

# Inheritance



# Task 4 Tips: Tip #1

## Do not test undefined behavior

- If there is any behavior that is not explicitly defined in the HW doc -> **Do not write a test case for that behavior**
- Many students are testing the `bayesianAverage` method with invalid ratings or negative numbers of additional ratings
  - The behavior of the method on these inputs is not defined in the HW doc
  - What should `bayesianAverageRating(3, -2)` return? I don't know. If you are guessing what it should return, you can't write a test case for it
- If a method is not defined in the HW doc, you cannot test it since that method won't exist in the correct solution (eg. you can't test `playlist.getComparator()` since it will crash when testing the correct solution)
- It's subtle, but undefined behavior is different from edge cases which are defined



# Task 4 Tips: Tip #2

**To test a class, you need to create an object of that type**

- Refer to the TestClasses1 code as an example
  - To test the Song class, we first create a new object of type Song and call methods through that object
- This will apply to both your Comparators as well
  - Create a new object of the the you are testing, then call compare through the object
  - Note: If a class does not have a constructor, it automatically has the "default constructor" which takes no parameters and an empty body



# Task 4 Tips: Tip #3

~~Have fun!~~ Try not to get too frustrated..

- Task 4 will be much more time consuming than three previous tasks
- Tasks 5 and 6 will be less time consuming than task 4
- There is less code to write for each of these tasks than for task 4
- You will already have the tests written when you start these tasks which will make the code easier to debug
- **If you can survive task 4 -> You can survive CSE116!**



# Override



# Inheritance

- Recall that we used inheritance to add all of the state and behavior of one class to another class
- HealthPotion extends (or, inherits from) GameItem
- HealthPotion objects have all the instance variables (State) of both HealthPotion and GameItem
- GameItem is the super class of HealthPotion

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

```
public class HealthPotion extends GameItem {  
    private int increase;  
    public HealthPotion(double xLoc, double yLoc, int increase) {  
        super(xLoc, yLoc);  
        this.increase = increase;  
    }  
}
```



# Inheritance

- HealthPotion objects have all the methods (Behavior) of both HealthPotion and GameItem
- We add a use method to the GameItem class
- All HealthPotion objects now have a use method

```
public class GameItem {  
    private double xLoc;  
    private double yLoc;  
    public GameItem(double xLoc, double yLoc) {  
        this.xLoc = xLoc;  
        this.yLoc = yLoc;  
    }  
    public void use() {  
        System.out.println("Item Used");  
    }  
}
```

```
public class HealthPotion extends GameItem {  
    private int increase;  
    public HealthPotion(double xLoc, double yLoc, int increase) {  
        super(xLoc, yLoc);  
        this.increase = increase;  
    }  
}
```



# Inheritance

- What if we want to extend a class, but don't want 100% of the inherited state and behavior?
- We want a class to inherit the location code from GameItem, but want the use method to something else
- Override!

```
public class GameItem {  
    private double xLoc;  
    private double yLoc;  
    public GameItem(double xLoc, double yLoc) {  
        this.xLoc = xLoc;  
        this.yLoc = yLoc;  
    }  
    public void use() {  
        System.out.println("Item Used");  
    }  
}
```

```
public class HealthPotion extends GameItem {  
    private int increase;  
    public HealthPotion(double xLoc, double yLoc, int increase) {  
        super(xLoc, yLoc);  
        this.increase = increase;  
    }  
}
```



# Override

- Weapon will also inherit the state and behavior from GameItem
- We will **Override** the use method with a new definition specific to the Weapon class
- The inherited method is **replaced** by this new definition

```
public class GameItem {  
    private double xLoc;  
    private double yLoc;  
    public GameItem(double xLoc, double yLoc) {  
        this.xLoc = xLoc;  
        this.yLoc = yLoc;  
    }  
    public void use() {  
        System.out.println("Item Used");  
    }  
}
```

```
public class Weapon extends GameItem {  
    private int damage;  
    public Weapon(double xloc, double yLoc, int damage) {  
        super(xloc, yLoc);  
        this.damage = damage;  
    }  
    @Override  
    public void use() {  
        System.out.println("Damage dealt: " + this.damage);  
    }  
}
```

```
public class HealthPotion extends GameItem {  
    private int increase;  
    public HealthPotion(double xLoc, double yLoc, int increase) {  
        super(xLoc, yLoc);  
        this.increase = increase;  
    }  
}
```



# Override

- To Override a method definition
- Use the `@Override` annotation before the method
- The annotation makes your intentions clear and tells the compiler that this method will replace an inherited method

```
public class GameItem {  
    private double xLoc;  
    private double yLoc;  
    public GameItem(double xLoc, double yLoc) {  
        this.xLoc = xLoc;  
        this.yLoc = yLoc;  
    }  
    public void use() {  
        System.out.println("Item Used");  
    }  
}
```

```
public class Weapon extends GameItem {  
    private int damage;  
    public Weapon(double xloc, double yLoc, int damage) {  
        super(xloc, yLoc);  
        this.damage = damage;  
    }  
    @Override  
    public void use() {  
        System.out.println("Damage dealt: " + this.damage);  
    }  
}
```

```
public class HealthPotion extends GameItem {  
    private int increase;  
    public HealthPotion(double xLoc, double yLoc, int increase) {  
        super(xLoc, yLoc);  
        this.increase = increase;  
    }  
}
```



# Override

- The @Override annotation is optional [but recommended]
- When overriding a method, your method must have the same *signature* as the method being overwritten
  - Same name
  - Same number of parameters
  - Same parameter types
  - Same return type
- If there are any differences between the methods, the method is not overridden

```
public class GameItem {  
    private double xLoc;  
    private double yLoc;  
    public GameItem(double xLoc, double yLoc) {  
        this.xLoc = xLoc;  
        this.yLoc = yLoc;  
    }  
    public void use() {  
        System.out.println("Item Used");  
    }  
}
```

```
public class Weapon extends GameItem {  
    private int damage;  
    public Weapon(double xloc, double yLoc, int damage) {  
        super(xloc, yLoc);  
        this.damage = damage;  
    }  
  
    public void use() {  
        System.out.println("Damage dealt: " + this.damage);  
    }  
}
```

```
public class HealthPotion extends GameItem {  
    private int increase;  
    public HealthPotion(double xLoc, double yLoc, int increase) {  
        super(xLoc, yLoc);  
        this.increase = increase;  
    }  
}
```



# Override

- If you have the `@Override` annotation
- The compiler will let you know if you have mistakes in the method signature
- This code will not compile since `uSe` does not match the signature of any inherited method
- Without the `@Override` annotation:
  - This code will compile and run, but will not do what you want or expect

```
public class GameItem {  
    private double xLoc;  
    private double yLoc;  
    public GameItem(double xLoc, double yLoc) {  
        this.xLoc = xLoc;  
        this.yLoc = yLoc;  
    }  
    public void use() {  
        System.out.println("Item Used");  
    }  
}
```

```
public class Weapon extends GameItem {  
    private int damage;  
    public Weapon(double xloc, double yLoc, int damage) {  
        super(xloc, yLoc);  
        this.damage = damage;  
    }  
    @Override  
    public void uSe() {  
        System.out.println("Damage dealt: " + this.damage);  
    }  
}
```

```
public class HealthPotion extends GameItem {  
    private int increase;  
    public HealthPotion(double xLoc, double yLoc, int increase) {  
        super(xLoc, yLoc);  
        this.increase = increase;  
    }  
}
```



# Incoming Memory Diagram!!



```
public class GameItem {
    private double xLoc;
    private double yLoc;
    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }
    public void use() {
        System.out.println("Item Used");
    }
}
```

```
public class Weapon extends GameItem {
    private int damage;
    public Weapon(double xloc, double yLoc, int damage) {
        super(xloc, yLoc);
        this.damage = damage;
    }
    @Override
    public void use() {
        System.out.println("Damage dealt: " + this.damage);
    }
}
```

```
public class HealthPotion extends GameItem {
    private int increase;
    public HealthPotion(double xLoc, double yLoc, int increase) {
        super(xLoc, yLoc);
        this.increase = increase;
    }
}
```

```
public static void main(String[] args) {  
    ➡ Weapon weapon = new Weapon(1.3, 0.7, 100);  
    HealthPotion potion = new HealthPotion(10.0, 0.0, 6);  
    weapon.use();  
    potion.use();  
}
```

[illegible]

- What will happen when the use method is called?
- There are 2 definitions of the method in 2 different classes

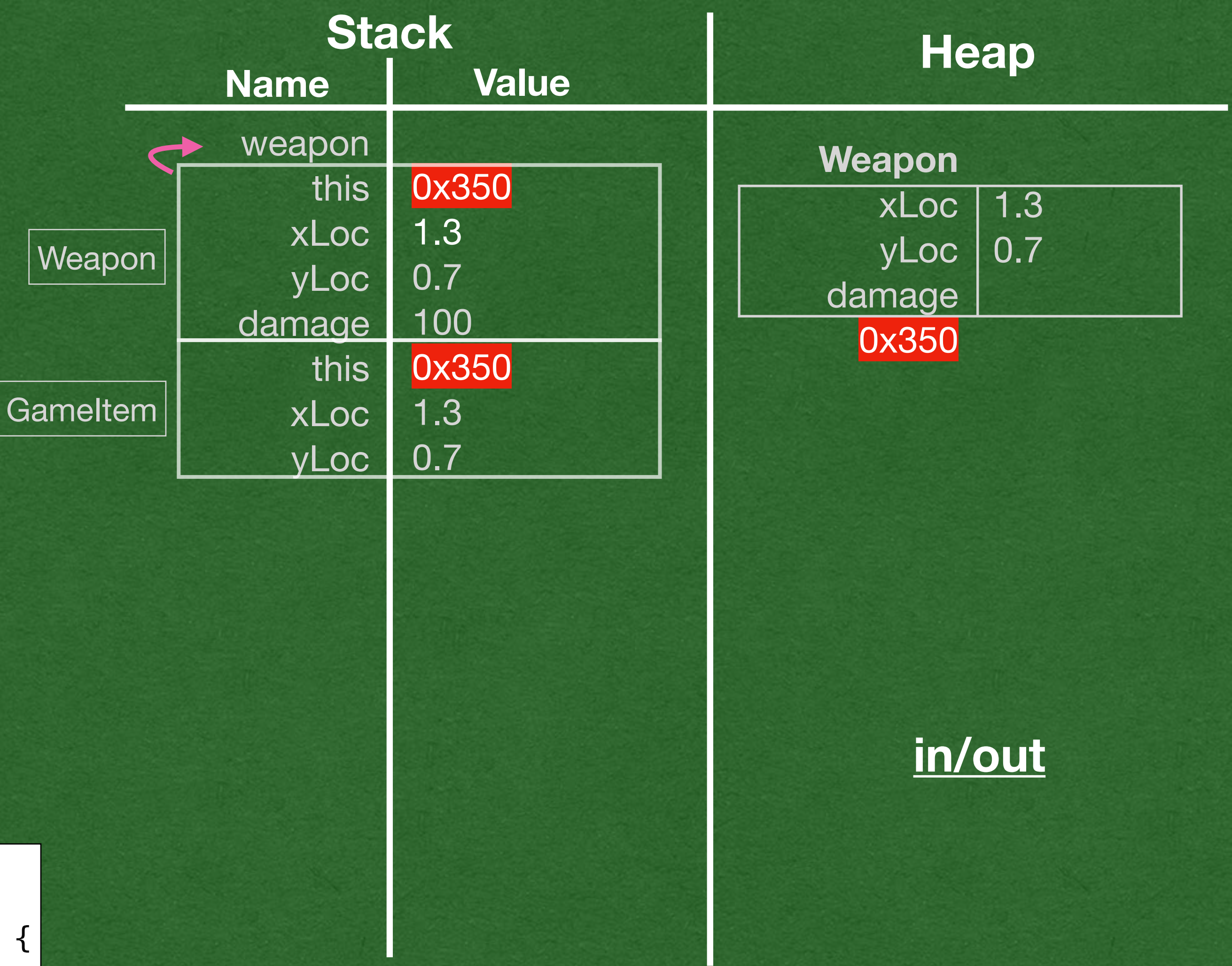


```
public class GameItem {
    private double xLoc;
    private double yLoc;
    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }
    public void use() {
        System.out.println("Item Used");
    }
}
```

```
public class Weapon extends GameItem {
    private int damage;
    public Weapon(double xloc, double yLoc, int damage) {
        super(xloc, yLoc);
        this.damage = damage;
    }
    @Override
    public void use() {
        System.out.println("Damage dealt: " + this.damage);
    }
}
```

```
public class HealthPotion extends GameItem {
    private int increase;
    public HealthPotion(double xLoc, double yLoc, int increase) {
        super(xLoc, yLoc);
        this.increase = increase;
    }
}
```

```
public static void main(String[] args) {
    Weapon weapon = new Weapon(1.3, 0.7, 100);
    HealthPotion potion = new HealthPotion(10.0, 0.0, 6);
    weapon.use();
    potion.use();
}
```



