Java inheritance notes

Prof Bill, Feb 2020

Our “textbook” links:

➢ Sedgewick Java 3.3 Designing data types, introcs.cs.princeton.edu/java/33design
➢ Oracle Java, Interfaces and Inheritance section, docs.oracle.com/javase/tutorial/java

Thank you - I bow down to Noctrl’s own Dr. Godfrey Muganda for his excellent Java textbook. I have liberally borrowed ideas from this book. media.pearsoncmg.com/bc/abp/cs-resources/products/product.html#product.isbn=0134038177

Sections:

1) Definitions, 2) UML, 3) Inheritance and ctors, 4) Override methods, 5) Access to methods and variables, 6) Classes, abstract classes, and interfaces, 7) Polymorphism, 8) The Object class

Terms:

<table>
<thead>
<tr>
<th>inheritance</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>concrete class</td>
<td>extends, implements</td>
</tr>
<tr>
<td>abstract class</td>
<td>this, super</td>
</tr>
<tr>
<td>interface</td>
<td>public, private, protected</td>
</tr>
<tr>
<td>polymorphism</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Versus battles:</th>
<th>Object class</th>
</tr>
</thead>
<tbody>
<tr>
<td>is-a, has-a relationships</td>
<td>inner classes</td>
</tr>
<tr>
<td>super class, subclass</td>
<td></td>
</tr>
<tr>
<td>override, overload</td>
<td></td>
</tr>
<tr>
<td>inheritance, composition</td>
<td></td>
</tr>
<tr>
<td>single inheritance, multiple inheritance</td>
<td></td>
</tr>
</tbody>
</table>

| UML diagram: class name, variables, methods; has-a arrow, is-a arrow |
1. Definitions

Inheritance - allows a new class to extend an existing class; the new inherits the member methods and variables

In definition above, existing class is the **superclass**; the new class is the **subclass**.

Java snippet:

```java
public class NewExample extends ExistingExample {
    // new inherits methods and variables from existing
    // NewExample is subclass; ExistingExample is superclass
}
```

Inheritance is often called the **is-a relationship**; example: Grasshopper is-a Insect; another example: every class is-a Object implicitly in Java

Java has two mechanisms for inheritance:

1. **Interface**, using `implements` keyword; methods only
2. **Class**, using `extends` keyword; methods and variables

Composition - allows a new class to specify other existing classes that are a part of it

In Java, composition simply means that one object is a member variable of another.

Composition is often call the **has-a relationship**; example: Grasshopper has-a Leg

More Grasshopper:

```java
public class Grasshopper extends Insect {
    // Grasshopper is-a Insect; methods and vars inherited!

    // You can create Grasshopper-specific methods/vars
    Leg backLeft; // composition has-a Leg
    Leg backRight;

    int jump() {
        // code
    }
}
```
You can have levels of inheritance. Example: C is-a B, B is-a A.

```java
public class A {
    // super class methods and variables
}

public class B extends A {
    // B is-a A; B is subclass, A superclass
}

public class C extends B {
    // C is-a B; C is subclass, B is superclass
}
```

Libraries like Java Collections Framework (JCF) have many, many levels of inheritance. It’s fair to describe these libraries as “complex”.

Some people call this an inheritance chain. Some call it the inheritance hierarchy.

Clash of the keywords: implements vs. extends

➢ A class may only extend only one other class
➢ However, Java allows you to implement as many interfaces as you like
➢ Why the difference? Interfaces don’t have variables or ctors that can complicate inheritance

Clash of the relationships: inheritance vs. composition, is-a vs. has-a

❖ Select relationship that best models your design
❖ This is often a difficult design decision

/* Inheritance is the core of OOP in Java. Much of this stuff is very simple and makes sense; that’s its power. */
2. UML

UML class diagrams are an easy, short-hand way to describe classes and the relationships between classes.

In UML, a class is defined as a rectangle with its name, variables, and methods

<table>
<thead>
<tr>
<th>Professor</th>
</tr>
</thead>
<tbody>
<tr>
<td>String name</td>
</tr>
<tr>
<td>String collegeId</td>
</tr>
<tr>
<td>int years</td>
</tr>
<tr>
<td>teach( Course c)</td>
</tr>
<tr>
<td>grade( int hours)</td>
</tr>
</tbody>
</table>

Your diagram can include +/- to indicate public/private members. If they’re missing, we will assume that variables are private and methods are public. Data hiding!

This is a nice overview, and it’s where I got my figures below:

Inheritance, the is-a relationship, is shown with an open arrow between classes
Example: SavingsAccount is-a BankAccount

Composition, the has-a relationship, is shown with a diamond arrow between classes
Example: Person has-a Hand
We won’t cover:

➔ Some people worry about the subtle difference between composition and aggregation...we will not
➔ The UML standard is HUGE and includes many different diagrams; we’ll only care about the class diagram
3. Inheritance and ctors

General ctor rules for all classes:

→ The ctor method name is the same as the class name; example: for the Apple class, the default ctor is Apple().

→ The default ctor has no parameters.

→ If a class doesn’t have any ctor specified, then Java implicitly calls a default ctor to create the variables for a new object.

→ If you specify any ctors in a class, Java won’t implicitly do anything...you must use the ctor(s) specified.

