Java Programming I

CHAPTER 5

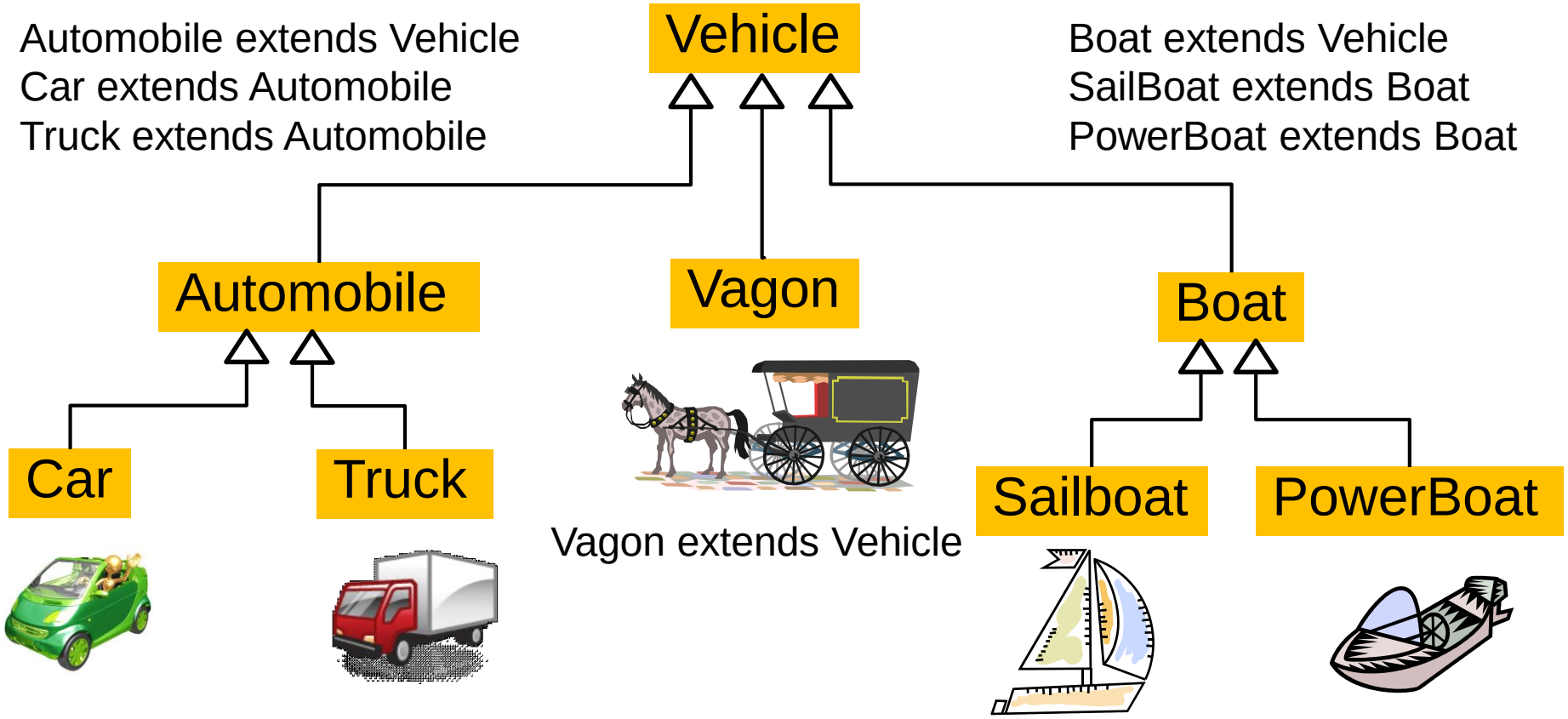
Inheritance
Part 2

Debugging

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.

