



Java Foundations

Composition and Inheritance

Copyright

Introduction to Java Programming by Eric Matthews is licensed under a Creative Commons Attribution 3.0 Unported License. This license allows you to

- Copy, distribute and transmit the work
- Adapt the work
- Make commercial use of the work

Under the following conditions:

You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work

With the understanding that:

Waiver — Any of the above conditions can be waived if you get permission from the copyright holder.

Public Domain — Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.

Other Rights — In no way are any of the following rights affected by the license:

- Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
- The author's moral rights;
- Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.

Notice — For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to one or more of the following web pages.

www.DeveloperGeekResources.com

www.EducationAnytime.com

Table of Contents

Composition and Inheritance4
Composition4
Inheritance8
 Can a class instantiate itself?11
 Initialization of classes involved in instantiation11
 Upcasting.....13
Put into practice – Inheritance16

COMPOSITION AND INHERITANCE

Hopefully you are at this stage and starting to get comfortable with the object oriented programming techniques previously covered. We are now going to explore the idea of inheritance and composition. This section is going to introduce some new ideas and principles that may be very different from what you are accustomed to.

To compare this to the Martial Arts, you have just received your green belt and are starting to feel pretty good about coding in Java. Here we are going to show you a new set of moves for your next belt. Take this slowly as it could prove to be a most humbling experience. Hopefully it will not be.

COMPOSITION

Let's discuss what appear to be four simple entities: Phone, Person, Address and Business. Anyone reading this will have a pretty reasonable understanding of each of these entities. You also are likely to have a reasonable understanding of how these entities relate to one another.

In a pure sense we examine the minimum data requirements for each entity:

Phone

- Area code
- Exchange
- Number

Person

- Last name
- First name
- Middle name
- Birth date

Address

- Street
- City
- State
- Zip

Business

- Name

We can say 206-555-5555 is a phone number. Great, care to make the call? Whose phone number is it? A person? A business? A person can have many phone numbers, work, home, pager, cell phone. A business has many phones and phone numbers. Addresses belong to people and they also belong to businesses that contain people conducting business. People can have more than one job. Businesses can have more than one location. I could go on and on in quantifying these relationships as they reflect the "real world". These relationships can get quite complex and difficult to quantify and capture in a technological form.

The term “object” is thrown around a lot, but in reality it is hard to achieve containment for an object because seldom do objects stand on their own two feet. In practical reality most objects need to form a relationship with other objects to have meaning. Having a phone number without any other association is not all that practical.

What ends up happening is entities get created that are really combinations in part or in whole of many entities. This results in a fragmentation of both data and code. Even worse, the idea of discrete objects (phone, address, invoice) gets lost.

With a little encapsulation and then a little inheritance I can contain an object at a more discrete and granular level, yet allow it to have association to other objects in both a practical and a flexible manner.

For these examples, I will not be coding all the methods that you would want to associate with a particular object. You of course could and would want to do this when implementing an object. And, in practical application of these principles you will still run up against the limitations of the DBMS that you are using to store your data. Actually limitations may have been a poor choice of words. Somewhere in your application, your OOP code will need to IO into a world that is somewhat relational in nature. In other words, the design and implementation of the database will be a determining factor on how your code gets written.

Again, I wish to emphasize that my goal is that you understand and can apply the concept of inheritance to the practical issues you have to code.

Let me demonstrate a small example where we use inheritance to create a business object that contains a person, their phone number and address.

If you recall we already built an object earlier named phone. A little reuse is in order and most appropriate here.

Source Phone.java

```
class Phone
{
    String area_cde;
    String exchange;
    String extension;

    void phone_nbr_us() {
        String phn = "";
        phn = (area_cde + "-" + exchange + "-" + extension);
        System.out.println(phn);
    }

    void phone_nbr_europe() {
        String phn = "";
        phn = (area_cde + "." + exchange + "." + extension);
        System.out.println(phn);
    }
}
```

Next we need to build a person class

Introduction to Java Programming

Source Person.java

```
class Person
{
    String last="";
    String first="";
    String middle="";

    void fullname_fml()
    {
        String Person = "";
        if (middle == "")
        {
            Person = (first + " " + last);
            System.out.println(Person);
        }
        else
        {
            Person = (first + " " + middle + " " + last);
            System.out.println(Person);
        }
    }

    void fullname_lmf()
    {
        String Person = "";
        if (middle == "")
        {
            Person = (last + " " + first);
            System.out.println(Person);
        }
        else
        {
            Person = (last + " " + middle + " " + first);
            System.out.println(Person);
        }
    }

    public static void main(String[] args)
    {
        Person prsn = new Person();
        prsn.first = "Albert";
        prsn.last = "Together";
        prsn.fullname_fml();
        Phone hm_phn = new Phone();
    }
}
```

Finally we need to build an address class.

