CSE1030 – Introduction to Computer Science II

Lecture #9

Inheritance I

Goals for Today

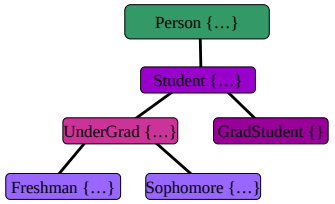
- Today we start discussing Inheritance (continued next lecture too)
- This is an important fundamental feature of Object Oriented Programming
  - This is the last feature of the language you need to learn, after this you can call yourselves Java programmers
- The Theory is easy, but there are lots of practical details:
  - What Inheritance looks like in Java

CSE1030 – Lecture #9

- "is-a" and Inheritance
- Example 1: Introduction to Inheritance
- Example 2: Constructors
- Example 3: Inheriting Code and Data
- Example 4: equals()
- Example 5: Undergrad
- We're Done!

Review "is-a" versus "has-a"

- "is-a"
  - e.g., Class Hierarchy:

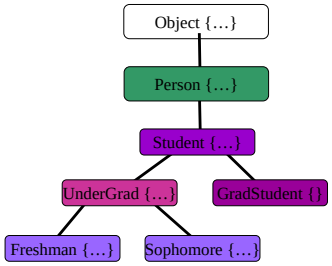


- "has-a"
  - e.g., Person Class:

```
public class Person
{
   // attributes
   private String Name;
   private int    Age;
   private int    Weight;

   Person(String name, int age,
                int weight)
   {
      Name = name;
      Age = age;
      Weight = weight;
   }
...
```

"is-a" and Inheritance



- "is-a" implies that a lower class "is" an instance of the higher class
  - has to do everything that the upper class does
  - so **we can only "add"** not "take away" functionality
- The lower class is called the **SubClass**
- The Higher class is the **SuperClass**
- The subclass **is a kind of** the super-class

CSE1030 – Lecture #9


```
public class Person
{
   // attributes
   private String name;
   private int    age;

   // constructors
   Person(String name, int age)
      { this.name = name; this.age = age; }

   Person(Person otherPerson)
      { name = otherPerson.name; age = otherPerson.age; }

   // methods
   public String getName() { return name; }
   public void   setName(String name)
      { this.name = name; }

   public int  getAge() { return age; }
   public void setAge(int age)
      { this.age = age; }
}
```



```
public class Student extends Person
{
   // attributes
   private String ID;
   private int    year;

   // constructor
   Student(String name, int age, String ID, int year)
   {
      super(name, age);
      this.ID   = ID;
      this.year = year;
   }

   // copy constructor
   Student(Student otherStudent)
   {
      super(otherStudent);
      ID   = otherStudent.ID;
      year = otherStudent.year;
   }
```

This is how we do Inheritance in Java. This is how we denote the "is-a" relationship.

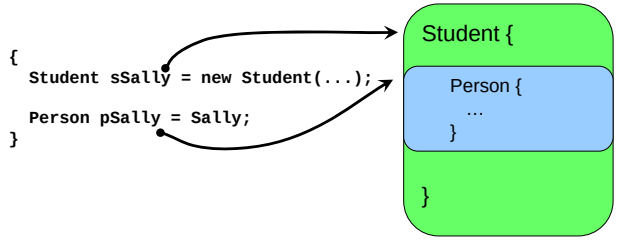
Otherwise, this class looks like the classes we've seen already – we declare some instance data (static data too, if we want) and include functions to do things with the data (next).

```
    // methods
    public String getID()          { return ID;    }
    public void   setID(String ID) { this.ID = ID; }

    public int  getYear()         { return year;      }
    public void setYear(int year) { this.year = year; }
}
```


Relationship Between Super and Sub

- The Superclass exists inside the Subclass
- So, pointers to the subclass can be treated as pointers to the superclass (even though they're not…)




```
public class test
{
   public static void main(String[] args)
   {
      Student sSally = new Student("Sally",23,"2012-12345",2);
      Person  pSally = sSally;

      System.out.println("sSally: " + sSally);
      System.out.println("pSally: " + pSally);
   }
}
```

Output:

```
> java test
sSally: Student@addbf1
pSally: Student@addbf1
```


But the Other Way is Invalid

```
{
   Person  pSally = new Person("Sally", 23);
   Student sSally = pSally;
}
```

```
test2.java:8: incompatible types
found   : Person
required: Student
      Student sSally = pSally;
                       ^
1 error
```

- This substitution is invalid because,
  - Although a *Student* **is** a *Person*
  - A *Person* **is not** a *Student*
    - Because the Student has extra Stuff that the Person class doesn't know about, like **`ID`** and **`year`**.

