Module 1 Day 13

Inheritance Part II:
Abstract Classes

Module 1 Day 13

Can you / Do You?

1. … define and use abstract in the context of a class and a method
2. … define and use final in the context of a class and a method
3. … understand what a design pattern is and how to research them
4. … explain the differences between public, private, and protected access modifiers
5. … that many keywords in Java are not for security, but rather for design and letting other developers know how to use your code

Design Patterns: What are they?

"It's a pattern!" - How many times have you heard this in class?

- A design pattern is a well-defined, tested, and proven approach that is used to solve common coding challenges.
- Being well-defined and documented, Design Patterns make code more maintainable than custom solutions.**
- Design Patterns help developers focus on the specialization of the application, not the implementation of the base application.

Design Patterns: What you need to know starting Out

There are many design patterns. They cover procedural solutions and Object Oriented solutions. In common use, they most commonly refer to 23 well documented patterns for OOP.

These 23 Design Patterns came into mainstream development starting in 1995 following the publication of a book by a group of passionate programmers at a 1994 conference.

These programmers became known as the Gang Of Four and their book, *Design Patterns: Elements of Reusable Object-Oriented Software* is still a First Edition in its 21st printing.

Design Patterns: Where you can start

- K.I.S.S(illy). - You're just learning OOP principles
- Patterns are largely an advanced topic, awareness is the goal, not proficiency
- Start by exploring one or two ***Creational Patterns***
- The Factory Pattern and, to a lesser degree, the Singleton (Lazy Singleton) are great patterns to start playing with; they are appropriate for where you are in your exposure to OOP
- Resources:
  - *Design Patterns: Elements of Reusable Object-Oriented Software*
  - Java Design Patterns – Example Tutorial

Inheritance: The ***final*** Keyword

- In member variables, the *final* keyword prevents a variable assignment from being changed; a common example can be seen in constant declarations.
- The final keyword can also be applied to classes and methods.

- When applied to a member variable, the assignments become “locked”.
- When applied to Classes and Methods, they too become “locked”.

Inheritance: The ***final*** Keyword as a ***Design*** tool

- Making methods final means that any extending children can't override what the parent has defined
    - Prevents logic that is integral to the application from being overridden by a poorly behaving subclass
    - Just a design decision that should have a good reason for being
- Making classes final means that another class can't inherit from it
    - Again, just a design decision. You should have a good reason for doing it, it is something that you should be able to articulate and defend.

Abstract Classes

Abstract classes are another tool in our designers' chest to design a system following OOP principles.

While they may, at first, appear to be more fully fleshed-out Interfaces; they are different in functionality and design intent.

Today we will talk about those considerations and usages in terms of functional design.

	Interface	Abstract Class
Constructors	✗	✓
Static Fields	✓	✓
Non-static Fields	✗	✓
Final Fields	✓	✓
Non-final Fields	✗	✓
Private Fields & Methods	✗	✓
Protected Fields & Methods	✗	✓
Public Fields & Methods	✓	✓
Abstract methods	✓	✓
Static Methods	✓	✓
Final Methods	✗	✓
Non-final Methods	✓	✓
Default Methods	✓	✗

Pramodbablad. (2019, April 08). Interface Vs Abstract Class After Java 8. Retrieved from https://javaconceptoftheday.com/interface-vs-abstract-class-after-java-8/

Abstract Classes

Abstract Classes combine some of the features we've seen in interfaces along with inheriting from a concrete class.

- Abstract methods can be extended by concrete classes.
- Abstract classes can have abstract methods
- Abstract classes can have concrete methods
- Abstract classes can have constructors
- Abstract classes, like Interfaces, cannot be instantiated

Abstract Classes : Declaration

We use the following pattern to declare and use an abstract class.

- Declaring the abstract class:

  public **abstract** class **<<Name of the Abstract Class>>** {...class body...}

- The child class (sub-class) that inherits (extends) the abstract class is declared as:

  public class **<<Name of Child Class>>** **extends** **<<Name of Abstract Class>>** {...class body...}

Abstract Classes: A Basic Example

```
package te.mobility;

public abstract class Vehicle {

        private int numberOfWheels;
        private double tankCapacity;
        private double fuelLeft;

        public double getFuelLeft() {
                return fuelLeft;
        }

        public double getTankCapacity() {
                return tankCapacity;
        }

        public Vehicle(int wheels) {
                this.numberOfWheels = wheels;
        }

        public abstract Double calcFuelPct();
}
```

```
package te.mobility;

public class Car extends Vehicle {

        public Car(int wheels) {
                super(wheels);
        }

        @Override
        public Double calcFuelPct() {
                return (super.getFuelLeft() /
                        super.getTankCapacity()) * 100;
        }
}
```

1) We need to implement the **constructor**

2) We need to implement the **abstract method**

3) We can call **concrete methods** in the abstract class!

Abstract Classes and The ***final*** Keyword

Declaring methods as final **prevents** them from being overridden by an extending subclass.

```
package te.mobility;

public abstract class Vehicle {


        public final void refuel() {
                this.fuelLeft = tankCapacity;
        }

}
```

```
package te.mobility;

public class Car extends Vehicle {

        @Override
        public void refuel() {
                this.fuelLevel = 100;
        }

}
```

(!) Attempting to implement this override will cause an error; the method is marked as final in Vehicle.

Abstract Classes: Multiple Inheritance

- Java **does** allow for the implementation of multiple interfaces; the following declaration is valid.

    public class Car **implements Driveable, Rentable** {...}

- Java **does not** allow multiple inheritance of concrete classes ***or*** abstract classes; the following declaration throws an error.

    public class Car **extends Vehicle, Asset** {...}

TO THE CODE !


The last day of being blinded by Object Oriented Programming … maybe