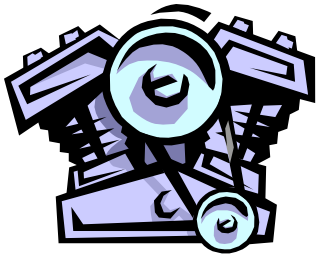
Example: Composition (HAS-A Relationship)

```
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
```

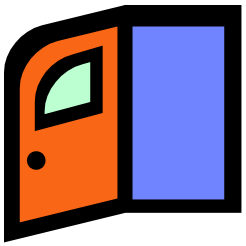
```
public class Car {
  protected Engine engine = new Engine();
  protected Wheel[] wheel = new Wheel[4];
  protected Door
    left = new Door(),   // first door
    right = new Door(); // 2-door
  public Car() { // constructor
    for(int i = 0; i < 4; i++)
      wheel[i] = new Wheel();
  } // end 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 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: Key Definitions

- Bug: An error in a program.
- Testing: A process of analyzing, running a program, looking for bugs.
- Test case: A set of input values, together with the expected output.
- Debugging: A process of finding a bug and removing it.
- Exception: An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.
  - When an error occurs, like dividion by 0, Java throws an exception. A lot (generally too much) information is provided.

Common Bugs

- **Compilation or syntactical errors** are the first that you will encounter and the easiest to debug. They are usually the result of typing errors.
- **Run-time errors** occur during the execution of the program and typically generate Java exceptions.
- **Logic errors** are different from run-time errors because there are no exceptions thrown, but the output still does not appear as it should. These errors can range from buffer overflows to memory leaks.
- **Threading errors** are the most difficult to replicate and track down.

General Advice: Syntactical Errors

- Pay attention to the first error message.
- If the number of errors is large, use the following command to redirect errors to the file:
  - The C shell

    ```
    javac  name.java  >&  error.txt
    ```

  - The Borne shell and the Korn shell

    ```
    javac  name.java  2>  error.txt
    ```

General Advice: Run-time and Logic Errors

- When an error occurs, you have to play detective and find it. That process is called debugging.
- The place where the bug is may be far removed from the place where an error is revealed.

General Advice: Run-time and Logic Errors

- Strategy 1: Find a simplest possible test case that exhibits the error.
- Strategy 2: put print statements, suitably annotated, at the carefully chosen places in the program.
- Strategy 3: Use Java assert-statements at good places:
  - **assert** <boolean expression> ;
- Strategy 4: Use the debugging feature of Java.

Strategy 1: A Test Case

- You should create a set of input values, together with the expected output.
- This strategy works in combination with strategy 2, 3 and 4.

Strategy 2: Print Statements

- It is important to make enough information visible but not too much. It can help highlight important information in various ways so that it stands out from the output.

```
void m1(int a, float b) {
   System.out.println(
   "**** Start of m1 method, arguments " + a + ", " + b );
. . .
}
```

Strategy 3: Assert Statements

- Assertions state that you expect some condition to be true at some point in your program. If that condition is not true at that point, then you want to be told about it.

```
public class AssertClass { // two numbers from  command line
   public static void main( String[] args ) {
            int a = Integer.parseInt( args[ 0 ] );
            int b = Integer.parseInt( args[ 1 ] );
            int sum = a + b + 1; // bug, + 1 is wrong.
            assert sum == a + b;
   }
}
```

  - Compile: javac AssertClass.java
  - Run: java AssertClass 50 60
  - Run with assertions: java -ea AssertClass 50 60

Strategy 4: jdb Debugger

- jdb is a command-line utility that enables you to debug Java programs.
- Terminology
  - A breakpoint is a line of code specified by the user using the debugger that the interpreter will halt at each time it reaches that point in the program.
  - Single-stepping is the process of executing your code one line at a time (in single steps).
- https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jdb.html

Strategy 4: How to Use jdb

- Compile your program
  - javac -g AssertClass.java
- Run the debugger (two numbers are command line parameters for the AssertClass program)
  - jdb AssertClass 40 70
- Setup break-points in the program
  - Stop in Assertclass.main
- Run the program
  - run
- Execute the single stepping process
  - step
- Print a value
  - print sum

Strategy 4: Key Commands of jdb

Command	Comments
help	displays the list of recognized commands with a brief description.
run	After starting **jdb**, and setting any necessary breakpoints, you can use this command to start the execution the debugged application.
cont	Continues execution of the debugged application after a breakpoint, exception, or step.
print name	Displays Java objects and primitive values.
stop at Class:line	stop at MyClass:22 *(sets a breakpoint at the first instruction for line 22 of the source file containing MyClass)*
stop in Class.method	*sets a breakpoint at the beginning of the method*
step	Execute the current line (go into the the method)
next	Execute the current line (step over calls)
clear Class.line	Clear a breakpoint at a line
exit	Finish debugging

Summary

- Debugging is not always easy. Some bugs can take a long time to find.
- The general strategies to debug code:
  - Create test cases;
  - Add print statements;
  - Add assert statements;
  - Utilize jdb (a command line debugger).