Source Address.java

```
class Address
{
```

```
String street="";
String city="";
String state="";
String zip="";

void Address()
{
    System.out.println(street);
    System.out.println(city);
    System.out.println(state);
    System.out.println(zip);
}

public static void main(String[] args)
{
    Address a = new Address();
    a.street = "1234 Bonney Ave SE";
    a.city = "Renton";
    a.state = "WA";
    a.zip = "98058";
    a.Address();
}
}
```

Now that we have all the classes we want to use we can create the business object.

Source Composition.java

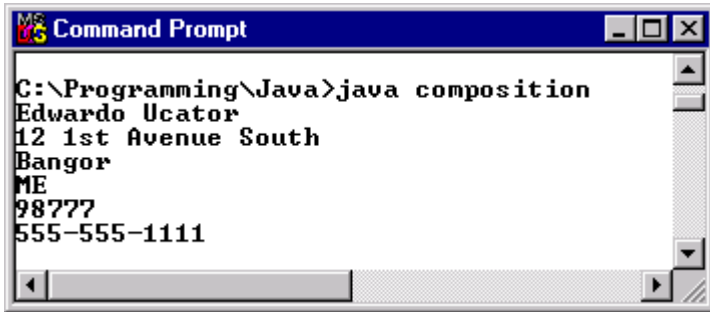
```
class Composition
{
    // Composition is a method of reuse that you have already seen

    public static void main(String[] args)
    {
        Person p = new Person();
        Phone ph = new Phone();
        Address a = new Address();

        p.last      = "Ucator";
        p.first     = "Edwardo";
        ph.area_cde = "555";
        ph.exchange = "555";
        ph.extension = "1111";
        a.street    = "12 1st Avenue South";
        a.city      = "Bangor";
        a.state     = "ME";
        a.zip       = "98777";

        p.fullname_fml();
        a.Address();
        ph.phone_nbr_us();
    }
}
```

}



I named the program “Composition” because this is the term used to denote this type of inheritance. Note that technically “Composition” is a form of reuse but is not technically considered to be inheritance. I personally believe that “composition” is a restrictive form of inheritance. Now here is my rationale and feel free to disagree with me. Inheritance is a process through which an object can acquire the properties and methods of another object. Inheritance also allows for adding and changing the methods and properties related to the object acquired. An object can add methods and properties to a method it has acquired. These methods and properties would now be available only to that object and any other object that inherited it or called it using composition. Well, semantically speaking I can add functionality to an object whether or not I use true inheritance or not. Granted I may not be able to preserve true hierarchical classification using “composition”. But I still can none-the-less extend the functionality of an object using “composition”. I also believe that the definition of inheritance in OOP is too narrow and should be more in line with the real dictionary definition of the word, which simply stated is “to acquire something”.

When you are done with this chapter I am more concerned that you have a practical understanding of “composition” and “inheritance” and how these techniques can be practically applied to solve real problems.

INHERITANCE

This is a very important aspect of OOP. The ability to not only reuse a class, but to extend it is a concept that requires a great deal of architectural consideration. We need to look at a trivial example in terms of code, but I want us to look at an example that is practical and really demonstrates the power of this concept.

Think about the person class we just covered. It is a good start to begin to quantify a human being. But what if we were building an application for an academic institution and we needed to quantify students and teachers. Well, both are people (hey no jokes). But both have attributes and needs that are different from one another. This is where inheritance can play a key role is letting us have our proverbial cake and eat it too.

