CMSC 131

Object-Oriented Programming I

Inheritance Intro, Iterators

Dept of Computer Science
University of Maryland College Park

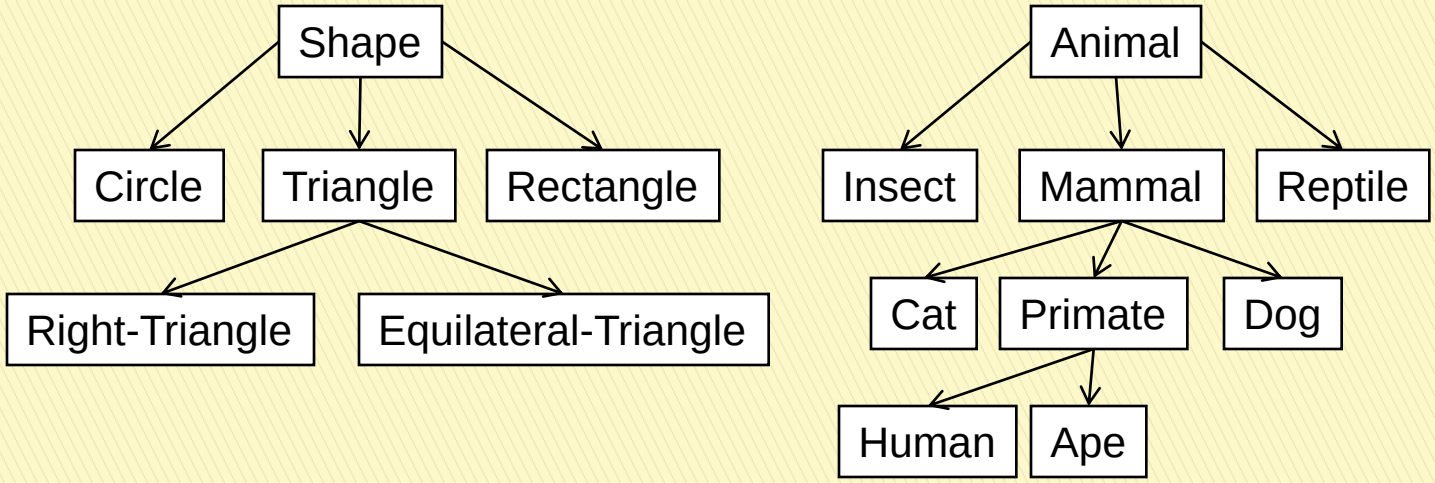
This material is based on material provided by Ben Bederson, Bonnie Dorr, Fawzi Emad, David Mount, Jan Plane

Overview

- Introduction Inheritance
- Iterators

Inheritance

- **Inheritance:** is the process by which one new class, called the **derived class**, is created from another class, called the **base class**
  - The **derived class** is also called: **subclass** or **child class**
  - The **base class** is also called: **superclass** or **parent class**
- **Motivation**: In real life objects have a hierarchical structure:



- We want to do the same with our program objects.

Inheritance

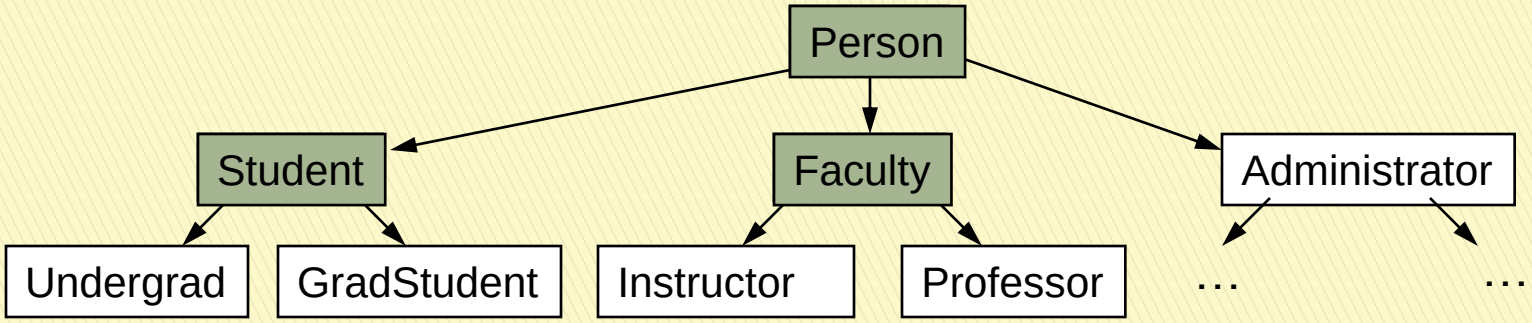
- **Object Inheritance**: What does inheritance mean within the context of object-oriented programming?
- Suppose a **derived class**, **Circle**, comes from a **base class**, **Shape**:
  - Circle should have **all the instance variables** that Shape has.
    (E.g., Shape stores a color, and thus, Circle stores a color.)
  - Circle should have **all the methods** that Shape has (E.g.,
    Shape has an accessor, getColor( ), and thus, Circle has getColor( ).)
  - Circle is allowed to define **new instance variables** and **new methods** that are particular to it:

