CSE 2123: Object-oriented Programming: Classes & Inheritance

Jeremy Morris
Extending classes - Inheritance

- Classes fall into a hierarchical structure
  - *parents* (or *super-classes*)
  - *children* (or *sub-classes*)

- For example:
  
  ```java
  java.lang.Object
  └── java.io.Writer
      └── java.io.PrintWriter
  
  *java.lang.Object* is the parent class of all other classes
  - This includes user-defined classes!
Extending classes - Inheritance

- Child classes *inherit* methods from their parent classes
  - By default, a child has all of the *public* methods of its parent
  - However, these methods may or may not be usefully implemented by the parent
  - To create a class that is a child of another class we use the *extends* keyword:
    ```java
    public class listMinTest extends TestCase { ... }
    ```
  - By default, all classes are children of the Object class – java.lang.Object
Extending classes - Inheritance

Example: toString()

- toString() is implemented in java.lang.Object
  - java.lang.Object is the parent of every class
    - So every class inherits this method

- Called automatically in print() methods if you try to print an object:
  Student student1 = new Student();
  System.out.println(student1);

- Or you can call it explicitly
  System.out.println(student1.toString());

- Same output regardless (sample):
  studentExample.Student@19821f
Extending classes - Inheritance

- Same output regardless (sample):
  ```java
  studentExample.Student@19821f
  ```

- This is not really useful output
  - Fortunately we can write our own `toString()` method to replace the inherited one
  - This is known as **overriding** the method
Extending classes - Inheritance

Example:

```java
public String toString() {
    String msg;
    msg = "(" + this.studentId + " : " + 
           this.lastName + ", " + this.firstName + ")"
    return msg;
}

// (123456789: Marley, Bob)
```
Extending classes - Inheritance

- **Another example:** `equals()`
  - Compares two objects to see if they are the same
  - **Note:** The default `equals()` inherited by a new class may not act the way you expect:
    ```java
    Student student1 =
        new Student("William","Tell","123456789");
    Student student2 =
        new Student("William","Tell","123456789");
    Student student3 = student1;
    // student1.equals(student2) will return false
    // student1.equals(student1) will return true
    // student1.equals(student3) will return true
    ```
Extending classes - Inheritance

- By default, `equals()` and `==` operate the same way
  - Return true if the same `object` is being tested
  - Return false if we are looking at different `objects`
    even if the contents of those objects are the same

- We can override `equals()` to behave differently
Extending classes - Inheritance

**Example:**

```java
public boolean equals(Object s) {
    boolean testResult;
    if (s instanceof Student) {
        Student s1 = (Student) s;
        if (this.firstName.equals(s1.getFirstName()))
            && this.lastname.equals(s1.getLastName())
            && this.studentId.equals(s1.getStudentId()) {
            testResult=true;
        }
        else { testResult=false; }
    }
    else { testResult=false; }
    return testResult;
}
```
Note that in the equals example, we pass in an object of type Object:

```java
public boolean equals(Object s)
```

We can do this because Object is the parent class of any other Java class.

- We can treat any child class as if it were actually of the same type as its parent class.
Extending classes - Polymorphism

For example:

```java
Object obj1 = new Student();
```

- This will work
  - Because `Student` is a child class of the `Object` class
  - However, we can ONLY use methods defined for the `Object` class (e.g. `equals`, `toString`, etc.)
    - Java compiler *thinks* that `obj1` is an `Object`
    - Restricted to only things an `Object` could do
Extending classes - Polymorphism

Why do we use polymorphism?

- Sometimes we need to treat a class as a different class
  - equals example
    - We need to allow objects of any class type to be passed in for comparison purposes
    - Allows us to have a single method that tests `Object` objects rather than trying to define a method for every possible class
      - (Imagine trying to do that...)
    - Used where appropriate, allows us to make for cleaner, easier to maintain code
Extending classes - Inheritance

- Recall: by default, all classes are children of java.lang.Object
  - But we can make a new class a child of any class we choose
  - We accomplish this with the extends keyword in our class declaration:

    ```java
    public class GradStudent extends Student
    ```
Extending classes - Inheritance

```java
public class GradStudent extends Student
```

Why would we want to do this?

- Child classes automatically inherit all of the parent’s methods and data
  - If the functionality is mostly the same, we get it for “free”
  - Suppose Students and GradStudents are almost the same, but the prereqCheck method needs to be different for GradStudents
- Write GradStudent as a child class of Student
  - Get all of the methods we’ve already written for Student for “free”
  - Override just the prereqCheck method
  - (And write new constructors – don’t get those for free)
public class GradStudent extends Student {

    public GradStudent(String fname, String lname) {
        // need to write new constructors
        ...
    }

    public boolean checkPrereq(String newClass) {
        // override just this class for grad student behavior!
        ...
    }

}
Extending classes - Polymorphism

- Remember Polymorphism?
  ```java
  Object obj1 = new Student();
  ```

- This works because **Object is the parent class of Student**
  - Still works for **GradStudent** – because **Object is a parent of GradStudent**:
    ```java
    Object obj1 = new GradStudent();
    ```

  - But **GradStudent also has Student as a parent class**:
    ```java
    Student st1 = new GradStudent();
    ```
Extending classes - Polymorphism

Student st1 = new GradStudent();

- When would we want to do this?
  - When most of our processing is the same for Student and GradStudent objects
  - Keep one piece of code that works for any kind of Student
    - Easier to write
    - Easier to maintain