Java Fundamentals

Section 2 - Relationships between classes

Topics

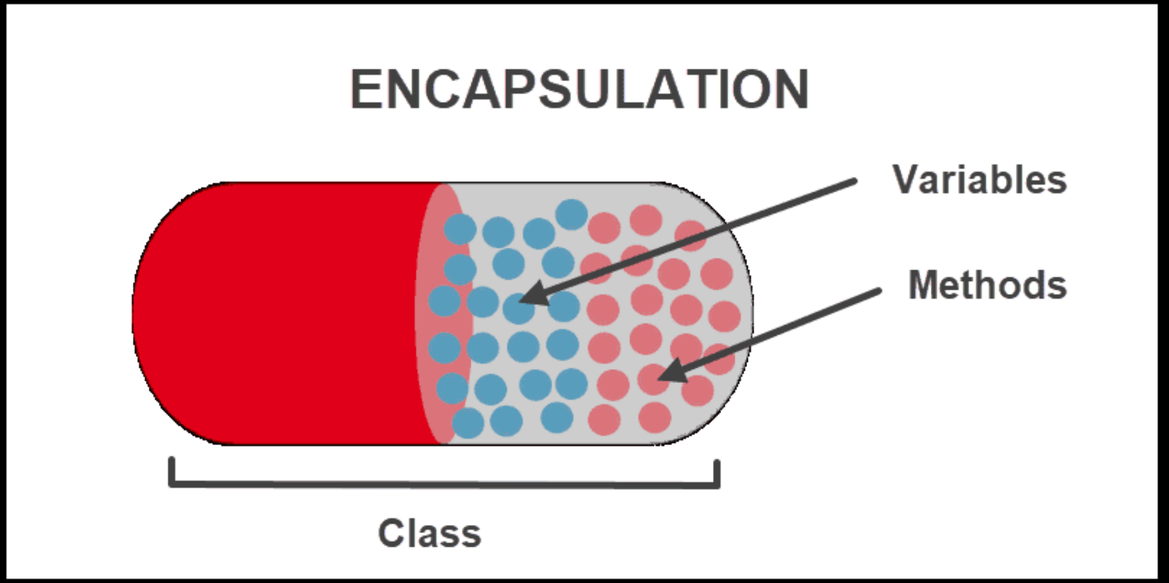
- Object oriented programming principles
- Aggregation & Composition
- What is inheritance?
- Class inheritance
- Interfaces
- Abstract classes
- Inner classes

Object oriented programming principles

Object oriented programming principles

Encapsulation

- All important information is **contained** inside an object
- only select information is **exposed**.
- implementation and state of each object are privately held inside the class.
- Other objects do not have access to the private informations / can’t modify it; they are only able to call a list of **public** methods
- This data hiding provides greater program security and control over who can modify the inner data
- Java keywords: **access modifiers (public, private, protected)**