    (New) Circle Instance variables: Center, radius.

    (New) Methods: draw( ), getArea( ), getPerimeter( ).
- **Code reuse**: Code/Data that is common to all the derived classes can be stored in the base class. This allows us to **avoid code duplication**, and so makes development and maintenance easier.

Example: University People

- Consider the following **Inheritance Hierarchy** (University Database)
- Stores information on various people at the university. The various objects form a hierarchy:



- We will consider the design of the **Person**, **Student**, and **Faculty** classes
- These classes will be very simple (almost trivial). Watch for the relationships between these classes

Base Class: Person (Part 1)

Base class: Person (part 1)

We will put all this in a package called university.

```
package university;

public class Person {
   private String name;          // person's name
   private String idNum;         // ID number

   public Person( ) {
      name = "No Name";
      idNum = "000-00-0000";
   }

   public Person( String n, String id ) {
      name = n;
      idNum = id;
   }

   public Person( Person p ) {
      name = p.name;
      idNum = p.idNum;
   }
   // ...other methods in part 2
}
```

Default constructor

Standard constructor

Copy constructor

Base Class: Person (Part 2)

Base class: Person (part 2)

```
public class Person {
   private String name;          // person's name
   private String idNum;         // ID number

   // ... constructors in part 1

   public String getName( ) { return name; }

   public String getIdNum( ) { return idNum; }

   public void setName( String n ) { name = n; }

   public void setIdNum( String id ) { idNum = id; }

   public String toString( ) {
      return "[" + name + "] " + idNum;
   }

   public boolean equals( Person p ) {
      return name.equals( p.name ) && idNum.equals( p.idNum );
   }
}
```

Instance variables (part 1)

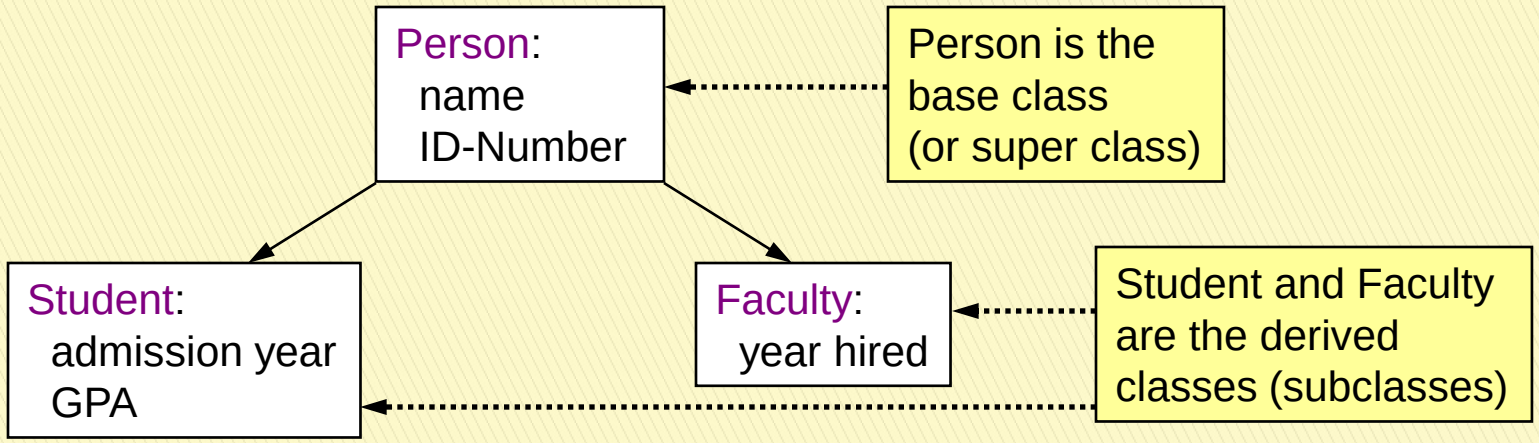
Accessors and setters

This is all standard stuff. Eclipse will set much of this up for you with:
Source → Generate Getters and Setters

