| | | |
|---|---|---|
| .. | | |
| Conway.java | Add Conway interface for assignment | 2 months ago |
| README.md | Add proper image links to Conway assignment | 2 months ago |

README.md

# Session 6

## Table of Contents

### RapidFire Revision

1. What is a class? (Blueprint)
2. What is an instance variable? (property of an instance)
3. What is a static variable? (property of the class)
4. What are the default values for primitives (numeric, boolean) and object references? (0, false, null)
5. What is an instance method? (behaviour of an instance)
6. What is a static method? (behaviour of the class)
7. Overloading is _ polymorphism (compile-time, static)
8. If I don't define a constructor, I get a _? (default constructor)
9. If I define any constructor, what happens to the default? (disappears)
10. What does the final modifier do to variables? (makes them initializable only once)
11. Where can you assign to a final instance variable? (declaration, all constructors, instance initialization block)
12. Where can you assign to a final static variable? (declaration, static block)
13. When I say "new", what happens? (object is created on the heap_
14. What is the first method to be pushed onto the call stack? (main)

15. Where are primitives stored? (call stack)
16. Where are object references stored? (call stack)
17. Where are instance variables stored? (heap)
18. When you pass a primitive data type to a function, it is call by _? (value/copy)
19. When you pass an object reference to a function, it is call by _? (copy of reference)
20. When you pass an array to a function, it is call by _? (copy of reference)
21. Arrays.sort sorts the array in place or returns a new array? (in-place)
22. If I pass an object reference to a function, and assign a new object to it by means of new, will it reflect in the caller? (no)
23. We use inheritance to avoid duplication of logic. This is in lieu with the design principle of _? (DRY - Don't Repeat Yourself)
24. What inheritance is not supported in Java? Because of what problem? (multiple, diamond)
25. Which is the least restrictive access modifier? (public)
26. Which is the most restrictive access modifier? (private)
27. For a variable/method defined with public/protected/default/private modifier can be accessed in _?
28. Why do we have accessor methods along with a private variable, rather than keeping the variable public? (implementation hiding, validation)
29. One example of a package? (java.util, java.io)
30. A stack has operations of _? (push, pop, peek)
31. A queue has operations of _? (enqueue, dequeue)
32. The constraint on a set is _? (unique elements)
33. The constraint on a list is _? (insertion order)

## The `final` modifier (again)

The `final` modifier can be applied to:

1. variables
2. methods
3. classes

When you apply it to *variables*, they become write-once. You can initialize them, but they can never be re-assigned to.
When you apply it to *methods*, it makes them non-overridable. You cannot override final methods in subclasses.
When you apply it to *classes*, it makes them non-inheritable. You cannot extend final classes. (You'll see later that this is a common pre-requisite for immutable classes)

## The `this` keyword (again)

The `this` keyword can be used in two contexts:

1. As a reference to the current instance
2. As a constructor of the same class.

Below is an example of why you want to use `this` as a constructor call:

```java
class Point3D {
    private int x;
    private int y;
    private int z;

    public Point3D() {
        this.x = 0;
        this.y = 0;
        this.z = 0;
    }

    public Point3D(int x) {
        this.x = x; // "this" is necessary here to distinguish from the local variable x
        this.y = 0;
        this.z = 0;
    }

    public Point3D(int x, int y) {
        this.x = x; // "this" is necessary here to distinguish from the local variable x
        this.y = y; // "this" is necessary here to distinguish from the local variable y
        this.z = 0;
    }

    public Point3D(int x, int y, int z) {
        this.x = x; // "this" is necessary here to distinguish from the local variable x
        this.y = y; // "this" is necessary here to distinguish from the local variable y
        this.z = z; // "this" is necessary here to distinguish from the local variable z
    }
}
```

