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.

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:

    public class FooBar ( ... ) ↔ 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.
The class **Object** has no instance variables, but defines the 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, override these two methods with something that makes sense for your class.

### Early and Late Binding

Consider the following example:

```java
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. Defining 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:

```
glass( ): returns a description of a class. You can test whether two objects x and y are of the same class with:

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:

- To define *symbolic constants*:

  ```java
  public static final int MAX_BUFFER_SIZE = 1000;
  ```

- It means that a method cannot be overridden by derived classes.

  ```java
  public class Parent {
    ...
    public final void someMethod() { ... } // subclasses cannot override
  }

  public class Child extends Parent {
    ...
    public void someMethod() { ... } // Illegal!
  }
  ```

### Polymorphism and Abstract Methods

**Polymorphism**: is allows us to write generic methods that leave the low-level details for the subclasses to implement.

**Example**: Typical graphics drawing program.

- Define a base class, *Shape*, and derive various subclasses for specific shapes. Each subclass defines its own method `drawMe()`.

  ```java
  public class Shape {
    public void drawMe() { ... } // generic drawing method
  }

  public class Circle extends Shape {
    public void drawMe() { ... } // draws a Circle
  }

  public class Rectangle extends Shape {
    public void drawMe() { ... } // draws a Rectangle
  }
  ```
Polymorphism and Abstract Methods

Example: Graphics drawing program.
- The entire picture consists of an array shapes of type Shape[ ].
- To draw the picture, we invoke drawMe( ) for all the shapes.

    Shape[ ] shapes = new Shape[...];
    shapes[0] = new Circle( ... );
    shapes[1] = new Rectangle( ... );
    ...
    for ( int i = 0; i < shapes.length; i++ )
        shapes[i].drawMe( );

Polymorphism and Abstract Methods

Problem: The Shape object does not represent a specific shape. So how do we implement Shape’s drawMe( ) method?

- Draw some special "undefined shape".
- Just ignore the operation.
- Issue an error message or exception.

None of these solutions is very natural.

Better solution: Abstract methods/classes

- What you really want is for the compiler to know that Shape is incomplete class.
- It declares but does not define some methods. These must be implemented in the derived classes.

Abstract Method: a method that contains no body. It is just a placeholder. The derived classes will provide the actual implementation.
Polymorphism and Abstract Methods

Abstract Method:

- It behaves much like a method in an interface. You give a signature, but no body. The class descendents provide the implementation.

- An abstract method cannot be final. (Obviously...since it must be overridden by a descendent class, and final would prevent this.)

Abstract Class:

- A class with at least one abstract method is an abstract class.
- You must include the modifier abstract in the class heading.
  public abstract class Shape { ... }

- An abstract class is a incomplete:
  
  • It cannot be created using "new"
    Shape s = new Shape( ... );   // Illegal! Shape is abstract

  • ...but you can create concrete shapes (Circle, Rectangle) and assign them to variables of type Shape.
    Shape s = new Circle( ... );

Example: Shapes

```java
public abstract class Shape {
    private int color;
    Shape ( int c ) { color = c; }  
    public abstract void drawMe( );
}

public class Circle extends Shape {
    private double radius;
    public Circle( int c, double r ) { ... details omitted ... }
    public void drawMe( ) { ... Circle drawing code goes here ... }
}

public class Rectangle extends Shape {
    private double height;
    private double width;
    public Rectangle( int c, double h, double w ) { ... details omitted ... }
    public void drawMe( ) { ... Rectangle drawing code goes here ... }
}
```
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.

- When a method in a base class is not provided, the method and class are said to be abstract. It must be implemented in a derived class.

getClassName 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 a operator (!) in Java, and 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

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 Downcasting

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:

- `int size( )`: Returns the size of the list.
- `void add( Object x )`: Adds x to the end of the list.
- `Object get( int i )`: Returns a reference to the object at position i.

