Midterm Review

CSE1030 – Introduction to Computer Science II

Goals for Today

- Theoretical
  - Surviving the Midterm
- Practical:
  - Surviving Lab Test #1
- Don't forget – There are Two Tests!!
  - Midterm: **in Class Tuesday Oct 16**
  - Lab Tests: **in Your Registered Lab Time**
    - Sect 01: **Tues** Oct 16
    - Sect 02: **Thurs** Oct 18

Don't forget to…

- Review your Assignments
- Review the Lecture notes
- Review the Readings
  - Textbook
  - Course Notes
- If you want more practice coding:
  - **Look in the Textbook!**
  - Every chapter contains "**Programming Projects**"

CSE1030 – Lecture #2

- Intro to Object Oriented Programming
- Elements of a Java Class
- Utility Classes
- JavaDoc
- We're Done!

Idea Behind OOP

- Make it easier to develop and maintain large or complex software systems
- Originated in the original Graphical User Interface research projects (complex!)

Sketchpad (1963)

- Fundamental Ideas:
  - Organise Data and Code into Modules
  - Formalise the way one module interacts with another (We call this the **Interface** between the Modules)

Why OOP?

- Encapsulation
  - Data & Code* in single well-defined location
  - Hide complexity away, only expose a simple API**
- Take Advantage of Inherent Relationships
  - Polymorphism
    - Objects that do similar things are often used similarly
  - Inheritance
    - Many things are "a kind of…" something else

*Code = Software
**API = Application Programming Interface

Java Classes

- Classes describe Objects ←Important Idea! (Every Object has a Class)
- Java Class Definition: (we'll come back to this)
  1. **Names** the Class
  2. Describes **How to Construct** an Object of the Class
  3. Stipulates **Who** can use our Objects, and **How**
  4. **Defines the Data** in the Objects (and in the Class)
  5. **Contains all of the Code** pertaining to the Objects

Elements of a Java Class

1. Name the Class
2. How to Construct an Object
3. Who can use, and How
4. Defines the Data
5. Contains the Code

```
// any needed package statement
// any needed import statements
public class ClassName
{
   // data declarations
   private int i;

   // constructor
   ClassName(){ i = 0; };

   // method definitions
   int getI() { return i; }
   void setI(int pi) {i = pi; }
}
```

Definition of a Utility Class

- A Class that contains a common often re-used function (or family of functions)…
- No Objects – usually they are collections of functions
- Examples:
  - java.lang.Math
  - java.lang.System
  - java.util.Collections

The `main()` Function

- The main function is where execution of all java programs begins
- All classes can have a main function
  - Even if there are more than one class, each can have it's own main function
  - The only main function that matters is the one in the controlling class – that is the one that will be run
- The main function is labelled static, meaning that an object is not needed to run the main function
  - That's great if we don't want the added complexity of having objects around

Preconditions

- Preconditions are instructions made to the users of your function
- You should **always check** the validity of **your function's parameters**
- But **if you have limits** in what you can handle, **tell the user** – use a precondition!

JavaDoc Comments

```
/**
 * This class defines a function for
 * adding two numbers
 */
public class AdditionUtility
{
   private AdditionUtility() {};

   /**
    * This function adds two numbers.
    */
   public static int add(int A, int B)
   {
      return A + B;
   }
}
```

Adding Details to **`add()`**

```
/**
 * This function adds two numbers.
 *
 * @param A A number to add
 * @param B Another Number to add
 * @return The sum, A + B
 */
public static int add(int A, int B)
{
   return A + B;
}
```

CSE1030 – Lecture #3

- Review
- The Person Class – Holding Data
- The Default Constructor
- Grouping Data and Code Together
- Copy Constructors
- **`Main()`** as a Testing Facility
- We're Done!

Data / Attributes

- A lot of the OOP Philosophy has to do with Accessing and Changing the Data
- Advice: Keep Data **`private`**
- Allow Access via Accessor & Mutator Functions:
  - Accessors → getData() / Mutators → setData()
  - This gives the API creator Control
    - You can **Act** when something has Changed because you Made them **Call a Function**
  - Isolation & Implementation Independence
    - You can freely **Change** the **Implementation** No one will Know, No one will have to Change their Code!