- When a class extends another class:
- Objects inherit all the instance variables of the super class
  - The super class constructor is called (Do not forget this stack frame)

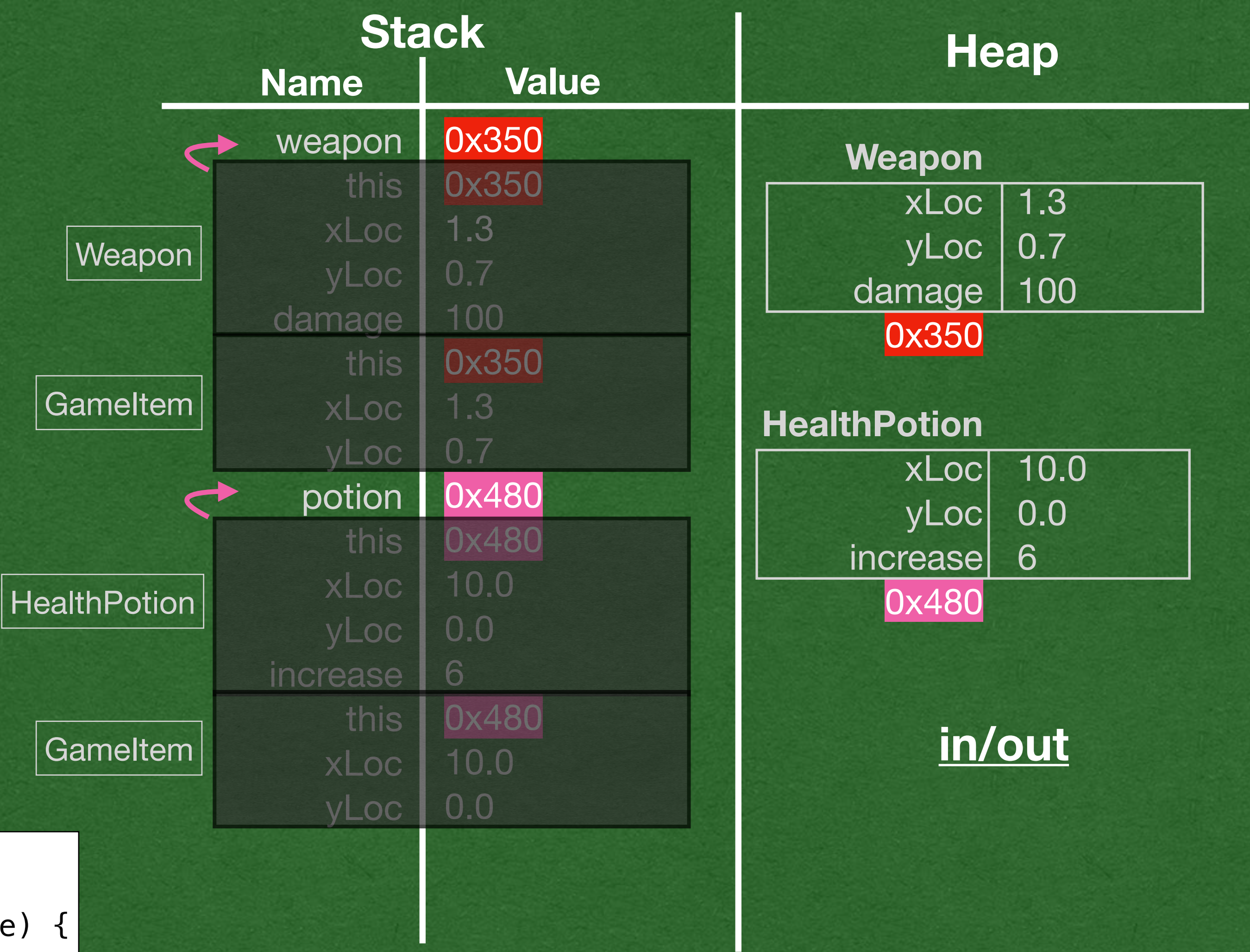


```
public class GameItem {
    private double xLoc;
    private double yLoc;
    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }
    public void use() {
        System.out.println("Item Used");
    }
}
```

```
public class Weapon extends GameItem {
    private int damage;
    public Weapon(double xloc, double yLoc, int damage) {
        super(xloc, yLoc);
        this.damage = damage;
    }
    @Override
    public void use() {
        System.out.println("Damage dealt: " + this.damage);
    }
}
```

```
public class HealthPotion extends GameItem {
    private int increase;
    public HealthPotion(double xLoc, double yLoc, int increase) {
        super(xLoc, yLoc);
        this.increase = increase;
    }
}
```

```
public static void main(String[] args) {
    Weapon weapon = new Weapon(1.3, 0.7, 100);
    HealthPotion potion = new HealthPotion(10.0, 0.0, 6);
    weapon.use();
    potion.use();
}
```



- The same applies to HealthPotion
- Do not forget the super constructor stack frame

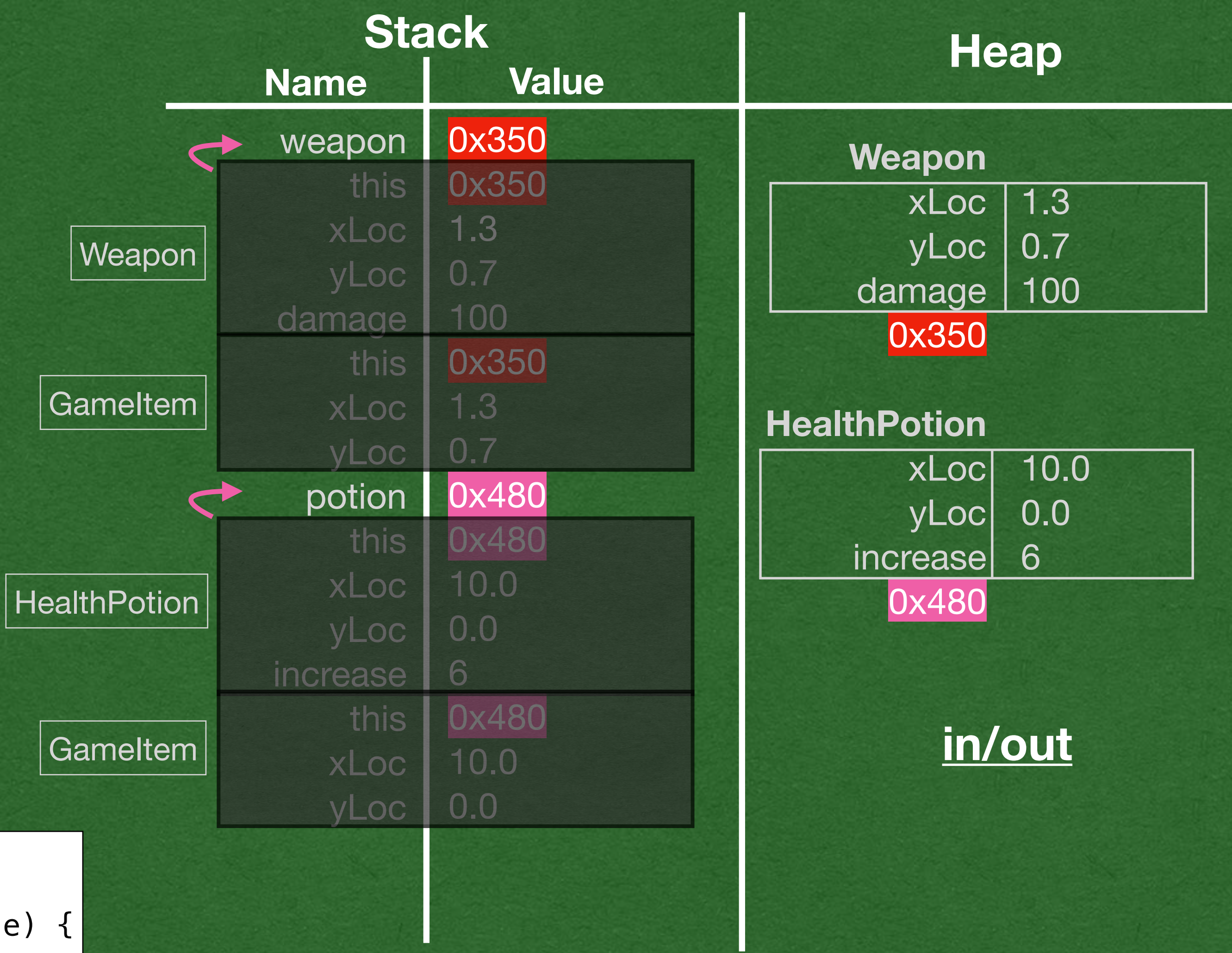


```
public class GameItem {
    private double xLoc;
    private double yLoc;
    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }
    public void use() {
        System.out.println("Item Used");
    }
}
```

```
public class Weapon extends GameItem {
    private int damage;
    public Weapon(double xloc, double yLoc, int damage) {
        super(xloc, yLoc);
        this.damage = damage;
    }
    @Override
    public void use() {
        System.out.println("Damage dealt: " + this.damage);
    }
}
```

```
public class HealthPotion extends GameItem {
    private int increase;
    public HealthPotion(double xLoc, double yLoc, int increase) {
        super(xLoc, yLoc);
        this.increase = increase;
    }
}
```

```
public static void main(String[] args) {
    Weapon weapon = new Weapon(1.3, 0.7, 100);
    HealthPotion potion = new HealthPotion(10.0, 0.0, 6);
    ➡ weapon.use();
    potion.use();
}
```



- We are calling the use method
- What method will be called?
  - There are 2 different use methods

