CSE1030 – Introduction to Computer Science II

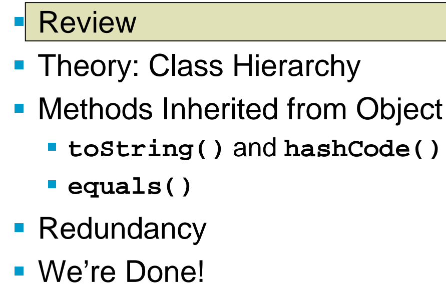
Lecture #4

Non-Static Features
of Java Classes II

Goals for Today

- Goals:
  - Theory: Introduction to Class Extension
  - More (Non-Static) Parts of a Typical Class
- Practical: (Assignment #2!)
  - How to create a typical Java class
  - Details, details, details…
    - toString / equals / hashCode
    - Code Redundancy and Private Member Functions

CSE1030 – Lecture #4



- Review
- Theory: Class Hierarchy
- Methods Inherited from Object
  - **`toString()`** and **`hashCode()`**
  - **`equals()`**
- Redundancy
- We're Done!

Review: Methods / Code

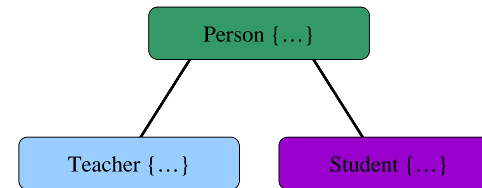
- Grouping Data and Code Together
  - Inherent Relationships
- Methods
  - Constructors
    - Default, "Regular", Copy Constructors, Overloading
  - Data
    - Keep it **`private`**
    - **`final`**
    - Accessor & Mutator Methods
  - Specialised Methods
    - Code Related to a Class, should be in the Class
  - Test cases in **`main()`**

CSE1030 – Lecture #4

- Review
- Theory: Class Hierarchy
- Methods Inherited from Object
  - `toString()` and `hashCode()`
  - `equals()`
- Redundancy
- We're Done!

Extending Classes

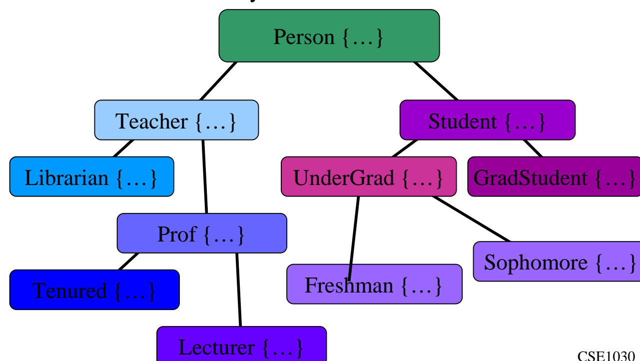
- In Java you can create a class that is based upon another class…



We won't talk about How to do this for another couple of lectures, but you will know all about this soon.

The Class Hierarchy

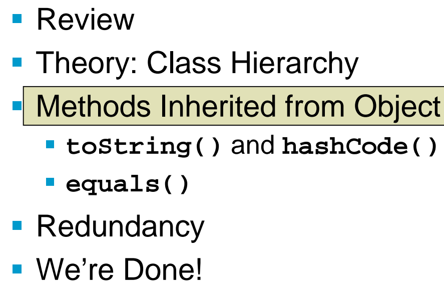
- … by doing this repeatedly, we construct a Class Hierarchy:



The Class Hierarchy

- The Class Hierarchy is one of the great organisational tools of OOP
- Great way to exploit Inherent Relationships
- For now we just need to understand the "Big Picture" - we'll cover the details in a few weeks
- But there are ramifications that affect us today…

CSE1030 – Lecture #4



- Review
- Theory: Class Hierarchy
- Methods Inherited from Object
  - **`toString()`** and **`hashCode()`**
  - **`equals()`**
- Redundancy
- We're Done!