The Person Class

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

   // no constructors

   // 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; }
}
```

Constructors

- Person Class uses the Default Constructor
  - No Constructor ➔ Default Constructor
  - Default Constructor Initialises:
    - numerics = 0
    - booleans = false
    - objects = null
- Why would you use the Default Constructor?
  - Because it's Easy
  - Less Coding
- For simple Classes, this is Fine
  - But the Person Class is not Simple…

Grouping Data & Code Together (1)

- Good Organisation supports even Large or Complex Programs
- Groups / Modules / Classes should reflect the **Inherent Relationships**
- Example: **Minimum Age to Drive**

Overloaded Constructors

- More than 1 constructor!
  - Basic Constructor:
    `Person(String name, int age)`
  - More Advanced Constructor:
    `Person(String name, int age, int weight)`
  - Copy Constructor:
    `Person(Person p)`
- Overloading
  - Two functions with the same name?
    - They are different if their **Parameters are Different Types**
    - Terminology: **Method's Signature must be Unique**

`main()` for Testing – Summary

- **`main()`** is a part of the class, so
  - It has **Access to All Data and Code**
  - Even **Private** Data and Code
- Using main to do Unit Testing means
  - Your tests are **in one easy to find place**
  - And they are **With the Code** that they Test!

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.)

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;    }
```

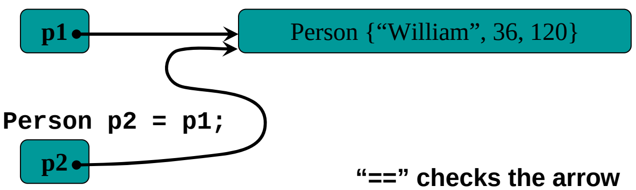
toString and hashCode examples (2)

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

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

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

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
```

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
```

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…

CSE1030 – Lecture #5

- Review
- Variable Scope
  - Parameters vs. Arguments
- Objects as Parameters / Arguments
- Privacy Leaks
- We're Done!

Variable Scope

- What is "Scope"?
  - Variable Scope refers to the areas within your program in which a variable is available
- Why do we care?
  - So we don't write confusing code
  - So we control access to our data

Aside: Parameters versus Arguments

- A **Parameter** is the variable: **`x`**
- An **Argument** is the value: **`10`**

```
double calc(double x)
{
   return x * Slope + Offset;
}

System.out.println("the answer is: " + calc(10));
```

Objects as Parameters, Arguments, and Return Values

- **When an object is passed to a function's Parameter as an Argument**, the object is not copied! Instead, the **arrow (pointer) is passed**, yielding access to the original object.
- The same thing happens when an object is returned from a function.

Object Parameter Passing

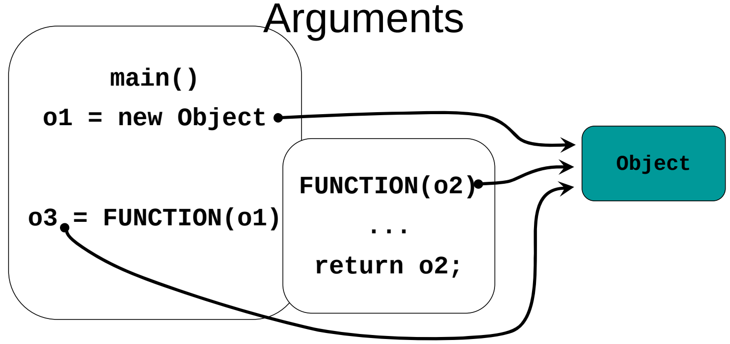
```
public class Int
{
   // data
   public int I;

   // Constructors
   public Int(int i) { I = i;   }  // regular
   public Int(Int i) { I = i.I; }  // copy

   // toString
   public String toString() { return Integer.toString(I); }

   // an example function
   static Int FUNCTION(Int i2)
   {
      System.out.println("i2 (before) = " + i2 + "  == 100?");
      i2.I = 200;
      System.out.println("i2 (after)  = " + i2 + "  == 200?");
      return i2;
   }
```

```
public static void main(String[] args)
  {
     Int i1 = new Int(100);
     System.out.println("i1 = " + i1 + "  == 100?");

     System.out.println("Calling FUNCTION!");
     Int i3 = FUNCTION(i1);

     System.out.println("i1 = " + i1 + "  == 100?");
     i1.I = 400;
     System.out.println("i3 = " + i3 + "  == 200?");
  }
}
```

Results

```
i1 = 100   == 100?
Calling FUNCTION!
i2 (before) = 100   == 100?
i2 (after)  = 200   == 200?
i1 = 200   == 100?
i3 = 400   == 200?
```

Objects as Parameters and Arguments



The **arrows (pointers)** to the objects are what get copied on the way into (parameter/argument) and out of (return) a function.

