Inheritance IV

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

- getClass/instanceof
- Upcasting/downcasting
- equals method
Inheritance: Yet Another 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 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
**getClass and instanceof**

- 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( ... );
  
  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

- **Example**: InstanceGetClass.java
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

• Illegal downcasting results in a `ClassCastException` run-time error

• Example: `UpCastingDownCasting.java`
Safe Downcasting

• **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.
• **Example:** SafeDownCasting.java
equals: The Right Way

- We defined an `equals` methods for our various classes. Here is an example from `Student`:

```java
public boolean equals(Student s) {
    if (s == null) {
        return false;
    } else if (s == this) {
        return true;
    } else {
        /* Notice call of person's equals */
        return super.equals(s) && admitYear == s.admitYear;
    }
}
```

- Although this will correctly compare two Student objects, there will be problems if you try to compare a `Student` with other members of the Person hierarchy
equals: The Right Way

**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()**?

**Example**: FindLauraIncorrect.java
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?**
  
  ```java
  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
• **Example:** package correctEqualsMethods
equals: Options

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

• In option #2 instanceof handles a null parameter
• There are some cases where option #1 and option #2 produce different results
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. **getClass()** is a final method
• 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.