The **Object** Class
(is the root of all classes)

- In Java All Classes (All Objects) are Derived from the **Object Class**
- The important implication is that we get some things for free: (example coming…)
  - **`toString()`**
  - **`hashCode()`** ← more on next slide
  - **`getClass()`**
  - **`equals()`**
- (We get more than this for free, but we won't worry about the rest for now.)

Quick Aside: What's a Hash Code?

- A Hash Code is a number that (hopefully) is unique for a set of data
- Hash Codes:
  - Provide a quick way to compare data
  - Used in **Hash Table** data structures (2nd year stuff)
  - Different data may produce same hash code, Not perfect but good enough
- Example:
  - Data: Bank Balance = $1,271.87
  - Hash Code (by summing the digits) = 1 + 2 + 7 + 1 + 8 + 7 = **26**

Recall The 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; }

   // methods
   public String getName()        { return Name; }
   public void   setName(String n) { Name = n;    }

   public int  getAge()      { return Age; }
   public void setAge(int a) { Age = a;    }

   public void setWeight(int w) { Weight = w; }
}
```

Demo: Methods Inherited from Object

```
public class example
{
   public static void main(String[] args)
   {
      Person p = new Person("William", 36, 120);

      System.out.println("p.toString() = " + p.toString());
      System.out.println("p = " + p);

      System.out.println("p.hashCode() = " + p.hashCode());

      System.out.println("p.getClass() = " + p.getClass());
   }
}
```


The Output:

```
>java example

p.toString() = Person@addbf1

p = Person@addbf1

p.hashCode() = 11394033

p.getClass() = class Person
```

- `toString()` produces a descriptive String
- and automatically too
- `hashCode()` produces a unique number based on a guess of the **Index Data**
- `getClass()` finds our Class object, which gets us neat info

Methods Inherited from Object

- What?
  - `toString()` / `hashCode()` / `getClass()` / `equals()`
  - + a few others…
- Why?
  - These are useful features that every object should have
  - `toString()` is great for debugging
  - `equals()` is very a common function
  - `hashCode()` makes finding an object fast
- How? …

The Object Class – Source Code

```
public class Object
{
   // no attributes
   // no constructors

   // methods
   public final Class getClass();

   public int hashCode();

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

   public String toString()
   {
      return getClass().getName() + "@" + hashCode;
   }
}
```

From: http://www.docjar.com/html/api/java/lang/Object.java.html

CSE1030 – Lecture #4

- Review
- Theory: Class Hierarchy
- Methods Inherited from Object
  - **toString()** and **hashCode()**
  - **equals()**
- Redundancy
- We're Done!

Replacing Inherited Functions

- The Cool Thing is you can Replace the Default Implementations with Better and More Suitable Ones!
  - Particularly: toString / equals / hashCode
- Why?
  - **equals()** and **toString()** really should be customised to handle your class properly
  - **hashCode()** is complicated, because it should be a unique number depending upon the contents of the Object's Data **used for Indexing and Searching**
- Example…

toString and hashCode examples (1)

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

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

   // methods
   public String getName()       { return Name; }
   public void setName(String n) { Name = n;    }

   public int getAge()       { return Age; }
   public void setAge(int a) { Age = a;    }
```


toString and hashCode examples (2)

```
   public void setWeight(int w) { Weight = w; }

   // toString()
   public String toString()
   {
      return "Person:" + Name + "," + Age;
   }

   // hashCode()
   public int hashCode()
   {
      return Name.hashCode() + Age;
   }
}
```

toString and hashCode examples (3)

```
public class example
{
   public static void main(String[] args)
   {
      // First we want to create a person
      Person p = new Person("William", 36, 120);

      System.out.println("p.toString() = " + p.toString());
      System.out.println("p = " + p);

      System.out.println("p.hashCode() = " + p.hashCode());

      System.out.println("p.getClass() = " + p.getClass());
   }
}
```