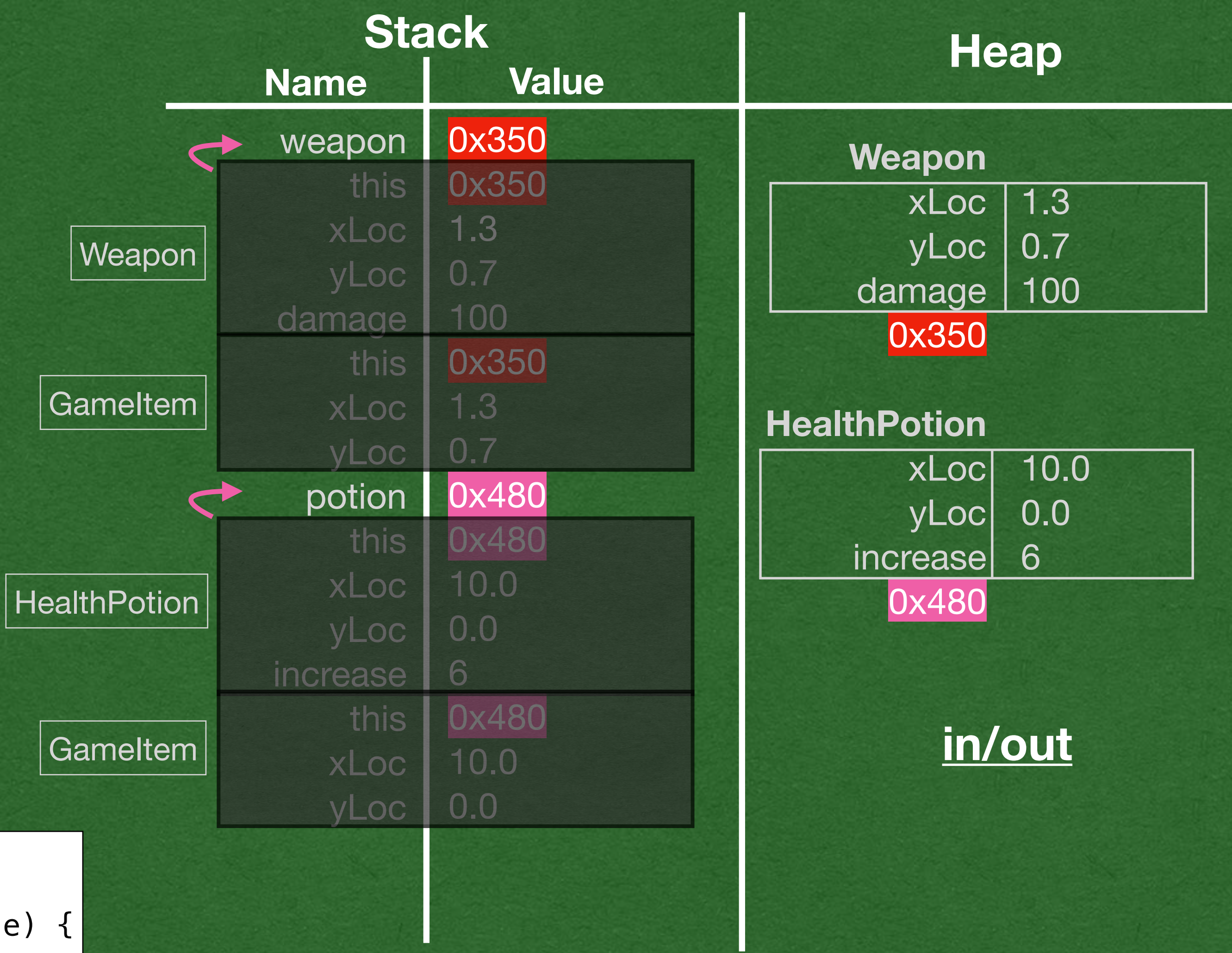


```
public class GameItem {
    private double xLoc;
    private double yLoc;
    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }
    public void use() {
        System.out.println("Item Used");
    }
}
```

```
public class Weapon extends GameItem {
    private int damage;
    public Weapon(double xloc, double yLoc, int damage) {
        super(xloc, yLoc);
        this.damage = damage;
    }
    @Override
    public void use() {
        System.out.println("Damage dealt: " + this.damage);
    }
}
```

```
public class HealthPotion extends GameItem {
    private int increase;
    public HealthPotion(double xLoc, double yLoc, int increase) {
        super(xLoc, yLoc);
        this.increase = increase;
    }
}
```

```
public static void main(String[] args) {
    Weapon weapon = new Weapon(1.3, 0.7, 100);
    HealthPotion potion = new HealthPotion(10.0, 0.0, 6);
    ➡ weapon.use();
    potion.use();
}
```



- Follow the type of the calling object!
- This call is from an object of type **Weapon**
  - Look in the **Weapon** class

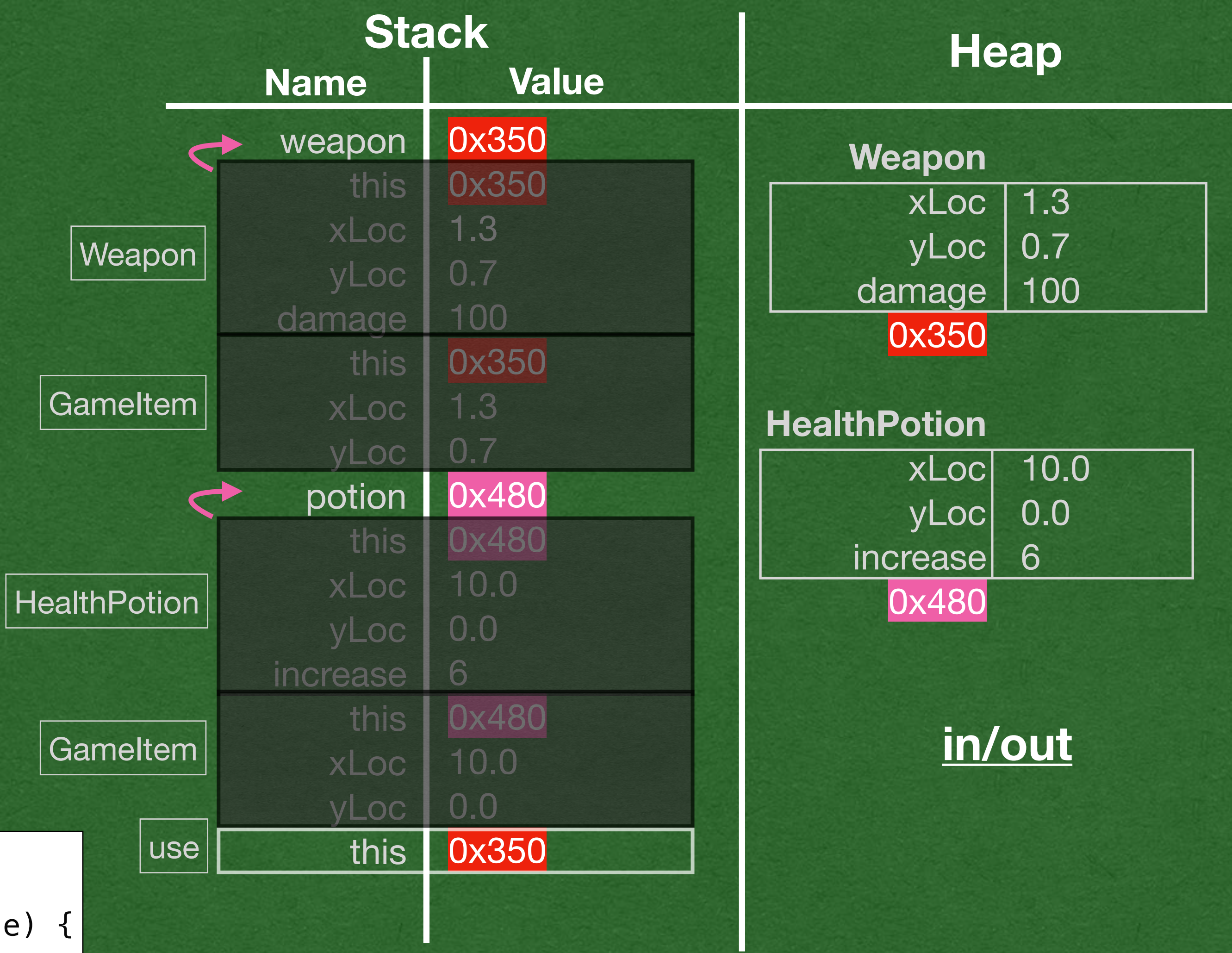


```
public class GameItem {
    private double xLoc;
    private double yLoc;
    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }
    public void use() {
        System.out.println("Item Used");
    }
}
```

```
public class Weapon extends GameItem {
    private int damage;
    public Weapon(double xloc, double yLoc, int damage) {
        super(xloc, yLoc);
        this.damage = damage;
    }
    @Override
    public void use() {
        System.out.println("Damage dealt: " + this.damage);
    }
}
```

```
public class HealthPotion extends GameItem {
    private int increase;
    public HealthPotion(double xLoc, double yLoc, int increase) {
        super(xLoc, yLoc);
        this.increase = increase;
    }
}
```

```
public static void main(String[] args) {
    Weapon weapon = new Weapon(1.3, 0.7, 100);
    HealthPotion potion = new HealthPotion(10.0, 0.0, 6);
    weapon.use();
    potion.use();
}
```