Privacy Leaks

- Privacy Leaks are accidental access to private data members caused by incorrect treatment of parameters that are objects
- The following code looks like it's doing everything correctly (**`private`** data and accessor / mutator methods)
- But something is wrong…

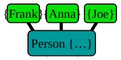
The Solution? Pass Copies of Objects!

- This is why we have Copy Constructors
- By passing a copy of an object, we retain our version of the object, and nobody else can modify it on us.
- We can still provide mutator functions to allow changes to objects, but so long as we copy our own versions of objects, nobody else can modify our objects behind the scenes!

CSE1030 – Lecture #6

- Review
- Static Data versus Instance Data
- Java Notation
- Static Utility Class Revisited
- Variable Hiding & Shadowing
- **`this`**
- We're Done!

Important Concepts from Past Lectures

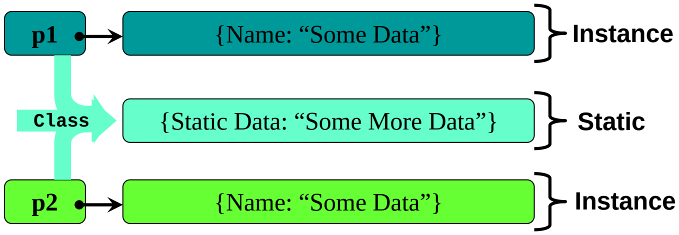
- In Java, Everything is a Class
- Classes Define Objects
  - 

- An Object Variable is
  - A Name,
  - An Arrow (pointer) to memory, and,
  - A Block of Memory
  - 

- Static Utility Classes have no Objects

Review: Regular Classes:

- Regular Classes have:
  - Instance Data (in the Objects)
  - Instance Code (does things with Objects)
  - Static Data (Shared by All Objects)
  - Static Code (Only does things with static data)



Inherent Relationships: Static versus Non-Static Data

- Static Data is Best for
  - Summary Statistics
    - Counting, Serial Numbers, Profiling (Frequency, Time)
  - Class-wide **`final`**s (Constants)
- Static Code is Best for
  - Static Functions
    (Little Utilities that don't need an Object)
  - **`main()`**
- Why?
  - Pertain to a Class, Not Tied to an Object

Initialisation

- Initialise **`static`**s when they are defined (because the constructor is called once for each object created)

```
private static int Number = 42;
```

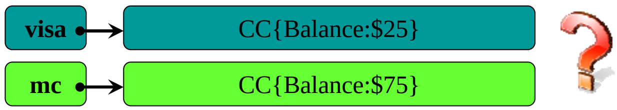
- Initialise instance variables when the object is constructed (i.e., in the Constructor)

```
class example {
   private int Number;

   example() { Number = 42; }
}
```

Initialising **`final`**s

- **`final`** denotes a constant
  within **a Class** (i.e. static)
  or within **an Instance (Object)**
- Why?
  - Some constants pertain to the whole Class, whereas other only to an object
- Example…

How does Java know which Object?

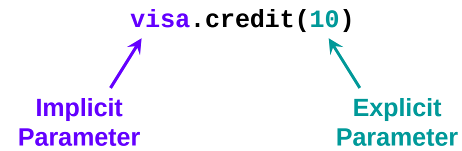



```java
// credit the credit card
public boolean credit(double amount)
{
   if(amount < 0)
      return false;

   Balance -= amount;
   TotalBalance -= amount;

   return true;
}
```

- In **`credit()`**, we just write "Balance", java implicitly figures-out which object (**`visa`** or **`mc`**) we are using

Implicit Parameter / Argument

- The idea is that the object by which an instance function is called is an **Implicit Parameter**, whereas our regular parameters are **Explicit**:

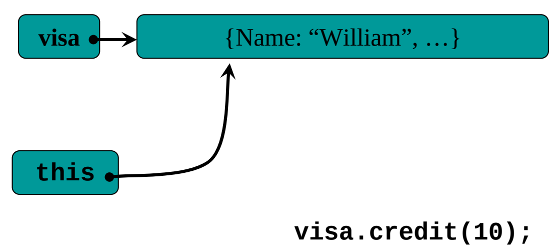


Variable Hiding / Shadowing

- You can define a "Local Variable" or parameter to have the same name as a Class Data Member
- Why?
  - **It's confusing, so it's a bad programming practice**
- Example…

