Inheritance & Polymorphism

Ibrahim Albluwi
A GuitarString **has a** RingBuffer.

A MarkovModel **has a** Symbol Table.
A Symbol Table **has a** Binary Search Tree.

A Deque **has a** Node.
A Node **has a** Node.

A Solver **has a** Node.
A Node **has a** Board and a Node.

A KdTreeST **has a** Node.
A Node **has a** Point2D and a Node.

Code reuse through **Composition**.

Classes are related with a **has-a** relationship.
Inheritance *is-a* Basic OOP Feature!

Definition of **OBJECT-ORIENTED PROGRAMMING**

: a type of computer programming in which programs are composed of objects (see \(^1\text{OBJECT }6a\)) which communicate with each other, which may be arranged into hierarchies, and which can be combined to form additional objects.

**Object-oriented programming**

*From Wikipedia, the free encyclopedia*

Languages that support classes almost always support inheritance. This allows classes to be arranged in a hierarchy that represents "is-a-type-of" relationships.

- Found in (almost) every Java, C++ or Python book.
- Very difficult to find a CS1/CS2 set of courses that does not cover it.

**But ...**

- Not covered explicitly in COS126/COS226!

**Goal Today:** — Know what Inheritance and Polymorphism *are*.
— Relate them to what we have seen in 126 and 226 so *far*. 
Which of the following is a valid Java Statement?

A. `Iterable<Integer> myStack = new Stack<Integer>();`
B. `Stack<Integer> myStack = new Stack<Integer>();`
C. `Object myStack = new Stack<Integer>();`
D. A and B only.
E. A, B and C.
Which of the following is a valid Java Statement?

A. `Iterable<Integer> myStack = new Stack<Integer>();`
B. `Stack<Integer> myStack = new Stack<Integer>();`
C. `Object myStack = new Stack<Integer>();`
D. A and B only.
E. A, B and C.

By the end of this class, you will be able to explain what these statements mean and what implications they have.
A Circle Class

public class Circle {
    private double centerX;
    private double centerY;
    private double radius;
    private Color color;

    public void move(double newX, double newY) {
        centerX = newX;
        centerY = newY;
    }

    public int getX() { return centerX; }
    public int getY() { return centerY; }

    public void draw() {
        StdDraw.setPenColor(color);
        StdDraw.circle(centerX, centerY, radius);
    }

    public void setColor(int r, int g, int b) {
        String errorMsg;
        boolean isValid = true;
        if (r < 0) {
            errorMsg = "Red is < 0";
            isValid = false;
        } else if (g < 0) {
            errorMsg = "Green is < 0";
            isValid = false;
        } else if (b < 0) {
            errorMsg = "Blue is < 0";
            isValid = false;
        } else if (r > 255) {
            // Other cases...
        }
    }
}
A Rectangle Class

```java
public class Rectangle {
    private double centerX;
    private double centerY;
    private double width;
    private double height;
    private Color color;

    public void move(double newX, double newY) {
        centerX = newX;
        centerY = newY;
    }

    public int getX() { return centerX; }
    public int getY() { return centerY; }

    public void draw() {
        StdDraw.setPenColor(color);
        StdDraw.rectangle(centerX, centerY, width / 2, height / 2);
    }

    public void setColor(int r, int g, int b) {
        String errorMsg;
        boolean isValid = true;
        if (r < 0) {
            errorMsg = "Red is < 0";
            isValid = false;
        } else if (g < 0) {
            errorMsg = "Green is < 0";
            isValid = false;
        } else if (b < 0) {
            errorMsg = "Blue is < 0";
            isValid = false;
        }
    }
}
```

+ Other Rectangle methods
Classes for Shapes

**Circle**
- centerX : double
- centerY : double
- color : Color
- radius : double

+ getX(): double
+ getY(): double
+ move(int,int): void
+ setColor(int,int,int): void
+ draw(): void
+ area() : double
+ circumference(): double
+ toString() : String
...

**Rectangle**
- centerX : double
- centerY : double
- color : Color
- width : double
- height : double

+ getX(): double
+ getY(): double
+ move(int,int): void
+ setColor(int,int,int): void
+ draw(): void
+ area() : double
+ circumference(): double
+ toString() : String
...

**Triangle**
- centerX : double
- centerY : double
- color : Color
- side1 : double
- side2 : double
- side3 : double

+ getX(): double
+ getY(): double
+ move(int,int): void
+ setColor(int,int,int): void
+ draw(): void
+ area() : double
+ circumference(): double
+ toString() : String
...

