CSE1030 – Introduction to Computer Science II

Lecture #10

Inheritance II

Goals for Today

- Advanced Topics in Inheritance:

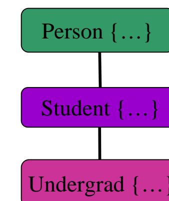
- Polymorphism
- Abstract Classes
- Multiple Inheritance
- Interfaces

CSE1030 – Lecture #10

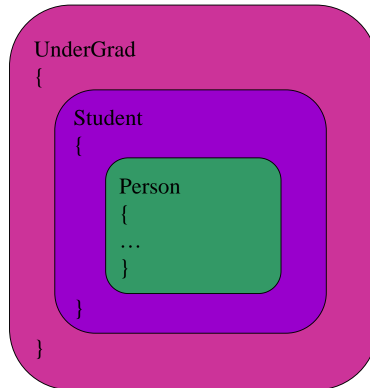
- Review
- Polymorphism
- Abstract Classes
- Interfaces
- We're Done!

Inheritance is "is-a" Relationship

- In Java the "is-a" relationship is realized through **Inheritance**
- The **Person** Class
- is extended to be a **Student** Class
- which is extended to become the **Undergrad** Class



Code and Data are Inherited



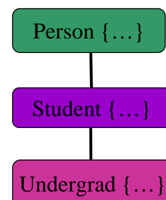
- Any Code or Data defined in a superclass is available in the subclasses (scoping)
- Code can be replaced (overridden)

CSE1030 – Lecture #10

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

Polymorphism

- Let's look at the final example from the last lecture again, **Person**, **Student**, and **Undergrad** …



Review: The Person Class

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

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

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

   public 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; }
}
```


Review: The Student Class

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

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

   private 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; }
}
```

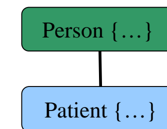

Review: The Undergrad Class

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

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

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

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


Polymorphism

- … and let's also remember the **Patient** class, which also extended the person class…



Review: The Patient Class

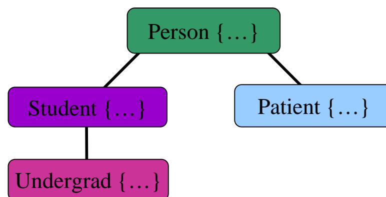
```
public class Patient extends Person
{
   // attributes
   protected String ID;
   protected String problem;
   protected String treatment;

   // constructor
   public 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; }
```



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

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

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

   public String getProblem() { return problem; }
   public void   setProblem(String problem)
      { this.problem = problem; }

   public String getTreatment() { return treatment; }
   public void   setTreatment(String treatment)
      { this.treatment = treatment; }
}
```

Polymorphism

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



The Problem:

- We want a **Contacts** class
  - Basically an **Address Book**
- Needs to be able to record:
  - The **contact details** of the people we know
  - Including **how we know them**
- The solution…

The Solution:

- We should use the **Person** class as the basis for the **Contacts** class
  - It can store objects of Person type (obviously)
  - It can store objects of all classes that extend Person
    - **Student**
    - **Undergrad**
    - **Patient**
- Let's see it …


```
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();
   }
}
```

Client Code that uses Contacts

```
import java.util.*;

public class test
{
   public static void main(String[] args)
   {
      Contacts addressbook = new Contacts();

      Person sally = new Person("Sally", 23);
      addressbook.add(sally);

      Student susy = new Student("Susy",20,"2012-12345",2);
      addressbook.add(susy);
```



```
      Undergrad jim = new Undergrad("James", 23,
                               "2012-12345", 2, "CompSci");
      addressbook.add(jim);

      Patient frank = new Patient("Frank", 27,
                          "2012-12345", "fever", "aspirine");
      addressbook.add(frank);

      System.out.println("Here's my list of contacts");
      Iterator<Person> i = addressbook.getIterator();
      while(i.hasNext())
         System.out.println("  " + i.next());
   }
}
```

Output:

```
Here's my list of contacts
  Patient: Frank,27,2012-12345,fever,aspirine
  Undergrad: James,23,2012-12345,2,CompSci
  Student: Susy,20,2012-12345,2
  Person: Sally,23
```


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()`

New Question

- What if I want to know who on my contact list has been a student for at least 2 years?
- Problem: `years` is defined in Student, so
  - Person **does not** have it
  - Patient **does not** have it
  - Student **does** have it
  - Undergrad **does** have it
- We need to identify the students from the others

```java
import java.util.*;

public class test2
{
   public static void main(String[] args)
   {
      Contacts addressbook = new Contacts();

