Abstract

Abstract Classes

- Methods can be abstract
  - Specify the method signature (name, return type, parameters)
  - Do not define the method (no body)
  - End the method with a semicolon

```
public abstract class GameItem {
    private double xLoc;
    private double yLoc;

    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }

    abstract void use(Player player);

}
```

- Abstract methods cannot be called
  - What would you expect to happen? Nothing? What if it has a return type?

Abstract Classes

- If a class has >0 abstract methods, the class itself must be abstract

- **Abstract classes cannot be instantiated**
  - Cannot create a new GameItem if GameItem is abstract
  - Prevents anyone from calling an abstract method

- They only exist to be inherited

```
public abstract class GameItem {
    private double xLoc;
    private double yLoc;

    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }

    abstract void use(Player player);

}
```

Abstract Classes

- Any class inheriting from an abstract class has a requirement to implement all abstract methods
  - If the extending class overrides the abstract method, it then exists and can be called

```
public abstract class GameItem {
    private double xLoc;
    private double yLoc;

    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }

    abstract void use(Player player);

}
```

- If a subclass does not implement all abstract methods, it too must be abstract

Abstract Classes

- **Why use abstract methods/classes?**

- You can only call methods that are known to your variable type
- Abstract methods are known to the abstract class
- You can call abstract methods using polymorphism

- Use an abstract method when you want all inheriting classes to have a method, but there's no clear default behavior for the method

```
public abstract class GameItem {
    private double xLoc;
    private double yLoc;

    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }

    abstract void use(Player player);

}
```

Interfaces

- If we take this one step further, we can create **interfaces**
- Interfaces are similar to classes
- Interfaces can **only** have abstract methods
  - No instance variables
  - No constructor
  - No methods with definitions
- To inherit an interface, use the implements keyword instead of extends

```
public interface Comparator<T> {
   boolean compare(T a, T b);
}
```

```
public class IntDecreasing implements Comparator<Integer> {
    @Override
    public boolean compare(Integer a, Integer b) {
        return a > b;
    }
}
```

Interfaces

- Why interfaces?
- You can only extend one class
- You can implement as many interfaces as you'd like
- *This avoids the potential of multiple definitions for the same method

```
public interface Comparator<T> {
   boolean compare(T a, T b);
}
```

```
public class IntDecreasing implements Comparator<Integer> {
    @Override
    public boolean compare(Integer a, Integer b) {
        return a > b;
    }
}
```