Inheritance & Polymorphism

Derived Classes

We'll often have classes that are very similar to each other, but with some small additions or changes.

Following the principle of DRY, we'll need a way of efficiently expressing a class that derives from another.

Example: Candy Crush

To program this game, you'll implement a grid of game pieces.

Each piece has, at least, a position and an image representation.

What about the `Candy`?

```
public class Candy {
    private Point position;
    private Image sprite;
    private Color color;

    public Point getPosition() {...}
    public void draw() {...}
    public Point moveTo(Point p) {...}
    public void drop() {...}
    public void clear() {...}
}
```

A `Candy` game piece can be cleared, and will drop when space is opened beneath it.

What about the `Obstacle`?

```
public class Obstacle {
    private Point position;
    private Image sprite;

    public Point getPosition() {...}
    public void draw() {...}
    public Point moveTo(Point p) {...}
    public void grow() {...}
}
```

An `Obstacle` game piece prevents other pieces from being moved into the space, and they will also grow over time.

Too Much Overlap!

`Candy` objects and `Obstacle` objects are almost exactly the same, except for a handful of small changes.

Idea:

- Can we "factor out" the common fields and methods into a single class?
- If we have the commonalities in one class, we can find a way to automatically import them into some specific classes
- The specific individual classes can have their own additions, too

the `GamePiece`

```
public class GamePiece {
    private Point position;
    private Image sprite;

    public Point getPosition() {...}
    public void draw() {...}
    public Point moveTo(Point p) {...}
}
```

These are the methods and fields that `Candy` and `Obstacle` both have.

Derived Classes (subclasses)

A **derived class (subclass)** is a class that is derived from another **base class (superclass)**.

Any class many serve as the superclass for another derived class.

A subclass inherits all properties of the base class, including member variables (fields) and methods.

`extends`

```
class DerivedClass extends BaseClass { ... }
```

Now `DerivedClass` has access to all public fields and methods contained in `BaseClass`.

- These can be referenced in the normal way
- New methods and fields can still be added in the normal way
- Derived methods can be modified by **overriding** the original method signature.

`Candy` as derived from `GamePiece`

```
public class Candy extends GamePiece{
    private Color color;

    public void drop() {...}
    public void clear() {...}
}
```

`position`, `sprite`, `getPosition()`, `draw()`, and `moveTo(Point p)` are all included in `Candy` now for free!

Using `Candy`

```
public class Tester {
    public static void main(String[] args) {
        Candy c = new Candy(); // imagine we wrote a useful constructor

        Point cPosition = c.getPosition(); // valid!

        c.drop(); // valid!

        System.out.println(c.color); // doesn't work, why not?

        System.out.println(c.position); // doesn't work, why not?
    }
}
```

Overriding

```
public class Obstacle extends GamePiece{

    @Override
    public Point moveTo(Point p) {
        return this.getPosition();
    }

    public void grow() {...}
}
```

The user should be able to *try* to move any game piece, but if you try to move an `Obstacle`, it should stay put.

Overriding

Because an `Obstacle` extends the `GamePiece`, it must be the case that we can call `moveTo()` on an `Obstacle`.

It's OK, though, if we need a subclass to implement a required method differently than the superclass.

More Rules on Inheritance

- A subclass can serve as a superclass for another class.
  - `class ProduceItem extends GenericItem {...}` and also `class FruitItem extends ProduceItem {...}`
  - `FruitItem` gets everything from `ProduceItem`, which also therefore gets everything from `GenericItem`.
- A class can serve as a superclass for multiple derived classes
  - In addition to above, `class FrozenFoodItem extends GenericItem {...}`

zyBook Activity 10.1.5 is **excellent** for visualizing this.

Example

```
public class Polygon {
    private ArrayList<Side> sides;

    public Polygon(ArrayList<Side> sides) {
        this.sides = sides;
    }

    public Side getSide(int i) {
        return sides.get(i);
    }

    public double perimeter() {
        double sum = 0.0;
        for (Side side : sides) {
            sum += side.length();
        }
        return sum;
    }
```

Example

```
public class Square extends Polygon {

    @Override
    public double perimeter() {
        return 4 * getSide(0);
    }

    public double area() {
        return getSide(0) * getSide(0);
    }

}
```

Your Turn: `Triangle`

Write a `Triangle` subclass of `Polygon`. Include a new method `isEquilateral()` to return `true` if all sides of the triangle are equal in length.

🚨 🚨 🚨 Remember, DRY! 🚨 🚨 🚨

Example

```
public class Triangle extends Polygon {

    public boolean isEquilateral() {
        double sideOne = getSide(0).length();
        double sideTwo = getSide(1).length();
        double sideThree = getSide(2).length();

        return sideOne == sideTwo && sideTwo == sideThree;
    }

}
```

Where Does It All Come From?

It turns out that every single class in Java is a (possibly distant) subclass of the `Object` class.

A sketch of `Object`:

```
public class Object {

    public String toString() {...}

    public boolean equals(Object other) {...}
}
```

Revealing a Minor Miracle

`System.out.println()` is actually a miraculous method.

We've called `System.out.println()` with `ArrayLists`, `Strings`, `Scanners`, and all sorts of wacky objects.

Does that mean someone wrote a `public void println(ArrayList a)` and also a `public void println(String s)`?

Revealing a Minor Miracle

It's all just:

```
public void println(Object o) {
    println(o.toString());
}
```

where `public void println(String s)` is actually an interesting implementation somewhere else.

Polymorphism

Polymorphism refers to determining which program behavior to execute depending on data types.

- **compile-time polymorphism** happens when the compiler determines which of several identically-named methods to call based on the method's arguments
  - Do we call `add(int x, int y, int z)` or `add(int x, int y)` if we write `add(3, 9)`?
- **runtime polymorphism** happens when the compiler makes the determination is made while the program is running

Runtime Polymorphism

```
public static void main(String[] args) {

    ArrayList<Polygon> shapesList = new ArrayList<>();

    Triangle t = new Triangle(3, 4, 5);

    Square s = new Square(5);

    Polygon octagon = new Polygon(8);

    shapesList.add(t);
    shapesList.add(s);
    shapesList.add(octagon);
}
```

Reference Conversion

Java is happy to turn a reference to a subclass into a reference to its superclass without any fuss.

When appropriate, it's fine to treat a `Triangle` as a `Polygon` since `Triangle extends Polygon`.

What Happends to the Subclasses?

```
public class A {
    public void print() {
        System.out.println("This is an A.");
    }
}
public class B extends A{
    public void print() {
        System.out.println("This is a B.");
    }
}
public class C extends A{
    public void print() {
        System.out.println("This is a C.");
    }
}
```

What Happends to the Subclasses?

```
A a1 = new A();
A a2 = new A();
B b1 = new B();
C c1 = new C();

ArrayList<A> objs = new ArrayList<>();
objs.add(a1);
objs.add(a2);
objs.add(b1);
objs.add(c1);
for (A obj : objs) {
    obj.print();
}
```

```
This is an A. This is an A. This is a B. This is a C.
```

A Frightening Corollary

If an `ArrayList<SuperClass>` can store objects of any type that's a subclass of `SuperClass`...

And if `Object` is a superclass of all other classes...

A Frightening Corollary

```
ArrayList<Object> objs = new ArrayList<>();

objs.add("Hello");
objs.add(new Random());
objs.add(new Scanner(System.in));

for (Object obj : objs) {
    System.out.println(obj);
}
```

```
Hello
java.util.Random@6442b0a6
java.util.Scanner[delimiters=\p{javaWhitespace}+][position=0][match valid=false]....
```