CSE1030 – Lecture #9

- "is-a" and Inheritance
- Example 1: Introduction to Inheritance
- Example 2: Constructors
- Example 3: Inheriting Code and Data
- Example 4: equals()
- Example 5: Undergrad
- We're Done!

Subclass Constructors

- The subclass must call the superclass's constructor
  - Previous Example:
    The {Student} is a {Person}, and so one of the Person constructors must be called
- You can do this explicitly, as we did in our example
  - as **the 1st statement** in subclass's constructor
- or if you leave it out, Java will insert a call to the default constructor of the superclass for you
  - The default constructor is the one that takes no parameters, equivalent to: **`super()`**


```
public class Student extends Person
{
   // attributes
   private String ID;
   private int    year;

   // constructor
   Student(String name, int age, String ID, int year)
   {
      super(name, age);
      this.ID   = ID;
      this.year = year;
   }

   // copy constructor
   Student(Student otherStudent)
   {
      super(otherStudent);
      ID   = otherStudent.ID;
      year = otherStudent.year;
   }
```

Must be 1st Statement in subclass's Constructor

Or

- The next example uses the default constructor instead…
- Note that we have to have a default constructor in the superclass
  - This is a constructor that has no parameters

```
public class Person
{
   // attributes
   private String name;
   private int    age;

   // constructors
   Person()
      { name = "no name yet"; age = 0; }

   Person(String name, int age)
      { this.name = name; this.age = age; }

   Person(Person otherPerson)
      { name = otherPerson.name; age = otherPerson.age; }

   // methods
   public String getName() { return name; }

   ...
}
```



```
public class Patient extends Person
{
   // attributes
   private String ID;
   private String problem;
   private String treatment;

   // default constructor
   Patient()
   {
      this.ID        = "";
      this.problem   = "";
      this.treatment = "";
   }

   // methods
   public String getID() { return ID; }
   public void   setID(String ID)
      { this.ID = ID; }

   // ...
```

Uses the Person class default constructor

CSE1030 – Lecture #9

- "is-a" and Inheritance
- Example 1: Introduction to Inheritance
- Example 2: Constructors
- Example 3: Inheriting Code and Data
- Example 4: equals()
- Example 5: Undergrad
- We're Done!


```
public class Person
{
   // attributes
   protected String name;
   protected int    age;

   // constructors
   Person(String name, int age)
      { this.name = name; this.age = age; }

   public String toString()
      { return "Person: " + name + "," + age; }

   // methods
   public String getName() { return name; }
   public void   setName(String name)
      { this.name = name; }

   public int  getAge() { return age; }
   public void setAge(int age)
      { this.age = age; }
}
```

Need these to be **protected** if subclass is to have direct access to them, while still keeping the implementation safe from users of the API

```
public class Patient extends Person
{
   // attributes
   private String ID;
   private String problem;
   private String treatment;

   // constructor
   Patient(String name, int age, String ID,
                   String problem, String treatment)
   {
      super(name, age);
      this.ID        = ID;
      this.problem   = problem;
      this.treatment = treatment;
   }

   public String toString()
      { return "Patient: " + name + "," + age + ","
         + ID + "," + problem + "," + treatment; }

   ...
```

Access to protected data from Person class!

Important Point about Inheritance

- All of the **public** or **protected** data and code members of the superclass are accessible in the subclass (e.g., **`name`**, **`age`**, **`toString()`**, etc.)
- The subclass can (should?) probably use the accessors and mutators where possible
  - Because the superclass may change its implementation
- But it is important to keep the code understandable, and sometimes directly accessing the data members is unavoidable

