

# Topic 7

## Inheritance and Polymorphism

"Question: What is the object oriented way of getting rich?

Answer: Inheritance."

"Classes struggle, some classes triumph, others are eliminated."

-Mao Zedong



## Outline

- ▶ Explanation of inheritance.
- ▶ Using inheritance to create a SortedIntList.
- ▶ Explanation of polymorphism.
- ▶ Using polymorphism to make a more generic List class.

## Explanation of Inheritance

## Main Tenets 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."

## Things and Relationships

- ▶ Object oriented programming leads to programs that are models
  - sometimes models of things in the real world
  - sometimes models of contrived or imaginary things
- ▶ There are many types of relationships between the things in the models
  - chess piece has a position
  - chess piece has a color
  - chess piece moves (changes position)
  - chess piece is taken
  - a rook is a type of chess piece

## The “has-A” Relationship

- ▶ Objects are often made up of many parts or have sub data.
  - chess piece: position, color
  - die: result, number of sides
- ▶ This “has-a” relationship is modeled by composition
  - the instance variables or fields internal to objects
- ▶ Encapsulation captures this concept

## The “is-a” relationship

- ▶ Another type of relationship found in the real world
  - a rook is a chess piece
  - a queen is a chess piece
  - a student is a person
  - a faculty member is a person
  - an undergraduate student is a student
- ▶ “is-a” usually denotes some form of specialization
- ▶ it is not the same as “has-a”

## Inheritance

- ▶ The “is-a” relationship, and the specialization that accompanies it, is modeled in object oriented languages via inheritance
- ▶ Classes can inherit from other classes
  - base inheritance in a program on the real world things being modeled
  - does “an A is a B” make sense? Is it logical?

# 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

```
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

# Results of Inheritance

```
public class A
```

```
public class B extends A
```

- ▶ the sub class inherits (gains) all instance variables and instance methods of the super class, **automatically**
- ▶ additional methods can be added to class B (specialization)
- ▶ the sub class can replace (redefine, override) methods from the super class

# 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

```
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
```

# 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

# A Rectangle Constructor

```
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

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

    public Rectangle()
    {   this(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;
    }
}
```

# Initialization method

```
public 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;
    }
}
```

## 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
- ▶ The `Rectangle` class
  - adds data, 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` by calling `super.toString()`
- ▶ without the `super` calling `toString` would result in infinite recursive calls
- ▶ Java does not allow nested `super`s
  - `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`

## Result of Inheritance

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

```
Rectangle r = new Rectangle(1, 2, 3, 4);
ClosedShape s = new CloseShape(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() );
System.out.println( r.getWidth() );
```

## Inheritance in Java

- ▶ 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

## Inheritance in Java

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

```
public class Die
```

  - `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*.

## The Real Picture

A  
Rectangle  
object

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

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

## 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

```
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 );
    }
}
```

## 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 `protected`
  - What `protected` really means
- ▶ solving with a `set` method

```
int set(int location, int newValue)
[1, 5, 2].set(1, 3) -> [1, 3, 2] and
returns 5
```

## Problems

- ▶ What about this method?

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

```
void insertAll(int location,
               IntList otherList)
```
- ▶ `SortedIntList` is not the cleanest application of inheritance.

## Explanation of Polymorphism

- ▶ Another feature of OOP
- ▶ literally “having many forms”
- ▶ object variables in Java are polymorphic
- ▶ object variables can refer to objects or their declared type AND any objects that are descendants of the declared type

```
ClosedShape s = new
ClosedShape();
s = new Rectangle(); // legal!
s = new Circle(); //legal!
Object obj1; // = what?
```

## Data Type

- ▶ object variables have:
  - a declared type. Also called the static type.
  - a dynamic type. What is the actual type of the pointer at run time or when a particular statement is executed.
- ▶ Method calls are syntactically legal if the method is in the declared type or any ancestor of the declared type
- ▶ **The actual method that is executed at runtime is based on the dynamic type**
  - dynamic dispatch

## What's the Output?

```
ClosedShape s = new ClosedShape(1,2);
System.out.println( s.toString() );
s = new Rectangle(2, 3, 4, 5);
System.out.println( s.toString() );
s = new Circle(4, 5, 10);
System.out.println( s.toString() );
s = new ClosedShape();
System.out.println( s.toString() );
```

## Method LookUp

- ▶ To determine if a method is legal the compiler looks in the class based on the declared type
  - if it finds it great, if not go to the super class and look there
  - continue until the method is found, or the Object class is reached and the method was never found. (Compile error)
- ▶ To determine which method is actually executed the run time system
  - starts with the actual run time class of the object that is calling the method
  - search the class for that method
  - if found, execute it, otherwise go to the super class and keep looking
  - repeat until a version is found
- ▶ Is it possible the runtime system won't find a method?

## Why Bother?

- ▶ Inheritance allows programs to model relationships in the real world
  - if the program follows the model it may be easier to write
- ▶ Inheritance allows code reuse
  - complete programs faster (especially large programs)
- ▶ Polymorphism allows code reuse in another way (We will explore this next time)
- ▶ Inheritance and polymorphism allow programmers to create *generic algorithms*

## Genericity

- ▶ One of the goals of OOP is the support of code reuse to allow more efficient program development
- ▶ If a 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, there are 2 ways of doing this
    - polymorphism and the inheritance requirement
    - generics

## the createASet example

```
public Object[] createASet(Object[] items)
{
    /*
     pre: items != null, no elements
     of items = null
     post: return an array of Objects
     that represents a set of the elements
     in items. (all duplicates removed)
     */

    {5, 1, 2, 3, 2, 3, 1, 5} -> {5, 1, 2, 3}
```

## createASet examples

```
String[] sList = {"Texas", "texas", "Texas",
                 "Texas", "UT", "texas"};

Object[] sSet = createASet(sList);
for(int i = 0; i < sSet.length; i++)
    System.out.println( sSet[i] );

Object[] list = {"Hi", 1, 4, 3.3, true,
                new ArrayList(), "Hi", 3.3, 4};

Object[] set = createASet(list);
for(int i = 0; i < set.length; i++)
    System.out.println( set[i] );
```

## A Generic List Class

## Back to IntList

- ▶ We may find `IntList` useful, but what if we want a List of `Strings`? `Rectangles`? `Lists`?
  - What if I am not sure?
- ▶ Are the List algorithms going to be very different if I am storing `Strings` instead of `ints`?
- ▶ How can we make a generic List class?

## Generic List Class

- ▶ required changes
- ▶ How does `toString` have to change?
  - why?!?!
- ▶ What can a `List` hold now?
- ▶ How many `List` classes do I need?

## Writing an `equals` Method

- ▶ How to check if two objects are equal?

```
if(objA == objA)
    // does this work?
```

- ▶ Why not this

```
public boolean equals(List other)
```

- ▶ Because

```
public void foo(List a, Object b)
    if( a.equals(b) )
        System.out.println( same )
```

- what if `b` is really a `List`?

## `equals` method

- ▶ read the javadoc carefully!
- ▶ don't rely on `toString` and `String`'s `equal`
- ▶ lost of cases