      Person sally = new Person("Sally", 23);
      addressbook.add(sally);

      Student susy = new Student("Susy",20,"2012-12345",2);
      addressbook.add(susy);

      Undergrad jim = new Undergrad("James", 23,
                             "2012-12345", 2, "CompSci");
      addressbook.add(jim);

      Patient frank = new Patient("Frank", 27,
                      "2012-12345", "fever", "aspirine");
      addressbook.add(frank);
```



```java
      System.out.println("Here's my list of contacts");
      Iterator<Person> i = addressbook.getIterator();
      while(i.hasNext())
      {
         Person p = i.next();

         if(p instanceof Student)
         {
            Student s = (Student)p;
            if(s.getYear() >= 2)
               System.out.println("  " + s);
         }
      }
   }
}
```

Output:

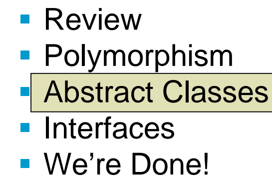
```
Here's my list of contacts
  Undergrad: James,23,2012-12345,2,CompSci
  Student: Susy,20,2012-12345,2
```


`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”)

CSE1030 – Lecture #10

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

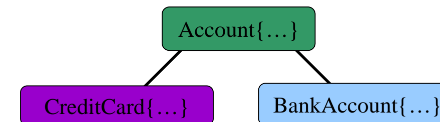


Abstract Classes

- Let's revisit the CreditCard Class
- It allows us to manage a credit card account
  - **`charge(double amount)`**
  - **`credit(double amount)`**
  - **`double getBalance()`**
- If we had a BankAccount class, it would be useful to have a superclass for both…

Abstract Classes

- We are proposing a **Class Hierarchy** that looks like this:



Account Class – First Try

```
public class Account
{
   protected double Balance;

   // constructor
   public Account() { Balance = 0; }

   // accessor
   public double getBalance() { return Balance; }
```



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

      Balance += amount;    // bank account
      Balance -= amount;    // credit card

      return true;
   }

   // mutator - debit
   public boolean debit(double amount)
   {
      if(amount < 0)
         return false;

      Balance -= amount;    // bank account
      Balance += amount;    // credit card

      return true;
   }
}
```

Account Class Problems

- The question of whether debits and credits are added or subtracted from the balance depends upon the context
  - There is no single clear implementation
- The Problem is, "Account" doesn't represent anything real
  - Absent an institution (a bank or credit company) it does not make sense to create an instance of an "Account"
- "Account" is really only a related set of concepts
  - A Balance
  - A means to Credit or Debit that balance
- Solution is to make "Account" an Abstract Class

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

Account Class – As Abstract Class

```
public abstract class Account
{
   protected double balance;

   // constructor
   public Account() { balance = 0; }

   // accessor
   public double getBalance() { return balance; }

   // mutator - credit
   public abstract boolean credit(double amount);

   // mutator - debit
   public abstract boolean debit(double amount);
}
```


Abstract Account Class – Summary

- **`abstract`** Account Class
  - We have included the parts we could,
  - And omitted the parts we couldn't
- Contains some Data
  - **`double Balance`**
- Contains some Code
  - **`getBalance()`**
- Missing some Code (abstract functions define what we want subclasses to implement)
  - **`credit()`**
  - **`debit()`**
- "Abstract" Missing Data would be omitted

BankAccount Class

```
public class BankAccount extends Account
{
   // instance variables/attributes/fields
   protected String name;
   protected String number;

   // constructor
   public BankAccount(String name, String number)
   {
      super();
      this.name   = name;
      this.number = number;
   }

   // accessors
   public String getName()   { return name;   }
   public String getNumber() { return number; }
```



```
   // withdrawl from the account
   public boolean debit(double amount)
   {
      if(amount < 0)
         return false;

      balance -= amount;
      return true;
   }

   // deposit into the account
   public boolean credit(double amount)
   {
      if(amount < 0)
         return false;

      balance += amount;
      return true;
   }
}
```


CreditCard Class

```
public class CreditCard extends Account
{
   // instance variables/attributes/fields
   protected String name;
   protected String number;
   protected double limit;

   // constructor
   public CreditCard(String name, String number,
                                       double limit)
   {
      super();
      this.name   = name;
      this.number = number;
      this.limit  = limit;
   }
```



```
// accessors
public String getName()   { return name;   }
public String getNumber() { return number; }
public double getLimit()  { return limit;  }


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

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

         if(balance + amount > limit)
            return false;
         else
         {
            balance += amount;
            return true;
         }
      }

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

         balance -= amount;

         return true;
      }
   }
```