Source Student.java

```
class Student extends Person
```

```
{
String major="";
String minor="";
double gpa=0;

void GPA()
{
System.out.println("GPA is " + gpa);
}

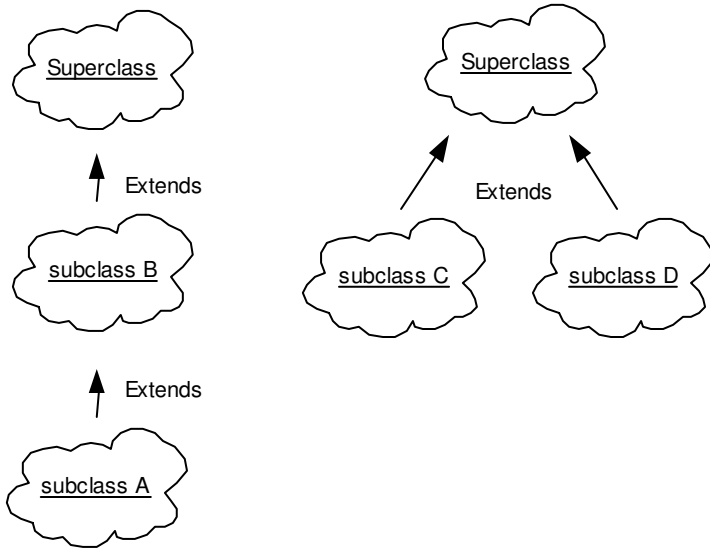
void Major()
{
System.out.println("Major is " + major);
}

void Minor()
{
System.out.println("Major is " + minor);
}

public static void main(String args[])
{
Student sp = new Student();
sp.major = "Computer Science";
sp.minor = "Zoology";
sp.gpa = 3.65;
sp.last = "Bob";
sp.first = "Jim";
sp.fullname_fml();
sp.Major();
sp.Minor();
sp.GPA();
}
}
```

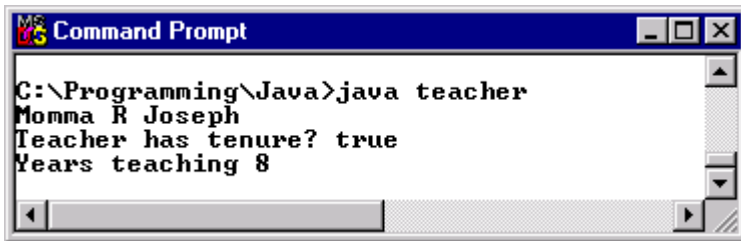
As you can see we can create a new student class that adds the methods and attributes that are unique to a student, while inheriting the general attributes and methods of the person class. Inheritance is achieved by using the “**extends**” keyword. In our example the person class is referred to as the **superclass** and the student class is referred to as the **subclass**.

Scenarios for Superclass/Subclass



To wrap this diagram up I present to you one more class.

Source Teacher.java



```
class Teacher extends Person
{
    boolean tenure_flg;
    String years_in_service="";

    void tenure_flg()
    {
        System.out.println("Teacher has tenure? " + tenure_flg);
    }

    void years_in_service()
    {
        System.out.println("Years teaching " + years_in_service);
    }

    public static void main(String args[])
    {
        Teacher t = new Teacher();
        t.tenure_flg = true;
        t.years_in_service = "8";
        t.last = "Joseph";
        t.middle = "R";
    }
}
```

```
t.first = "Momma";
t.fullname_fml();
t.tenure_flg();
t.years_in_service();
}
}
```

I hope your light bulb has come on in analyzing these examples and their application to the ideas I have presented. Inheritance and composition are not for every programming task. However, where hierarchical classification and re-classification is obvious then it should not only be given consideration, but should be your first choice.

Can a class instantiate itself?

Yes. This does not have anything directly to do with composition or inheritance, but this is an opportune time to demonstrate object recursion.

Source Instantiate_self.java

```
public class Instantiate_self
{
    Instantiate_self()
    {
        System.out.println("instance of self");
    }
    public static void main(String args[])
    {
        Instantiate_self sc = new Instantiate_self();
    }
}
```

Initialization of classes involved in instantiation

It is important to understand the order by which instantiation occurs where inheritance is involved. The order is linear from superclass through subclass. This is really quite logical when you think about it. If "A" is going to inherit "B", then it only makes sense that "B" is in existence before "A". Or put another way, your grandmother is born before your mother, whom was born before you.