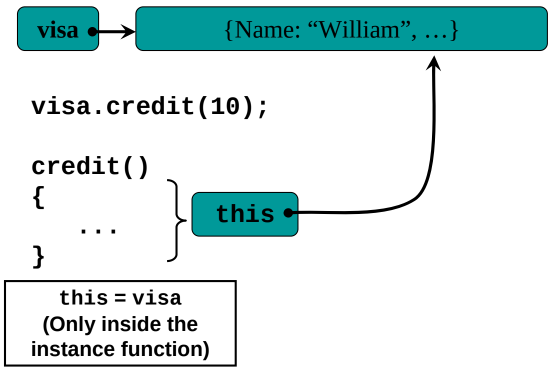
`this`

- **In instance code**, the **`this`** variable is an alias for the name of our object



this

- **this** equals the implicit argument



Why do we need **this**?

- Since we can easily directly refer to:
  - Instance Data (Data inside Objects)
  - Static Data (Data in the Class)

  why do we need **this**?

- **this** allows us to explicitly refer to Instance Data
  - Sometimes good for clarity
  - Solves Variable Hiding Problems
  - Solves Inheritance Problems

Java Documentation Uses for **this**

- **this** is frequently overused
- The Java documentation only lists 5 situations where you need to use **this:**

1. To call from one constructor to another
2. Nested Classes (one class defined inside another one)
3. Passing References
4. Calling subclasses (Inheritance)
5. Fixing Variable Hiding Problems…

this and Cool Variable Hiding?

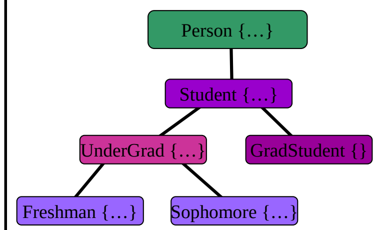
```
public class Cool
{
   String Name;
   int    Age;

   public Cool(String Name, int Age)
   {
      this.Name = Name;
      this.Age  = Age;
   }

   public void setName(String Name)
   {
      this.Name = Name;
   }

   ...  // rest of class
}
```

CSE1030 – Lecture #7

- **Review**
- Theory: "is-a" versus "has-a"
- Special Case 1: Has 1
- Special Case 2: Has a "Known" Number
- General Case: Collections
- Retrieving Data from a Collection
- We're Done!

We have seen both kinds of relationship before…

- "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;
  }
...
```

Recall The Person Class:

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

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

Reminders:
Style Suggestions:
javaNamingConvention
CapitalClasses
Don't Forget Comments!

Baseball Fielders

- In Baseball, when a team plays the field, they have exactly 9 players
- This is a "has-a" relationship
  (teams **are not** players, they **have** players)
- What would the corresponding Java Class look like?

```
public class BaseballFielders
{
   private Person pitcher;
   private Person catcher;
   private Person firstBaseman;
   private Person secondBaseman;
   private Person thirdBaseman;
   private Person shortstop;
   private Person LeftFielder;
   private Person centreFielder;
   private Person rightFielder;
```

What if you don't know how many?

- Java provides **Collections** to conveniently store an unknown number of objects
- Can store collections of any type of object
- There are 3 main families (types) of collection:
  - Sets
  - Lists
  - Maps

Sets

- Are like the mathematical notion of "set", or like a shopping list:
  - {Eggs, Milk, Bread, Chocolate, …}
- No Duplicates
- No notion of numerical or alphabetic "order"

Reminder: Import generic

```
import java.util.*;

public class set
{
   public static void main(String[] args)
   {
      // create a set to store my friends
      HashSet<Person> friends = new HashSet<Person>();

      // create some friends
      Person sally = new Person("Sally", 32);
      Person frank = new Person("Frank", 44);
      Person billy = new Person("Billy", 36);

      // add them to my collection
      friends.add(sally);
      friends.add(frank);
      friends.add(billy);

      System.out.println("I have " + friends.size()
                                          + " friends");
   }
}
```

Lists

- Are like a "To Do" list, a sequence of objects:
  1. Weekly Readings
  2. Go to Class
  3. Work on Assignment
  4. Send e-mail to Prof telling him how riveting his lectures are
  5. Send e-mail to Prof telling him how riveting his lectures are
  6. Submit Assignment
- Can have Duplicates
- Does have a notion of "order" (not necessarily numeric or alphabetic)

