..

PointRunner.java	Add Session05	3 months ago
README.md	Add max element problem	3 months ago
inheritance-models.png	Remove transparency in image	3 months ago

README.md

Session 5

Table of Contents

Call By Value / Call By Copy Of Reference

In Java, arguments are ALWAYS passed to functions by means of a copy.

Primitives example:

```
class PrimitiveArg {

    // This "a" is a variable local to this function
    public static void modify(int a) {
        a = a + 1;
    }

    public static void main(String[] args) {
        int a = 10;
        modify(a);
        System.out.println(a);          // Prints out 10
        }
    }
```

This is fairly obvious. However, when it comes to object references, things may "look" complicated, but they aren't really.

Object reference example:

```
class ObjectRefArg {
    public static void modifyFail(Student s) {
        s = new Student("PETER");
    }

    public static void main(String[] args) {
        Student s = new Student("JON");
        modifyFail(s);
        System.out.println(s.getName());        // Prints JON
    }
}
```

When you pass an object reference to a function, this is what does NOT happen:

1. A replica of the object is made in the heap, and a new reference is created to point to it.
2. The original reference to the object is passed.

This is what actually happens: *A copy of the object reference is created, and passed to the function*
This means that the `s` in the `modifyFail` is only a copy of the reference in main. When you create a new object and point it there, you are only pointing the local copy of the reference to a new object.

```
class ObjectRefArg {
    public static void modifySuccess(Student s) {
        s.setName("PETER");
    }

    public static void main(String[] args) {
        Student s = new Student("JON");
        modifySuccess(s);
        System.out.println(s.getName());        // Prints PETER
    }
}
```

This example works because you use the copy of the reference to access the object on the heap, and modify it.

A good analogy is the following:
Imagine that I have a balloon (object on the object heap). I write a name (instance variable) on the balloon with a marker. I then tie a rope (object reference) to the balloon. I can now pull the balloon down anytime I want, read the name, erase it, and write a new one.

I want a friend (function) to be able to read the name, and write a new one on the balloon. If I give away my rope, I'll lose my own access to the balloon. It is also way too tedious to make a new balloon, write the name on it, tie a new rope to it, and give it to my friend (replica of object on the heap). Instead, I'll tie a new rope (copy of the reference) to my balloon, and give that to my friend.

He can now pull down the balloon using his rope (copy of the reference), and modify the name on the balloon. Since I have a rope to the same balloon, when I pull it down and check, I can see the modified values.
If my friend decides to cut the rope, blow a new balloon, and tie a new one (`s = new Student("PETER")`), he loses access to my balloon.

The `this` keyword

When you write constructors or methods, it is important that you have meaningful argument names. This is because the constructors and methods are part of the API of your class, and it is your responsibility to make it easy to use. As an example, imagine seeing this constructor in a JavaDoc:

```
Student(String n, int a, String c, int y)
```

What a nightmare! Compare it with this:

`Student(String name, int age, String college, int yearOfAdmission)`

This clearly tells you what the constructor expects, and what each argument means. However, this poses a problem when writing the constructor:

```
class Student {
    String name;

    Student(String name) {
        // The instance variable name is being hidden/shadowed by the local argument "name"

        // This is simply assigning the value of the local variable name to itself!
        name = name;
    }
}
```

To clarify, the instance variable `name` is not even visible inside the constructor as is. To prove a point:

```
// This compiles perfectly well
class Student {
    Student(String name) {
        name = name;
    }
}
```

In order to resolve this problem, we have the `this` keyword.

