Topic 4
Inheritance

“Question: What is the object oriented way of getting rich?
Answer: Inheritance.”

Features of OO Programming

- Encapsulation
  - abstraction
  - information hiding
  - breaking problem up based on data types
- Inheritance
  - code reuse
  - specialization
  - "New code using old code."

Encapsulation

- Create a program to allow people to play the game Monopoly
  - Create classes for money, dice, players, the bank, the board, chance cards, community chest cards, pieces, etc.
- Some classes use other classes:
  - the board *consists of* spaces
  - a player *has* money
  - a piece *has* a position
  - also referred to as *composition*

Inheritance

- Another kind of relationship exists between data types
- There are properties in Monopoly
  - a street *is* a kind of property
  - a railroad *is* a kind of property
  - a utility *is* a kind of property
Inheritance

- In Monopoly there is the concept of a Property
- All properties have some common traits
  - they have a name
  - they have a position on the board
  - they can be owned by players
  - they have a price to buy
- But some things are different for each of the three kinds of property
  - How to determine rent when another player lands on the Property

What to Do?

- If we have a separate class for Street, Railroad, and Utility there is going to be a lot of code copied
  - hard to maintain
  - an anti-pattern
- Inheritance is a programming feature to allow data types to build on pre-existing data types without repeating code

Mechanics of Inheritance

1. extends keyword
2. inheritance of instance methods
3. inheritance of instance variables
4. object initialization and constructors
5. calling a parent constructor with super()
6. overriding methods
7. partial overriding, super.parentMethod()
8. inheritance requirement in Java
9. the Object class
10. inheritance hierarchies

Inheritance in Java

- Java is designed to encourage object oriented programming
- 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
Nomenclature of Inheritance

- In Java the `extends` keyword is used in the class header to specify which preexisting class a new class is inheriting from.
- `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`

Clicker Question 1

What is the primary reason for using inheritance when programming?

A. To make a program more complicated
B. To copy and paste code between classes
C. To reuse pre-existing code
D. To hide implementation details of a class
E. To ensure pre conditions of methods are met.

Clicker Question 2

What is output when the `main` method is run?

```java
public class Foo {
    public static void main(String[] args) {
        Foo f1 = new Foo();
        System.out.println(f1.toString());
    }
}
```

A. 0
B. null
C. Unknown until code is actually run.
D. No output due to a syntax error.
E. No output due to a runtime error.

Simple Code Example

- Create a class named `Shape`
  - what class does `Shape` inherit from
  - what methods can we call on `Shape` objects?
  - add instance variables for a position
  - override the `toString` method
- Create a `Circle` class that extends `Shape`
  - add instance variable for radius
  - debug and look at contents
  - try to access instance var from `Shape`
  - constructor calls
  - use of key word `super`
Overriding methods

- any method that is not final may be overridden by a descendant class
- same signature as method in ancestor
- may not reduce visibility
- may use the original method if simply want to add more behavior to existing

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 call a one of the parent's constructors
  - 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 no be called
  - no super();this(); allowed. One or the other, not both
  - good place for an initialization method

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 ClosedShapes toString

Creating a SortedIntList
A New Class

- Assume we want to have a list of ints, but that the ints must always be maintained in ascending order
  
  \[ [-7, 12, 37, 212, 212, 313, 313, 500] \]
  
  `sortedList.get(0)` returns the min
  
  `sortedList.get( list.size() - 1 )` returns the max

Implementing SortedIntList

- Do we have to write a whole new class?
- Assume we have an `IntList` class.
- Which of the following methods would have to be changed?
  
  `add(int value)`
  
  `int get(int location)`
  
  `String toString()`
  
  `int size()`
  
  `int remove(int location)`

Overriding the `add` Method

- First attempt
- Problem?
- solving with insert method
  - double edged sort
- solving with `protected`
  - What `protected` really means

Problems

- What about this method?
  
  `void insert(int location, int val)`

- What about this method?
  
  `void insertAll(int location, IntList otherList)`

- `SortedIntList` is not a good application of inheritance given the `IntList` we developed
More Example Code

ClosedShape and Rectangle classes

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

A ClosedShape class

```java
public class ClosedShape
{
    private double myX;
    private double myY;

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

    public ClosedShape (double x, double y)
    {
        myX = x;
        myY = y;
    }

    public String toString()
    {
        return "x: " + getX() + " y: " + getY();
    }

    public double getX() { return myX; }
    public double getY() { return myY; }

    // Other methods not shown
}
```

A Rectangle Constructor

```java
public class Rectangle extends ClosedShape
{
    private double myWidth;
    private double myHeight;

    public Rectangle( double x, double y,
                      double width, double height )
    {
        super(x,y);
        // calls the 2 double constructor in
        // ClosedShape
        myWidth = width;
        myHeight = height;
    }

    // other methods not shown
}
```
A Rectangle Class

```java
class Rectangle extends ClosedShape {
    private double myWidth;
    private double myHeight;

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

    public Rectangle(double width, double height) {
        myWidth = width;
        myHeight = height;
    }

    public Rectangle(double x, double y, double width, double height) {
        super(x, y);
        myWidth = width;
        myHeight = height;
    }

    public String toString() {
        return super.toString() + " width " + myWidth + " height " + myHeight;
    }
}
```

Result of Inheritance

Do any of these cause a syntax error? What is the output?

```java
Rectangle r = new Rectangle(1, 2, 3, 4);
ClosedShape s = new ClosedShape(2, 3);
System.out.println( s.getX() );
System.out.println( s.getY() );
System.out.println( s.toString() );
System.out.println( r.getX() );
System.out.println( r.getY() );
System.out.println( r.toString() );
```

Initialization method

```java
class Rectangle extends ClosedShape {
    private double myWidth;
    private double myHeight;

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

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

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

    private void init(double width, double height) {
        myWidth = width;
        myHeight = height;
    }
}
```

The Real Picture

### Fields from Object class
- Instance variables declared in Object

### Fields from ClosedShape class
- Instance Variables declared in ClosedShape

### Fields from Rectangle class
- Instance Variables declared in Rectangle

A Rectangle object

Available methods are all methods from Object, ClosedShape, and Rectangle
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

---

Why private Vars and not protected?

- In general it is good practice to make instance variables private
  - hide them from your descendants
  - if you think descendants will need to access them or modify them provide protected methods to do this

Why?

Consider the following example

---

Required update

```java
public class GamePiece {
    private Board myBoard;
    private Position myPos;

    // whenever my position changes I must
    // update the board so it knows about the change

    protected void alterPos( Position newPos ) {
        Position oldPos = myPos;
        myPos = newPos;
        myBoard.update( oldPos, myPos );
    }
}
```