CMSC 131
Object-Oriented Programming I

Inheritance III

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

- Object class
- Early/Late Binding
- `getClass()`, `instanceof`
- Up-casting, down-casting
- `equals` method
News

- http://www.boutiques.com/
  - Sponsored by Google
- Game Class Recommendations
Inheritance: Quick Recap

- Recap:
  - Inheritance is when one class (derived class or subclass) is defined from another class (the base class or superclass).
  - The derived class inherits all the instance variables and the methods from the base class. It can also:
    - Define its own instance variables and methods
    - Redefine methods from the base class, which is called **overriding**
  - A derived class can explicitly refer to entities from the base class using **super**
  - A reference to a derived class can be used anywhere where a reference to the base class is expected
  - By declaring members to be **protected**, a base class makes them accessible to derived classes and further descendents
Class inheritance defines a hierarchy:

- GradStudent is a Student
- Student is a Person
- Person is a ???

There is a class at the top of the hierarchy, called Object. Every class is derived (either directly or indirectly) from Object

- If a class is not explicitly derived from some class, it is automatically derived from Object. The following are equivalent:

  \[
  \text{public class FooBar \{ \ldots \}} \leftrightarrow \text{public class FooBar extends Object \{ \ldots \}}
  \]

- This means that if you write a method with a parameter of type Object, you can call this method with an object reference of any class
- Object is defined in java.lang, and so it is available to all programs
The class **Object** has no instance variables, but defines a number of methods. These include:

- `toString()`: returns a String representation of this object
- `equals(Object o)`: test for equality with another object `o`

Every class you define can, and probably should, overrides these two methods with something that makes sense for your class (hashCode method is also included in the group)

http://download.oracle.com/javase/6/docs/api/java/lang/Object.html
Motivation: Consider the following example:
Faculty carol = new Faculty( "Carol Tuffteacher", "999-99-9999", 1995 );

Person p = carol;
System.out.println(p.toString());

Q: Should this call Person’s toString or Faculty’s toString?
A: There are good arguments for either choice:
Early (static) binding: The variable p is declared to be of type Person. Therefore, we should call the Person’s toString.
Late (dynamic) binding: The object to which p refers was created as a “new Faculty”. Therefore, we should call the Faculty’s toString.

Pros and cons: Early binding is more efficient, since the decision can be made at compile time. Late binding provides more flexibility.
Java uses late binding (by default): so Faculty toString is called.
(Note: C++ uses early binding by default.)
Polymorphism

- Java’s **late binding** makes it possible for a single reference variable to refer to objects of many different types. Such a variable is said to be **polymorphic** (meaning having many forms)

- **Example**: Create an array of various university people and print

  ```java
  Person[ ] list = new Person[3];
  list[0] = new Person("Col. Mustard", "000-00-0000");
  list[1] = new Student("Ms. Scarlet", "111-11-1111", 1998, 3.2);
  list[2] = new Faculty("Prof. Plum", "222-22-2222", 1981);
  for ( int i = 0; i < list.length; i++ )
      System.out.println( list[i].toString( ) );
  
  Output:
  [Col. Mustard] 000-00-0000
  [Ms. Scarlet] 111-11-1111 1998 3.2
  [Prof. Plum] 222-22-2222 1981
  
- **What type is list[i]?** It can be a reference to any object that is derived from Person. The appropriate toString will be called
Disabling Overriding with “final”

- Sometimes you do not want to allow method overriding
  
  **Correctness**: Your method only makes sense when applied to the base class. Redefining it for a derived class might break things
  
  **Efficiency**: Late binding is less efficient than early binding. You know that no subclass will redefine your method. You can force early binding by disabling overriding

- **Example**: The class **Object** defines the following method:
  
  `getClass()`: returns a description of a class. You can test whether two objects `x` and `y` are of the same class with:

  ```java
  if ( x.getClass() == y.getClass() ) ...
  ```

  This is a very useful function. But clearly we do not want arbitrary classes screwing around with it

- We can disable overriding by declaring a method to be “final”
Disabling Overriding with "final"

- **final**: Has two meanings, depending on context:
  - Define **symbolic constants**:
    ```java
    public static final int MAX_BUFFER_SIZE = 1000;
    ```
  - Indicate that a method **cannot be overridden by derived classes**
    ```java
    public class Parent {
        ...
        public final void someMethod() { ... }
    }
    
    public class Child extends Parent {
        ...
        public void someMethod() { ... }
    }
    ```
    - Subclasses cannot override this method
    - Illegal! someMethod is final in base class.
Recap:
• Inheritance is when one class (derived class or subclass) is defined from another class (the base class or superclass). The derived class inherits variables and methods from the base class and can override their definitions or define its own
• A reference to a derived class can be used anywhere where a reference to the base class is expected
• All objects are derived (directly or indirectly) from Object
• Java uses late (or dynamic) binding, which means that the method that is called depends on an object’s actual type, and not the declared type of the referring variable
• Late binding and inheritance allows you to create polymorphic variables. The behavior (based on method calls) depends on what the variable refers to
Objects in Java can access their type information dynamically.

**getClass( ):** Returns a representation of the class of any object.

