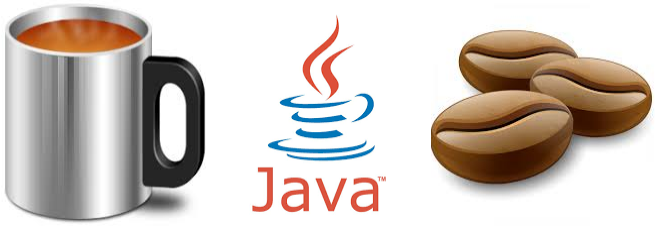
CSC3301 Java Workshop



Mohammed Hassan

mhassan.se@buk.edu.ng

Inheritance and Composition

An Example of Inheritance

```
public class Bicycle {        // the Bicycle class has three fields
    protected int cadence;
    protected int gear;
    protected 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;
    }
}
```

An Example of Inheritance

- A class declaration for a MountainBike class that is a subclass of Bicycle might look like this:

```
public class MountainBike extends Bicycle {
                    // the MountainBike subclass adds one field
    protected 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;
    }
}
```

- MountainBike inherits all the fields and methods of Bicycle and adds the field *seatHeight* and a method to set it. a new MountainBike class has four fields and five methods.

What You Can Do in a Subclass

- You can use the inherited members as is, replace them, hide them, or supplement them with new members:
  - The inherited fields can be used directly, just like any other fields.
  - You can declare new fields in the subclass that are not in the superclass.
  - The inherited methods can be used directly as they are.
  - You can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus ***overriding*** it.
  - You can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus *hiding* it.
  - You can declare new methods in the subclass that are not in the superclass.
  - You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

Private Members in a Superclass

- A subclass has no access to a private field or method of its superclass.
- If the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

```
class AA {
    private int aak;
    protected float aaf;
    public  setAAK(int aak) {
            this.aak = aak;
    }
}
class BB extends AA {
    private int bbk;
    BB() {
        aak = 5;            // error
        setAAK(5);          // correct
        aaf = 0F;
        bbk = 4;
    }
}
```

```
class AA {
    public AA(int i) { … }
    private AA(float f) { … }

    private void m1() { … }
    public void m2() { … }
}

class BB extends AA {
    BB() {
        super(5.0F);        // error
    }
    BB(int i,float f) {
        super(i);           // correct
        m1();               // error
        m2();               // correct
    }
}
```

Casting Objects

- We have seen that an object is of the data type of the class from which it was instantiated:

  MountainBike myBike = new MountainBike();

- myBike is of type MountainBike in the example.
- MountainBike is descended from Bicycle and Object. Therefore, a MountainBike is a Bicycle and is also an Object, and it can be used wherever Bicycle or Object objects are called for.
- The reverse is not necessarily true: a Bicycle *may be* a MountainBike, but it **isn't** necessarily. Similarly, an Object *may be* a Bicycle or a MountainBike, but it **isn't** necessarily.

Casting Objects

- *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 MountainBike();
```

- obj is both an *Object* and a *Mountainbike* (until such time as obj is assigned another object that is *not a Mountainbike*). This is called *implicit casting*.
- If, on the other hand, we write:

```
MountainBike  myBike = obj;      // error
```

- we would get a compile-time error because *obj* is not known to the compiler to be a *MountainBike.*

Casting Objects

- We can *tell* the compiler that we promise to assign a MountainBike to obj by *explicit casting:*

```
MountainBike myBike = (MountainBike)obj;
```

- This cast inserts a runtime check that *obj* is assigned a *MountainBike* so that the compiler can safely assume that *obj* is a *MountainBike*. If *obj* is not a *Mountainbike* at runtime, an exception will be thrown.
- To avoid run-time errors, use the *instanceof*:

```
if (obj instanceof MountainBike) {
        MountainBike myBike = (MountainBike)obj;
}
```

  - This code verifies that *obj* refers to a *MountainBike* so that we can make the cast with knowledge that there will be no runtime exception thrown.

Overriding Instance Methods

- *Overriding* means that a subclass redefines a method from a superclass when:
  - Both methods have the same signature;
  - Both methods have the same return type.
- A *covariant* return type – an overriding method can also return a subtype of the type returned by the overridden method.
- By using the keyword *super*, the overridden method can be invoked.

