Abstraction

Lesson 4.2

Learning Outcomes

LO 4.2.1 **Create** licit abstract classes to apply abstraction

LO 4.2.2 **Create** licit interfaces to apply abstraction

LO 4.2.3 **Apply** inclusion polymorphism on object identifiers applied with abstraction

Learning Outcomes

LO 4.2.4 **Identify** correct and incorrect applications of abstraction

LO 4.2.5 **Assert** correct use of either abstract classes or interface when applying abstraction on object identifiers or classes

Abstraction

There are times that when you construct a class design applied with *inheritance*, you don’t want superclasses/ supertypes to be instantiated. You only want them to serve as a data type of identifiers for *subtyping*.

This can be done if these types are **abstract**.

Abstraction

Abstraction is an important concept in OOP that allows programmers to focus on the essential details of an object and ignore the rest. This can make it easier to *reason* about code, simplify *code maintenance*, and make it easier to *develop complex systems*.

In OOP, abstraction is achieved through the use of *abstract classes* and *interfaces*.

Abstract Classes

An **abstract class** is a class that is declared **abstract**—it may or may not include abstract method/s.

An **abstract method** is a method that is declared without an implementation (*without braces, and followed by a semicolon*), like this:

```
public abstract void move(double deltaX, double deltaY);
```

Abstract Classes

All abstract constructs; that includes abstract classes, *cannot be instantiated/created*. It can only be inherited or used in subtyping.

When an abstract class is inherited into a non-abstract class, the non-abstract class *must provide implementation* for all the inherited methods that are still abstract. However if you don't want it to, then you must *declare* that class, *abstract*.

Abstract Classes

Example:

```
public abstract class GraphicObject
{
    protected int x;
    protected int y;
    public abstract void draw(Graphics g);
}

public class GameObject extends GraphicObject
{
    private Image img;
    @Override
    public void draw(Graphics g)
    {
        g.drawImage(img, super.x, super.y);
    }
}
```

Abstract Classes

Example:

```
public abstract class GraphicObject
{
    protected int x;
    protected int y;
    public abstract void draw(Graphics g);
}
```

```
public abstract class GameObject
   extends GraphicObject
{
    protected Image img;
}
```

```
public class Character extends GameObject
{
    @Override
    public void draw(Graphics g)
    {
        g.drawImage(super.img, super.x, super.y);
    }
}
```

LO 4.2.1 Create licit abstract classes to apply abstraction

Redesign the current implementation of the *Plants Vs Zombies* classes from the previous lesson while applying necessary abstraction with abstract classes. Supply the reason with regards to your design.

Interfaces

In Java OOP, an **interface** is a group of related methods with empty bodies. All methods in an interface is, by default, *abstract*.

An interface can *inherit 1 or more existing interfaces*. A class can *implement 1 or more interfaces*. With the inheritance of the *abstract methods* from the interfaces, it will still follow the same procedure done on abstract methods inherited from *abstract classes*.

Interfaces

Example:

```
public interface MovingObject
{
   public void moveUp();
   public void moveDown();
   public void moveLeft();
   public void moveRight();
}
```

```
public interface IntelligentObject extends
   MovingObject
{
   public boolean moveAI(int[][] heuristics);
}
```

All abstract methods in *interface* `MovingObject` will be inherited into *interface* `IntelligentObject`.

Interfaces

Example:

```
public interface MovingObject
{
   public void moveUp();
   public void moveDown();
   public void moveLeft();
   public void moveRight();
}
```

```
public class Pacman implements MovingObject
{
   private int x;
   private int y;

   @Override
   public void moveUp(){ this.y++; }
   @Override
   public void moveDown(){ this.y--; }
   @Override
   public void moveLeft(){ this.x++; }
   @Override
   public void moveRight(){ this.x--; }
}
```

All *abstract methods* in *interface* `MovingObject` that is implemented into a *non-abstract class* `Pacman` must implement all *abstract methods* from the implemented *interface*.

