeed()
   {
       System.out.println("Max speed is 1000 mph");
   }

}
```

```java
public class PlaneTester {

    public static void main(String arg[])
    {
        SRBlackbird sr =new SRBlackbird();

        sr.maxSpeed();
    }

}
```

Output:

Max speed is ? mph

Rules for Method Overriding

EXACT argument list matching.

```
public class Plane
{
    public void maxSpeed(int speed)
    {
        System.out.println("Max speed is " +speed+ " mph");
    }
}

class SRBlackbird extends Plane
{
    public void maxSpeed(double speed)// This is NOT an
                                      // overriding
    {
        System.out.println("Max speed is " +speed+ " mph");
    }

}
```

Rules for Method Overriding

The return type must be the SAME as, OR a SUBTYPE of, the return type declared in the original overridden method in the superclass.

```
class Alpha
{
    Alpha doStuff(char c)
    {
        return new Alpha();
    }
}

class Beta extends Alpha
{
    Beta doStuff(char c)
    {
        return new Beta();// legal override
    }
}
```

Rules for Method Overriding

The access level can't be more restrictive than the overridden method's.

```
public class Plane
{
   public void maxSpeed()
   {
       System.out.println("Max speed is 570 mph");
   }
}

class SRBlackbird extends Plane
{
   private void maxSpeed() // This is NOT an
                           // overriding

   {
       System.out.println("Max speed is 1000 mph");
   }
}
```

Rules for Method Overriding

The access level CAN be less restrictive than that of the overridden method.

```
public class Plane
{
    protected void maxSpeed()
    {
        System.out.println("Max speed is 570 mph");
    }
}

class SRBlackbird extends Plane
{
    public void maxSpeed() // Legal override
    {
        System.out.println("Max speed is 1000 mph");
    }

}
```

Rules for Method Overriding

Possible only through inheritance.

```
Public class Plane
{
   protected void maxSpeed()
   {
       System.out.println("Max speed is 570 mph");
   }
}


class SRBlackbird
{
   public void maxSpeed() // This is NOT an
                          // overriding

   {
       System.out.println("Max speed is 1000 mph");
   }

}
```

Rules for Method Overriding

- You cannot override a method marked `final`.
- You cannot override a method marked `static`.
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code.

Invoking a Superclass Version of an Overridden Method

- Use the code in the overridden method.
- Also add extra features in overriding method.

```
public class Plane
{
    protected void maxSpeed()
    {
        System.out.println("Max speed is 570 mph");
    }
}

class SRBlackbird extends Plane
{
    public void maxSpeed()
    {
        super.maxSpeed();

        System.out.println("I can race");
    }

}
```

Method Overloading

Method Overloading

- *Same name, different arguments*.
- It CAN have a different return type.
- Argument list MUST be different.
- Access modifier CAN be different.
- A method can be overloaded in the *same class* or in a *subclass*.

1. Overloading in Same Class

```
public class Sort
{
    public void sortItems(int[] a)
    {
        //Program to sort integers
        System.out.println("Sorting INTEGERS");
    }

    public void sortItems(String[] a)
    {
      //Program to sort Strings
        System.out.println("Sorting STRINGS");

    }
}
```

1. Overloading in Same Class (cont.)

```
public class SortTester
{

    public static void main(String[] args)
    {
        int[] a={3,8,6,1,2};
        String[] s={"Sachin","Sourav","David"};

        Sort st = new Sort();
        st.sortItems(a);
        st.sortItems(s);
    }
}
```

2. Overloading in Subclass

```
public class Sort
{
    public void sortItems(int[] a )
    {
        //Program to sort integers
        System.out.println("Sorting INTEGERS");
    }
}

class DoubleSort extends Sort
{
    public void sortItems(double[] a )
    {
        //Program to sort doubles
        System.out.println("Sorting DOUBLES");

    }
}
```

2. Overloading in Subclass (cont.)

