Inheritance Introduction

Department of Computer Science
University of Maryland, College Park
Announcements

• Make sure you check your projects results in the submit server
• Do not wait until the day of the project to try submitting your project
  • Submission problems are not a valid excuse for a project extension
• Remember we take academic integrity matters seriously
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:

![Inheritance Diagram](image)

- 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
University Database

- We derive two classes, Student and Faculty from Person. 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.
- The above diagram is referred to as an inheritance tree/hierarchy.

```
Person:
  name
  ID-Number

Student:
  admission year
  GPA

Faculty:
  year hired

Person is the base class (super class)
Student and Faculty are the derived classes (subclasses)
```
extends and super

- **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 **super class**). 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( ... )`”.
- **Example**: `university package`
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 (what explains why super() should appear first)
- Example:

  ```java
  Student bob = new Student( "Bob", 457, 2004, 4.0);
  Person ted = new Person( "Ted", 331);
  ```
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", 457, 2004, 4.0 );
  String bobsName = bob.getName( ) ;
  bob.setName( "Robert" );
  System.out.println( "Bob's new info: " + bob.toString( ) );

• **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
Overriding Methods

- **New Methods**: A derived class can define **entirely new** instance variables and new methods (e.g. `hireYear` and `getHireYear()`)
- **Overriding (“redefining”, changing what is does)**: A derived class can also **redefine existing** methods

```java
public class Person {
    ...
    public String toString() { ... }
}

public class Student extends Person {
    ...
    public String toString() { ... }
}
```

Student `bob` = new `Student( "Bob", 457, 2004, 4.0);`

System.out.println("Bob's info: " + bob.toString());
Overriding and Overloading

- Don’t confuse method **overriding** with method **overloading**
  - **Overriding ("redefining")**: occurs when a derived class defines a method with the **same name** and **parameters** as the base class
  - **Overloading**: occurs when two or more methods have the **same name**, but have **different parameters** (different signature)

**Example:**

```java
public class Person {
    public void setName(String n) { name = n; }
    ...
}

public class Faculty extends Person {
    public void setName(String n) {
        super.setName("The Evil Professor " + n);
    }
    public void setName(String first, String last) {
        super.setName(first + " " + last);
    }
}
```

The base class defines a method `setName()`

Overriding: Same name and parameters; different definition.

Overloading: Same name, but different parameters.
Overriding Variables: Shadowing

• We can override methods, can we override instance variables too?
• **Answer:** Yes, it is possible, but not recommended
  • Overriding an instance variable is called **shadowing**, because it makes the base instance variables of the base class inaccessible (we can still access it explicitly using `super.varName`). You are creating a new variable with the same name

```java
public class Person {
    String name;
    // ...
}

public class Staff extends Person {
    String name;
    // ... name refers to Staff’s name
}
```

• This can be **confusing** to readers, since they may not have noticed that you redefined `name`. **Better to just pick a new variable name**
super and this

- **super**: refers to the base/super class object
  - We can invoke any base class constructor using `super( ... )`
  - We can access data and methods in the base class (Person) through `super`. E.g., `toString()` and `equals()` invoke the corresponding methods from the `Person` base class, using `super.toString()` and `super.equals()`

- **this**: refers to the current object
  - We can refer to our own data and methods using “this.”
  - In a class, we can invoke one constructor from another constructor using `this( ... )`. As with the `super` constructor, this can only be done within a constructor, and must be the first statement of the constructor. Example:

    ```java
    public Toy(Toy toy) {
        this(toy.name, toy.releasedYear);
    }
    ```

- Can `super()` and `this()` calls appear simultaneously in a constructor?