```
class AA {

    Object method(int i) {
        Object oo;
                …
        return oo;
    }       // end of the method
}           // end of the AA class

class BB extends AA {
    String method(int k) {
        String os;
        Object oo = super.method(5);
                …
        return os;
    }       // end of the method
}           // end of the BB class

String os = new BB().method(4);
```

Overriding Class Methods

- If a subclass defines a class method with the same signature as its superclass, the subclass' method *hide*s the superclass' method.
- The distinction between hiding and overriding is important when invoking:
  - The subclass version of an overridden method gets invoked.
  - The version that gets invoked depends on the namespace from which it is invoked.

Example: Overriding and Hiding Methods

```
class Animal {      //  the file name: Cat.java
   public static void testClassMethod() {
      System.out.println("The class method in Animal.");
   }
   public void testInstanceMethod() {
      System.out.println("The instance method in Animal.");
   }
} // end of the Animal class
public class Cat extends Animal {
   public static void testClassMethod() {
      System.out.println("The class method in Cat.");
   }
   public void testInstanceMethod() {
      System.out.println("The instance method in Cat.");
   }
   public static void main(String[] args) {
      Cat myCat = new Cat();
      Animal myAnimal = myCat;
      myAnimal.testClassMethod();
      myAnimal.testInstanceMethod();
   }
} // end of the Cat class
```

- Compile and run:
  - Save the text to the file: Cat.java
  - Compile the program typing: javac Cat.java
  - Run the program typing: java Cat
  - Output of the program:

```
The class method in Animal.
The instance method in Cat.
```

Comments on the Previous Slide

- The Cat class overrides the instance method in Animal and hides the class method in Animal.
- The main method in this class creates an instance of Cat and calls testClassMethod() on the class and testInstanceMethod() on the instance.
- The version of the hidden method that gets invoked is the one in the superclass, and the version of the overridden method that gets invoked is the one in the subclass.

Example: Overriding Methods

```
class AA {
    void insMethod() { … }
}

class BB extends AA {
    void insMethod() { … }
}

AA oa = new AA();
oa.insMethod();          // AA

BB ob = new BB();
ob.insMethod();          // BB

oa = ob;
oa.insMethod()           // BB
```

```
class AA {
    static void stcMethod() { … }
}

class BB extends AA {
    static void stcMethod() { … }
}

AA.stcMethod();
BB.stcMethod();

AA oa = new AA();
oa.stcMethod();          // AA

oa = new BB();
oa.stcMethod();          // AA
```

Overriding Methods: Summary

- A subclass can redefine the methods it inherits from its superclass:
  - Overriding instance methods
  - Hiding class methods
- Defining a method with the same signature:

	Superclass instance methods	Superclass static methods
Subclass instance methods	Overrides	Generates a compile-time error
Subclass static methods	Generates a compile-time error	Hides

Hiding Fields

- A subclass field that has the same name as a superclass field *hide*s the superclass' field.
- Use the keyword *super* to access a hidden field of the superclass.
- Avoid hiding fields: It makes code difficult to read.

```
class AA {
    int field1;
    int field2;

}

class BB extends AA {
    int field1;

    void method(){
        field1 = 0;
        super.field1 = 2;
        field2 = 4;
    }

}
```

Accessing Superclass Members

```
class Father{  // the file name Son.java
   public void printMethod(){
      System.out.println("Printed in Father class.");
   }
}          // end of the Father class
public class Son extends Father{
          //overrides printMethod in Father class
   public void printMethod(){
      super.printMethod();
      System.out.println("Printed in Son class");
   }
   public static void main(String[] args){
   Son s = new Son();
   s.printMethod();
   }
}          // end of the Son class
```

- Compile and run:
  - Save the text to the file: Son.java
  - Compile the program typing: javac Son.java
  - Run the program typing: java Son
  - Output of the program:

```
Printed in Father class
Printed in Son class
```

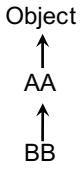
If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword super.

Example: *super* and Members