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 can be applied to any subclass of Account

(Example on next 5 slides)

…Behave like Regular Classes

```
public class client
{
   public static void main(String[] args)
   {
      // first we create some credit-cards
      CreditCard  visa = new CreditCard("William",
                         "1234 5678 9012 3456", 20000);
      BankAccount pc   = new BankAccount("William",
                         "5678 9012 3456 7890");

      // transactions
      visa.debit(100);
      visa.credit(75);
      pc.credit(250);
      pc.debit(225);

      // what are the balances?
      System.out.println("visa Balance: " + visa.getBalance());
      System.out.println("pc Balance:   " + pc.getBalance());
   }
}
```


…Polymorphism → Easy to Collect

```
import java.util.*;

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

      // create some accounts
      CreditCard  visa = new CreditCard("William",
                                "1234 5678 9012 3456", 20000);
      BankAccount pc   = new BankAccount("William",
                                "5678 9012 3456 7890");

      // add them to my collection
      accounts.add(visa);
      accounts.add(pc);

      System.out.println("Here they are:");
      for(Account a : accounts)
         System.out.println("   " + a);   // do something...
   }
}
```

…Polymorphism → Generic Utilities

```
public abstract class Account
{
   ...

   // useful static convenience function
   public static boolean transfer(Account from,
                                  Account to,
                                  double amount)
   {
      // try debiting the 'from' account first
      // if it succeeds then credit the 'to' account
      // otherwise we bail
      if(from.debit(amount))
         return to.credit(amount);
      else
         return false;
   }
}
```


Example of `Account.transfer()`

```
public class client2
{
   public static void main(String[] args)
   {
      // first we create some credit-cards
      CreditCard  visa = new CreditCard("William",
                              "1234 5678 9012 3456", 20000);
      CreditCard  mc   = new CreditCard("William",
                              "5678 9012 3456 7890", 20000);
      BankAccount pc   = new BankAccount("William",
                              "9012 3456 7890 1234");
```



```
      // transactions
       pc.credit(1000);
       visa.debit(100);
       mc.debit(225);

       Account.transfer(pc,   visa, 100);
       Account.transfer(pc,   mc,   200);
       Account.transfer(visa, mc,    25);

       // what are the balances?
       System.out.println("visa Balance: " + visa.getBalance());
       System.out.println("mc Balance:   " + mc.getBalance());
       System.out.println("pc Balance:   " + pc.getBalance());
   }
}
```

Output:

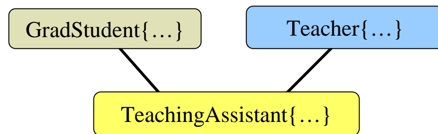
```
visa Balance: 25.0
mc Balance:   0.0
pc Balance:   700.0
```


CSE1030 – Lecture #10

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

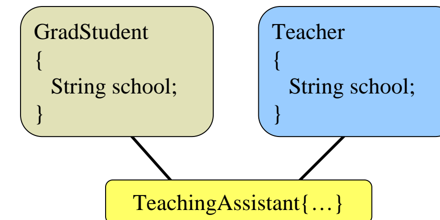
Multiple Inheritance

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



Multiple Inheritance

- Some problems can occur with multiple inheritance when **two different things** with the **same name** are inherited, if they don't both mean exactly the same thing…

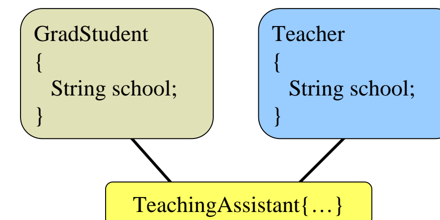


Multiple Inheritance

- Some languages support multiple inheritance
  - e.g., C++, Lisp, Perl, Python, Eiffel
- In order to avoid these complicated scenarios, Java does not support full multiple inheritance
- But it does support a simplified form of multiple inheritance that avoids most of these problems
- The Java approach is called **Interfaces**
- But first, let's review the problems…

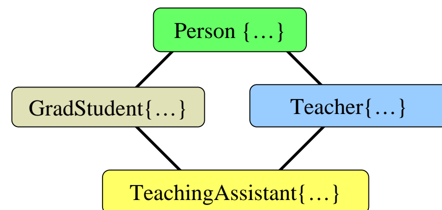
Multiple Inheritance

- Example of **Big Problem #1**:
  - Same Name, Different Meaning
  - Is `school` where the TA studies or teaches?