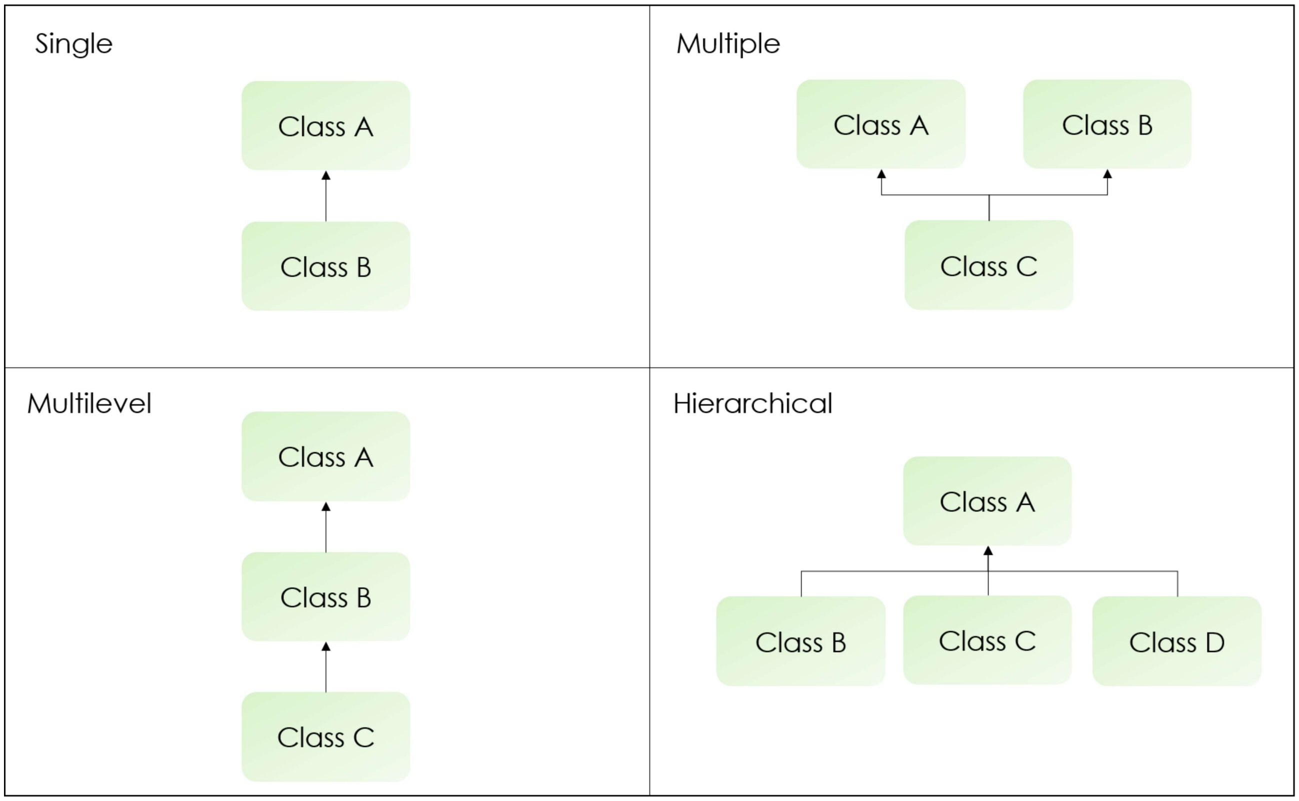
```
class Student {
    String name;

    Student(String name) {
        // this.name ensures that we access the instance variable.
        this.name = name;
    }
}
```

The `this` is simply a reference to the object on which this method is called.

Inheritance

Inheritance Models



Out of all of these, multiple inheritance is not supported in Java. This is because it suffers from the diamond problem.

Access Modifiers

In order to understand inheritance, it is important to first understand what access modifiers are, and what they do.

In Java, there are four access modifiers (in decreasing order of access):

1. public
2. protected
3. default*
4. private

A small access matrix makes it very clear what acccess they provide:

Modifier	Within class	Within package	Outside package, to sub-classes only	Outside package
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default*	Y	Y	N	N
private	Y	N	N	N

*default is not actually a keyword. It is the default access applied when you don't specify anything.

The `extends` keyword

For inheriting from classes, we use the `extends` keyword

This is best seen with an example, which is here.

Practice:

1. https://www.hackerrank.com/challenges/java-inheritance-1

2. https://www.hackerrank.com/challenges/java-inheritance-2

Overriding

Overriding is simply rewriting or redefining the implementation of an inherited method.

```java
class Rectangle {
    private int width;
    private int height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int getArea() {
        // We don't need to use the this keyword here,
        // because there are no variables that hide the instance variable
        return width * height;
    }

    public int getWidth() {
        return width;
    }

    public int getHeight() {
        return height;
    }
}

// A square is a rectangle
class Square extends Rectangle {
    private int length;

    public Square(int length) {
        this.length = length;
    }

    public int getArea() {
        return length * length;
    }

    public int getWidth() {
        return length;
    }

    public int getHeight() {
        return length;
    }
}
```

Practice:

1. https://www.hackerrank.com/challenges/java-method-overriding

Abstract Data Structures - an Overview

This is only an overview of a few common abstract data structures. They are called abstract because they are only a concept, the implementation can vary.

1. Stack
   Rule: LIFO
   Operations: push, pop, peek
   Problems:

   1. Balanced brackets
2. Queue
   Rule: FIFO
   Operations: enqueue, dequeue
   Problems:
   1. the Petrol Pump Problem
3. List
   Rule: Insertion order Operations: add, get
4. Set Rule: Unique elements
   Operations: put, contains, remove
   Problems:
   1. Counting unique characters in a string

Assignments

HackerRank

Solve these AFTER you have solved the miscellaneous section, since the classes you create there will help.

1. https://www.hackerrank.com/challenges/balanced-brackets
2. https://www.hackerrank.com/challenges/simple-text-editor
3. https://www.hackerrank.com/challenges/maximum-element

Miscellaneous

1. Implement a class called `CharStack`. It should have the following methods:
   i. `char pop()`
   ii. `void push(char c)`
   iii. `char peek()`
   iv. `boolean isEmpty()`
   v. `boolean isFull()` It should also have a maximum capacity, beyond which the stack cannot grow.
2. Implement a similar stack for Strings.
3. Implement the Sieve of Eratosthenes. What kind of a data structure is the sieve?

When solving these problems, think about each variable and method with `private`, `protected`, `public`, `static` and `final` in mind.