Source Subcls2.java

```
public class Subcls2 extends Sprcls
{
    Subcls2()
    {
        System.out.println("this is from subclass");
    }
    public static void main(String args[])
    {
        Subcls2 sc = new Subcls2();
    }
}
```

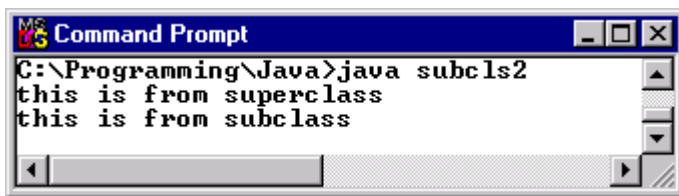
class Sprcls

Introduction to Java Programming

```
{  
  Sprcls()  
  {  
    System.out.println("this is from superclass");  
  }  
}
```

A superclass is initialized before a subclass. Initialization of subclasses happens in hierarchical order.

Notice that “Sprcls” constructor is initialized first.



```
MS-DOS Command Prompt  
C:\Programming\Java>java subcls2  
this is from superclass  
this is from subclass
```

Here is another example.

Source Subcls3.java

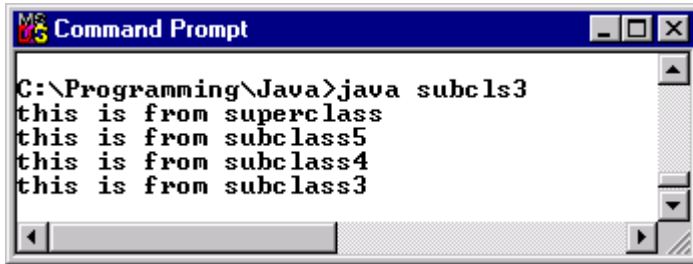
```
// This is meant to demonstrate initialization using inheritance  
  
public class Subcls3 extends Subcls4  
{  
  Subcls3()  
  {  
    System.out.println("this is from subclass3");  
  }  
  public static void main(String args[])  
  {  
    Subcls3 sc = new Subcls3();  
  }  
}  
  
class Subcls4 extends Subcls5  
{  
  Subcls4()  
  {  
    System.out.println("this is from subclass4");  
  }  
}  
  
class Subcls5 extends Supercls  
{  
  Subcls5()  
  {
```

```

    System.out.println("this is from subclass5");
}
}

class Supercls
{
    Supercls()
    {
        System.out.println("this is from superclass");
    }
}

```



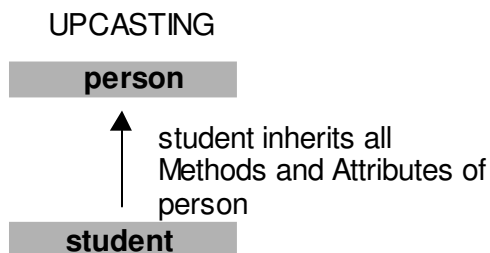
```

C:\Programming\Java>java subcls3
this is from superclass
this is from subclass5
this is from subclass4
this is from subclass3

```

Upcasting

We have actually been covering this concept since beginning the subject of inheritance. I made this a separate subject, as I want to point out that inheritance is really about ancestry, or derivation. One can easily get hung-up and think that the “person” class spawns the student class (reverse the arrow in the picture below). This however is not the case. The fact is that “person” exists as an independent piece of code that may or may not (in this case it does) stand on its own. Other classes we create can derive or inherit this code.



For me it is an incredibly subtle but major point that inheritance be used in situations where the base class (or superclass as it is termed) can meet some of our needs but not all of them. I think the person, student, teacher example though trivial in robustness illustrates this point very well. If we ponder the semantics of the relationship between a student entity and a person entity we can conclude the following:

- A student can be a person
- A person is not necessarily a student
- A student entity will have attributes that are different from a person entity
- A student entity will have attributes that are the same as the person entity

Given this assessment an upcasted solution of student to person makes the most sense as the student entity has a dependency on the person entity, though the person entity exists without dependency to student.

Java's solution for inheritance is one that is hierarchical in nature. The tree in singular, and the implementation is such that object in the upper part of the hierarchy do not have a dependency on the objects below them.

Another incredibly subtle point of inheritance is that when you inherit something you are really doing more than just instantiating the object. You are actually creating a **type** of that object that you can extend (as the Java keyword for inheritance suggests).

This can be a pretty abstract and mind-blowing concept. It is also a concept that can incur the wrath of opponents to OOP that considers such an idea something that can create convoluted code. I have heard the arguments from both sides on this subject and both sides offer compelling points. While I agree this is a very difficult concept to both understand and follow in code, I do not believe it is convoluted if properly thought out and implemented. In fact I believe that used properly and with a great deal of thought that this is as great an invention as pizza!

