

Abstract Classes

Inheritance is a Powerful Tool

If we know what an object is, we know what we can expect it to do.

- Any object that extends a `Polygon` can calculate its perimeter
- Any `GamePiece` can determine its position, be it an `Obstacle` or a `Candy`.

Superclasses Shouldn't Have to Be Concrete, Though

It makes sense that we might want to instantiate a `Polygon` to represent a real, concrete polygon of some irregular shape that's not a `Triangle` or a `Square`.

It makes **much less sense** to instantiate a `GamePiece` in CandyCrush that's not a `Candy` or an `Obstacle`.

Abstract Classes

An **abstract class** can be used to define a superclass that cannot itself be instantiated.

- Abstract classes are marked as `abstract`, e.g.
`public abstract class GamePiece`
- Abstract classes do not have constructors and cannot be instantiated.
- Abstract classes can contain **abstract methods**

Abstract Methods

An **abstract method** is a signature without a body.

- marked by writing a signature like so:

```
abstract returnType methodName(argType argName, argType  
argName...);
```

- any subclass that derives the abstract class containing this abstract method must provide an implementation for the abstract method
- Any class containing an abstract method has to itself be abstract.

Quick aside: `protected`

A field or method tagged as `protected` is accessible within the class it's defined inside of, and also from any class **within this package** that also **derives from this class**.

`public` to subclasses *you personally write* and `private` otherwise.

GamePiece, redux

```
public abstract class GamePiece {  
    protected Point position;  
    protected Image sprite;  
  
    public Point getPosition() {  
        return this.position;  
    }  
    public void draw() {  
        sprite.render(position);  
    }  
    abstract Point moveTo(Point p);  
}
```

Candy and **Obstacle** handle **moveTo** differently, so might as well make them deal with it!

Animals, wild and domestic

```
public abstract class Animal {  
    protected String name;  
    protected int age;  
  
    abstract void printInfo();  
  
    public String getNameAndAge() {  
        return this.name + ", " + age + " years";  
    }  
}
```

Animals, wild and domestic

```
public class Domestic extends Animal {  
    private String owner;  
  
    public Domestic(String domesticName, int domesticAge, String domesticOwner) {  
        this.name = domesticName;  
        this.age = domesticAge;  
        this.owner = domesticOwner;  
    }  
  
    public void printInfo() {  
        String nameAndAge = this.getNameAndAge();  
  
        System.out.println(nameAndAge);  
        System.out.println("Owner: " + this.owner);  
    }  
}
```

Animals, wild and domestic

```
public class Wild extends Animal {  
    private String species;  
  
    public Wild(String wildName, int wildAge, String wildSpecies) {  
        this.name = wildName;  
        this.age = wildAge;  
        this.species = wildSpecies;  
    }  
  
    public void printInfo() {  
        String nameAndAge = this.getNameAndAge();  
  
        System.out.println(nameAndAge);  
        System.out.println("Species: " + this.species);  
    }  
}
```

Interfaces for Full Abstraction

Perhaps you want to define a superclass that has **ONLY** abstract methods.

This is a fully abstract data type:

- you'd know everything that it can do
- you know nothing about how it works.

To do this, write `public interface InterfaceName`

Rules of Interfaces

- Interfaces cannot have private methods
- Interfaces can have `private static final` fields, i.e. **constants**
- Interfaces do not need to tag each method signature as abstract
 - They are all understood to be abstract!
- A class can implement multiple interfaces.
 - Remember that this is different from `extends`

Example

```
public interface DrawableASCII {  
    public static final char defaultChar = '#';  
  
    public void drawASCII(char drawChar);  
  
    public void drawASCII();  
  
}
```

Square

```
public class Square implements DrawableASCII {  
    private int sideLength;  
  
    public Square(int sideLength) {  
        this.sideLength = sideLength;  
    }  
  
    @Override  
    public void drawASCII(char drawChar) {  
        // TO DO  
    }  
  
    @Override  
    public void drawASCII() {  
        drawASCII(defaultChar);  
    }  
}
```

Circle

```
public class Circle implements DrawableASCII {  
    private int radius;  
  
    public Circle(int radius) {  
        this.radius = radius;  
    }  
  
    @Override  
    public void drawASCII(char drawChar) {  
        // TO DO  
    }  
}
```

Interfaces and Superclasses Define "is-a" Relationships

Contrast this with fields, which indicate "have-a" relationships.