One of the downsides of inheritance...it complicates the creation of objects. Here are some ctor guidelines:

● The superclass ctor always executes before the subclass ctor. This makes sense: a subclass may need superclass data in its own ctor. Java makes this happen automatically.

● In rare cases, you may want to call the super ctor yourself.
  ○ To do this, use the keyword super.
  ○ When calling the super ctor, it must be the first line in your subclass ctor.
  ○ One reason to do this: send super ctor arguments; example: super( 17)

● When interfaces are used for inheritance, these rules don’t apply. Interfaces don’t have ctors.

● If no ctor is specified, then Java

Note the important class vs. interface tradeoff here:

+ Interfaces are simpler and more elegant than classes for inheritance; they don’t have ctor complexity/issues
- But interfaces don’t have variables
- Interfaces are abstract and therefore can’t be created as objects

/* The impact of inheritance on ctors and creating objects is sometimes difficult and hard to follow. This is a weakness of inheritance. */
4. Override methods

It's common for a subclass to **override** the methods of a superclass. This is done by matching the method name and parameters exactly in your subclass.

Some guidelines:

- A subclass **must** override any abstract methods in the superclass
- All interface methods are abstract (they have no code)
- Why override a superclass method that is not abstract (and therefore has code)? Because the subclass needs to accomplish something different in the method.

Java snippet:

```java
public class A {
    public void exitMaze( int level) {
        // code here
    }
}

public class B extends A {
    public void exitMaze( int level) {
        // override code here!
    }
}
```

You can't override superclass variables. You can change their value, but not their type or anything like that.

/* I can **override** a method. I can **overload** a method. What's the difference? */
5. Access to methods and variables

Access to class methods and variables is controlled by these three keywords:

- **public** - can be called (method) or changed (variable) by anyone
- **private** - can only be called or changed within the class
- **protected** - can only be called or changed within the class or any subclass

For data hiding, we typically have private variables and public methods.

Java example:

```java
public class A {
    private int powerUps;
    protected String levelName;

    public void moveMario() {
        // code here
    }
}

public class B extends A {
    public void example() {
        // powerUps var - no access because it’s private
        // levelName var - can change in subclass because it’s protected
        levelName = “Rainbow road”;  
    }
}
```
6. Classes, abstract classes, and interfaces

A **class** has code for all its methods. This is sometimes called a **concrete class**.

An **interface** has no code, only method signatures. All methods are implicitly abstract.

An **abstract class** is a mixture, some abstract and some code. It must be specified explicitly using the keyword **abstract**.

These three options are tools to you as a Java coder. They are there for you to best model the design you are trying to implement.

Java abstract class example:

```java
public abstract class GhostWorld {
    private String[] actors;

    public double averageReview() {
        // code here
    }

    public abstract String favoriteActor(int appearances);
}
```

Interfaces and abstract classes **can not** be created as objects. They are missing code!

They can only be used as a superclass.
One common paradigm for abstract classes: add code to an interface.

Java snippet:

```java
public interface WordCounter {
    public void countWord(String w);
}

public abstract class WordCounterAbs implements WordCounter {
    public void countWordsInString(String sentence) {
        // code here, calls countWord() defined in interface
    }

    public void countWordsInFile(String fileName) {
        // code here, calls countWord() defined in interface
    }

    // NOTE: no countWord() method; so class is still abstract
}

public class MyWordCounter extends WordCounterAbs {
    public countWord(String w) {
        // code here; method is no longer abstract!
    }
}
```

So, motivation for abstract class is often code sharing. JCF example: the abstract class AbstractMap has code that is shared by HashMap and TreeHashMap classes.

/* Yes, abstract classes are more complex, a deeper dive. */
7. Polymorphism

Polymorphism literally means: many forms or shapes.

In Java, it means that a subclass method is given priority over the superclass. Example!

```java
public interface Shape {
    public void draw();
}

class Rectangle implements Shape {
    public void draw() {
        // code to draw a rectangle
    }
}

// define Circle is-a Shape, Square is-a Shape, etc
// snippet: ArrayList of Shapes draws correctly with polymorphism
ArrayList<Shape> shapes = new ArrayList<>();
Rectangle r = new Rectangle();
shapes.add(r);
Circle c = new Circle();
shapes.add(c);
Square sq = new Square();
shapes.add(sq);

for (Shape sh: shapes) {
    sh.draw(); // correct subclass method called, polymorphism!
}
```

Nice polymorphism example: Animal class with Cat, Horse subclasses.

[beginnersbook.com/2013/03/polymorphism-in-java/](http://beginnersbook.com/2013/03/polymorphism-in-java/)

/* Once you “get it”, polymorphism is easy to use and powerful. */
8. The Object class

Java snippet.

```java
public class Example {
    // methods and variables here
}
```

Implicit for every class: Example is-a Object, or public class Example extends Object

Here’s the Javadoc: [docs.oracle.com/javase/8/docs/api/java/util/Objects.html](docs.oracle.com/javase/8/docs/api/java/util/Objects.html)

Three important methods in Object:

- equals() - compare two objects; default use **pointer**
- hashCode() - get hash code for object; default use **pointer**
- toString() - return string for object; default create string for **pointer**

The method defaults are to use Object pointers, but that’s often not very helpful.

The answer: Override in your class.

```java
public class Example {
    // override Object method for nicer printing
    public String toString() { // override Object method for nicer printing
        return "Last example!";
    }
}
```