```
public class AA {
    private int field1;
    protected int field2;
}

public class BB extends AA {
    private int field1;

    void method() {
        field1 = 0;
        super.field1 = 2;      // error
        field2 = 4;
    }
}
```




```
class AA extends Object {

    public String toString() {
        String s = super.toString();

        return "AA: " + s;
    }

}

class BB extends AA {

    public String toString() {
        String s = super.toString();

        return "BB: " + s;
    }

}
```

super and Constructors

- MountainBike is a subclass of Bicycle. Here is the MountainBike (subclass) **constructor** that calls the superclass **constructor** and then adds initialization code of its own:

```
public MountainBike(int startHeight, int startCadence, int
    startSpeed, int startGear) {
      super(startCadence, startSpeed, startGear);
      seatHeight = startHeight;
  }
```

- Invocation of a superclass constructor **must be the first** line in the subclass constructor:

```
super();              // the superclass no-argument constructor is called
--or--
super(parameter list); // the superclass constructor with a matching
                       // parameter list is called.
```

super and Constructors

- If a constructor **does not** explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the **no-argument** constructor of the superclass.
- If the super class **does not** have a no-argument constructor, you will get a compile-time error.
  - A no-argument constructor is created **automatically** by Java, if there are **no any** constructors for the class provided. In other case, the no-argument constructor has to be defined explicitly.
- The *Object* class does have such a constructor, so if *Object* is the only superclass, there is **no problem**.
- If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that there will be a whole **chain** of constructors called, all the way back to the constructor of *Object*.
  - It is called *constructor chaining*, and you need to be aware of it when there is a long line of **class descent**.

Constructor Chaining

```
// File: Cartoon.java
class Art {
        Art() {
                System.out.println("Art constructor");
        }
}   // end of Art class
class Drawing extends Art {
        Drawing() {
          System.out.println("Drawing constructor");
        }
} // end of the Drawing class
public class Cartoon extends Drawing {
        Cartoon(){
          System.out.println("Cartoon constructor");
        }
        public static void main(String[] args){
                Cartoon x = new Cartoon();
        }
} // end of the Cartoon class
```

- Compile and run:
  - Save the text to the file: Cartoon.java
  - Compile the program typing: javac Cartoon..java
  - Run the program typing: java Cartoon.
  - Output of the program:

```
Art constructor
Drawing constructor
Cartoon constructor
```

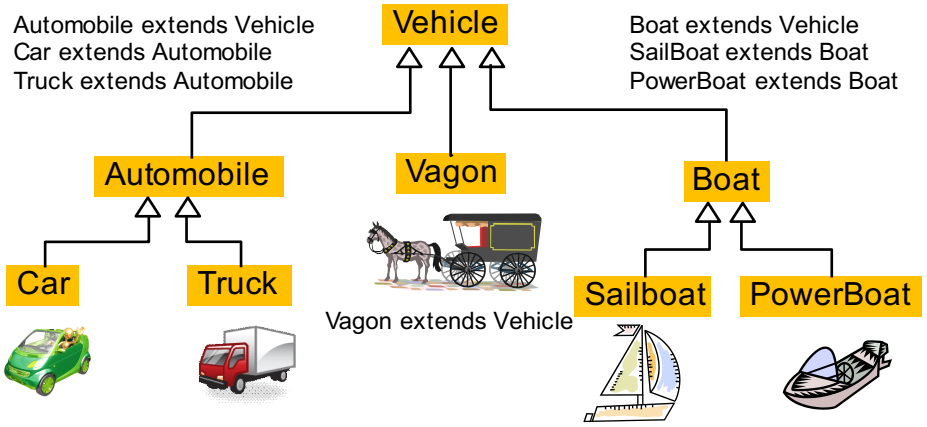
Summary

- The *Object* class is the root (or top) of any class hierarchy in Java.
- All other classes are inherited from *Object*, either directly or indirectly.
- A class inherits fields and methods from all its *superclasses*.
- A *subclass* may:
  - Override accessible inherited methods
  - Hide accessible fields or methods

Contents

- Inheritance
  - IS-A versus HAS-A Relations
  - The Object Class as a Superclass
- Debugging
  - Key Definitions
  - General strategy
  - jdb (a Command Line Debugger)

Hierarchy of Classes: IS-A Relationship



Sailboat s = new Sailboat();
Boat b = new Sailboat();
Vehicle v = new Sailboat();
Boat b1 = new Vehicle(); // error
// Vehicle IS NOT ALWAYS a Boat

Sailboat IS-A Boat (ALWAYS)
Boat IS- A Vehicle (ALWAYS) and
SailBoat IS-A Vehicle (ALWAYS)
A Sailboat can do anything
A Vehicle can do.

Reusing Classes

- **Inheritance**: A new class is created as a *type of* an existing class. You take the form of the existing class and add code to it **without** modifying the existing class. The compiler does most of the work.
  - **IS-A** relationship between classes.
- **Composition**: A new class is composed of objects of existing classes. You reuse the functionality of the code, not its form.
  - **HAS-A** relationship between classes.

Example: Composition (HAS-A Relationship)