Overriding Inherited Functions

- Remember overloaded functions?
  - Same name, but **different** parameters
  - Example: constructors
- **Overriding** is different:
  - Code in subclass **replaces** code in superclass
  - **same** name, **same** parameters
  - Example (coming up): **`toString()`**


```
public class Person
{
   // attributes
   protected String name;
   protected int    age;

   // constructors
   Person(String name, int age)
      { this.name = name; this.age = age; }

   public String toString()
      { return "Person: " + name + "," + age; }

   // methods
   public String getName() { return name; }
   public void   setName(String name)
      { this.name = name; }

   public int  getAge() { return age; }
   public void setAge(int age)
      { this.age = age; }
}
```

We are overriding the **`toString()`** function inherited from the Object base class

```
public class Patient extends Person
{
   // attributes
   private String ID;
   private String problem;
   private String treatment;

   // constructor
   Patient(String name, int age, String ID,
                   String problem, String treatment)
   {
      super(name, age);
      this.ID        = ID;
      this.problem   = problem;
      this.treatment = treatment;
   }

   public String toString()
      { return "Patient: " + name + "," + age + ","
         + ID + "," + problem + "," + treatment; }

   ...
```

Overridden function: toString()


```
import java.util.*;

public class test
{
   public static void main(String[] args)
   {
      Person sally = new Person("Sally", 23);
      Patient frank = new Patient("Frank", 27,
            "2012-12345", "fever", "aspirine");

      System.out.println("sally = " + sally);
      System.out.println("frank = " + frank);

      Person personFrank = frank;
      System.out.println("personFrank = " + personFrank);
   }
}
```

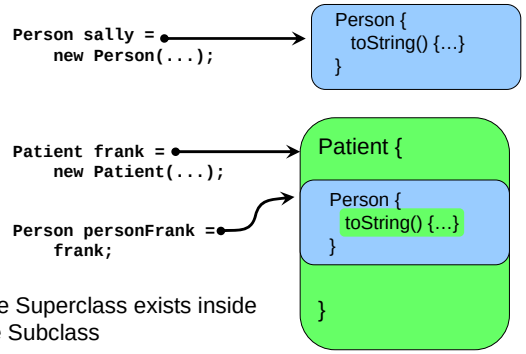
Which toString() is called here?
personFrank is a **Person**

Output

```
sally = Person: Sally,23
frank = Patient: Frank,27,2012-12345,fever,aspirine
personFrank = Patient: Frank,27,2012-12345,fever,aspirine
```

PersonFrank is a **Person**,
and yet the **Patient** toString() was called!!

Inheritance Between Super and Sub



- The Superclass exists inside the Subclass
- But the subclass can replace the code it wants to, by **overriding** it

What if I want to access the Inherited (original) code?

- The **`super.`** keyword can be used to access overridden functions
- For example, **`getPersonString()`**


```
public class Patient extends Person
{
   // attributes
   private String ID;
   private String problem;
   private String treatment;

   // constructor
   ...

   public String toString()
      { return "Patient: " + name + "," + age + ","
         + ID + "," + problem + "," + treatment; }

   public String getPersonString()
      { return super.toString(); }

   // methods
   ...
}
```

Access to superclass's code:
`super.toString()`
calls Person.toString()

Output

```
public class test
{
   public static void main(String[] args)
   {
      Patient frank = new Patient("Frank", 27,
            "2012-12345", "fever", "aspirine");

      System.out.println("toString()        = "
                         + frank.toString());
      System.out.println("getPersonString() = "
                         + frank.getPersonString());
   }
}
```

```
toString()        = Patient: Frank,27,2012-12345,fever,aspirine
getPersonString() = Person: Frank,27
```


CSE1030 – Lecture #9

- "is-a" and Inheritance
- Example 1: Introduction to Inheritance
- Example 2: Constructors
- Example 3: Inheriting Code and Data
- Example 4: equals()
- Example 5: Undergrad
- We're Done!

Review: `equals()`

- The function **`equals()`** provides a way to intelligently compare two objects for equality
- Works better than **==**, because it compares the data, not just the arrow (pointer)
- Last time we kept is simple, but it's time to have another look at **`equals()`** …

Remember this?

`equals()` parameter type matters!

- For casual usage, using the Class type is fine:

```
public boolean equals(Person p) {...}
```

- For maximum Compatibility with Java API, you'll need to use **Object** as the parameter type:

```
public boolean equals(Object o)
{
   ...
}
```


Remember this?