```java
import java.util.*;

public class list
{
   public static void main(String[] args)
   {
      // list of people I need to visit
      LinkedList<Person> visits = new LinkedList<Person>();

      // create some people to visit
      Person sally = new Person("Sally", 32);
      Person frank = new Person("Frank", 44);
      Person billy = new Person("Billy", 36);

      // construct list of upcoming visits
      visits.add(sally);
      visits.add(frank);
      visits.add(billy);
      visits.add(frank);

      System.out.println("I have planned " + visits.size()
                                          + " visits");
   }
}
```

Duplicates Allowed!

Maps

- Are like a dictionary: mapping one object (the **key**) to another (the **value**)
  - (Key → Value):
  - (“Hello” → “Bonjour”)
  - (“My Name Is” → “Je m’appelle”)
  - (“Croissant” → “Croissant”)
- Keys must be Unique, Values can be Duplicates

```java
import java.util.*;

public class map
{
   public static void main(String[] args)
   {
      // my list of contacts
      HashMap<String,Person> contacts
                        = new HashMap<String,Person>();

      // create some people to visit
      Person sally = new Person("Sally Yeh", 32);
      Person frank = new Person("Frank Sinatra", 44);
      Person billy = new Person("Billy Holiday", 36);

      // construct list of upcoming contacts
      contacts.put("Sally", sally);
      contacts.put("Frank", frank);
      contacts.put("Billy", billy);

      System.out.println("I have " + contacts.size()
                                       + " contacts");
   }
}
```

(Key, Value) Pairs

Automatic Iteration

- Automatic Iteration is an easy way to get access to the data stored in an (`Iterable`) Collection
- In Java code it looks like this:
  - ```java
    for(Class Variable : Collection)
    {
       do.somthing();
    }
    ```

“Type” or Class Name of the Objects

Variable Name

Collection Variable
    ```

```java
import java.util.*;

public class set
{
   public static void main(String[] args)
   {
      // create a set to store my friends
      HashSet<Person> friends = new HashSet<Person>();

      ...

      // add them to my collection
      friends.add(sally);
      friends.add(frank);
      friends.add(billy);

      System.out.println("I have " + friends.size()
                                       + " friends");
      System.out.println("Here they are:");
      for(Person p : friends)
         System.out.println("   " + p.getName());
   }
}
```

```java
import java.util.*;

public class map
{
   public static void main(String[] args)
   {
      // my list of contacts
      HashMap<String,Person> contacts
                        = new HashMap<String,Person>();

      // create some people to visit
      Person sally = new Person("Sally Yeh", 32);
      Person frank = new Person("Frank Sinatra", 44);
      Person billy = new Person("Billy Holiday", 36);

      // construct list of upcoming contacts
      contacts.put("Sally", sally);
      contacts.put("Frank", frank);
      contacts.put("Billy", billy);
```

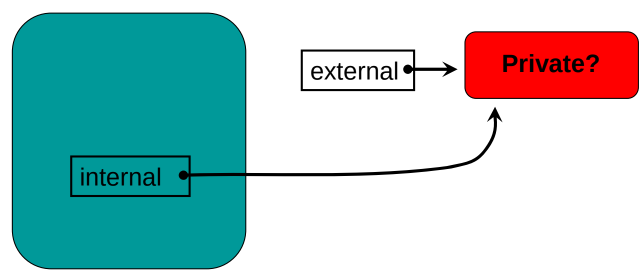
Keys → (the first argument of `contacts.put`)
Values → (the second argument of `contacts.put`)

CSE1030 – Lecture #8

- Review: "is-a" versus "has-a"
- Theory: Composition versus Aggregation
- Iteration
- Shallow vs. Deep Copy
- We're Done!