```
public class SortTester2
{
    public static void main(String[] args)
    {
        int[] a={3,8,6,1,2};
        double[] f={3.5,6.8,1.4,67.9};

        Sort s = new Sort();
        s.sortItems(a);

        DoubleSort s2=new DoubleSort();
        s2.sortItems(f);
    }
}
```

Methods That Are Both Overloaded and Overridden

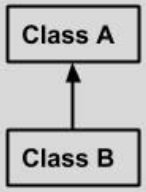
```
public class Animal
{
    public void eat()
    {
        System.out.println("Generic Animal Eating Generically");
    }
}
class Horse extends Animal
{
    public void eat()
    {
        System.out.println("Horse eating hay ");
    }
    public void eat(String s)
    {
        System.out.println("Horse eating " + s);
    }
}
```

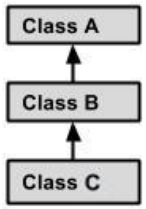
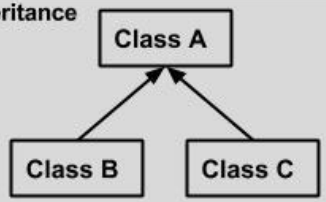
Overriding vs. Overloading

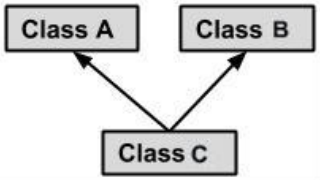
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overriding lets you define a similar operation in different ways for different object types

- Overloading deals with multiple methods in the same class or class/ subclass with the same name but different signatures
- Overloading lets you define a similar operation in different ways for different data

Types of inheritance

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

Multiple Inheritance

- Java supports *single inheritance*, meaning that a derived class can have only one parent class
- *Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents
- *Java does not support multiple inheritance*
  - *Why not multiple inheritance? By Tony Sintes*
    - *http://www.javaworld.com/article/2077394/core-java/why-not-multiple-inheritance.html*
- In most cases, the use of *interfaces* gives us aspects of multiple inheritance without the overhead

Interfaces

Interfaces

- An `interface` is a way to describe what classes should do, without specifying how they should do it. It's not a class but a set of requirements for classes that want to conform to the interface
- An *`interface`* declares (describes) methods but does not supply bodies for them

```
public interface Comparable
{
    int compareTo(Object otherObject);
}
```

- This requires that any class implementing the `Comparable` interface contains a `compareTo` method.

Interfaces

```
interface KeyListener {
   public void keyPressed(KeyEvent e);
   public void keyReleased(KeyEvent e);
   public void keyTyped(KeyEvent e);
}
```

- All the methods are implicitly **public**
- You CANNOT instantiate an interface
  - *none* of its methods are defined
- An interface may also contain constants (final static fields).

Implementing interfaces in a class (1)

Two steps to make a class implement an interface:

1. declare that the class intends to implement the given interface by using the `implements` keyword

```
class Robot implements KeyListener { . . . }
```

2. supply definitions for ***all*** methods in the interface

```
public void keyPressed (KeyEvent e) {
          System.out.println("A key has been pressed");
       }
public void keyReleased (KeyEvent e) {…}
public void keyTyped (KeyEvent e) {…}
```

Implementing interfaces in a class (2)

- A single class can implement multiple interfaces. Just separate the interface names by comma

```
class Employee implements Comparable, Cloneable {. . .}
```

Partially implementing an Interface

- It is possible to define some but not all of the methods defined in an interface:

```
abstract class MyKeyListener implements KeyListener{
      public void keyTyped(KeyEvent e) {...};
   }
```

- Since this class does not supply all the methods it has promised, it is an abstract class
- You must label it as such with the keyword `abstract`
- You can even *extend* an interface (to add methods):

```
interface FunkyKeyListener extends KeyListener { ... }
```

Interfaces, again

- When you implement an interface, you promise to define *all* the functions it declares

- There can be a *lot* of methods

```
interface KeyListener {
   public void keyPressed(KeyEvent e);
   public void keyReleased(KeyEvent e);
   public void keyTyped(KeyEvent e);
}
```

- What if you only care about a couple of these methods?

Adapter classes

- **<u>Solution</u>**: use an adapter class

- An `adapter class` implements an interface and provides empty method bodies