`equals()` Object Parameter Type

- We Have More Work if parameter type is **Object**:
  - Confirm that the Object is really of the correct type
  - Check that the object is not null
  - Convert to compatible (comparable) type

```
public boolean equals(Object o)
{
   if(o == null || getClass() != o.getClass())
      return false;

   Person p = (Person)o;
   return (Name.equals(p.Name) && Age == p.Age);
}
```


Now we can Understand Why

- Because a Patient is a Person, naive **`equals()`** code can be fooled
- So we have to check the class as a part of the comparison
- First, let's examine the simple approach we used before… (possibly incorrect)

```
public class Person
{
   // attributes
   protected String name;
   protected int    age;

   // constructors
   public Person(String name, int age)
      { this.name = name; this.age = age; }

   public boolean equals(Person p)
   {
      System.out.println("in Person.equals()");
      return name.equals(p.name);
   }

   // methods
   ...
}
```

This code assumes that the argument is going to be a **Person**, and not a **Patient**


```
public class Patient extends Person
{
   // attributes
   private String ID;
   private String problem;
   private String treatment;

   // constructor
   ...

   public boolean equals(Patient p)
   {
      System.out.println("in Patient.equals()");
      return ID.equals(p.ID);
   }

   // methods
   ...
}
```

This code appears correct, but in this case the incorrect assumption is that we have a **Patient**, not a **Person**


```
public class test
{
   public static void main(String[] args)
   {
      Patient frank1 = new Patient("Frank", 27,
            "2012-12345", "fever", "aspirine");

      Person frank2 = new Person("Frank", 27);

      System.out.println("frank1.equals(frank2)) = "
            + frank1.equals(frank2));
      System.out.println("frank2.equals(frank1)) = "
            + frank2.equals(frank1));
   }
}
```

Are these the same thing? They're not the same type, but that may or may not be of interest, depending upon the specific application.

Output

```
in Person.equals()
frank1.equals(frank2)) = true
in Person.equals()
frank2.equals(frank1)) = true
```

- Both times, Person.equals() was called
  - Did it do the right thing?
  - Did it do what you want?
  - Should these be considered equal to one another?

A Better Way

- When **Inheritance** is involved, it is more consistent, and usually better, to make the parameter an object of the **Object Class**, and to include the class comparison code…


```
public class Person
{
   // attributes
   protected String name;
   protected int    age;

   // constructors
   public Person(String name, int age)
      { this.name = name; this.age = age; }

   public boolean equals(Object o)
   {
      System.out.println("in Person.equals()");
      if(o == null || getClass() != o.getClass())
         return false;

      Person p = (Person)o;
      return name.equals(p.name);
   }

   ...
}
```



```
public class Patient extends Person
{
   private String ID;

   ...

   public boolean equals(Object o)
   {
      System.out.println("in Patient.equals()");
      if(o == null || getClass() != o.getClass())
         return false;

      Patient p = (Patient)o;
      return ID.equals(p.ID);
   }

   ...
}
```



```
public class test
{
   public static void main(String[] args)
   {
      Patient frank1 = new Patient("Frank", 27,
            "2012-12345", "fever", "aspirine");

      Person frank2 = new Person("Frank", 27);

      System.out.println("frank1.equals(frank2)) = "
            + frank1.equals(frank2));
      System.out.println("frank2.equals(frank1)) = "
            + frank2.equals(frank1));
   }
}
```

Output

```
in Patient.equals()
frank1.equals(frank2)) = false
in Person.equals()
frank2.equals(frank1)) = false
```

- This time, the respective **`.equals()`** functions are called (Person and Patient)
  - This probably makes more sense
  - Also, they are found to be different

One Last Good Idea

- In Patient{}, since we have inherited the **`Person.equal()`** code, we can use it to simplify our implementation of **`Patient.equal()`**


```
public class Patient extends Person
{
   private String ID;

   ...

   public boolean equals(Object o)
   {
      System.out.println("in Patient.equals()");
      if(!super.equals(o))
         return false;

      Patient p = (Patient)o;
      return ID.equals(p.ID);
   }

   ...
}
```

Because we called **Person.equals()**, we don't have to check whether the argument is null or the types are equal. Also, the comparison is better – it only returns true if **both** the **name** and **IDs** are equal.