- We find a use method in the Weapon class
- This is the method that's called

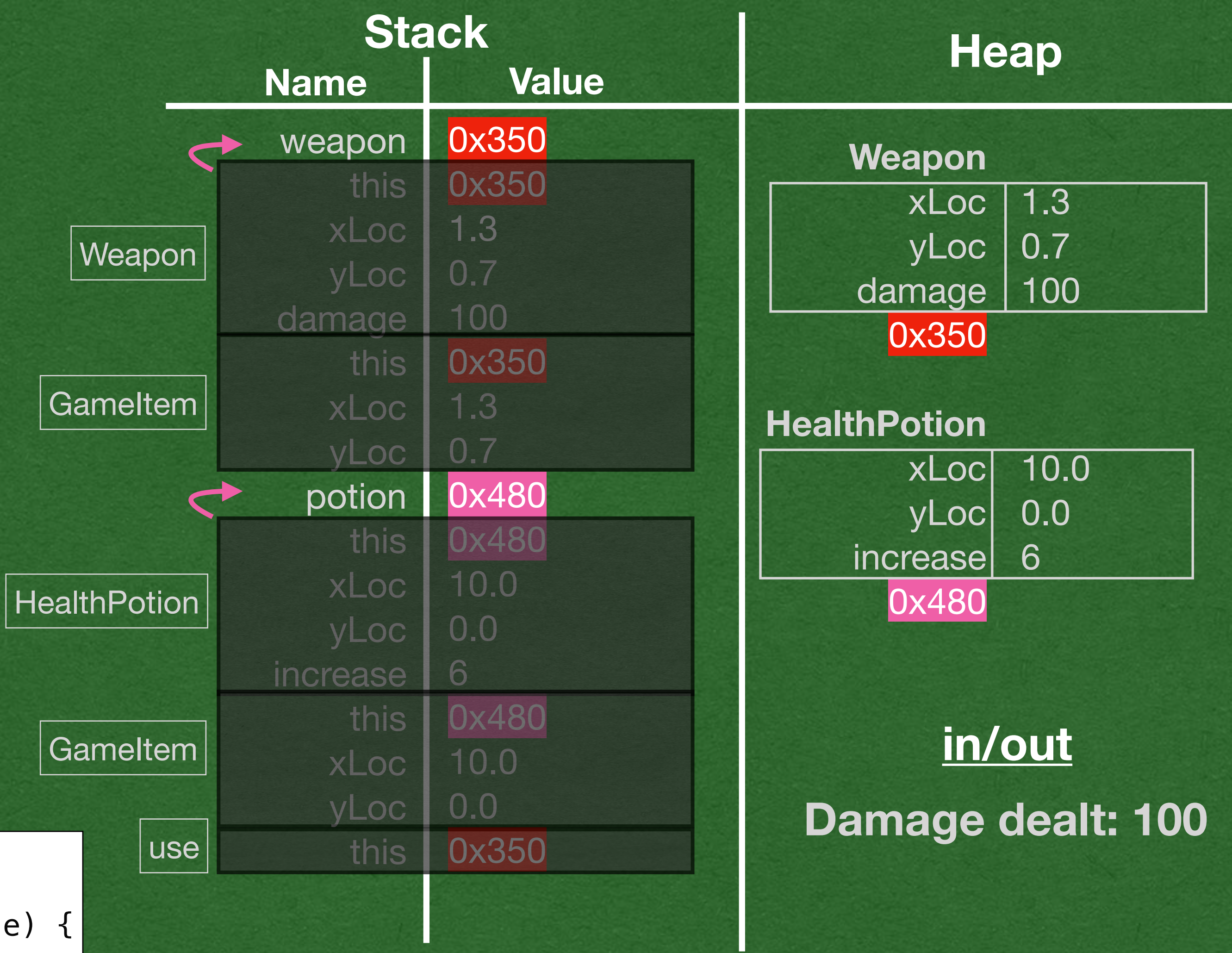


```
public class GameItem {
    private double xLoc;
    private double yLoc;
    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }
    public void use() {
        System.out.println("Item Used");
    }
}
```

```
public class Weapon extends GameItem {
    private int damage;
    public Weapon(double xloc, double yLoc, int damage) {
        super(xloc, yLoc);
        this.damage = damage;
    }
    @Override
    public void use() {
        System.out.println("Damage dealt: " + this.damage);
    }
}
```

```
public class HealthPotion extends GameItem {
    private int increase;
    public HealthPotion(double xLoc, double yLoc, int increase) {
        super(xLoc, yLoc);
        this.increase = increase;
    }
}
```

```
public static void main(String[] args) {
    Weapon weapon = new Weapon(1.3, 0.7, 100);
    HealthPotion potion = new HealthPotion(10.0, 0.0, 6);
    weapon.use();
    → potion.use();
}
```



- Follow the same steps for the next call
- The calling object has type HealthPotion
  - Look in the HealthPotion class

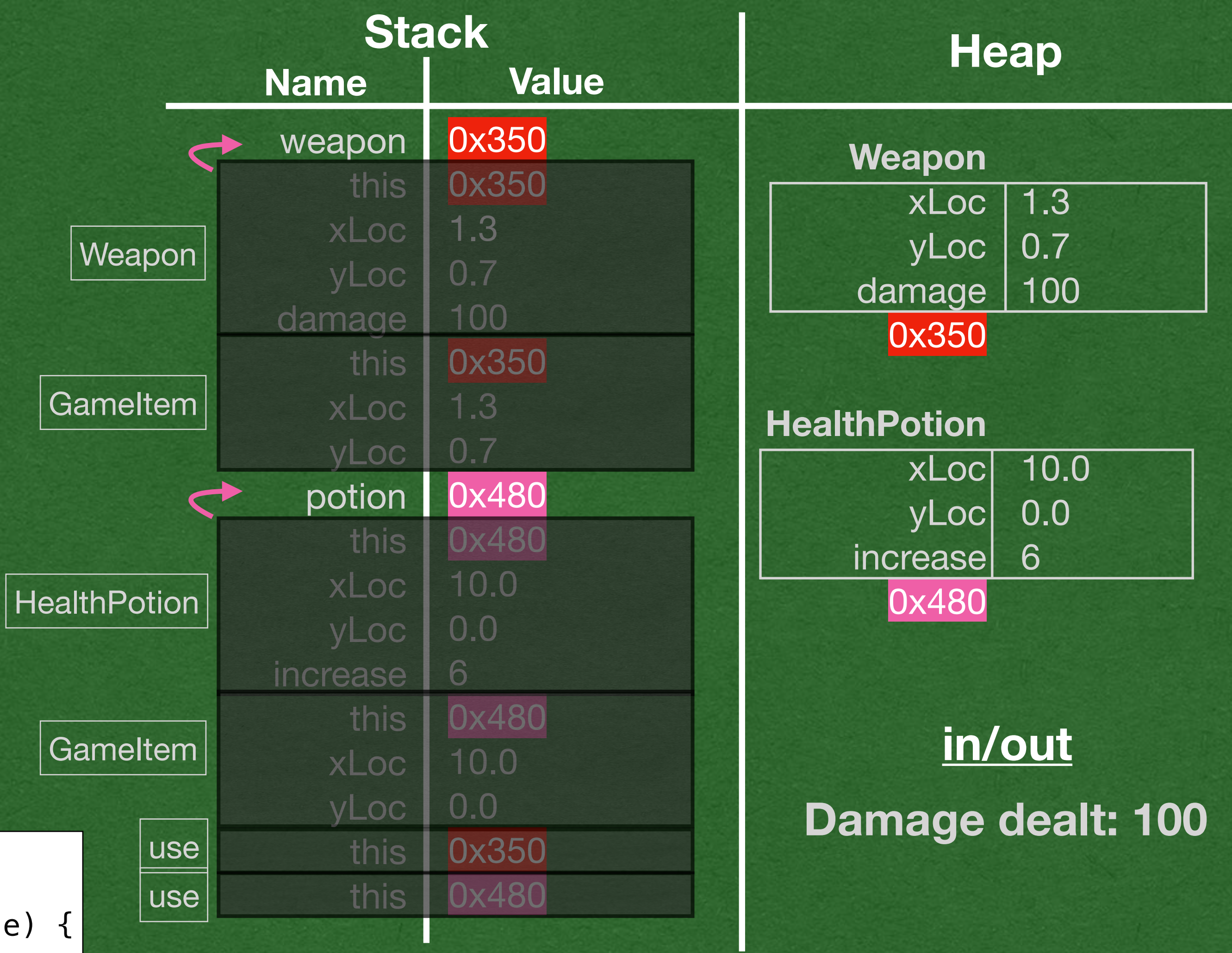


```
public class GameItem {
    private double xLoc;
    private double yLoc;
    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }
    ➡ public void use() {
        System.out.println("Item Used");
    }
}
```

```
public class Weapon extends GameItem {
    private int damage;
    public Weapon(double xloc, double yLoc, int damage) {
        super(xloc, yLoc);
        this.damage = damage;
    }
    @Override
    public void use() {
        System.out.println("Damage dealt: " + this.damage);
    }
}
```

```
public class HealthPotion extends GameItem {
    private int increase;
    public HealthPotion(double xLoc, double yLoc, int increase) {
        super(xLoc, yLoc);
        this.increase = increase;
    }
}
```

```
public static void main(String[] args) {
    Weapon weapon = new Weapon(1.3, 0.7, 100);
    HealthPotion potion = new HealthPotion(10.0, 0.0, 6);
    ➡ weapon.use();
    potion.use();
}
```



- We don't find a method named use in the HealthPotion class
- Continue our search in it's super class
  - We find and call the use method in the GameItem class

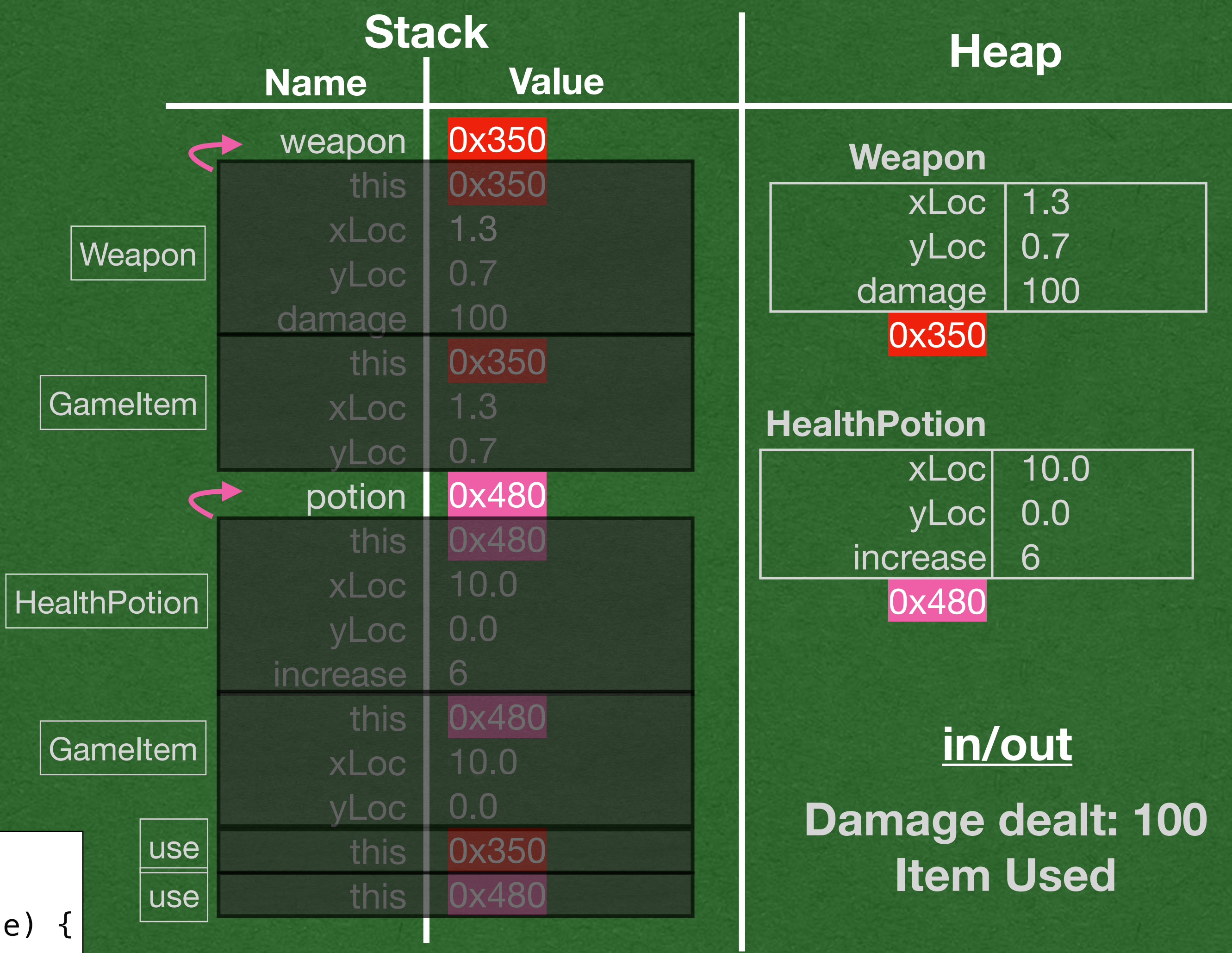


```
public class GameItem {
    private double xLoc;
    private double yLoc;
    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }
    public void use() {
        System.out.println("Item Used");
    }
}
```

```
public class Weapon extends GameItem {
    private int damage;
    public Weapon(double xloc, double yLoc, int damage) {
        super(xloc, yLoc);
        this.damage = damage;
    }
    @Override
    public void use() {
        System.out.println("Damage dealt: " + this.damage);
    }
}
```

```
public class HealthPotion extends GameItem {
    private int increase;
    public HealthPotion(double xLoc, double yLoc, int increase) {
        super(xLoc, yLoc);
        this.increase = increase;
    }
}
```

```
public static void main(String[] args) {
    Weapon weapon = new Weapon(1.3, 0.7, 100);
    HealthPotion potion = new HealthPotion(10.0, 0.0, 6);
    weapon.use();
    potion.use();
}
```