Object oriented programming principles

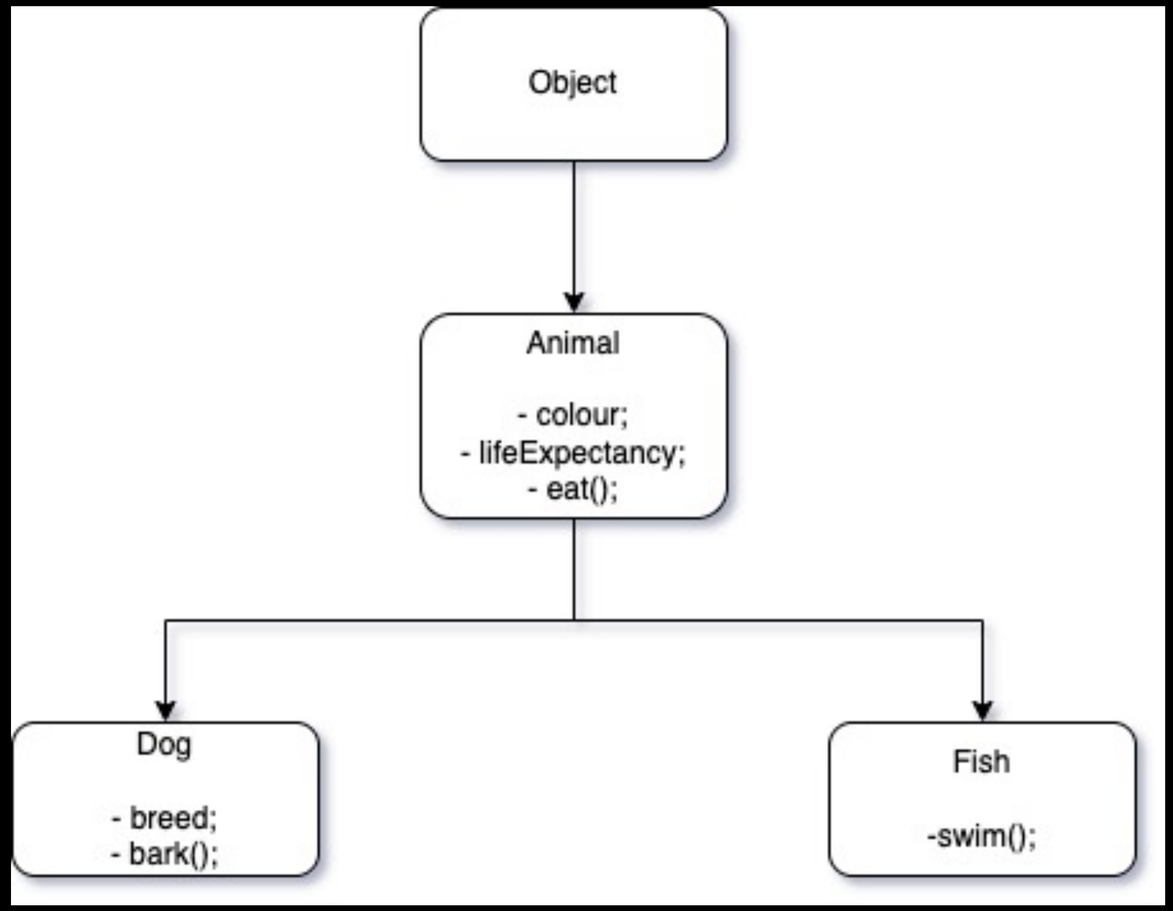
Abstraction

- Objects only **reveal** internal mechanisms that are **relevant** for the use of other objects
- Inner implementation code, data structure is **hidden**.
- Easier to use, maintain and extend when you don’t know the complexity and inner implementation
- Java keywords: **class, interface, abstract class**

Object oriented programming principles

Inheritance

- Classes can inherit (extend) other classes
- A child class (subclass) can use the functionality of the parent class (superclass).
- this property of OOP forces a more thorough data analysis and reduces development time
- All classes from java extend **Object** class
- Java keywords: **extends, implements**



Object oriented programming principles

Polymorphism

- Polymorphism: the ability of an object to take **multiple forms**.
- objects of different types can be accessed through the same interface.
- Eg: a Dog instance is at the same time an Animal and an Object
- Types of polymorphism:
  - Dynamic (runtime polymorphism)
  - Static (compile time polymorphism)

Aggregation & Composition

Aggregation & Composition

- We are using a reference of a object in a different class (as a member attribute)
- This way, the container class can reuse the code
- When using aggregation or composition, we need to be careful to instantiate and initialise the contained objects before using them

Aggregation & Composition

Aggregation

- represents a **Has-A** relationship.
- It is a unidirectional association, e.g.: a department can have students
- both entries can survive individually which means ending one entity will not affect the other entity

Aggregation and Composition

Composition

- It represents **part-of** relationship.
- In composition, both the entities are dependent on each other.
- the contained class cannot exist independently of the container. If the container is destroyed, the child is also destroyed (e.g.: a Book has Chapters, when a Book is destroyed, there is no need for its Chapters to exist anymore)

What is inheritance?

What is inheritance?

- It represents **is-a** relationship (eg: a Dog is an Animal)
- It helps us reuse the code (eg: the Dog class might benefit from the eat() method from Animal)
- New functionalities can also be added (eg: in the Dog class we can add a bark() method)
- The original class (eg: Animal is called) is called: **parent** class, **super** class or **base** class
- The class which extends the parent class (eg: Dog) is called: **child** class, **derived** class or **sub**-class

Class inheritance

Class inheritance

- Keyword: **extends**
- A child class inherits the functionality of the parent class, and can add new more specialized functionality or **override** its existing methods
- All classes in Java extend by default the **Object** class
- In Java, a class can extend only one class (no multiple inheritance).

- Keyword **super** – parent class reference. It can be used to call an overriden method from the parent class, or a constructor from the parent class.

Class inheritance

Override vs overload

- **Overriding** - replacing the functionality of a parent class method with something new
- **Overloading** - adding extra functionality, in a new method, by providing a new method with the same name, but different signature

Interfaces

Interfaces

- Keyword: **interface**
- can only contain method signatures and constant fields (public static final - constants; must be initialized)
- since Java 8 - we can provide default implementation for methods (keyword: **default**)
- methods and fields are all public (even if the public keyword is missing)
- access modifiers: public or default package
- since Java 9 – **private** methods
- before using an interface, we need a class to implement it (an interface cannot be instantiated!) - keyword **implements**
- a class can implement multiple interfaces
- a class that implements an interface must implement all its methods (except default ones)
- interfaces can have static methods; they need to be implemented
- an interface can inherit other interfaces (keyword **extends**) - for interfaces we have multiple inheritance

Abstract classes

Abstract classes

- Keyword: **abstract**
- class that can contain 'abstract' methods (without implementation)
- can contain non-constant fields
- can use different access modifiers (not everything is public by default, like with interfaces)
- can have constructors but can't be instantiated!

Inner classes

Inner classes

What are inner classes?

- **Inner** classes (or **nested** classes) are classes which are declared inside a regular class (named external)
- An inner class behaves like a member of the external class
- An inner class has access to all the members of the external class (including **private** ones)
- It can have class access modifiers (public, protected, private) but also it can be static, final, abstract.

Inner classes

Why?

- We need to solve a more complex issue, that needs a class, but we don’t want that class to be usable from the exterior

Inner classes

Types of inner classes

- regular inner classes
- anonymous inner classes
- static nested classes
- method-local inner classes / block-local inner classes

Inner classes

Regular inner classes

- It is defined as a member of a regular class (eg: Car class has an internal Engine class)
- It can be accessed by creating an instance of the external class, in a similar manner to any non-static member.
- From the internal class we can access the external class using its name and the keyword **this** (eg: Car.this)

Inner classes

Anonymous inner classes

- Anonymous classes are used in scenarios where we need a class, but we don’t have a more general use for it, it’s just needed in a specific context, used just in a few places
- Advantage: less code written
- Anonymous classes can’t have constructors
- It can extend a class **or** implement just **one** interface
- It can use parameters or variables from the method where it is declared only if they are final or effectively final(not declared final, but not changed after initialisation)

Inner classes

Static nested classes

- If we declare a static inner class, we can access it without needing an instance of the external class
- We cannot access non-static fields of the external class.