Output

```
in Patient.equals()
in Person.equals()
frank1.equals(frank2)) = false
in Person.equals()
frank2.equals(frank1)) = false
```

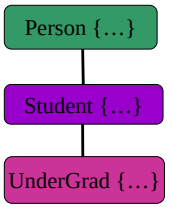
- This time, both **`.equals()`** functions are called (**`Person`** and **`Patient`**)
  - This provides the most thorough comparison

CSE1030 – Lecture #9

- "is-a" and Inheritance
- Example 1: Introduction to Inheritance
- Example 2: Constructors
- Example 3: Inheriting Code and Data
- Example 4: equals()
- Example 5: Undergrad
- We're Done!

One Final Complete Example

- The point here is to provide a complete working example
- We start with the **Person** Class:
- We extend it to be a **Student** Class:
- And Finally we extend Student to be the **Undergrad** Class:



The Person Class

```
public class Person
{
   // attributes
   protected String name;
   protected int    age;

   // constructors
   Person() { name = "no name yet"; age = 0; }

   Person(String name, int age)
      { this.name = name; this.age = age; }

   Person(Person otherPerson)
      { name = otherPerson.name; age = otherPerson.age; }
```



```
   // methods inherited from Object
   public String toString()
      { return "Person: " + name + "," + age; }

   public boolean equals(Object o)
   {
      if(o == null || getClass() != o.getClass())
         return false;

      Person p = (Person)o;
      return name.equals(p.name);
   }

   // accessors and mutators
   public String getName() { return name; }
   public void   setName(String name)
      { this.name = name; }

   public int  getAge() { return age; }
   public void setAge(int age)
      { this.age = age; }
}
```

The Student Class

```
public class Student extends Person
{
   // attributes
   protected String ID;
   protected int    year;

   // constructors
   Student(String name, int age, String ID, int year)
   {
      super(name, age);
      this.ID   = ID;
      this.year = year;
   }

   Student(Student otherStudent)
   {
      super(otherStudent);
      ID   = otherStudent.ID;
      year = otherStudent.year;
   }
```



```
   // methods inherited from Object
   public String toString()
      { return "Student: " + name + "," + age + ","
         + ID + "," + year; }

   public boolean equals(Object o)
   {
      if(!super.equals(o))
         return false;

      Student s = (Student)o;
      return ID.equals(s.ID);
   }

   // methods
   public String getID() { return ID; }
   public void   setID(String ID)
      { this.ID = ID; }

   public int  getYear() { return year; }
   public void setYear(int year)
      { this.year = year; }
}
```


The Undergrad Class

```
public class Undergrad extends Student
{
   // attributes
   protected String major;

   // constructors
   Undergrad(String name, int age,
             String ID, int year, String major)
   {
      super(name, age, ID, year);
      this.major = major;
   }

   Undergrad(Undergrad otherUndergrad)
   {
      super(otherUndergrad);
      major = otherUndergrad.major;
   }
```



```
// methods inherited from Object
public String toString()
   { return "Undergrad: " + name + "," + age + ","
      + ID + "," + year + "," + major; }

public boolean equals(Object o)
{
   if(!super.equals(o))
      return false;

   Undergrad u = (Undergrad)o;
   return major.equals(u.major);
}
```

```
    // methods
    public String getMajor() { return major; }
    public void   setMajor(String major)
       { this.major = major; }

    public void setYear(int year)
    {
       if(year > 4)
          System.err.println("Attempt to set Invalid Year! "
                             + year);
       else
          super.setYear(year);
    }
}
```


Some Code to Test

```
public class test
{
   public static void main(String[] args)
   {
      Undergrad sally
            = new Undergrad("Sally", 23,
                      "2012-12345", 2, "CompSci");

      System.out.println("Sally: " + sally);

      sally.setYear(10);
      System.out.println("Sally: " + sally);
   }
}
```


The Output

```
Sally: Undergrad: Sally,23,2012-12345,2,CompSci
Attempt to set Invalid Year! 10
Sally: Undergrad: Sally,23,2012-12345,2,CompSci
```


CSE1030 – Lecture #9

- “is-a” and Inheritance
- Example 1: Introduction to Inheritance
- Example 2: Constructors
- Example 3: Inheriting Code and Data
- Example 4: equals()
- Example 5: Undergrad
- We’re Done!

Next topic…

Inheritance II