- Weapon overrides the use method, so it's use method is called
- HealthPotion does not override the use method so the inherited use method is called



toString



# The Object Class

- Every class in Java extends Object either directly or indirectly
- Every object in Java has a toString and equals method that it inherited from Object
- We can override toString if we want custom behavior



# toString

- When calling toString on HealthPotion or Weapon:
- We don't find a toString method in the class matching the type of the object
- Continue the search in GameItem
- Don't find toString in GameItem
- Continue the search in Object
- Call the method defined in Object

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

```
public class Weapon extends GameItem {  
    private int damage;  
    public Weapon(double xloc, double yLoc, int damage) {  
        super(xloc, yLoc);  
        this.damage = damage;  
    }  
}
```

```
public class HealthPotion extends GameItem {  
    private int increase;  
    public HealthPotion(double xLoc, double yLoc, int increase) {  
        super(xLoc, yLoc);  
        this.increase = increase;  
    }  
}
```

```
package java.lang;  
  
// Most code removed for space on the slide  
public class Object {  
  
    public Object() {}  
  
    public String toString() {  
        return getClass().getName() + "@" + Integer.toHexString(hashCode());  
    }  
}
```



# toString

- The toString method inherited from the Object class will return:
- {object\_type}@{hex\_value}
- week6.Weapon@452b3a41
- week6.HealthPotion@4a574795

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

```
public class Weapon extends GameItem {  
    private int damage;  
    public Weapon(double xloc, double yLoc, int damage) {  
        super(xloc, yLoc);  
        this.damage = damage;  
    }  
}
```

```
public class HealthPotion extends GameItem {  
    private int increase;  
    public HealthPotion(double xLoc, double yLoc, int increase) {  
        super(xLoc, yLoc);  
        this.increase = increase;  
    }  
}
```

```
package java.lang;  
  
// Most code removed for space on the slide  
public class Object {  
  
    public Object() {}  
  
    public String toString() {  
        return getClass().getName() + "@" + Integer.toHexString(hashCode());  
    }  
}
```



# toString

- The default behavior of toString is mostly useless
- Even the official documentation says - *"It is recommended that all subclasses override this method."*
- We will override this method

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

```
public class Weapon extends GameItem {  
    private int damage;  
    public Weapon(double xloc, double yLoc, int damage) {  
        super(xloc, yLoc);  
        this.damage = damage;  
    }  
}
```

```
public class HealthPotion extends GameItem {  
    private int increase;  
    public HealthPotion(double xLoc, double yLoc, int increase) {  
        super(xLoc, yLoc);  
        this.increase = increase;  
    }  
}
```

```
package java.lang;  
  
// Most code removed for space on the slide  
public class Object {  
  
    public Object() {}  
  
    public String toString() {  
        return getClass().getName() + "@" + Integer.toHexString(hashCode());  
    }  
}
```



# toString

- GameItem implicitly extends Object and inherits toString
- We override this default behavior to return something meaningful to our GameItems
- In previous lectures, we did this without the @Override annotation
- Weapon and HealthPotion inherit the override method from GameItem

```
public class GameItem {  
    private double xLoc;  
    private double yLoc;  
    public GameItem(double xLoc, double yLoc) {  
        this.xLoc = xLoc;  
        this.yLoc = yLoc;  
    }  
    @Override  
    public String toString() {  
        return "x: " + this.xLoc + " y:" + this.yLoc;  
    }  
}
```

```
public class Weapon extends GameItem {  
    private int damage;  
    public Weapon(double xloc, double yLoc, int damage) {  
        super(xloc, yLoc);  
        this.damage = damage;  
    }  
}
```

```
public class HealthPotion extends GameItem {  
    private int increase;  
    public HealthPotion(double xLoc, double yLoc, int increase) {  
        super(xLoc, yLoc);  
        this.increase = increase;  
    }  
}
```



# toString

- We can also override a method that has already been overridden
- In both Weapon and HealthPotion
  - Override toString again to return Strings specific to each type
- Note: In Weapon we omitted the annotation and in HealthPotion we used the annotation
- Both have the same result on our program
- No reason to mix using and not using the annotation except for an example

```
public class GameItem {
    private double xLoc;
    private double yLoc;
    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }
    @Override
    public String toString() {
        return "x: " + this.xLoc + " y:" + this.yLoc;
    }
}
```

```
public class Weapon extends GameItem {
    private int damage;
    public Weapon(double xloc, double yLoc, int damage) {
        super(xloc, yLoc);
        this.damage = damage;
    }
    public String toString() {
        return "Weapon Damage: " + this.damage;
    }
}
```

```
public class HealthPotion extends GameItem {
    private int increase;
    public HealthPotion(double xLoc, double yLoc, int increase) {
        super(xLoc, yLoc);
        this.increase = increase;
    }
    @Override
    public String toString() {
        return super.toString() + " - Health Potion";
    }
}
```



# super

- We saw the super keyword when calling the super classes constructor
- Another use is to call an override method
- Here, we call the GameItem's toString method
- It's common to add functionality to a method instead of completely replacing it
- Override the method, but still call the method you are replacing with *super*

```
public class GameItem {  
    private double xLoc;  
    private double yLoc;  
    public GameItem(double xLoc, double yLoc) {  
        this.xLoc = xLoc;  
        this.yLoc = yLoc;  
    }  
    @Override  
    public String toString() {  
        return "x: " + this.xLoc + " y:" + this.yLoc;  
    }  
}
```

```
public class Weapon extends GameItem {  
    private int damage;  
    public Weapon(double xloc, double yLoc, int damage) {  
        super(xloc, yLoc);  
        this.damage = damage;  
    }  
    public String toString() {  
        return "Weapon Damage: " + this.damage;  
    }  
}
```

```
public class HealthPotion extends GameItem {  
    private int increase;  
    public HealthPotion(double xLoc, double yLoc, int increase) {  
        super(xLoc, yLoc);  
        this.increase = increase;  
    }  
    @Override  
    public String toString() {  
        return super.toString() + " - Health Potion";  
    }  
}
```



# Another Memory Diagram

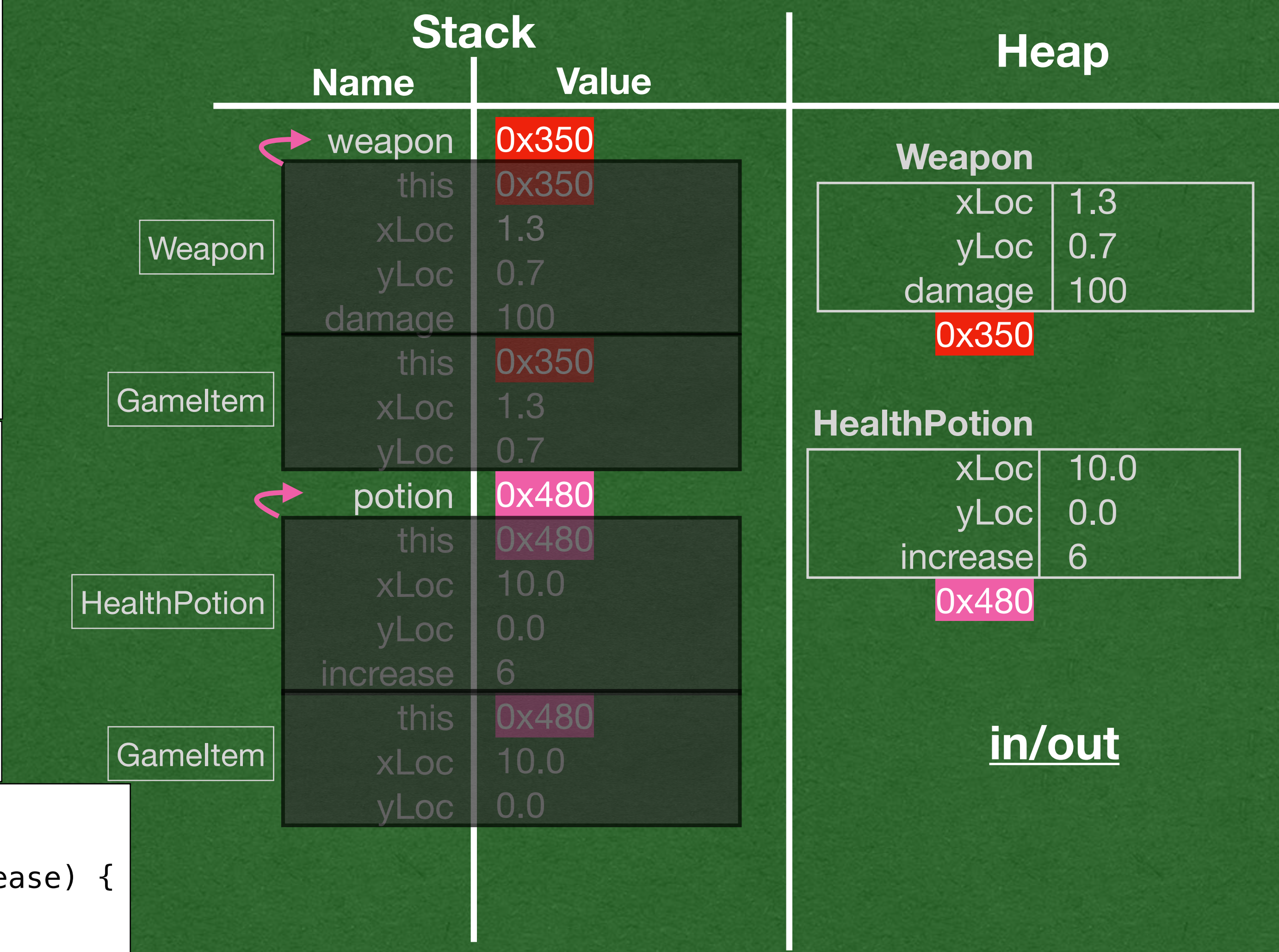


```
public class GameItem {
    private double xLoc;
    private double yLoc;
    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }
    @Override
    public String toString() {
        return "x: " + this.xLoc + " y:" + this.yLoc;
    }
}
```

```
public class Weapon extends GameItem {
    private int damage;
    public Weapon(double xloc, double yLoc, int damage) {
        super(xloc, yLoc);
        this.damage = damage;
    }
    public String toString() {
        return "Weapon Damage: " + this.damage;
    }
}
```

```
public class HealthPotion extends GameItem {
    private int increase;
    public HealthPotion(double xLoc, double yLoc, int increase) {
        super(xLoc, yLoc);
        this.increase = increase;
    }
    @Override
    public String toString() {
        return super.toString() + " - Health Potion";
    }
}
```

```
Weapon weapon = new Weapon(1.3, 0.7, 100);
HealthPotion potion = new HealthPotion(10.0, 0.0, 6);
➡ System.out.println(weapon);
System.out.println(potion);
```