toString and equals

Derived Classes: Student and Faculty

- We derive two classes Student and Faculty. Each class inherits all the data and methods from Person, and adds data and methods that are particular to its particular function.



- **Student**: In addition to name and ID, has **admission year** and **GPA**.
- **Faculty**: In addition to name and ID, has the **year they were hired**.

Derived Class Structure

- **Person**: (base class)
  Instance Data: Name and ID-number
    String name
    String idNum
  Methods:
    Constructors: default, standard, copy constructors.
    Accessors/Setters: getname( ), setName( ), getIdNum( ), setIdNum( ).
    Standard methods: toString( ), equals( ).

- **Student**: (derived from **Person**)
  Instance Data: Admission year and GPA
    int admitYear
    double gpa
  Methods: (same structure as Person)

- **Faculty**: (derived from **Person**)
  Instance Data: Year hired
    int hireYear
  Methods: (same structure as Person)

Derived class: Student (Part 1)

Derived class: Student (part 1)

```
package university;
public class Student extends Person {
   private int admitYear;
   private double gpa;

   public Student( ) {
      super( );
      admitYear = -1;
      gpa = 0.0;
   }
   public Student( String n, String id, int yr, double g ) {
      super( n, id );
      admitYear = yr;
      gpa = g;
   }
   public Student( Student s ) {
      super( s );
      admitYear = s.admitYear;
      gpa = s.gpa;
   }
   // ...other methods in part 2
}
```

Tells Java that Student is derived from Person

Additional instance variables

This calls the default constructor for our base class (superclass), Person, to set name and idNum.

Calls Person constructor

Calls Person copy constructor.

Dissecting the Student Class

- **Extends**: To specify that Student is a **derived class** (subclass) of Person we add the descriptor "extends" to the class definition:

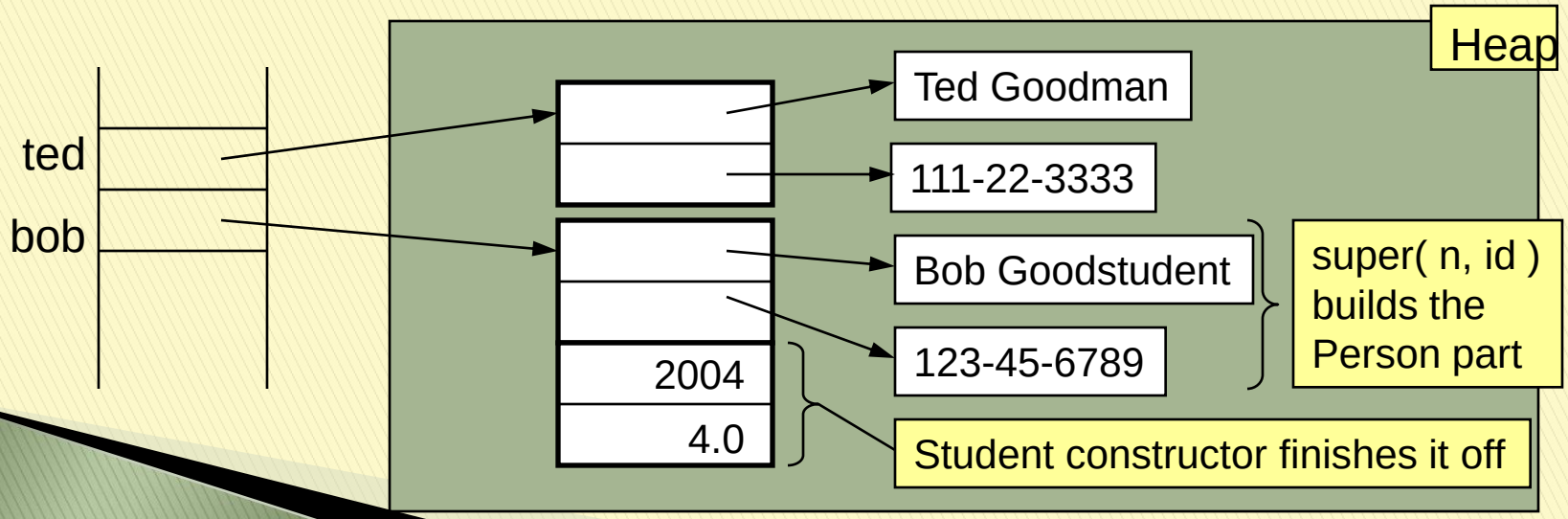
  public class Student extends Person { … }

