Inheritance, Polymorphism, and Interfaces

Advanced Placement Computer Science

Inheritance and Polymorphism

What’s past is prologue.

Don’t write it twice — write it once and reuse it.

Mike Scott
The University of Texas at Austin
Main Tenants of OO Programming

- Encapsulation
  - abstraction, information hiding, responsibility driven programming

- Inheritance
  - code reuse, specialization "New code using old code."

- Polymorphism
  - do X for a collection of various types of objects, where X is different depending on the type of object
  - "Old code using new code."
Explanation of Inheritance

¬ 1 of the fundamental principles of OOP
  – allows code reuse
¬ Models the IS-A relationship
  – a student is-a person
  – an undergraduate is-a student
  – a rectangle is-a shape
  – a rook is-a piece
¬ Contrast with the Has-A relationship (or uses a)
  – a student has-a name
  – a rook has-a position
  – a Stack uses a List
¬ Is-a relationships indicate inheritance, has-a relationships indicate composition (fields)
Nomenclature of Inheritance

- The `extends` keyword is used to specify which preexisting class a new class is inheriting from:
  ```java
  public class Student extends Person
  ```

- Person is said to be
  - the parent class of Student
  - the super class of Student
  - the base class of Student
  - an ancestor of Student

- Student is said to be
  - a child class of Person
  - a sub class of Person
  - a derived class of Person
  - a descendant of Person
The Mechanics of Inheritance

- Java is a pure object oriented language
- all code is part of some class
- all classes, except one, must inherit from exactly one other class

The `Object` class is the `cosmic super class`
- The `Object` class does not inherit from any other class
- The `Object` class has several important methods: `toString`, `equals`, `hashCode`, `clone`, `getClass`

Implications:
- all classes are descendants of `Object`
- all classes, and thus all objects, have a `toString`, `equals`, `hashCode`, `clone`, and `getClass` method
  - `toString`, `equals`, `hashCode`, `clone` normally overridden
Inheriting from a Class

- If a class header does not include the extends clause the class extends the `Object` class by default

  ```java
  public class Card
  - Object is an ancestor to all classes
  - it is the only class that does not extend some other class
  ```

- A class extends exactly one other class
  - extending two or more classes is *multiple inheritance*. Java does not support this directly, rather it uses *Interfaces*. 
Implications of Inheritance

- The sub class gains all of the behavior (methods) and data regarding state (instance variables) of the super class and all ancestor classes.

- Sub classes can:
  - add new fields
  - add new methods
  - override existing methods (change behavior)

- Sub classes may not
  - remove fields
  - remove methods

- Note, even though an object may have instance variables from its parent they may not be accessible by the code of the child class if the fields are private.
A String object

<table>
<thead>
<tr>
<th>Fields from Object class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance variables</td>
</tr>
<tr>
<td>declared in Object</td>
</tr>
<tr>
<td>Fields from String class</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Instance Variables</td>
</tr>
<tr>
<td>declared in String</td>
</tr>
</tbody>
</table>

Behaviors (methods) from String class and Object class.
Access Modifiers and Inheritance

- public
  - accessible to all classes

- private
  - accessible only within that class. Hidden from all sub classes.

- protected
  - accessible by classes within the same package and all descendant classes

- Instance variables should be private

- protected methods are used to allow descendant classes to modify instance variables in ways other classes can't
Instance Variables - Private or Protected

- Why is it good design to make the instance variables of an object private instead of protected?
- protected also allows classes in the same package to access the data
  - a class in a package does not necessarily inherit from other classes in the same package
- What if when the data changes something else must be done? How would the descendant classes know to do the required changes?
  - Excellent example in the MBCS
MBCS Example

```java
public class Fish {
    private Location myLoc;
    // Why private?
    // What if subclasses override move() and need to change
    // the Location? Don't subclasses need access to it?
    private Environment theEnv;
    // If a Fish changes its location theEnv must be updated

    protected void changeLocation(Location newLoc) {
        // Change location and notify Environment
        Location oldLoc = location();
        myLoc = newLoc;
        environment().recordMove(this, oldLoc);

        // object is again at location myLoc in environment
    }
}
```

Making myLoc private and forcing sub classes to call changeLocation to alter the location of a fish guarantees the environment is correctly updated with the now location.
Shape Classes