- What happens when we print Weapons and HealthPotions to the screen?



```

public class GameItem {
    private double xLoc;
    private double yLoc;
    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }
    @Override
    public String toString() {
        return "x: " + this.xLoc + " y:" + this.yLoc;
    }
}

```

```

public class Weapon extends GameItem {
    private int damage;
    public Weapon(double xloc, double yLoc, int damage) {
        super(xloc, yLoc);
        this.damage = damage;
    }
    public String toString() {
        return "Weapon Damage: " + this.damage;
    }
}

```

```

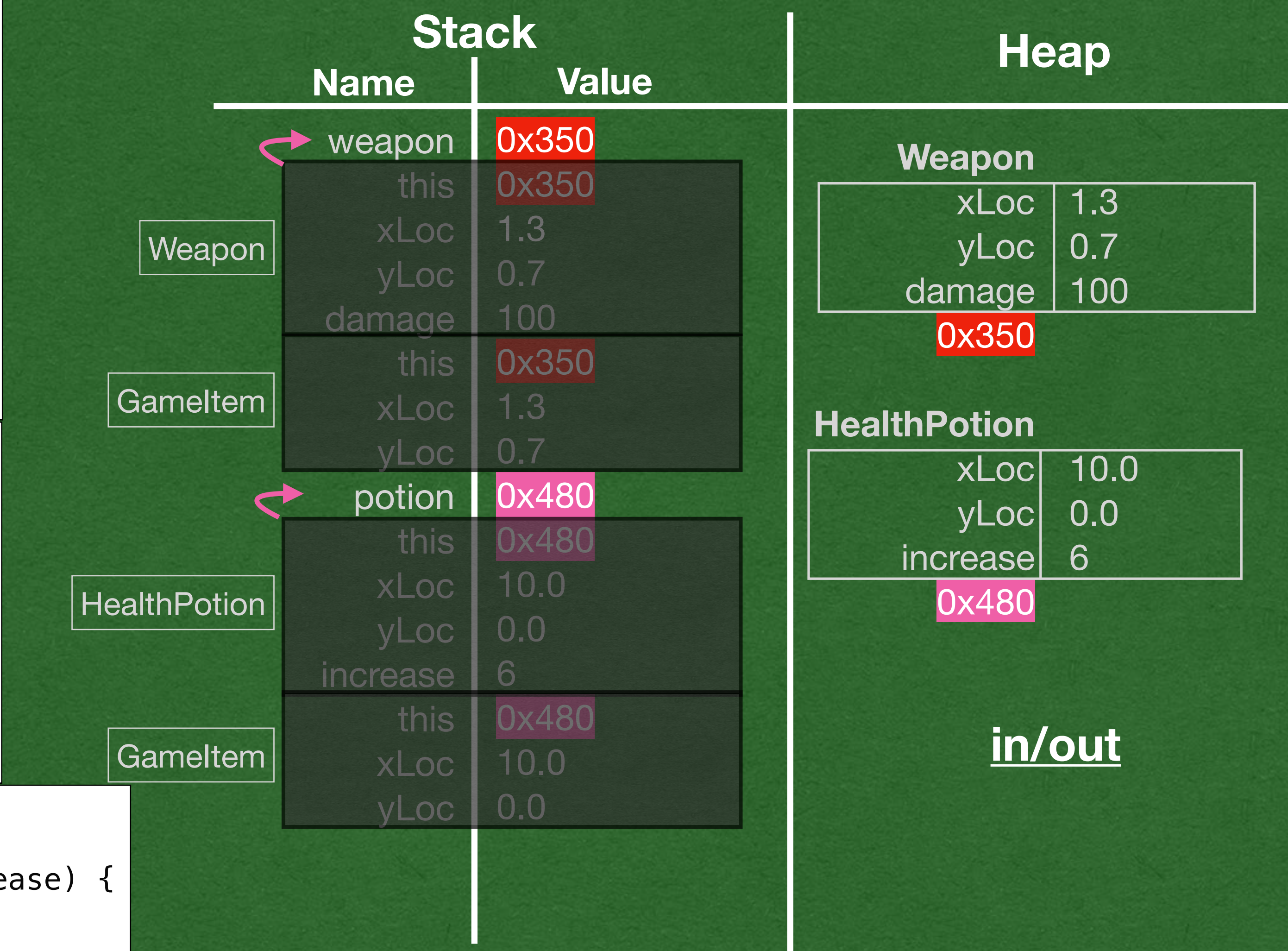
public class HealthPotion extends GameItem {
    private int increase;
    public HealthPotion(double xLoc, double yLoc, int increase) {
        super(xLoc, yLoc);
        this.increase = increase;
    }
    @Override
    public String toString() {
        return super.toString() + " - Health Potion";
    }
}

```

```

Weapon weapon = new Weapon(1.3, 0.7, 100);
HealthPotion potion = new HealthPotion(10.0, 0.0, 6);
➡ System.out.println(weapon);
System.out.println(potion);

```



- System.out.println will call toString
- You must call toString in your memory diagrams if you have the code for a toString method

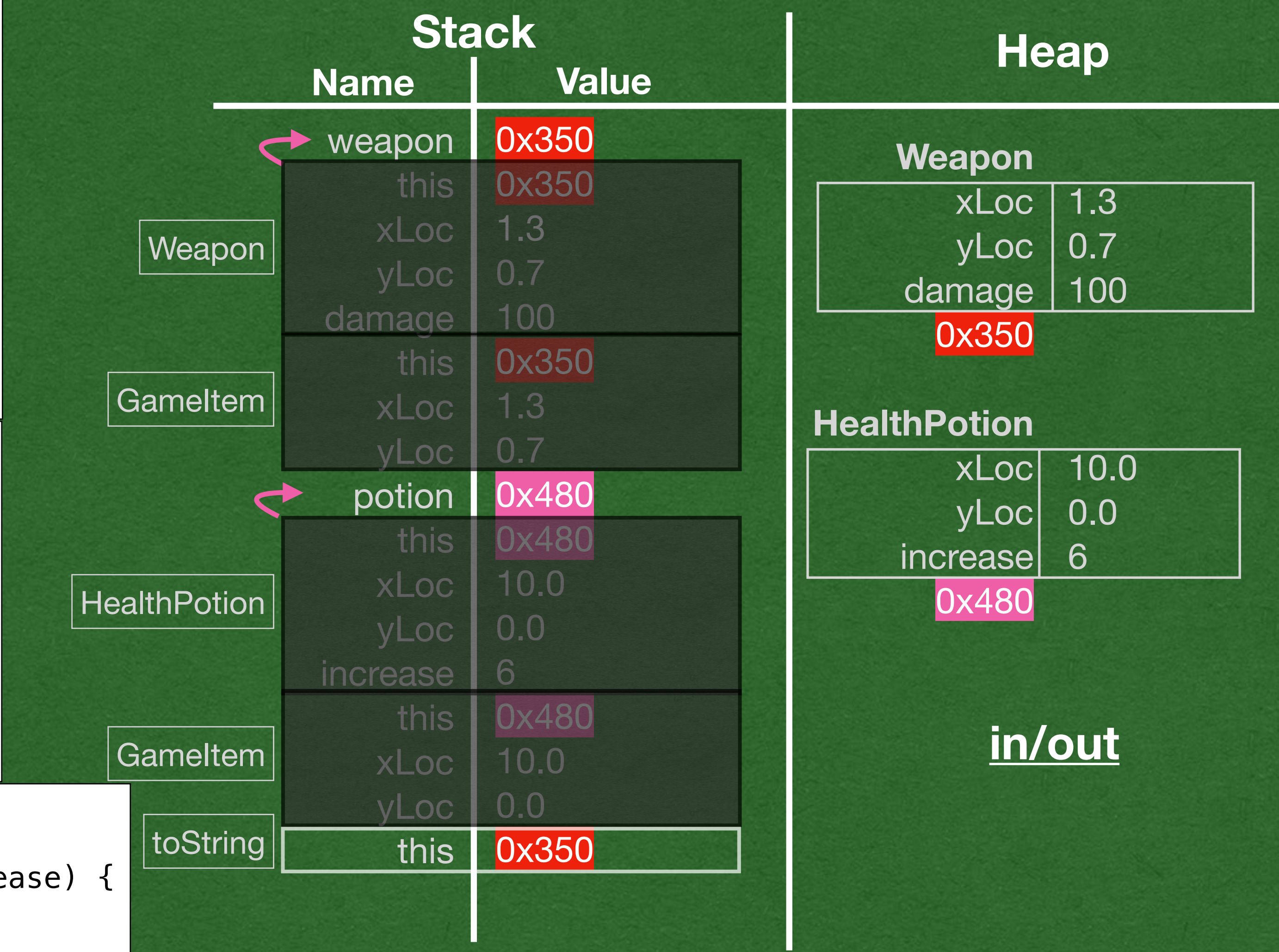


```
public class GameItem {
    private double xLoc;
    private double yLoc;
    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }
    @Override
    public String toString() {
        return "x: " + this.xLoc + " y:" + this.yLoc;
    }
}
```

```
public class Weapon extends GameItem {
    private int damage;
    public Weapon(double xloc, double yLoc, int damage) {
        super(xloc, yLoc);
        this.damage = damage;
    }
    ➡ public String toString() {
        return "Weapon Damage: " + this.damage;
    }
}
```

```
public class HealthPotion extends GameItem {
    private int increase;
    public HealthPotion(double xLoc, double yLoc, int increase) {
        super(xLoc, yLoc);
        this.increase = increase;
    }
    @Override
    public String toString() {
        return super.toString() + " - Health Potion";
    }
}
```

```
Weapon weapon = new Weapon(1.3, 0.7, 100);
HealthPotion potion = new HealthPotion(10.0, 0.0, 6);
➡ System.out.println(weapon);
System.out.println(potion);
```



- The calling object has type Weapon
- Find toString in the Weapon class
  - The Override is implicit since there's no annotation