- Notice that a Student class
  - Inherits everything from the Person class
  - A Student IS-A Person (wherever a Person is needed, we can use a Student).
- **super( )**: When initializing a new Student object, we need to initialize its **base class** (or **superclass**). This is done by calling **super(…)**. For example, **super(name, id)** invokes the constructor **Person(name, id)**
  - super(…) must be the **first statement** of your constructor
  - If you **do not** call super( ), Java will automatically invoke the base class's **default constructor**
  - What if the base class's default constructor is **undefined**? **Error**
  - You must use "**super(…)**", not "**Person(…)**".

Memory Layout and Initialization Order

- When you create a new derived class object:
  - Java allocates space for **both** the **base class** instance variables and the **derived class** variables
  - Java initializes the **base class variables first**, and then initializes the derived class variables
- **Example**:

```
Student bob = new Student( "Bob Goodstudent", "123-45-6789",
                                                2004, 4.0 );
Person ted = new Person( "Ted Goodman", "111-22-3333" );
```



Derived class: Student (Part 2)

Derive class: Student (part 2)

```
public class Student extends Person  {
   private int admitYear;
   private double gpa;

     // ... constructors in part 1

   public int getAdmitYear( ) { return admitYear; }
   public double getGpa( ) { return gpa; }

   public void setAdmitYear( int yr ) { admitYear = yr; }
   public void setGpa( double g ) { gpa = g; }

   public String toString( ) {
      return super.toString( ) + " " + admitYear + " " + gpa;
   }

   public boolean equals( Student s ) {
      return super.equals( s ) &&
             admitYear == s.admitYear &&
             gpa == s.gpa;
   }
}
```

Instance variables (part 1)

Standard accessors and setters

We do not need accessors for the base class; we inherit them.

Calls Person toString for name and idNum.

Calls Person equals to test name and idNum.

Inheritance

- **Inheritance:** Since Student is derived from Person, a Student object can invoke any of the Person methods, it **inherits** them.

```
Student bob = new Student( "Bob Goodstudent", "123-45-6789", 2004, 4.0 );
String bobsName = bob.getName( ) );
bob.setName( "Robert Goodstudent" );
System.out.println( "Bob's new info: " + bob.toString( ) );
```

bob is a Student, but by inheritance we can invoke Person methods

- **A Student "is a" Person**:
  - By inheritance a Student object is also a Person object. We can use a Student reference anywhere that a Person reference is needed.

```
Person robert = bob;          // Okay: A Student is a Person
```

  - We cannot reverse this. (A Person need not be a Student.)

```
Student bob2 = robert;        // Error! Cannot convert Person to Student
```

Iterators

- **ArrayList** is inherited from an class called **AbstractList**.
  - Java provides **many different data structures** that are inherited from AbstractList, e.g. linked lists, binary trees, hash tables.
  - They all provide a device for enumerating all the elements of the data structure: **Iterator**. An **iterator** is an **object** that allows you to **enumerate** the elements of a collection, one by one.
- **How Iterators work**: Let **list** be an ArrayList (or any class inherited from AbstractList).

  Iterator x = list.Iterator( ): Creates a **new iterator** object for list. It is positioned at the **start** of the list.

  x.next( ): Returns the **next element** of the list, and advances the iterator. (Throws an exception if none left.)

  x.hasNext( ): Returns **true** if more elements remain in the list

Iterator Example

```
ArrayList<String> names = new ArrayList<String>();
names.add("Mary");
names.add("Kelly");
names.add("John");

Iterator<String> iter = names.iterator();
while (iter.hasNext()) {
    String value = iter.next();
    System.out.println(value);
}
```