- Declare a class called ClosedShape
  - assume all shapes have x and y coordinates
  - override Object's version of toString

- Possible sub classes of ClosedShape
  - Rectangle
  - Circle
  - Ellipse
  - Square

- Possible hierarchy
  ClosedShape -> Rectangle -> Square
public class ClosedShape
{
  private int iMyX;
  private int tMyY;

  public ClosedShape()
  {
    this(0,0);
  }

  public ClosedShape (int x, int y)
  {
    iMyX = x;
    iMyY = y;
  }

  public String toString()
  {
    return "x: " + iMyX + " y: " + iMyY;
  }

  public int getX(){ return iMyX; }
  public int getY(){ return iMyY; }
}

// Other methods not shown
Constructors

- Constructors handle initialization of objects.
- When creating an object with one or more ancestors (every type except Object) a chain of constructor calls takes place.
- The reserved word `super` may be used in a constructor to specify which of the parent's constructors to call:
  - must be first line of constructor.
- If no parent constructor is explicitly called the default, 0 parameter constructor of the parent is called:
  - if no default constructor exists a syntax error results.
- If a parent constructor is called another constructor in the same class may not be called:
  - no `super(); this();` allowed. One or the other, not both.
  - good place for an initialization method.
A Rectangle Constructor

public class Rectangle extends ClosedShape
{
    private int iMyWidth;
    private int iMyHeight;

    public Rectangle(int width, int height, int x, int y)
    {
        super(x, y);
        // calls the 2 int constructor in
        // ClosedShape
        iMyWidth = width;
        iMyHeight = height;
    }

    // other methods not shown
}

Inheritance, Polymorphism, and Interfaces
A Rectangle Class

```java
public class Rectangle extends ClosedShape {
    private int iMyWidth;
    private int iMyHeight;

    public Rectangle()
    {
        this(0, 0);
    }

    public Rectangle(int width, int height)
    {
        this(width, height, 0, 0);
    }

    public Rectangle(int width, int height, int x, int y)
    {
        super(x, y);
        iMyWidth = width;
        iMyHeight = height;
    }

    public String toString()
    {
        return super.toString() + " width: " + iMyWidth
                   + " height: " + iMyHeight;
    }
}
```
Initialization method

```java
public class Rectangle extends ClosedShape {
    private int iMyWidth;
    private int iMyHeight;

    public Rectangle() {
        init(0, 0);
    }

    public Rectangle(int width, int height) {
        init(width, height);
    }

    public Rectangle(int width, int height, int x, int y) {
        super(x, y);
        init(width, height);
    }

    private void init(int width, int height) {
        iMyWidth = width;
        iMyHeight = height;
    }
}
```
Overriding methods

- any method that is not `final` may be overridden by a descendant class
  - overriding is a replacement of a behavior
  - overloading is the addition of a behavior
- same signature as method in ancestor
- may not reduce visibility
- may use the original method if simply want to add more behavior to existing
  - also called *partial overriding*
- The Rectangle class
  - adds data, partially overrides `toString`
The Keyword `super`

- `super` is used to access something (any protected or public field or method) from the super class that has been overridden.
- Rectangle's `toString` makes use of the `toString` in `ClosedShape` my calling `super.toString()`
- without the super calling `toString` would result in infinite recursive calls.
- Java does not allow nested supers
  
  `super.super.toString()`
  
  results in a syntax error even though technically this refers to a valid method, `Object's toString`
- Rectangle `partially` overrides `ClosedShape's toString`
What Can Rectangles Do?

Rectangle r1 = new Rectangle();
Rectangle r2 = new Rectangle(10, 15);
Rectangle r3 = new Rectangle(10, 15, 2, 3);
System.out.println( r1 );
System.out.println( r2 );
System.out.println( r3 );
int a = r1.getX() + r1.getY();
ClosedShape s = new Rectangle(5, 10, 3, 4);
System.out.println( s.toString() );
a += s.getX();
ClosedShape[] sList = new ClosedShape[3];
sList[0] = new ClosedShape(5, 10);
sList[1] = new Rectangle(10, 25, 10, 7);
sList[2] = r2;
for(int i = 0; i < sList.length; i++)
    System.out.println( sList[i].toString() );
The role of `final` in Inheritance