```
class Engine {
  public void start() {}
  public void rev() {}
  public void stop() {}
}  // end of the Engine class
class Wheel {
  public void inflate(int psi) {}
} // end of the Wheel class
class Window {
  public void rollup() {}
  public void rolldown() {}
} // end of the Window class
class Door {
  protected Window window =
                          new Window();
  public void open() {}
  public void close() {}
} // end of the Door class
```

```
public class Car {
  protected Engine engine = new Engine();
  protected Wheel[] wheel = new Wheel[4];
  protected Door
    left = new Door(),    // first door
    right = new Door(); // 2-door
  public Car() { // constructor
    for(int i = 0; i < 4; i++)
      wheel[i] = new Wheel();
  } // end of the constructor
  public static void main(String[] args) {
    Car car = new Car();
    car.left.window.rollup();
    car.wheel[0].inflate(72);
  } // end of the main method
} // end of the Car class
```

Comments on the Previous slide

- We have classes: *Engine*, *Wheel*, *Window*, *Door*, and *Car*.
- The *Door* class is **composed** of the object of class *Window*.
- The *Car* class is composed of the objects of classes *Engine*, four *Wheels*, two *Doors*.

Car HAS-A Engine
Car HAS-A Wheel
Car HAS-A Door

Door HAS-A Window

The Object Class as a Superclass

- The *Object* class, in the java.lang package, is the root of the class hierarchy tree.
- Every class inherits the instance *methods* of *Object*.
- The methods defined by *Object* are:
  - clone – creates and returns a copy of itself;
  - equals – checks whether another object is equal to this one;
  - getClass – returns the runtime class of an object;
  - toString – returns a string representation of the object.

The *equals* Method

- This method compares 2 objects for equality and returns *true* if they are equal.
- The implementation by *Object* tests whether the references are equal, i.e., if it is the same object:

```
public boolean equals(final Object obj) {
   return obj == this;
}
```

The *equals* Method: Example 1

```
class Book {
  private int price;
  private String ISBN;
  public Book(int price,
              String ISBN) {
              this.price = price;
              this.ISBN = ISBN;
  }
  public int getPrice() {
              return price;
  }
  public getISBN() {
              return ISBN;
  }
}
```

```
Book firstBook  = new Book(1250, "0201914670");
Book secondBook = new Book(1250, "0201914670");
Book thirdBook = secondBook;
if (firstBook.equals(secondBook)) {
   System.out.println("objects  1 and 2 are equal");
} else {
   System.out.println("objects 1 and 2 are not equal");
}
if (thirdBook.equals(secondBook)) {
   System.out.println("objects 2 and 3 are equal");
} else {
   System.out.println("objects 2 and 3 are not equal");
}
```

OUTPUT:
objects 1 and 2 are not equal
objects 2 and 3 are equal

- secondBook and thirdBook are two names for the same object
- Values of firstBook and secondBook are different references.

The *equals* Method

- To test in the sense of equivalency (containing the same information) each class must override the equal() method.

The *equals* Method: Example 2