As you can clearly see, this is a violation of the DRY principle; the lines for initializing the instance variables are almost copy-pasted in each constructor. Instead, the following is a much cleaner way of doing the exact same thing:

```
class Point3D {
    private int x;
    private int y;
    private int z;

    public Point3D() {
        this(0, 0, 0);
    }

    public Point3D(int x) {
        this(x, 0, 0);
    }

    public Point3D(int x, int y) {
        this(x, y, 0);
    }

    public Point3D(int x, int y, int z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}
```

## The `super` keyword (again)

Similar to the `this` keyword, the `super` keyword can be used in two contexts:

1. As a reference to the super class
2. As a constructor of the super class

### The `super` keyword as a reference

The first case is particularly useful when you want to use the functionality provided by a super class, in an overridden method in the child class:

```
class Employee {
    ...
    public double getSalary() {
        return 8.3*(0.5 * yearsOfExperience + 0.8 * referencePoints) + 2.6*(skillPoints);
    }
}


class SpecialEmployee extends Employee {
    ...
    // SpecialEmployee only has a small bonus
    // Doing it this way is bad because you are violating DRY
    public double getSalary() {
        return 8.3*(0.5 * yearsOfExperience + 0.8 * referencePoints) + 2.6*(skillPoints) + 5000;
    }
}


class SpecialEmployee extends Employee {
    ...
    // SpecialEmployee only has a small bonus
    // Doing it this way is cleaner
    public double getSalary() {
        return super.getSalary() + 5000;
    }
}
```

## The super keyword as a super-class constructor

In every constructor of a sub class, the first line is an *implicit* call to the parameterless constructor of the parent class. This is best demonstrated by the following example:

```
class A {
    A() {
        System.out.println("In A constructor");
    }
}

class B extends A {
    B() {
        // An implicit call "super()" exists here
        System.out.println("In B constructor");
    }
}

class Runner {
    public static void main(String[] args) {
        B b = new B();
    }
}

/* Output is:
In A constructor
In B constructor
*/
```

This behaviour can sometimes lead to inexplicable errors. Can you figure out why the following is a compilation failure?

```
class A {
    private int x;
    A(int t) {
        this.x = t;
        System.out.println("In A constructor");
    }
}

class B extends A {
    B() {
        System.out.println("In B constructor");
    }
}
```

```
class Runner {
	public static void main(String[] args) {
		B b = new B();
	}
}
```

As we have seen before, you can explicitly call the parent class constructor by using the `super` keyword as a constructor call.

```
class Point {
	private int x;

	Point(int x) {
		this.x = x;
	}
}

class Point2D {
	private int y;

	Point2D(int x, int y) {
		super(x);
		this.y = y;
	}
}
```

## Abstract Classes

Abstract classes are classes that are declared abstract, and may or may not have abstract methods.

Important points:

1. Abstract classes cannot be instantiated, only subclassed.
2. An abstract method is a method with the abstract modifier, and no implementation.
3. If you extend an abstract class, but don't override ALL the abstract methods, then your class MUST be abstract as well.

```
abstract class BankAccount {
	abstract void deposit(double amount);
	abstract void withdraw(double amount);
	abstract void getBalance();

	// A default implementation since 4% is a standard interest rate
	double getInterestRate() {
		return 4.0;
	}
}

class UCUCUBankAccount extends BankAccount {
	private double balance;

	void deposit(double amount) {
		if (amount < 1000) {
			System.out.println("Cannot deposit less than 1000");
		}
		else {
			balance += amount;
		}
	}

	void withdraw(double amount) {
		// Ugh, 1% transaction charge
		balance = balance - amount - 0.01 * amount;
	}
```

```
    void getBalance() {
        return balance;
    }

    // Does not override getInterestRate, gives the usual interest rate of 4%
}

class LenaBankAccount extends BankAccount {
    private double balance;

    void deposit(double amount) {
        // More lenient :)
        if (amount < 200) {
            System.out.println("Cannot deposit less than 200");
        }
        else {
            balance += amount;
        }
    }

    void withdraw(double amount) {
        // No transaction charge :)
        balance = balance - amount;
    }

    void getBalance() {
        return balance;
    }

    // Override getInterestRate to get a better interest rate
    double getInterestRate() {
        return 6.0;
    }
}
```

An excellent resource is the Oracle Javadoc for abstract classes

## Interfaces

An interface establishes a contract between software developers developing two separate pieces of software. It allows a developer to use a library without worrying about the internal implementation.