...
## Classes for Shapes

### Circle
- centerX : double
- centerY : double
- color : Color
- radius : double

+getX(): double
+getY(): double
+move(int,int): void
+setColor(int,int,int): void
+draw() : String
+area() : double
+circumference(): double
+toString() : String

### Rectangle
- centerX : double
- centerY : double
- color : Color
- width : double
- height : double

+getX(): double
+getY(): double
+move(int,int): void
+setColor(int,int,int): void
+draw() : void
+area() : double
+circumference() : double
+toString() : String

### Triangle
- centerX : double
- centerY : double
- color : Color
- side1 : double
- side2 : double
- side3 : double

+getX(): double
+getY(): double
+move(int,int): void
+setColor(int,int,int): void
+draw() : void
+area() : double
+circumference() : double
+toString() : String
A Shape Base Class

Shape
- centerX : double
- centerY : double
- color : Color

+ getX(): double
+ getY(): double
+ move(int,int): void
+ setColor(int,int,int): void

Circle
- radius : double

...
A Shape Base Class

Observations.  (1) Lots of **common code** between the classes.
(2) A Circle is a Shape, so is a Rectangle and a Triangle.

Solution.  — Create a **Shape class** that has the common code.
— Declare that Circle is a Shape. Do the same for Triangle and Rectangle.
— Circle, Triangle and Rectangle **inherit** the code from class Shape.

In Java:  
```java
public class Circle extends Shape { ... }
public class Triangle extends Shape { ... }
public class Rectangle extends Shape { ... }
```
Demo!
**Notes**

**Terminology.** Shape is a *parent* class, a *superclass* and a *base* class. Circle is a *child* class, a *subclass* and a *derived* class.

**Access Modifiers.**

- **Public:** Accessible to everyone.
- **Protected:** Accessible to subclasses and package.
- **No Modifier:** Accessible to package.

**Super** and **this.**

- **this.x** Can be an $x$ in the parent or child class.
- If both classes have $x$, **this.x** refers to the $x$ in the child class

- **super.x** Always refers to $x$ in the *superclass.*
What did we gain?

- Code Reuse!
- Is-A Relationship!

Can do great things!

Example 1

```java
Circle c = new Circle();
Triangle t = new Triangle();
Rectangle r = new Rectangle();

Shape [] shapes = new Shape[3];
shapes[0] = c;
shapes[1] = t;
shapes[2] = r;

for (int i = 0; i < 3; i++)
    shapes[i].setColor(128, 128, 128);
```

Example 2

```java
myObj.doSomething(c);
myObj.doSomething(t);
myObj.doSomething(r);
```

Method doSomething accepts an argument of type Shape.
Rules of the Game

Circle $c = \text{new Circle}()$

Reference of type Circle

Object of type Circle

pointing to an

⚠️ Can invoke on $c$ any method in class Circle (or Shape).
- $c.setRadius()$ ← Valid
- $c.setColor()$ ← Valid

Shape $c = \text{new Circle}()$

Reference of type Shape

Object of type Circle

pointing to an

⚠️ Can invoke on $c$ only methods in class Shape.
- $c.setRadius()$ ← Invalid
- $c.setColor()$ ← Valid
Q. Do we want to allow instantiating objects of type Shape?

If not, then declare class Shape as **abstract**.

```java
public abstract class Shape { ... }
```

Q. Do we want method draw() to be defined in class Shape?

Yes! Since all shapes need to be drawn.

However, since each shape is drawn differently, draw() should be **abstract**.

```java
public abstract void draw();
```

An **abstract method**: Has no body (note the semicolon).

Derived classes MUST either be also abstract or implement all abstract methods in the base class.
Assume that Circle overrides method setColor. Which method will get called when `shapes[0].setColor` is called?

A. `setColor` of Shape.
B. `setColor` of Circle.
C. The compiler will complain because there are two `setColor` methods.
D. Armagedon.
Assume that Circle overrides method setColor. Which method will get called when `shapes[0].setColor` is called?

A. setColor of Shape.
B. setColor of Circle.
C. The compiler will complain because there are two setColor methods.
D. Armageddon.
Welcome *Polymorphism*

**What?** If a subclass defines it’s own version of a base class method (*overrides* it), the subclass version is invoked if the reference points to an object of the subclass type.

**Example.** Assume class Circle overrides method `setColor` in Shape.

```
Shape c = new Circle()
```

- `c.getX()`: Valid. Implemented in Shape.
- `c.setColor()`: Valid. Implemented in both Shape and Circle. Circle’s version gets called.
Déjà vu!

Inheritance & Polymorphism you have already seen.
All Java classes implicitly extend a class named `Object` that has the following methods:

- `equals(Object obj)` Checks if the object is equal to obj.
- `toString()` Returns a string representation of the object.
- `hashCode()` Returns a hash code value for the object.
- `clone()` Returns a copy of the object.
- `getClass()` Returns the type of the object.
- `Others...`
When you implement `equals` or `toString`, you are actually overriding the default implementation of `equals` and `toString` in the `Object` class.