The Output

```
>java example

p.toString() = Person:William,36

p = Person:William,36

p.hashCode() = -1282435769

p.getClass() = class Person
```

- now `toString()` produces a more meaningful String
- and `hashCode()` produces a unique number based on **Index-able Data**
- Can't change `getClass()`

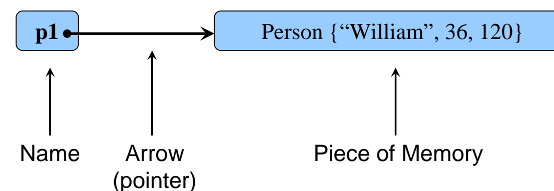
CSE1030 – Lecture #4

- Review
- Theory: Class Hierarchy
- Methods Inherited from Object
  - `toString()` and `hashCode()`
  - `equals()`
- Redundancy
- We're Done!

Comparing Objects

- Theory Idea:
  - Objects are little pieces of memory, with a name:

```
Person p1 = new Person("William", 36, 120);
```



Comparing with ==

- The == operator checks whether the names point to the same memory block (the arrows!)

```
Person p1 = new Person("William", 36, 120);
```

```
Person p2 = p1;
```

"==" checks the arrow

`p1 == p2 is true`

Comparing Objects

- Objects created separately are not == equal
  - Even if they contain the same data!
  - Because the arrow points somewhere else

```
Person p1 = new Person("William", 36, 120);
```

```
Person p2 = new Person("William", 36, 120);
```

`p1 == p2 is false`

So, how to Meaningfully Compare?

- To properly compare, we need the API designer (the person who understands the Class) to write custom comparison code:

```
public boolean equals(Person p)
{
    return (Name.equals(p.Name) && Age == p.Age);
}
```


equals()

- equals() compares data inside the object
  - so it works as you'd expect
  - not by default – only if you replace the default code

```
Person p1 = new Person("William", 36, 120);
```

```
Person p2 = new Person("William", 36, 120);
```

`p1.equals(p2) is true`

equals example (1)

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

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

   public String getName()       { return Name; }
   public void setName(String n) { Name = n;    }

   public int getAge()       { return Age; }
   public void setAge(int a) { Age = a;    }

   public void setWeight(int w) { Weight = w; }
```


equals example (2)

```
   public String toString()
   {
      return "Person:" + Name + "," + Age;
   }

   public int hashCode()
   {
      return Name.hashCode() + Age;
   }

   // equals()
   public boolean equals(Person p)
   {
      return (Name.equals(p.Name) && Age == p.Age);
   }
}
```


Alternate Implementation

- We can use **`hashCode()`** to quickly check for equality of the Index or Search data in the class:

```
public boolean equals(Person p)
{
   return (hashCode() == p.hashCode());
}
```


`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)
{
   ...
}
```

`equals()` **Object** Parameter Type:

- More Work if parameter type is **Object**:
  - Confirm that the Object is really of the correct type
  - Check that the object is not null
    - Maybe?
  - Convert to compatible type

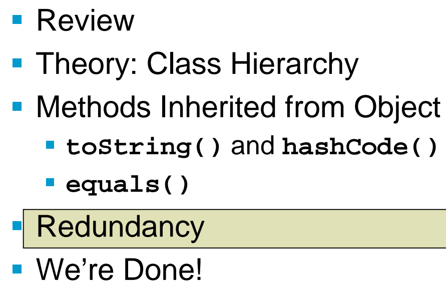
```
public boolean equals(Object o)
{
   if(o == null || getClass() != o.getClass()
      return false;

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


`equals()` Hash Code Simplification

- If we have to use Object as the parameter type, we can simplify our code by using the **`hashCode()`** function

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

   return (hashCode() == o.hashCode());
}
```


CSE1030 – Lecture #4



- Review
- Theory: Class Hierarchy
- Methods Inherited from Object
  - **`toString()`** and **`hashCode()`**
  - **`equals()`**
- Redundancy
- We're Done!