Before I present this next piece of code for your review let me set it up in this way. You are already familiar with how you can pass values into a method. Well, it is also possible to pass objects as well. In the following example I am demonstrating this more to solidify the idea of inheritance than I am to suggest doing this (at least now) as a coding technique. It is offered to drive a point home about objects. It is my opinion that a more detailed discussion of the practical implementation of this technique falls outside of the parameters of an introductory discussion of the subject.

Source Subcls6.java

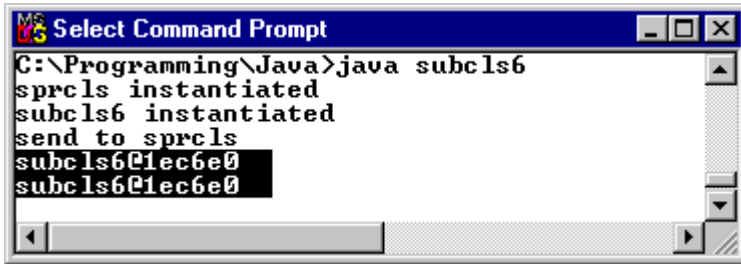
```
public class Subcls6 extends Sprcls
{
    Subcls6()
    {
        System.out.println("subcls6 instantiated");
    }

    public static void main(String args[])
    {
        String s = "send to Sprcls";
        Subcls6 sc = new Subcls6();
        Sprcls.sprcls_methd(s);
        Sprcls.sprcls_methd2(sc);
        System.out.println(sc);
    }
}

class Sprcls
{
    Sprcls()
    {
        System.out.println("Sprcls instantiated");
    }
}
```

```
static void sprcls_methd(String str)
{
    System.out.println(str);
}

static void sprcls_methd2(Subcls6 sc6)
{
    System.out.println(sc6);
}
}
```



```
MS-DOS Select Command Prompt
C:\Programming\Java>java subcls6
sprcls instantiated
subcls6 instantiated
send to sprcls
subcls6@1ec6e0
subcls6@1ec6e0
```

Notice that we passed the object “sc” (an instantiation of class “Subcls6”) into the method call to the other object. Also note that we did not again instantiate the class “Subcls6”, rather we passed a reference to the object “sc” that we created as part of a class that extends another class! Now you may be asking why would we do this? In this case I did it to show that the lineage of extends is a reference to the other object that was instantiated. Object reuse is what inheritance is all about.

I think this is a good point to discuss the merits of composition and inheritance. We could accomplish exactly the same thing that we did with the person, student, and teacher classes using composition. So which one is the best means? There are many passionate arguments on this subject and in my estimation there is no definitive answer. My personal bias is that composition is an easier concept to work with in code than is inheritance. However I also believe that inheritance offers a cleaner and more apparent relationship between two or more classes than composition.

I decided to use inheritance for the Teacher, Student, and Person classes instead of composition. It is true that I could have gone either way. It is also true that I still need to know the details of the person class regardless of the means selected. I still lean toward the camp that uses inheritance under circumstances where I need to extend or change a piece of code.

In business application programming things do not always fit neatly into boxes. Inheritance is a nice way of reusing code, and being able to extend its functionality. Hopefully my “Person” example turned your light bulb on to this concept. Too often I see examples that are so abstract that it is hard to get a clear picture as to how someone would implement a concept that is, itself, quite abstract once you get into the details.

It is also possible to override the methods of an inherited class. This is an area that can be abused, and as such I am not going to cover it. Philosophically I do not like this practice because it is easy to write code that is convoluted. Spaghetti code was a term attributed to crappy programming in structured languages though abuse of “goto”, and other ghoulish creations of bad programmers and hacks. Ravioli code is a term applied to OOP abuses. If you have to change the underlying code in an inherited class, then I would ask yourself the following question.

Should the class you need to change even be a candidate for reuse?

Also I must confess that I do not have a practical example for using this technique, another reason I am not presenting it.

We have covered enough ground on the subject of inheritance. There is certainly more to this subject, and you will need some time to apply these concepts to your code.

PUT INTO PRACTICE – INHERITANCE

This is such an important subject to get, that for this workshop we are going away from the canned labs. Your task is to come up with a “real world” problem and write some code using inheritance. The goal is not to write a fully functional program but rather to demonstrate that you understand and can apply this concept of OOP.