```
class  KeyAdapter implements KeyListener {

    public void keyPressed(KeyEvent e) { };
    public void keyReleased(KeyEvent e) { };
    public void keyTyped(KeyEvent e) { };
}
```

- You can override only the methods you care about
- This isn't elegant, but it does work
- Java provides a number of adapter classes

Implementing an Interface vs. Subclassing

- You extend a class, but you *implement* an interface

- A class can only *extend (subclass) one other class*, but it can *implement as many interfaces* as you like
  - This lets the class fill multiple "roles"

Class Hierarchies

- Two children of the same parent are called *siblings*
- *Common features* should be put as high in the hierarchy as is reasonable
- An inherited member is passed continually down the line
- Therefore, a child class inherits from all its ancestor classes

The *`Object`* Class

The `Object` Class

- A class called `Object` is defined in the `java.lang` package of the Java standard class library

- All classes are derived from the `Object` class
  - even if a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class
  - the `Object` class is therefore the ultimate root of all class hierarchies

- The `Object` class contains a few useful methods, which are inherited by all classes
  - `toString()`
  - `equals()`
  - `clone()`

The `Object` Class: the `toString` Method

- That's why the `println` method can call `toString` for any object that is passed to it – all objects are guaranteed to have a `toString` method via inheritance

- The `toString` method in the `Object` class is defined to return a string that contains the name of the object's class and a hash value

- Every time we have defined `toString`, we have actually been *overriding it*

- Correctly overridden `toString` method can help in logging and debugging of Java.

Java non-access modifiers

Java provides a number of non-access modifiers to achieve many other functionality.

- The **static** modifier for creating class methods and variables,
- The **final** modifier for finalizing the implementations of classes, methods, and variables.
- The **abstract** modifier for creating abstract classes and methods.
- The **synchronized** and **volatile** modifiers, which are used for threads. (not covered in this module)

`Static` modifier

The `static` modifier

- Can be applied to member variables and methods
- The static keyword is used when a *member variable of a class* has to be shared between all the instances of the class.
- All static variables and methods *belong to the class* and not to any instance of the class
- *Note*: Static keyword can also be used with specific type of classes (nested ones).

Example

```
class StaticDemo{
      public static int a = 100; // All instances of staticDemo have this variable as a common `
                                 //variable
      public int b =2 ;

      public static void showA(){

            System.out.println("A is " + a);
      }
}

class ExecClass{

      public static void main(String args[]){

            StaticDemo.a = 35; // when we use the class name, the class is loaded, direct access to
                               //a without any instance
            //StaticDemo.b=22; // ERROR this is not valid for non static variables

            StaticDemo demo = new StaticDemo();

            demo.b = 200;  // valid to set a value for a non static variable after creating an
                              instance.

            StaticDemo.showA();  //prints 35
      }
}
```

Why do we need this

- Static methods are identified to be mostly used when we are writing any utility methods.

```
Math.abs(); Math.random();
System.out.println(); Integer.parseInt();
main();
```

- We can also use static variables when sharing data.

Static vs. Non-static

- We can access *static* variables without creating an instance of the class
- As they are already available at class loading time, we can use them in any of our non static methods.
- We cannot use *non static* methods and variables without creating an instance of the class as they are bound to the instance of the class.
- They are initialized by the constructor when we create the object using `new` operator.

`Final` modifier

The `final` modifier

- If a *class* declared with the final modifier, then *it cannot be subclassed (inherited).*

- If a *field* is declared with final, then the value of it *cannot be changed*.

- If a *method* is declared with final, then it *cannot be overridden* by subclasses

`Abstract` modifier

The `abstract` modifier

- Abstract classes *cannot be instantiated*, but they *can be subclassed*.

- An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

- If a class includes abstract methods, then the class itself must be declared abstract, as in:

```
public abstract class GraphicObject {
        // declare fields
        // declare non-abstract methods
        abstract void draw();
    }
```

- When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.

Further Reading

- http://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html
- https://docs.oracle.com/javase/tutorial/java/IandI/final.html
- https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

Class Naming Conventions

- **Advantages**: By using standard Java naming conventions, you make your code easier to read for yourself and for other programmers.

- *Readability* of Java program is very important. It indicates that less time is spent to figure out what the code does.

- This reduces the time/cost needed for code *maintainability*

Class Naming Conventions

Name	Convention
class name	should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc.
interface name	should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc.
method name	should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc.
variable name	should start with lowercase letter e.g. firstName, orderNumber etc.
package name	should be in lowercase letter e.g. java, lang, sql, util etc.
constants name	should be in uppercase letter. e.g. RED, YELLOW, MAX_PRIORITY etc.

Java follows CamelCase syntax for naming the class, interface, method and variable.

If name is combined with two words, second word will start with uppercase letter always e.g. actionPerformed(), firstName, ActionEvent, ActionListener etc.