Redundancy & Private Member Functions

- The Idea:
  - Code that gets used in more than one place in a Class, should be made into a private member function to reduce redundancy
- Why? Reducing redundancy:
  - Reduces the number of lines of code, which:
  - Reduces the effort to maintain the code
  - Reduces the likelihood of an error
  - Makes the code more consistent
- Example…

CreditCard Class

```
public class CreditCard
{
   // instance variables/attributes/fields
   private String Name;
   private String Number;
   private double Balance;
   private double Limit;

   // constructor
   public CreditCard(String number,
                     String name, double limit)
   {
      Name    = name;
      Number  = number;
      Balance = 0;
      Limit   = limit;
   }
```



```
   // accessors
   public String getName()    { return Name;    }
   public String getNumber()  { return Number;  }
   public double getBalance() { return Balance; }
   public double getLimit()   { return Limit;   }

   // mutator
   public boolean setLimit(double limit)
   {
      if(limit > 0)
      {
         Limit = limit;
         return true;
      }
      else
         return false;
   }
```



```
   public boolean charge(double amount)
   {
      if(amount < 0)
         return false;

      if(Balance + amount > Limit)
         return false;
      else
      {
         Balance += amount;
         return true;
      }
   }

   public boolean credit(double amount)
   {
      if(amount < 0)
         return false;

      Balance -= amount;
      return true;
   }
}
```


The Problem:

- Need to add Bank Statement functionality
- This is involves writing the details of the transactions to a log file
- Easy, but it takes lots of lines of code…

```
private void WriteToBankStatement(String TransactionText, double Amount)
{
   final String FileName = "BankStatement.txt";
   PrintWriter Logfile   = null;
   SimpleDateFormat DateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

   try {
      Logfile = new PrintWriter(new BufferedWriter(
                    new FileWriter(FileName, true)
                 ));
   } catch (IOException e) {
      System.err.println("Cannot open log file for writing: \""
                                                    + FileName + "\"");
      return;
   }

   Logfile.println(
         DateFormat.format(new Date()) + "  "
       + String.format("%8.2f", Amount) + "  "
       + TransactionText);

   Logfile.close();
}
```


The Solution

- Because it is lots of lines of code, we want only 1 copy of the code in the Class
- So, we define a new **`private`** function:
  - **`private void WriteToBankStatement( String TransactionText, double Amount)`**
- This way, there's only 1 copy of the code
- And it can be called from wherever we need it

The `charge()` function

```
public boolean charge(double amount)
  {
     if(amount < 0)
        return false;

     if(Balance + amount > Limit)
     {
        WriteToBankStatement("Charge Declined", amount);
        return false;
     }
     else
     {
        WriteToBankStatement("Card Charged", amount);
        Balance += amount;
        return true;
     }
  }
```


The `credit()` Function

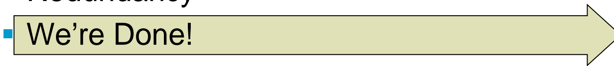
```
public boolean credit(double amount)
  {
     if(amount < 0)
        return false;

     WriteToBankStatement(
                 "Payment Received - Thank You", amount);
     Balance -= amount;

     return true;
  }
```

Summary: Methods / Code

- Methods
  - Constructors
  - Accessor & Mutator Methods
  - Methods inherited from Object
    - toString
    - equals(Object otherObject)
    - hashCode()
  - Specialised Methods
  - Redundancy → Private Member Functions
    - Return type: void
  - Test cases in main()

CSE1030 – Lecture #4

- Review
- Theory: Class Hierarchy
- Methods Inherited from Object
  - **`toString()`** and **`hashCode()`**
  - **`equals()`**
- Redundancy
- We're Done!



Next topic…

Mixing Static and Non-Static Features I