```
class Book {
    private int price;
    private String ISBN;
    piblic Book(int price, String ISBN) {
            this.price = price;
            this.ISBN = ISBN;
    }
    public int getPrice() {     return price;
    }
    public String getISBN() {   return ISBN;
    }
    public boolean equals(Object obj) {
            if (obj == null)
                    return false;
            else if (super.equals(obj))  // equal references → this == obj
                    return true;
            else if (getClass() == obj.getClass()) {     // equivalent objects
                    Book oa = (Book)obj;
                    return oa.getPrice() == price && oa.getISBN().equals(ISBN);
            }
            else       return false;
    } // end of the equals method
} // end of the Book class
```

The *equals* Method: Example 2

```
Book firstBook  = new Book(1250, "0201914670");
Book secondBook = new Book(1250, "0201914670");
if (firstBook.equals(secondBook)) {
    System.out.println("objects are equal");
} else {
    System.out.println("objects are not equal");
}
```

- This program displays objects are equal even though firstBook and secondBook reference two distinct objects. They are considered equal because the objects compared **contain the same ISBN number and the same price**.

The *getClass* Method

- getClass returns a *Class* object which stores information about the class.
- getClass is a final method.
- *java.lang.Class* defines these methods:
  - getName – returns the (class) name
  - getFields – returns all the public fields
  - getMethods – returns all the public methods
  - getPackage – returns the class' package
  - getSuperclass – returns the class' superclass
  - getConstructors – returns all the public constructors

Example: getClass

```
class AA {
    public int aak;

    public AA(int k) {
        aak = k;
    }

}

final AA oa = new AA(5);
Class oc = oa.getClass();
String ocname = oc.getName();          // → "AA"

final Class sc = oa.getSuperclass();
String scname = sc.getName();          // → "Object"
```

The *toString* Method

- A class may override the toString method.
- The Object's toString method produces output that is useful for debugging.
- The string representation of an object depends on the information (i.e., state) it stores.
  - See an example of the Bicycle class, Lecture 1. Here is a toString method for that class:

```
public String toString () {
        return getclass().getName() +  "cadence:" +  cadence+ " speed:"+speed+
                " gear:"+gear;
}
```

The *final* Keyword

- A **final** method cannot be overridden by a subclass, for example:
  - final void method() { … }
- Final methods protect the behavior that is critical to the consistent state of the object
- An entire class can be declared final to prevent the class from being subclassed:
  - public final class String { … }
  - public final class Class { … }

Example: *final* Method and Class

```
public class AA {
    private int aak;

    final void method(){
        …
    }

}

class BB extends AA {

    void method(){ … }

}
```

```
public final class AA {
    private int aak;

    void method(){
        …
    }

}

class BB extends AA {

}
```

Example: *final* Fields

```
public class AA {
   final int fi = 0; // initialized

   AA() {
        fi = 3;      // error
   }

   void method(){
        fi = 3;     // error
   }

}
```

```
public class AA {
   final int fi; // not initialized

   AA() {
        fi = 3; // initialized
   }

   void method(){
        fi = 5;     // error
   }

}
```

Example: *final* Variables

```
public class AA {

    void method() {
        final int k;

        k = 3;
        k = 5; // error
    }

}
```

```
public class AA {

    void method() {
        final int k = 3;

        k = 5;  // error
    }

}
```

Example: *final* Parameters

```
public class AA {

    Object aao;

    void mt(Object arg) {
        aao = arg;

        arg = null;
    }

}
```

```
public class AA {

    Object aao;

    void mt(final Object arg) {
        aao = arg;

        arg = null;    // error
    }

}
```

Summary

- **IS-A** and **HAS-A** are different relations between classes.
- The ***Object*** class is the top of the class hierarchy.
  - Useful methods inherited from *Object* include toString(), equals(), and getClass().
- A final class cannot be extended.
- A final method cannot be overridden.
- A final field or variable, once initialized, cannot change its value.

Debugging: Assignment

- Key Definitions
- Common Bugs
- General Advice:
  - Syntactical Errors
  - Run-time and Logic Errors
- Strategies
  - A Test Case
  - Print Statements
  - Assert Statements
  - jdb Debugger