- A class may be declared as `final`
  - that class may not be extended
- A method in a class may be declared as `final`
  - that method may not be overridden
  - guarantees behavior in all descendants
  - can speed up a program by allowing static binding (binding or determination at compile time what code will actually be executed)
Abstract Classes and Methods

- An abstract class is used to define a class to gather together behaviors but:
  - an object of that type never exists and can never be created or instantiated.
  - a Shape or a Mammal

- a method may be declared abstract in its header, after visibility modifier
  - no body to the method
  - all derived classes must eventually implement this method (or they must be abstract as well)
  - any class with 1 or more abstract methods must be an abstract class
An Abstract ClosedShape Class

```java
public abstract class ClosedShape {
    private int iMyX;
    private int iMyY;

    public ClosedShape() {
        this(0, 0);
    }

    public ClosedShape (int x, int y) {
        iMyX = x;
        iMyY = y;
    }

    public String toString() {
        return "x: " + iMyX + " y: " + iMyY;
    }

    public abstract int getArea();

    public int getX(){ return iMyX; }
    public int getY(){ return iMyY; }
}
```
Classes that Inherit from ClosedShape

- Rectangle inherits from ClosedShape
- What if Rectangle is unchanged
- Problem: If I have a Rectangle object what happens when I call:

```java
Rectangle r = new Rectangle(10, 5, 0, 0);
System.out.println(r.getArea());
```

- Undefined behavior = BAD
- As is the Rectangle class would not compile
- If a class inherits from an abstract class that has abstract methods those methods must be defined in the child or the child must be abstract as well
Implementing getArea()

```java
public class Rectangle extends ClosedShape
{
    private int iMyWidth;
    private int iMyHeight;

    public int getArea()
    {
        return iMyWidth * iMyHeight;
    }

    // other methods not shown
}

public class Square extends Rectangle
{
    public Square()
    {
    }

    public Square(int side)
    {
        super(side, side);
    }

    public Square(int side, int x, int y)
    {
        super(side, side, x, y);
    }
}
```
A Circle Class

public class Circle extends ClosedShape
{
   int iMyRadius;

   public Circle()
   {
      this(1);  
   }

   public Circle(int radius)
   {
      iMyRadius = radius;
   }

   public Circle(int radius, int x, int y)
   {
      super(x,y);
      iMyRadius = radius;
   }

   public int getArea()
   {
      return Math.PI * iMyRadius * iMyRadius;
   }

   public String toString()
   {
      return super.toString() + " radius: " + iMyRadius;
   }
}
Polymorphism in Action

```java
public class UsesShapes
{
    public static void go()
    {
        ClosedShape[] sList = new ClosedShape[10];
        int a, b, c, d;
        int x;
        for(int i = 0; i < 10; i++)
        {
            a = (int)(Math.random() * 100);
            b = (int)(Math.random() * 100);
            c = (int)(Math.random() * 100);
            d = (int)(Math.random() * 100);
            x = (int)(Math.random() * 3);
            if (x == 0)
                sList[i] = new Rectangle(a, b, c, d);
            else if (x == 1)
                sList[i] = new Square(a, c, d);
            else
                sList[i] = new Circle(a, c, d);
        }
        int total = 0;
        for(int i = 0; i < 10; i++)
        {
            total += sList[i].getArea();
            System.out.println(sList[i]);
        }
    }
}
```
The Kicker

- We want to expand our pallet of shapes
- Triangle could also be a sub class of ClosedShape.
  - it would *inherit* from ClosedShape
    
    ```java
    public int getArea()
    { return 0.5 * iMyWidth * iMyHeight; }
    ```

- What changes do we have to make to the code on the previous slide for totaling area so it will now handle Triangles as well?
- Power.
Genericity

- One of the goals of OOP is the support of code reuse to allow more efficient program development.