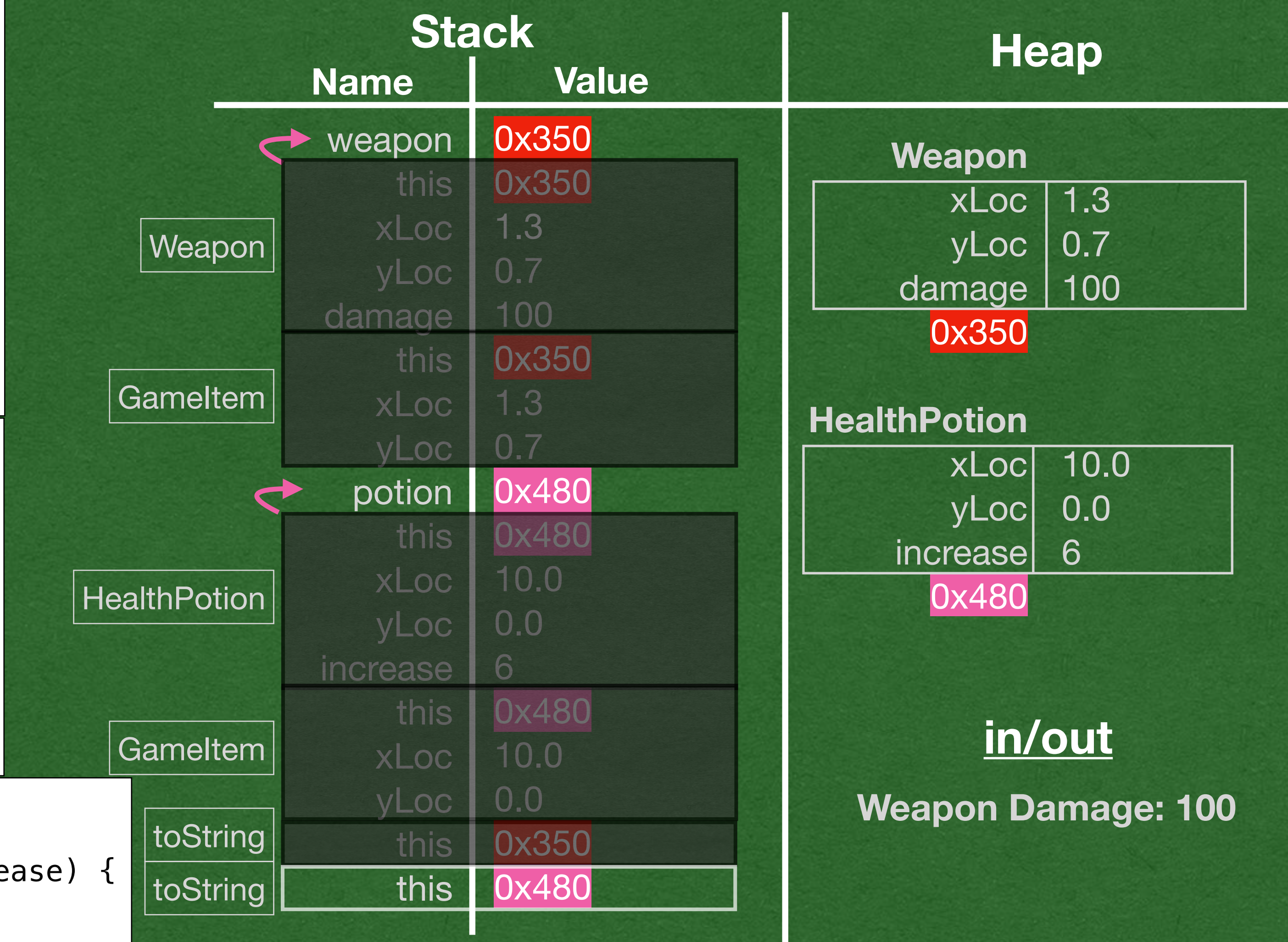


```
public class GameItem {
    private double xLoc;
    private double yLoc;
    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }
    @Override
    public String toString() {
        return "x: " + this.xLoc + " y:" + this.yLoc;
    }
}
```

```
public class Weapon extends GameItem {
    private int damage;
    public Weapon(double xloc, double yLoc, int damage) {
        super(xloc, yLoc);
        this.damage = damage;
    }
    public String toString() {
        return "Weapon Damage: " + this.damage;
    }
}
```

```
public class HealthPotion extends GameItem {
    private int increase;
    public HealthPotion(double xLoc, double yLoc, int increase) {
        super(xLoc, yLoc);
        this.increase = increase;
    }
    @Override
    ➔ public String toString() {
        return super.toString() + " - Health Potion";
    }
}
```

```
Weapon weapon = new Weapon(1.3, 0.7, 100);
HealthPotion potion = new HealthPotion(10.0, 0.0, 6);
System.out.println(weapon);
➔ System.out.println(potion);
```



- Similar for HealthPotion
- Look in the HealthPotion class and find a toString method
  - This time the Override is explicit with an annotation

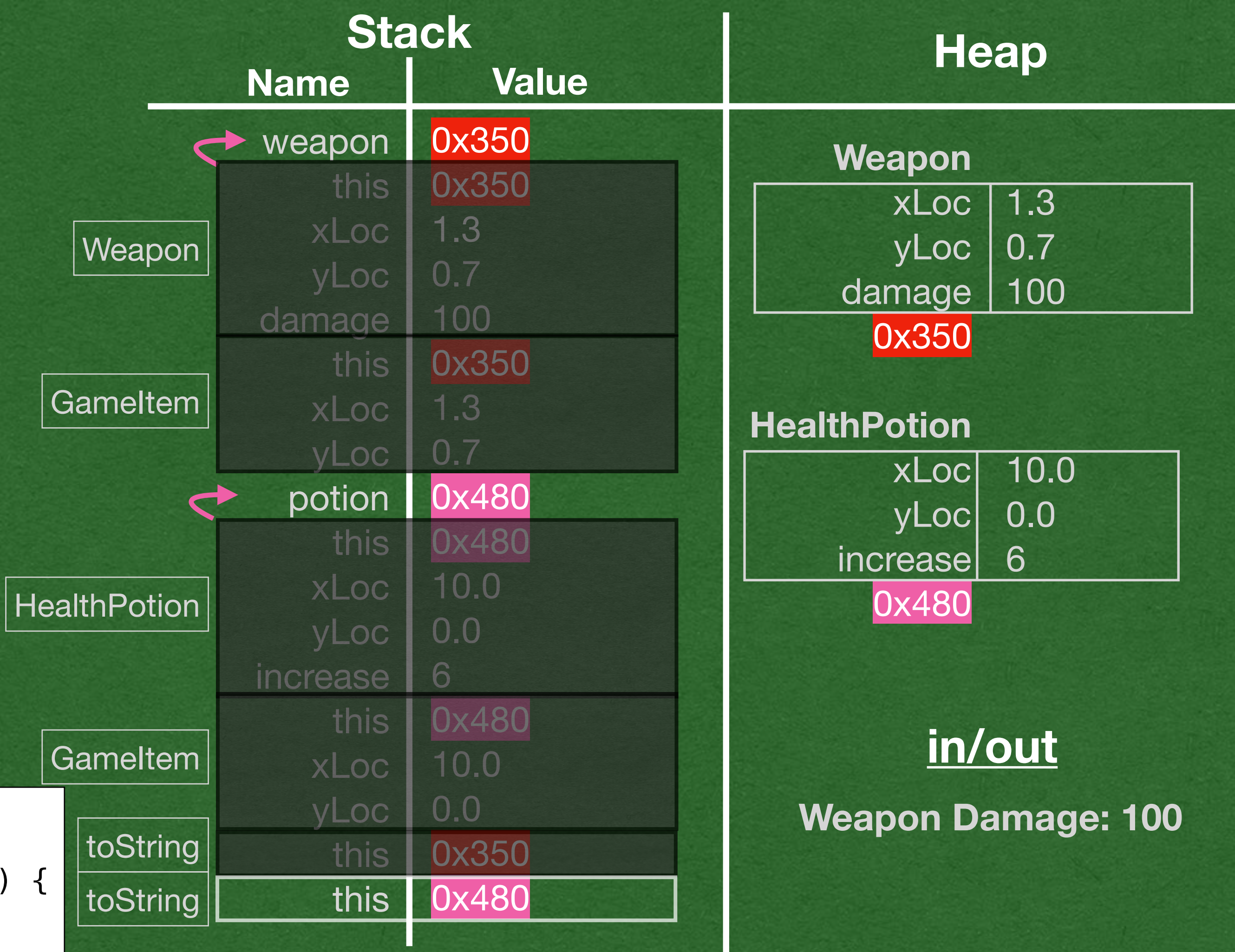


```
public class GameItem {
    private double xLoc;
    private double yLoc;
    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }
    @Override
    public String toString() {
        return "x: " + this.xLoc + " y:" + this.yLoc;
    }
}
```

```
public class Weapon extends GameItem {
    private int damage;
    public Weapon(double xloc, double yLoc, int damage) {
        super(xloc, yLoc);
        this.damage = damage;
    }
    public String toString() {
        return "Weapon Damage: " + this.damage;
    }
}
```

```
public class HealthPotion extends GameItem {
    private int increase;
    public HealthPotion(double xLoc, double yLoc, int increase) {
        super(xLoc, yLoc);
        this.increase = increase;
    }
    @Override
    public String toString() {
        return super.toString() + " - Health Potion";
    }
}
```

```
Weapon weapon = new Weapon(1.3, 0.7, 100);
HealthPotion potion = new HealthPotion(10.0, 0.0, 6);
System.out.println(weapon);
System.out.println(potion);
```



- For a super method call:
  - Look in the super class for a matching method

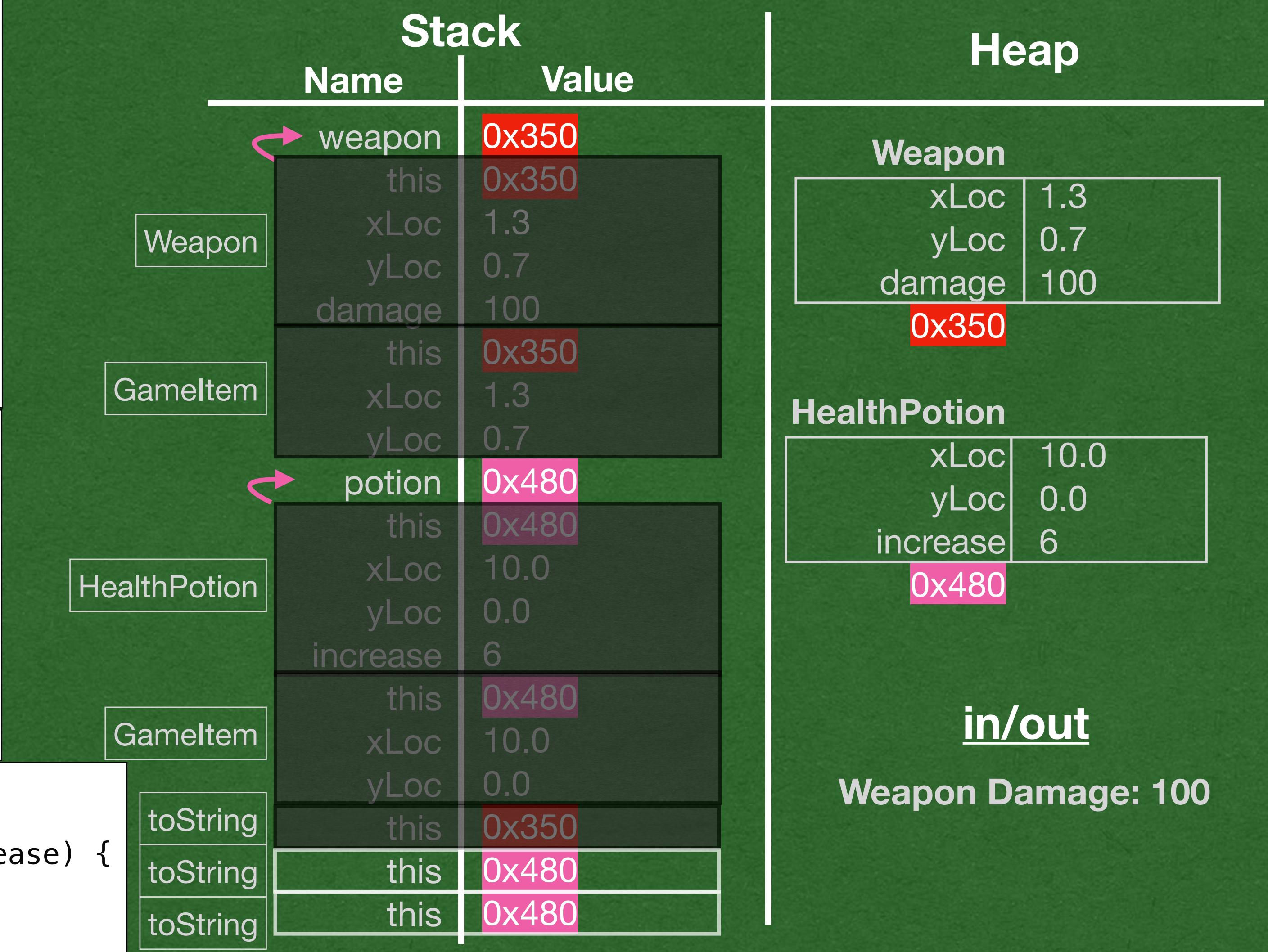


```
public class GameItem {
    private double xLoc;
    private double yLoc;
    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }
    @Override
    public String toString() {
        return "x: " + this.xLoc + " y:" + this.yLoc;
    }
}
```

```
public class Weapon extends GameItem {
    private int damage;
    public Weapon(double xloc, double yLoc, int damage) {
        super(xloc, yLoc);
        this.damage = damage;
    }
    public String toString() {
        return "Weapon Damage: " + this.damage;
    }
}
```

```
public class HealthPotion extends GameItem {
    private int increase;
    public HealthPotion(double xLoc, double yLoc, int increase) {
        super(xLoc, yLoc);
        this.increase = increase;
    }
    @Override
    public String toString() {
        return super.toString() + " - Health Potion";
    }
}
```

```
Weapon weapon = new Weapon(1.3, 0.7, 100);
HealthPotion potion = new HealthPotion(10.0, 0.0, 6);
System.out.println(weapon);
System.out.println(potion);
```