As elements are removed from the list they must be downcast from Object to Student, but this can only be done if the object really is a Student.
Safe Downcasting

```java
ArrayList list = new ArrayList();
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++ ) {
    Object o = list.get( i );
    if ( o instanceof Student ) {
        Student s = (Student) o;
        System.out.println( s.getName() + "'s GPA is " + s.getGpa() );
    }
}
```

Fry's GPA is 1.2
Leela's GPA is 3.8
Nibbler's GPA is 4.0

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 ) {
    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.
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 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( ) ? ArrayList.get( ) returns an Object reference

equals: The Right Way

Answer: Object equals( ) is called (Surprise!)

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 **Object**. So Object 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;
    }
}
```

Copying and Inheritance

We have seen how to make copies of objects using the copy constructor:

```java
Student amy = new Student( "Amy", "555", 3002, 3.5 );
Student amyCopy = new Student( amy );
```

```java
public Student( Student s ) {
    super( s );
    admitYear = s.admitYear;
    gpa = s.gpa;
}
```

As with equals, we will run into problems if we try to use this to copy other types of Person objects, but the problems are of a different nature here.
Copying and Inheritance

Let's suppose that we want to construct an array of Person objects (of various types) and write general purpose copy method.

```java
Person list[ ] = new Person[5];
list[0] = new Person( "Bender", "000" );
list[1] = new Student( "Fry", "111", 1999, 1.2 );
// ... (also Leela and Farnsworth)
list[4] = new Student( "Nibbler", "444", 1000, 4.0 );
Person list2[ ] = copy( list );

public static Person[ ] badCopy( Person[ ] list ) {
    Person[ ] list2 = new Person[ list.length ];
    for ( int i = 0; i < list.length; i++ ) {
        list2[i] = new Person( list[i] );
    }
    return list2;
}
```

Copying and Inheritance

**Problem:** The constructor call "Person ( list[i] )" invokes the Person constructor, irrespective of the actual type of list[i].

```java
public static Person[ ] badCopy( Person[ ] list ) {
    Person[ ] list2 = new Person[ list.length ];
    for ( int i = 0; i < list.length; i++ ) {
        list2[i] = new Person( list[i] );
    }
    return list2;
}
```

Huh? Isn't Java smart enough to know that I really wanted it to apply the copy constructor of the actual type of list[i]?

**Of course not:** If you call a method, it is not Java's job to read your mind and change the call to an entirely different method. (Overriding is different, because the methods have the same name and same argument types, so it must pick one.)
Copying: The Right Way

Solution: We use inheritance and method overriding, so Java will call the correct copying method.

clone:

- We define a method clone, which will be overridden for each class.
- The job of clone is to return a duplicate copy of this object.
- clone is overridden from class Object and returns an Object. Thus, we will need to downcast its result to Person.
- This works because (by late binding) each type (Person, Student, Faculty) calls its own clone, and so makes the proper copy.

```java
public static Person[] goodCopy(Person[] list) {
    Person[] list2 = new Person[list.length];
    for (int i = 0; i < list.length; i++) {
        list2[i] = (Person) list[i].clone();
    }
    return list2;
}
```

Defining Clone

How to define clone?

- There is an officially sanctioned way to define clone, but it will take too long to explain. (It involves the Cloneable interface.)

- Instead, we will just define it in a quick-and-dirty manner, using our copy constructors.

```java
public class Person {
    ...
    public Object clone() {
        return new Person(this);
    }
}

public class Student extends Person {
    ...
    public Object clone() {
        return new Student(this);
    }
}
```
Summary of equals and clone

General Warning:

- When dealing with class inheritance, you may not know the actual types of objects that you are manipulating. This complicates your job as a programmer.

- Set up your method parameters to be of the most general type that is applicable to your method (e.g. Object or Person rather than Student or Faculty).

- Whenever dealing with references to objects in the class hierarchy, use method overriding to produce the proper behavior. (E.g. using clone rather than calling a copy constructor directly).

- Avoid downcasting, but if you need to downcast to a particular type, use getClass and/or instanceof to make sure your cast is valid.