**Default Implementations.**

**Equals():** Reference comparison of memory locations using the `==` operator.

**ToString():** A String made of the class name + ‘@’ + `hashCode()`. The default implementation of `hashCode` returns the memory location of the object.

```java
Circle c1 = new Circle()
c1.equals(c2) ← Valid Uses default implementation of Object.
c1.toString() ← Valid A default implementation in Object and another in Circle.
```

Uses the implementation in Circle.
Consider. Circle c1 = new Circle()

Explain. How does Java handle the following two lines of code?

System.out.print(c1)
System.out.print(c1.toString())

Answer. Method print is overloaded:

<table>
<thead>
<tr>
<th>print(Object obj)</th>
<th>Calls method toString on obj.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Print an object.</td>
<td></td>
</tr>
<tr>
<td>print(String s)</td>
<td>Polymorphism in action!</td>
</tr>
<tr>
<td>Prints a string.</td>
<td></td>
</tr>
</tbody>
</table>
Again … What did we gain?

- Code Reuse!
- Is-A Relationship!
- A promise for an API

by inheriting state and behavior from parent class.

and Polymorphism!

through abstract methods

What if we care only about these? Define an abstract class where all methods are abstract.

Or …

Define and use an interface!
Welcome *Interfaces!*

Instead of:

```java
public abstract class Shape {
    public abstract double getX();
    public abstract double getY();
    public abstract void draw();
    public abstract void setColor(int, int, int);
    ...
}
```

Implement:

```java
public interface Shape {
    double getX();
    double getY();
    void draw();
    void setColor(int, int, int);
    ...
}
```

All methods are implicitly:

*public abstract.*

All fields are implicitly:

*public static final*
Examples of Interfaces in Java 7

```java
public interface Iterable<T> {
    Iterator<T> iterator();
    ...
}
```

```java
public interface Iterator<T> {
    boolean hasNext();
    T next();
    ...
}
```

```java
public interface Comparable<T> {
    int compareTo(T other);
}
```

```java
public interface Cloneable {}
```

Use with interfaces `implements` instead of `extends`.

Empty! Useful only for Is-A relationship.
Interfaces in Java 8

In Java 8, methods in interfaces are allowed to have a **default** implementation.

**Question.** What is the difference between an abstract class and an interface with default implementations?

**Answer.**

<table>
<thead>
<tr>
<th>extending a class</th>
<th>implementing an interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherits API, <strong>state</strong> and implementation.</td>
<td>Inherits only API and implementation.</td>
</tr>
</tbody>
</table>

*Multiple Inheritance* NOT allowed.  
*Multiple Inheritance* allowed.

A class **can extend only one class**, but **can implement several interfaces**. I.e., a class can be only one thing, but can play several roles!
Which of the following is a valid Java Statement?

A. `Iterable<Integer> myStack = new Stack<Integer>();`
B. `Stack<Integer> myStack = new Stack<Integer>();`
C. `Object myStack = new Stack<Integer>();`
D. A and B only.
E. A, B and C.

A Stack is Iterable.

A Stack is an Object.

Only iterator() can be invoked

Only Object methods can be invoked
Discussion

What is the difference between using Generics and using Object?

Example.

```java
public class Queue {
    public void enqueue(Object obj) {...}
    public Object dequeue() {...}
}
```

V.S.

```java
public class Queue<T> {
    public void enqueue(T element) {...}
    public T dequeue() {...}
}
```

```
Queue<Integer> qT = new Queue<Integer>();
Queue qObj = new Queue();
```

Type Safe.  qT.enqueue(myCat); // does not compile!
Not Type Safe. qObj.enqueue(myCat); // compiles!

No Need to Cast.  int element = qT.dequeue();
Needs a Cast.  int element = (Integer) qObj.dequeue();
Abusing Inheritance

1. Extending for implementation. To extend, Is-A should hold

**Example 1.** Make class Percolation extend class BeadFinder to make use of the DFS method. *Bad idea!* A Percolation object is not a BeadFinder.

**Example 2.** Make class Polygon extend class Circle to make use of `getX()`, `getY()`, `setColor()`, etc. *Bad idea!* A Polygon is not a Circle.

2. Methods or variables in base class not relevant in subclasses.

**Example.** Adding instance variable `radius` and method `getRadius` in class Shape. *Bad idea!* Useful for Circle, Oval but not for Triangle and Polygon.

3. Hierarchies that are long and complicated.
Hierarchies should be wide, not deep!
Inheritance Wars!

Anti-Inheritance Clan

- Inheritance violates encapsulation. Child class knows too much about Parent class.
- Widely abused.
- Leads to absurdities, especially with multiple inheritance.
- Code difficult to test and debug.

Anti-Anti-Inheritance Clan

- Inheritance is useful.
- Model’s real world entities more naturally.
- Code carefully and avoid abuse.

Widely Used Rules of Thumb:

— Favor Composition over Inheritance.
— Use Composition for code re-use and implement interfaces for defining Is-A relationships.