- We find a toString method in GameItem
  - This is the method called from super.toString
  - this in a super method call is the same as the original calling object

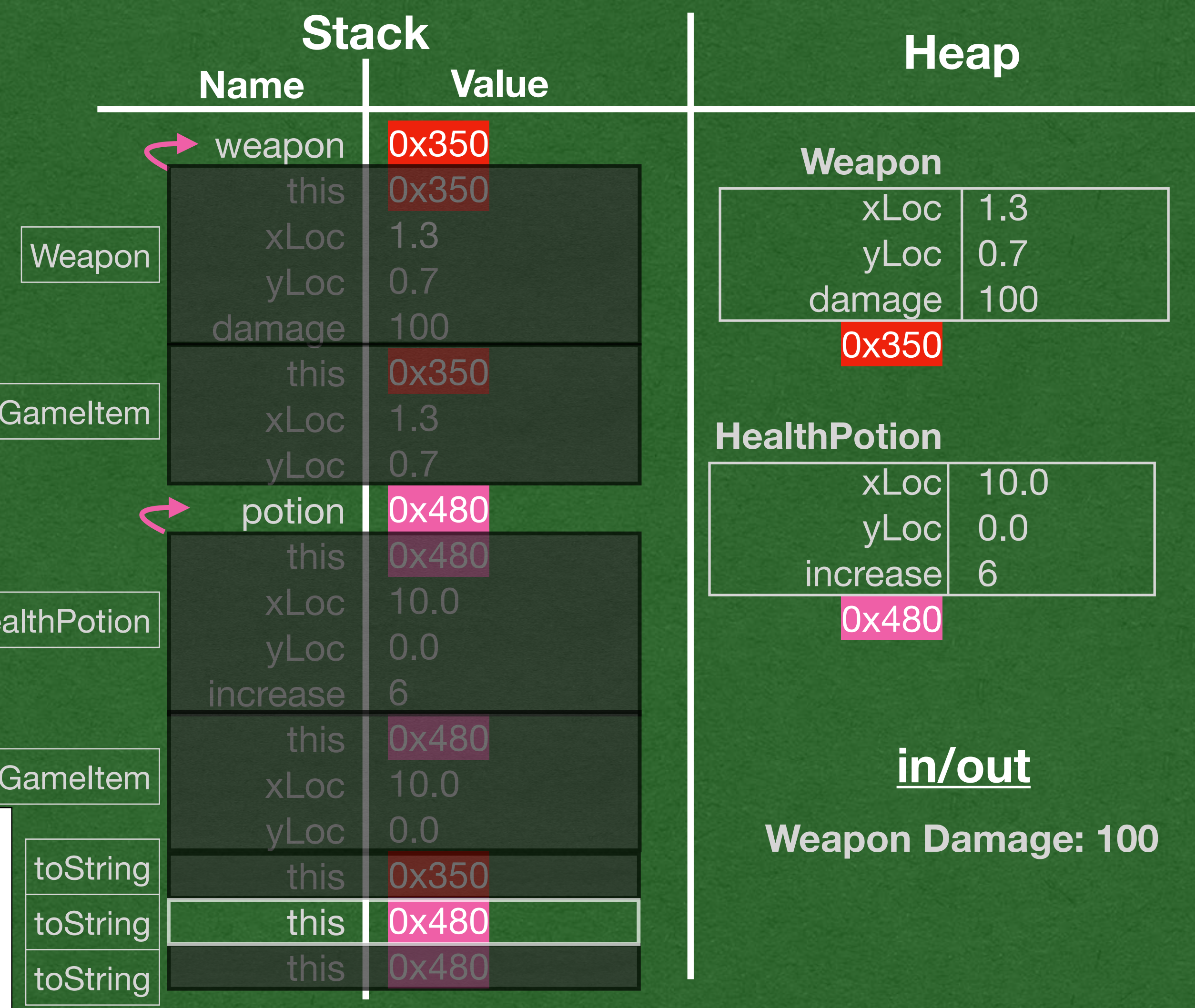


```
public class GameItem {
    private double xLoc;
    private double yLoc;
    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }
    @Override
    public String toString() {
        return "x: " + this.xLoc + " y:" + this.yLoc;
    }
}
```

```
public class Weapon extends GameItem {
    private int damage;
    public Weapon(double xloc, double yLoc, int damage) {
        super(xloc, yLoc);
        this.damage = damage;
    }
    public String toString() {
        return "Weapon Damage: " + this.damage;
    }
}
```

```
public class HealthPotion extends GameItem {
    private int increase;
    public HealthPotion(double xLoc, double yLoc, int increase) {
        super(xLoc, yLoc);
        this.increase = increase;
    }
    @Override
    public String toString() {
        return super.toString() + " - Health Potion";
    }
}
```

```
Weapon weapon = new Weapon(1.3, 0.7, 100);
HealthPotion potion = new HealthPotion(10.0, 0.0, 6);
System.out.println(weapon);
System.out.println(potion);
```



- The super method call returns "x: 10.0 y:0.0"
- The HealthPotion methods concatenates to this and returns

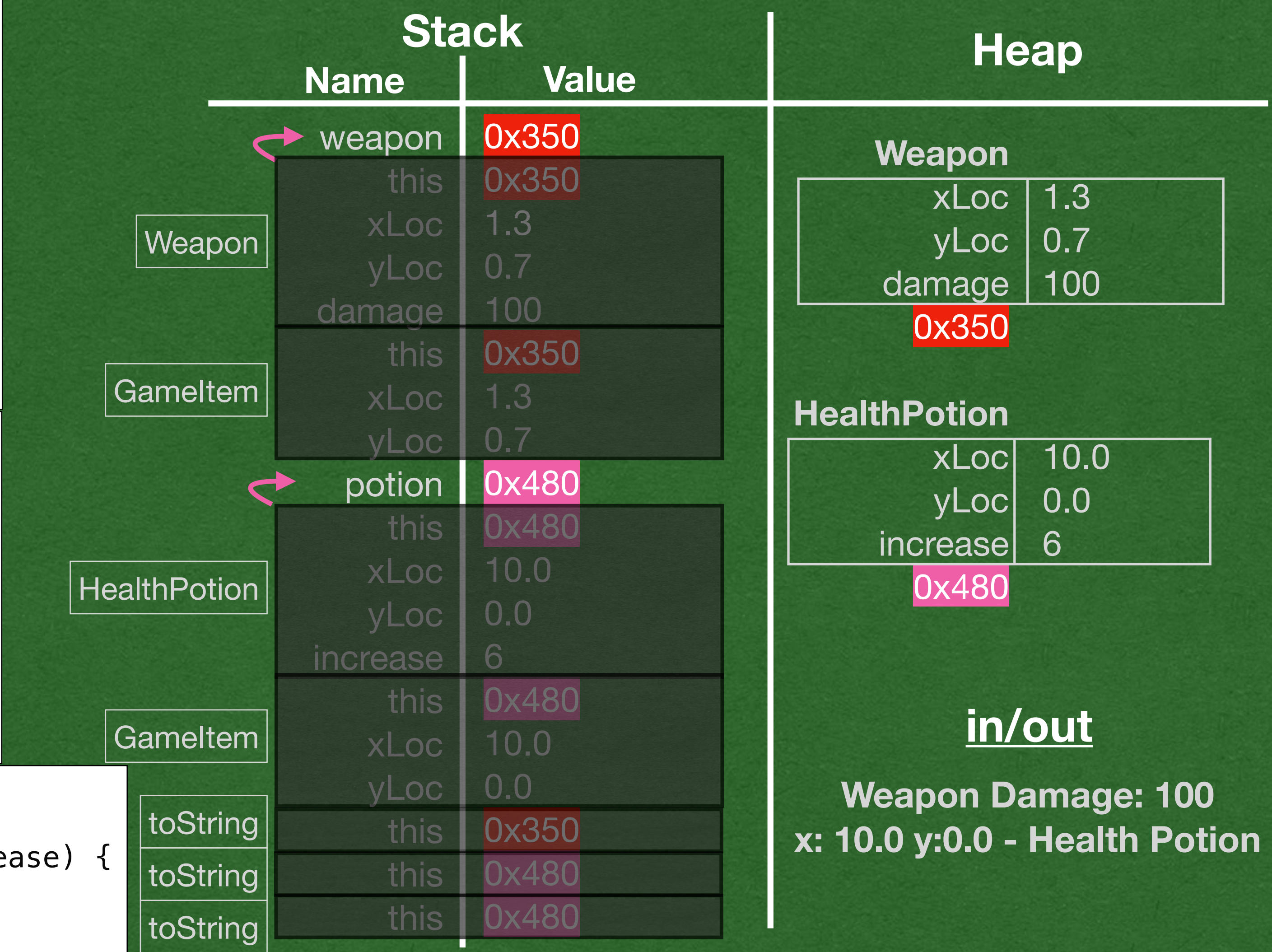


```
public class GameItem {
    private double xLoc;
    private double yLoc;
    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }
    @Override
    public String toString() {
        return "x: " + this.xLoc + " y:" + this.yLoc;
    }
}
```

```
public class Weapon extends GameItem {
    private int damage;
    public Weapon(double xloc, double yLoc, int damage) {
        super(xloc, yLoc);
        this.damage = damage;
    }
    public String toString() {
        return "Weapon Damage: " + this.damage;
    }
}
```

```
public class HealthPotion extends GameItem {
    private int increase;
    public HealthPotion(double xLoc, double yLoc, int increase) {
        super(xLoc, yLoc);
        this.increase = increase;
    }
    @Override
    public String toString() {
        return super.toString() + " - Health Potion";
    }
}
```

```
Weapon weapon = new Weapon(1.3, 0.7, 100);
HealthPotion potion = new HealthPotion(10.0, 0.0, 6);
System.out.println(weapon);
➡ System.out.println(potion);
```



- Print the final String to the screen
- End program