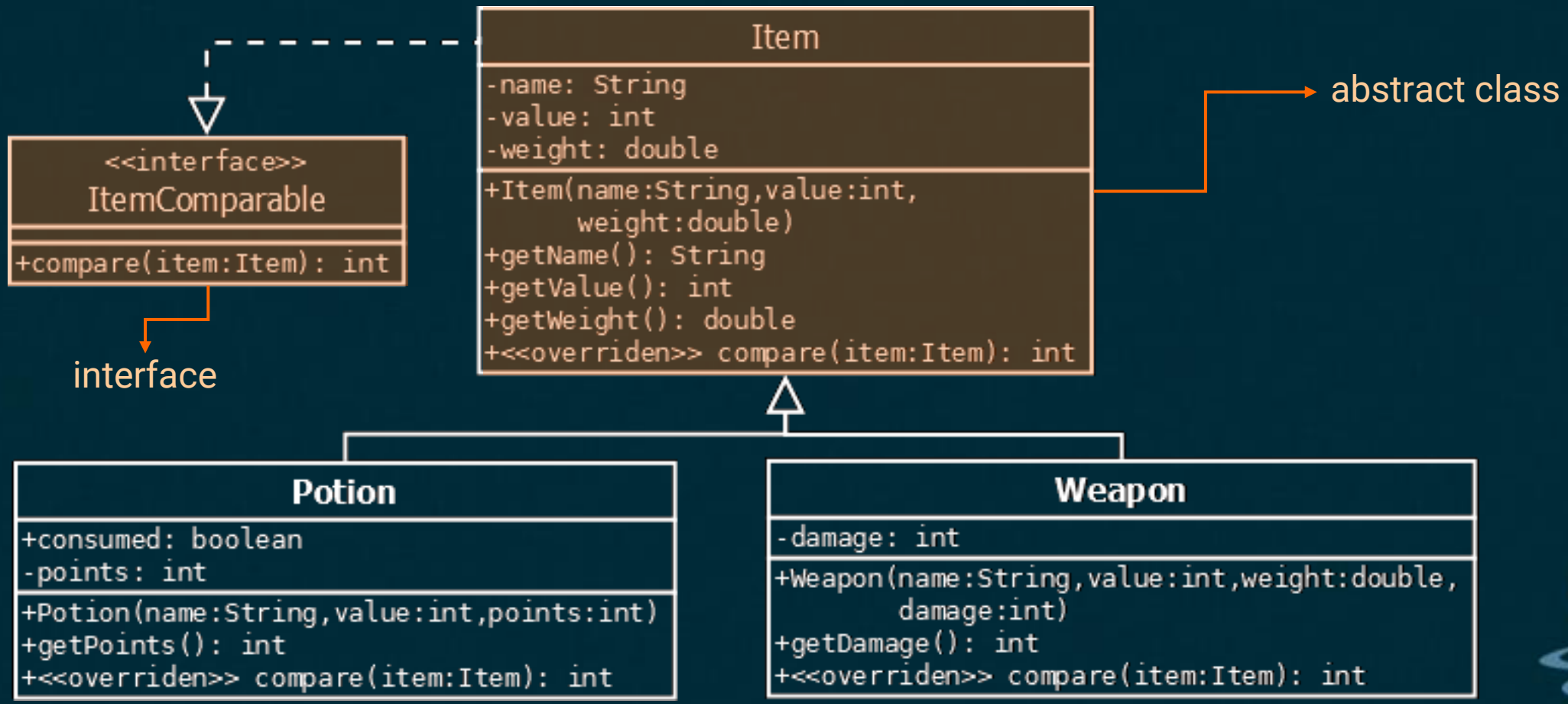
Interfaces

Example:

```
public interface Shooter
{
   public void shoot(Object o);
}

public interface RevivableObject
{
   public boolean revive();
}
```

```
public class Hero implements Shooter,
   RevivableObject
{
   private int currentHP;
   private int maxHP;
   private int damage;

   @Override
   public void shoot(Object o)
   {
      if(o instanceof Hero)
         ((Hero)o).currentHP -= this.damage;
   }

   @Override
   public boolean revive()
   {
      this.currentHP = this.maxHP;
      return true;
   }
}
```

Abstract Classes and Interfaces in UML



LO 4.2.2 Create licit interfaces to apply abstraction

Continue to redesign the current implementation of the *Plants Vs Zombies* classes from the previous lesson while applying necessary abstraction with interfaces. Supply the reason with regards to your design.

Abstraction + Polymorphism

Abstract constructs, i.e. *abstract classes* and *interfaces*, can be used as subtypes to objects instantiated from classes implementing such constructs.

Based from previous example, we can say:

```
RevivableObject ro = new Hero();
```

But we can only call the method `revive` from `ro` since `shoot` does not exist in *interface* `RevivableObject`.

LO 4.2.3 Apply inclusion polymorphism on object identifiers applied with abstraction

With your current implementation of the *Plants Vs Zombies* classes and interface/s, write code snippets demonstrating necessary inclusion polymorphism using abstract class and interface subtypes. You can either implement it with any collection. (e.g. arrays, lists, etc.)

LO 4.2.4 Identify correct and incorrect applications of abstraction

Can these code snippets successfully compile and run? Why or why not?

```
public interface Forgeable
{
   public Equipment enhance(Gem g);
}

public abstract class Equipment implements
   Forgeable
{
   private String name;
   public Equipment(String name)
   { this.name = name; }
   public abstract boolean merge(
      Equipment e);
}

public class Gem { }
```

```
public class Gun extends Equipment
{
   private int bullets;
   public Gun(String name, int bullets)
   {
      super(name); this.bullets = bullets;
   }
   @Override
   public boolean merge(Equipment e)
   {
      if(e instanceof Gun){
          this.bullets += ((Gun)e).bullets;
          ((Gun)e).bullets = 0;
          return true;
      }
      return false;
   }
}
```

LO 4.2.5 Assert correct use of either abstract classes or interface when applying abstraction on object identifiers or classes

```
abstract class Bird implements Livestock {...}
class Chicken extends Bird {...}
```

Given the following definition of Bird and Chicken, which of the given statements will not compile? Why?

1. `Bird bird = new Chicken();`
2. `Bird bird = new Bird();`
3. `Livestock livestock = new Chicken();`