Inheritance

Inheritance and Private

- Student objects inherit all the private data (name and idNum)
- However, private members of the base class cannot be accessed directly

Example: (Recall that name is a private member of Person.)

```java
public class Student extends Person {
    ...
    public void someMethod() { name = "Mr. Foobar"; } // Illegal
    public void someMethod2() { setName( "Mr. Foobar" ); } // Okay
}
```

Why is this? After you have gone to all the work of setting up privacy, it wouldn’t be fair to allow someone to simply extend your class and now have access to all the private information.

Protected and Package Access

- The derived class cannot access private base elements. So can a base class grant any special access to its derived classes?

  Special Access for Derived Classes:
  - Protected: When a class element (instance variable or method) is declared to be protected (rather than public or private) it is accessible:
    - To any derived class (and hence to all descendents), and
    - To any class in the same package

  Example:
  ```java
  protected void someMethod() { ... } // has protected access
  ```

  Package: When a class element is not given any access modifier (private, public, protected) it is said to have package access. It is accessible:
  - To any class in the same package

  Example:
  ```java
  void someOtherMethod() { ... } // has package access
  ```

Access to Base Class Elements

- Which should I use? : private, protected, package, or public?

  Public:
  - Methods of the object’s public interface
  - Constant instance variables (static final)

  Private:
  - Instance variables (other than constants)
  - Internal helper/utility methods (not intended for use except in this class)

  Protected/Package:
  - Internal helper/utility methods (for use in this class and related classes)
  - Note: Some style gurus discourage the use of protected. Package is safer, since any resulting trouble can be localized to the current package
The Class Hierarchy and Object

• **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:

    ```java
    public class FooBar {
        …
    }
    ```
    ```java
    public class FooBar extends Object {
        …
    }
    ```

• 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.

Early and Late Binding

• **Motivation**: Consider the following example:

  ```java
  Faculty carol = new Faculty("Carol Tuftteacher", "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++)
  ```java
  System.out.println(list[i].toString());
  ```

  ```java
  Output:
  [Col. Mustard] 000-00-0000
  [Ms. Scarlet] 111-11-1111 1998 3.2
  [Prof. Plum] 222-22-2222 1981
  ```

• **What type is list[2]??** It can be a reference to any object that is derived from **Person**. The appropriate **toString** will be called.
**get Class 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

**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 in 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(i)`: Adds element to the end of the list.
    - `get(i)`: 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

**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

**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;
}
```

* 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?
    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 overloading
  * 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: Options

```java
// Option #1 (we use in cmc131) */
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 */
    A s = (A) obj;
    /* Comparison based on A fields */
    */

// Option #2 */
public boolean equals(Object obj) {
    if (obj == this) return true;
    if (!((obj instanceof A)) 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:
  
  ```java
  public Class getClass() {
    return new Class();
  }
  ```
  
  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”

Inheritance versus Composition

• Inheritance is but one way to create a complex class from another. The other way is to explicitly have an instance variable of the given object type. This is called composition

  Common Object

  ```java
  public class ObjA {
    public void methodA() {
      // ... method A
    }
  }
  ```

  Derive a new class from ObjA

  ```java
  public class ObjB extends ObjA {
    public void methodA() {
      // call a method from ObjA
      // ... method A
    }
  }
  ```

  Add ObjA as an instance variable.

  ```java
  public class ObjC {
    private ObjA objA;
    public void methodA() {
      // call a method from ObjA and ObjC
      // ... method A
    }
  }
  ```

• When should I use inheritance vs. Composition?
  
  * ObjB “is a” ObjA: in this case use inheritance
  * ObjB “has a” ObjA: in this case use composition

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 final void someMethod() {
      // ... method
    }
    ```

    Subclasses cannot override this method

    ```java
    public final void someMethod() {
      // illegal method
    }
    ```

    Illegal! someMethod is final in base class.

Inheritance versus Composition

• University parking lot permits: A parking permit object involves a university Person and a lot name (“4”, “11”, “XX”, “Home Depot”)

  ```java
  public class Permit extends Person {
    String lotName;
    public Permit(String lotName) {
      this.lotName = lotName;
    }
    // ... method
  }
  ```

  Inheritance:

  ```java
  public class Permit extends Person {
    String lotName;
    public Permit(String lotName) {
      this.lotName = lotName;
    }
    // ... method
  }
  ```

  Composition:

  ```java
  public class Permit {
    String lotName;
    public Permit(String lotName) {
      this.lotName = lotName;
    }
    // ... method
  }
  ```

  ```java
  public class Permit {
    String lotName;
    public Permit(String lotName) {
      this.lotName = lotName;
    }
    // ... method
  }
  ```

  • Which to use?
  
  A parking permit “is a” person? Clearly no
  A parking permit “has a” person? Yes, because a Person is one of the two entities in a a permit object
  
  So composition is the better design choice here
  
  • Prefer Composition over Inheritance
  
  When in doubt or when multiple choices available, prefer composition over inheritance
Review of Overloading and Overriding

• Before discussing interfaces, let’s review some elements of method overloading and overriding

• When overriding a method the subclass method prototype must match exactly the prototype of the superclass (same name, same return type, same arguments)

• You may change access specifier (public, private, protected), but derived classes cannot decrease the visibility
  * Example: done() method in Object class

Example: You be the Compiler

```
public class Base {
    protected void doneMethod(int a) { ... }
}
public class Derived extends Base {
    public void doneMethod(int a) { ... }
    public int doneMethod(int a) { ... }
    public void doneMethod(double d) { ... }
}
```

21

22

Base class
Derived class
Overloading: with increased visibility
Error duplicate method declaration
Overloading
calls Base:doneMethod(int )
calls Derived:doneMethod(int )
Error Since d is declared Base, this attempts to call the overridden method doneMethod(int ). But the argument is of the wrong type.
calls Derived:doneMethod(double)