Multiple Inheritance

- Example of **Big Problem #2**:
- **The Diamond Problem** arises when there are two copies of something with different semantics…



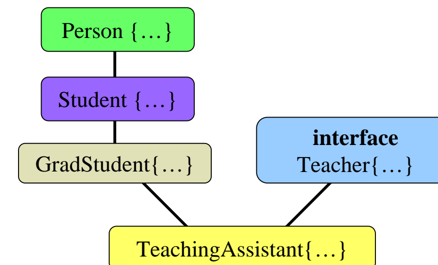

```
Person {
   String name;
   setName(String name) {
      this.name = name;
   }
}

GradStudent {
   setName(String name) {
      super(name + " B.Sc.");
   }
}

Teacher {
   setName(String name) {
      updatePayroll(
         this.name, name);
      super(name);
   }
}

TeachingAssistant{????}
```


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

Interface Example

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



Person Class

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

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


Student Class

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

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

GradStudent Class

```
public class GradStudent extends Student
{
   protected String thesis;

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

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

   // methods
   public String getThesis() { return thesis; }
   public void setThesis(String thesis)
      { this.thesis = thesis; }
}
```


Teacher Interface

```
import java.util.*;

public interface Teacher
{
   public Iterator<String> getCourses();
   public boolean          assignCourse(String coursename);
}
```

- Declared as an **interface**, not as a **class**
- Only **static final** Data is allowed!
  - This is implicit, don't need to declare them **static final**
- No Implementation (Code) is allowed
  - All functions are implicitly **abstract**
  - You don't need to (but you can) declare them **abstract**
  - Only function signatures (name & parameter types)

TA Class

```
import java.util.*;

public class TA extends GradStudent implements Teacher
{
   String coursename;

   // constructors
   public TA(String name, int age, String ID,
          int year, String thesis, String coursename)
   {
      super(name, age, ID, year, thesis);
      this.coursename = coursename;
   }

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



```
   // accessor and mutator methods
   public String getCourseName() { return coursename; }
   public void   setCourseName(String coursename)
      { this.coursename = coursename; }

   // methods to implement Teacher interface
   public Iterator<String> getCourses()
   {
      HashSet<String> set = new HashSet<String>();
      set.add(coursename);
      return set.iterator();
   }

   public boolean assignCourse(String coursename)
   {
      this.coursename = coursename;
      return true;
   }
}
```

test Program

```
import java.util.*;

public class test
{
   public static void main(String[] args)
   {
      TA frank = new TA("Frank", 22, "123456", 2,
                        "Java interfaces", "CSE1030");

      System.out.println("Here's my TA");
      System.out.println(frank);

      System.out.println("");
      System.out.println("Here's his teaching course load:");
      Iterator<String> i = frank.getCourses();
      while(i.hasNext())
         System.out.println("  " + i.next());
   }
}
```


Output

```
Here's my TA
TA: Frank,22,123456,2,Java interfaces,CSE1030

Here's his teaching course load:
  CSE1030
```


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

Prof Class

```
import java.util.*;

public class Prof extends Person implements Teacher
{
   HashSet<String> courses;

   // constructors
   public Prof(String name, int age)
   {
      super(name, age);
      courses = new HashSet<String>();
   }
```

Which constructor is called?

```
    // methods inherited from Object
    public String toString()
    {
       String retstring = "Prof: " + name + "," + age;
       for(String s : courses)
          retstring += "," + s;
       return retstring;
    }

    // methods to implement Teacher interface
    public Iterator<String> getCourses()
    {
       return courses.iterator();
    }

    public boolean assignCourse(String coursename)
    {
       return courses.add(coursename);
    }
}
```


set Test Code

```
import java.util.*;

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

      // create some TAs and Profs
      TA frank = new TA("Frank", 22, "123456", 2,
                        "Java interfaces", "CSE1030");

      Prof sally = new Prof("Sally", 37);
      sally.assignCourse("CSE1020");
      sally.assignCourse("CSE1030");
```



```
      // add teachers to my collection
      teachers.add(frank);
      teachers.add(sally);

      System.out.println("Here they are:");
      for(Teacher t : teachers)
        System.out.println("   " + t);
   }
}
```

Output:

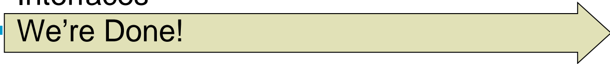
```
Here they are:
   Prof: Sally,37,CSE1020,CSE1030
   TA: Frank,22,123456,2,Java interfaces,CSE1030
```


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)

CSE1030 – Lecture #10

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



Next topic…

Graphical User Interface