Privacy Leaks

- When somebody "outside" gets a copy of an object meant to be securely "inside"…



Privacy Leaks

```
import java.util.*;

public class PrivacyLeak
{
   private HashSet<Person> students
                     = new HashSet<Person>();


   // constructor
   public PrivacyLeak()
      { students = new HashSet<Person>(); }


   // add
   public void add(Person p)
      { students.add(p); }
```

Privacy Leak

No Privacy Leak

```
import java.util.*;

public class PrivacyLeak
{
   private HashSet<Person> students
                     = new HashSet<Person>();


   // constructor
   public PrivacyLeak()
      { students = new HashSet<Person>(); }


   // add
   public void add(Person p)
      { students.add(new Person(p)); }
```

No Privacy Leak

Big Theory Idea for Today

- There is an important distinction between code that **uses** an object, and the code that is **responsible for managing** an object
- Ideally: **Responsibility** implies **Ownership**
- The terms we use for this are **Aggregation** versus **Composition**
  - Aggregation = **Using** or **Servicing** an object
  - Composition = **Ownership → Responsibility**

Big Theory Idea for Today

- Examples:
  - **Composition** (means **defining** / **constructing**)
    - Person owns Name
    - CreditCard owns Balance (and TotalBalance)
  - **Aggregation** (means **collecting**)
    - A Person doesn't own their Friend
    - CreditCard doesn't own the Interest Rate
- The idea is pure, but in the real world, the distinction is often arbitrary, and depends upon one's perspective

To Summarise Iterators

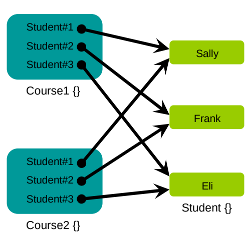
- They provide an easy way to access out data
- They are supported by all of the Java Collections
- The special “for-each” syntax makes them incredibly easy to use
  - Automatically retrieves the iterator
  - Reduces the amount of code we have to write

Comparison

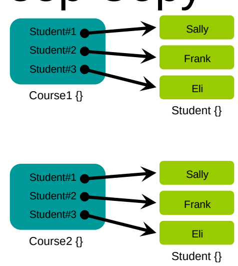
```
    System.out.println("I have " + friends.size()
                                   + " friends");
    System.out.println("Here they are:");
    for(Person p : friends)
       System.out.println("   " + p.getName());
  }
}
```

```
    System.out.println("I have " + friends.size()
                                   + " friends");
    System.out.println("Here they are:");
    Iterator<Person> it = friends.iterator();
    while(it.hasNext())
       System.out.println("   " + it.next().getName());
  }
}
```

Shallow versus Deep Copy



- Faster
- Uses Less Memory
- Aggregation
- Privacy Leak?



- Slower
- Uses More Memory
- Composition
- Protects the Data?

Shallow vs. Deep Summary

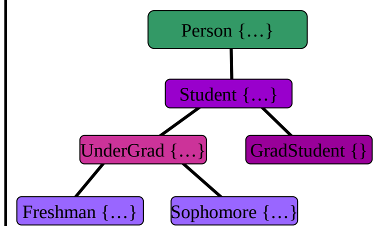
- The “Shallow versus Deep” issue is very similar to a Privacy Leak and it also relates to Aggregation / Composition
  - **If you own the data**, you want to ensure it doesn’t get changed without you knowing about it
  - **If you are using the data**, you probably want to use the latest (most accurate) data available
  - Be aware of the issues, and decide accordingly, by following the Inherent Relationships in the data

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;
   }
...
```

```
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;
   }
```

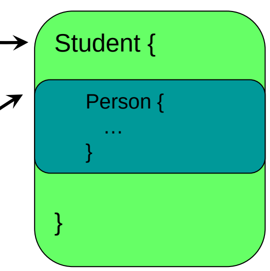
This is how we do Inheritance in Java. This is how we denote the “is-a” relationship.

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).

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…)

```
{
  Student sSally = new Student(...);

  Person pSally = Sally;
}
```



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

```
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

```
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

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()`**

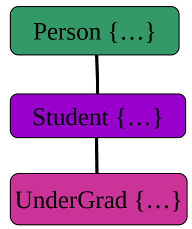
```
public class Patient extends Person
{
   // attributes
   private String ID;
   private String problem;
   private String treatment;

   // 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()

```
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);
   }

   ...
}
```

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:

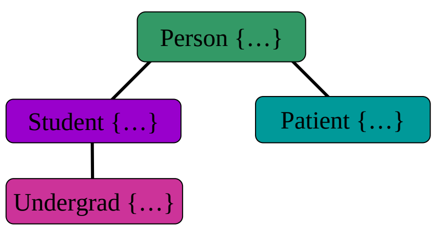


CSE1030 – Lecture #10

- Review
- Polymorphism
- Abstract Classes
- Interfaces
- We're Done!