Important points:

1. All methods in an interface are *implicitly* public and abstract.
2. All variables in an interface are *implicitly* public, static and final.
3. A class can implement one or more interfaces.

```
interface Shape {
    // This is effectively a constant
    // Since it is final and static
    int NO_OF_DIMENSIONS = 2;

    double getArea();
    double getPerimeter();
}

class Circle implements Shape {
    private int radius;

    public Circle(int radius) {
        this.radius = radius;
    }
```

```
        double getArea() {
            return Math.PI * radius * radius;
        }

        double getPerimeter() {
            return 2 * Math.PI * radius;
        }
    }

    class Square implements Shape {
        private int length;

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

        double getArea() {
            return length * length;
        }

        double getPerimeter() {
            return 4 * length;
        }
    }

    class VolumeCalculator {
        // This accepts a Shape s
        // And since every Shape subclass MUST override the getArea method
        // This can calculate the volume of any uniform solid, like a cylinder or cuboid
        // It doesn't care what the actual type of shape is, i.e.
        // It doesn't care whether it is given a circle or square or whatever else
        // As long as it implements the Shape interface
        public static double getVolume(Shape s, double h) {
            return s.getArea() * h;
        }

        public static void main(String[] args) {
            Circle c = new Circle(10);
            System.out.println(getVolume(c, 10));

            Square s = new Square(10);
            System.out.println(getVolume(s, 10));
        }
    }
```

Again, an excellent resource is the Oracle Javadoc for interfaces

## Dynamic Method Dispatch

Dynamic Method Dispatch is a mechanism by which a call to an overridden method on a parent reference is resolved at runtime.

For instance, simplifying the above example, we have:

```
    class DmdExample {
        public static void main(String[] args) {
            Shape c = new Circle(10);
            System.out.println(c.getArea());

            Shape s = new Square(10);
            System.out.println(s.getArea());
        }
    }
```

As you can see, the reference is of the parent `Shape`, and the instance is of the child `Circle` or `Square`. When the method `getArea` is called, the runtime checks for an overridden method in the **instance**.

This now clarifies the two types of polymorphism and their implementations in Java:

1. Static Polymorphism = Overloading (Decided at **Compile time** )
2. Dynamic Polymorphism = Overriding (Decided at **Runtime**)

## Order of Instance Variable Initialization

TODO

## Assignments

### HackerRank

1. https://www.hackerrank.com/challenges/game-of-stones-1
2. https://www.hackerrank.com/challenges/equality-in-an-array
3. https://www.hackerrank.com/challenges/picking-numbers
4. https://www.hackerrank.com/challenges/beautiful-triplets
5. https://www.hackerrank.com/challenges/missing-numbers
6. https://www.hackerrank.com/challenges/sam-and-substrings
7. https://www.hackerrank.com/challenges/the-birthday-bar
8. https://www.hackerrank.com/challenges/electronics-shop

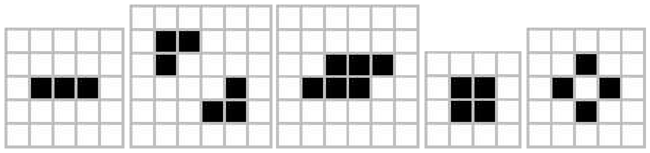
### Miscellaneous

1. Conway's Game of Life is an interesting concept. From Wikipedia

> The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970. The "game" is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how it evolves, or, for advanced "players", by creating patterns with particular properties.

You can read the rules of the game here

Here are some interersting images and gifs from the Wiki page, which make Conway easier to understand (somewhat):



Try to understand how the above gifs are behaving.

In the file Conway.java, there is an interface. Implement the interface.