- If an algorithm is essentially the same, but the code would vary based on the data type, genericity allows only a single version of that code to exist.
  - Some languages support genericity via *templates*.
  - In Java, Polymorphism and the inheritance requirement along with interfaces are used.
  - The remove duplicates example.
/**
   * pre: list != null, for all n such that 0 <= n <
   * list.length, list[n] != null
   * post: return an array of Objects with all duplicates in
   * list removed, i.e. each element in returned array
   * is unique
   */
   public static Object[] removeDups(Object[] list)
   {
       int numUnique = 0;
       Object[] temp = new Object[list.length];
       boolean unique;
       int index;
       for(int i = 0; i < list.length; i++)
       {
           unique = true;
           index = 0;
           while(unique && index < numUnique)
           {
               unique = list[i].equals( temp[index] );
               index++;
           }
           if(unique)
           {
               temp[numUnique] = list[i];
               numUnique++;
           }
       }
       Object[] result = new Object[numUnique];
       System.arraycopy(temp, 0, result, 0, numUnique);
   }
Object Variables

Rectangle r = new Rectangle(10, 20);
ClosedShape s = r;
System.out.println("Area is " + s.getArea());

- The above code works if Rectangle extends ClosedShape
- An object variable may point to an object of its base type or a descendant in the inheritance chain
  - The is-a relationship is met. A Rectangle object is-a shape so s may point to it
- This is a form of polymorphism and is used extensively in the Java Collection classes
  - Vector, ArrayList are lists of Objects
Type Compatibility

- polymorphism allows s to point at a Rect object but there are limitations
- The above code will not compile
- Statically, s is declared to be a shape
  - no changeWidth method in Shape class
  - must cast s to a Rectangle

```java
Rectangle r = new Rectangle(5, 10);
ClosedShape s = r;
s.changeWidth(20); // syntax error

Rectangle r = new Rectangle(5, 10);
Shape s = r;
((Rectangle)s).changeWidth(20); //Okay
```
Problems with Casting

- The following code compiles but a Class Cast Exception is thrown at runtime

```java
Rectangle r = new Rectangle(5, 10);
Circle c = new Circle(5);
Shape s = c;
((Rectangle)s).changeWidth(4);
```

- Casting must be done carefully and correctly
- If unsure of what type object will be the use the `instanceof` operator or the `getClass()` method

`expression instanceof ClassName`
Multiple Inheritance

- Inheritance models the "is-a" relationship between real world things
- one of the benefits is code reuse, completing programs faster, with less effort
- in the real world a thing can have "is-a" relationships with several other things
  - a Graduate Teaching Assistant is-a Graduate Student. Graduate Teaching Assistant is-a Faculty Member
  - a Student is-a Person. a Student is a SortableObject
The Power of Polymorphism

- Polymorphism allows collections, such as ArrayList, to hold anything
  - genericity, C++ achieves via templates

- Polymorphism also allows a method to work on multiple types
  - create a sorting method that accepts arrays of SortableObjects
  - it can sort anything
  - again, C++ has templated functions

- Java uses a limited form of multiple inheritance
Interfaces

- A Java interface is a "pure abstract class".
  - Design only, no implementation.
- Interfaces are declared in a way similar to classes but
  - consist only of public abstract methods
  - public final static fields
- A Java class extends exactly one other class, but can implement as many interfaces as desired
Common Interfaces in Java

One of the most interesting interfaces is: Comparable

```java
package java.lang

public interface Comparable
{
    public int compareTo( Object other );
}
```

- compareTo should return an int <0 if the calling object is less than the parameter, 0 if they are equal, and an int >0 if the calling object is greater than the parameter
Implementing an Interface

public class Card implements Comparable
{
    public int compareTo(Object otherObject)
    {
        Card other = (Card)otherObject;
        int result = iMySuit - other.iMySuit;
        if(result == 0)
            result = iMyValue - other.iMyValue;
    }
    // other methods not shown
}

- unlike the equals method no steps to prevent a miscast
- If a class declares that it will implement an interface, but does not provide an implementation of all the methods in that interface, that class must be abstract
Polymorphism Again

```java
public static SelectionSort(Comparable[] list)
{
    Comparable temp;
    int small;
    for(int i = 0; i < list.length - 1; i++)
    {
        small = i;
        for(int j = i + 1; j < list.length; j++)
        {
            if(list[j].compareTo(list[small]) < 0)
                small = j;
        } // end of j loop
        temp = list[i];
        list[i] = list[small];
        list[small] = temp;
    } // end of i loop
}
```