Polymorphism

- Altogether we have a **Class Hierarchy** that looks like this:




```
import java.util.*;

public class Contacts
{
   // a set in which to store the contacts
   private HashSet<Person> contacts;

   // constructor
   public Contacts() {
      contacts = new HashSet<Person>();
   }

   // add a person to the contacts
   public boolean add(Person contact) {
      return contacts.add(contact);
   }

   // get an iterator
   public Iterator<Person> getIterator() {
      return contacts.iterator();
   }
}
```

Polymorphism

- Look how short and easy the Contacts class is
- Look at how easy it is to use the Contacts class
- This is easy because of polymorphism
- Because all of the object types we are interested are subclasses of Person
  - We don't need 4 separate ways to store objects
  - We can treat all of our objects as Person objects – we don't need 4 separate ways to handle the objects
  - We greatly simplify our code
  - Also, polymorphic inheritance means we reduce the amount of code we need in each class, because the subclasses all do similar things, they can inherit that code from the superclass
    - Like: **`getName(), setName(), getAge(), setAge()`**

`instanceof`

- Polymorphism is great because it encapsulates the complexity of the individual classes
- But occasionally it is useful to do the opposite – to explicitly identify the class of an object
- **`instanceof`** allows us to determine the class of an object
  - Note that due to polymorphism, **`instanceof`** identifies members of a class or any of its subclasses ("is-a")

Abstract Classes

- An Abstract Class is similar to a regular class
  - It can define Data and Code
- But it is missing the implementation of some functions
  - The "missing" functions must be labeled **`abstract`**
  - Also, the class is labeled **`abstract`** as well
- But it includes the "signatures" (names & parameters) of the missing functions
  - This is important for polymorphism
  - We want objects of the abstract class to be useful, even though we are not able to implement some of the code
- Because there is code missing, no objects can be instantiated

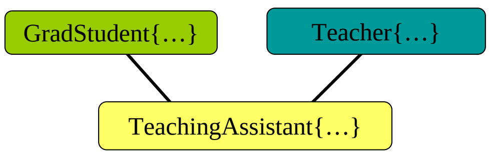
Abstract Account Class – Why?

- What's the advantage of Abstract Classes?
- In general they behave like regular classes
- Polymorphism makes them easy to collect
- Also, polymorphism makes it easy to write generic utility functions that that can be applied to any subclass of Account

(Example on next 5 slides)

Multiple Inheritance

- In general the idea is easy: Multiple Inheritance occurs when a subclass extends two superclasses.
- The **Class Hierarchy** would look like this:



Multiple Inheritance Problems

- Multiple Inheritance can give rise to two problems:
  - Same name with:
    - **#1** Different Meaning
    - **#2** Same Meaning but Different Semantics
- Java fixes Problem #2 by:
  - Multiple Inheritance of Classes is Not Allowed
  - Multiple inheritance can only occur with Interfaces, which are a special form of pure abstract classes
    - Because they have no implementations, they cannot have conflicting semantics
- Java doesn't fix Problem #1, so you have to be careful that all Data and Code names are distinct when doing multiple inheritance with interfaces

Interfaces are similar to classes

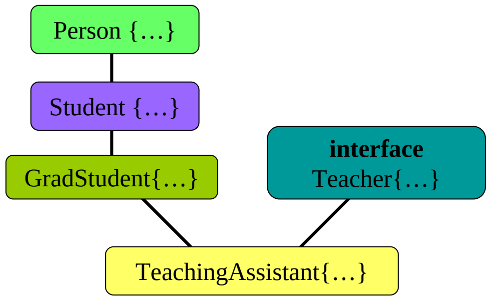
- But you cannot instantiate objects of the interface (no objects!)
  - Only subclasses (sub-interfaces) can be instantiated
- Kind of like a "fill in the blank" class
- But they do support multiple inheritance
  - A class can implement more than one interface
  - Because there's no code, the semantics of a function cannot differ between super-interfaces
- Interfaces can be used just like classes, which makes them very useful
  - The next example demonstrates a collection of Teachers

Summary Notes about Interfaces

- Subclasses may **`extend`** only **one** superclass
- A subclass can **`implement`** any number of interfaces
- (Subclasses do not **`extend`** an interface, they **`implement`** it)
- There is no support in Java to handle name clashes in inherited code – you'll have to change the interfaces to avoid these (inconvenient)
- Interfaces have:
  - no instance data (only **`static final`**)
  - no code
    - only function signatures (function name + parameter types)

Interface Example

- We will give an implementation of this class hierarchy, which includes an interface (Teacher)



Next…

The Midterm