```java
Person bob = new Person( ... );
Person ted = new Student( ... );
```

```java
if ( bob.getClass( ) == ted.getClass( ) )  // false (ted is really a Student)
```

**instanceof:** You can determine whether one object is an instance of (e.g., derived from) some class using `instanceof`. Note that it is an operator (!) in Java, not a method call.

```java
if ( bob instanceof Person )  // true
if ( ted instanceof Student )  // true
if ( ted instanceof Person )  // true
if ( bob instanceof Student )  // false
```

```java
Faculty carol = new Faculty( ... );
if ( carol instanceof Person )  // true
if ( carol instanceof Student )  // Illegal! Doesn’t compile
```
Up-casting and Down-casting

We have already seen that we can assign a derived class reference anywhere that a base class is expected.

**Upcasting**: Casting a reference to a base class (casting up the inheritance tree). This is done automatically and is always safe.

**Downcasting**: Casting a reference to a derived class. This may not be legal (depending on the actual object type). You can force it by performing an explicit cast.

```java
Person bob = new Person( ... );
Person ted = new Student( ... );
Student carol = new Student( ... );
GradStudent alice = new GradStudent( ... );

bob = ted; // okay: ted is a Person
carol = ted; // compile error! ted may not be a Student
carol = (Student) ted; // okay: ted is a Student
alice = (GradStudent) ted; // run-time error! ted isn’t a GradStudent
```
Safe Downcasing

- Illegal downcasting results in a `ClassCastException` run-time error.
- **Q:** Can we check for the **legality** of a cast before trying it?
- **A:** Yes, using `instanceof`.
- **Example:** Suppose that we want to store a list of university people references an `ArrayList`. We then want to print the GPA’s of all the students.

**Recall:** the following `ArrayList` methods:
- `size()`: Returns the size of the list
- `add()`: Adds element to the end of the list
- `get()`: Returns a reference to the object at position `i`
- As elements are removed from the list, they must be **downcast** from `Person` to `Student`, but this can only be done if the object really is a Student
Safe Downcasting

ArrayList<Person> list = new ArrayList<Person>();
list.add(new Person("Bender", "000");
list.add(new Student("Fry", "111", 1999, 1.2));
list.add(new Student("Leela", "222", 2999, 3.8));
list.add(new Faculty("Farnsworth", "333", 2841));
list.add(new Student("Nibbler", "444", 1000, 4.0));

for (int i = 0; i < list.size(); i++) {
    Person obj = list.get(i);
    if (obj instanceof Student) {
        Student s = (Student) obj;
        System.out.println(s.getName() + "'s GPA is " + s.getGpa());
    }
}
equals: The Right Way

- We defined an `equals` method for our various classes. Here is an example from Student:
  ```java
  public boolean equals(Student s) {
    return super.equals(s) &&
    admitYear == s.admitYear &&
    gpa == s.gpa;
  }
  ```
- Although this will correctly compare two students, there will be problems if you try to compare a `Student` with other members of the Person hierarchy.
Example: Write a method that looks up a person (Person, Student, or Faculty) in an ArrayList containing university person objects.

```java
public static boolean find(Person p, ArrayList<Person> list) {
    for (int i = 0; i < list.size(); i++) {
        if (p.equals(list.get(i))) return true;
    }
    return false;
}
```

Suppose that we have: Person p = new Student( ... ); find(p, list); Which equals method will be called here?

- **Person equals()**? p is declared to be type Person
- **Student equals()**? Late binding uses actual object type (Student)
- **Object equals()**?
equals: The Right Way

- **Answer:** **Person equals** is called
- **Huh?** Isn’t this a case of method overriding? Since p is a Student, we should call Student equals?
- **What are Java’s options?**
  - `class Student { ... boolean equals( Student s ) ...}
  - `class Person { ... boolean equals( Person p ) ... }
  - `...
  - `class Object { ... boolean equals( Object o ) ... }
- All of these methods take different parameter types
  - This is **not** a case of method **overriding**
  - This is a case of method **overloading**
- Java selects the option that **best matches** the parameter type, which is **Person** so Person equals( ) is called
equals: The Right Way

- What is the **right way** to define equals? It should:
  - Take an argument of type `Object`, not `Student`.
  - Check that the argument is **non-null** (just for robustness).
  - Check that the argument refers to an actual `Student`. (We could define `equals` less strictly, but we won’t.)
  - Proceed with the other equality checks.

```java
public boolean equals( Object o ) {
    if (o == null) return false;
    else if (getClass() != o.getClass()) return false;
    else {
        Student s = (Student) o;
        return super.equals(s) &&
                admitYear == s.admitYear &&
                gpa == s.gpa;
    }
}
```
equals: Options

/* Option 1 (we use in cmsc131) */
public boolean equals(Object o) {
  if (o == null)
    return false;
  if (getClass() != o.getClass())
    return false;
  A s = (A) o;
  /* Comparison based on A fields */
}

/* Option 2 */
public boolean equals(Object o) {
  if (o == this)
    return true;
  if (!(o instanceof A))
    return false;
  A s = (A) o;
  /* Comparison based on A fields */
}

The are some cases where they may produce different results.