National College of Ireland

Object Oriented Programming
Design Patterns

Static keyword

Static keyword

In the Java programming language, the keyword static means that the particular member belongs to a type itself, rather than to an instance of that type.

- This means we'll create only one instance of that static member that's shared across all instances of the class.
- Instance methods belong to the object of a class.
- Instance methods can only be triggered via the object in which they are to be used.
- An object must be declared and created first, in order to use an instance variable.
- Static methods belong to the class itself.
- A static method can be triggered on the class itself or on an instance of that class.

Static keyword

In Java, when we declare a field static, exactly a single copy of that field is created and shared among all instances of that class.

It doesn’t matter how many times we instantiate a class. There will always be only one copy of static field belonging to it. The value of this static field is shared across all objects of the same class.

From the memory perspective, static variables are stored in the heap memory.

```java
public class Car {
    private String name;
    private String engine;

    public static int numberOfCars;

    public Car(String name, String engine) {
        this.name = name;
        this.engine = engine;
        numberOfCars++;
    }

    // getters and setters
}
```

Static keyword

Since static variables belong to a class, we can access them directly using the class name. So, we don’t need any object reference.

We can only declare static variables at the class level.

We can access static fields without object initialization.

Finally, we can access static fields using an object reference

(such as ford.numberOfCars++).

But we should avoid this because it becomes difficult to figure out if it’s an instance variable or a class variable.

Instead, we should always refer to static variables using the class name (Car.numberOfCars++).

Static Methods (cont'd.)

Static methods in Java are resolved at compile time. Since method overriding is part of Runtime Polymorphism, static methods can’t be overridden.

Abstract methods can’t be static.
Static methods can’t use **this** or **super** keywords.

The following combinations of the instance, class methods, and variables are valid:

- instance methods can directly access both instance methods and instance variables
- instance methods can also access static variables and static methods directly
- static methods can access all static variables and other static methods
- static methods can’t access instance variables and instance methods directly. They need some object reference to do so.

Static Comparison

Static	Nonstatic
In Java, `static` is a keyword. It also can be used as an adjective.	There is no keyword for nonstatic items. When you do not explicitly declare a field or method to be static, then it is nonstatic by default.
Static fields in a class are called class fields.	Nonstatic fields in a class are called instance variables.
Static methods in a class are called class methods.	Nonstatic methods in a class are called instance methods.
When you use a static field or method, you do not use an object; for example: `JOptionPane.showDialog();`	When you use a nonstatic field or method, you must use an object; for example: `System.out.println();`
When you create a class with a static field and instantiate 100 objects, only one copy of that field exists in memory.	When you create a class with a nonstatic field and instantiate 100 objects, then 100 copies of that field exist in memory.
When you create a static method in a class and instantiate 100 objects, only one copy of the method exists in memory and the method does not receive a `this` reference.	When you create a nonstatic method in a class and instantiate 100 objects, only one copy of the method exists in memory, but the method receives a `this` reference that contains the address of the object currently using it.
Static class variables are not instance variables. The system allocates memory to hold class variables once per class, no matter how many instances of the class you instantiate. The system allocates memory for class variables the first time it encounters a class, and every instance of a class shares the same copy of any static class variables.	Instance fields and methods are nonstatic. The system allocates a separate memory location for each nonstatic field in each instance.

Table 3-1 Comparison of static and nonstatic

Static Methods - Examples

The Math class entirely comprises static methods, hence, there is no need to create an object of the Math class, in fact, it is not possible to create an object of the Math class:

https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

The JOptionPane class is another example of a class which uses static methods, for example, .showMessageDialog():

https://docs.oracle.com/javase/7/docs/api/javax/swing/JOptionPane.html#method_summary

Design Patterns

What are Design Patterns

- Design patterns are recurring solutions to software design problems you find again and again in real-world application development.
- Patterns are about design and interaction of objects.
- Describing elegant solutions to common problems in a given context .
- Can be customized to solve a new problem in the given context.
- They are also a method of communication.

Design Patterns - Origin

- The origin of design patterns is attributed to Christopher Alexander, an architect, who introduced the concept of design patterns as a common vocabulary for design discussions.
- Alexander described a pattern as a core **solution** to a **recurrent problem** that occurs in a given **context.**
- This solution can be reused every time this problem occurs and may deviate slightly depending on the problem being solved.

Advantages

- They are reusable in multiple projects.
- They provide the solutions that help to define the system architecture.
- They capture the software engineering experiences.
- They provide transparency to the design of an application.
- They are well-proved and testified solutions since they have been built upon the knowledge and experience of expert software developers.
- Design patterns don’t guarantee an absolute solution to a problem. They provide clarity to the system architecture and the possibility of building a better system.

Gang of Four - GoF

Design Patterns: Elements of Reusable Object-Oriented Software.

Erich Gamma, Richard Helm,Ralph Johnson, John Vlissides [GOF]

- It describes patterns in a consistent way
  - Pattern Name
  - Problem Description
- When to apply the pattern
  - Solution
- Participants and their relationships
  - Consequences

Pattern Name, Description, Consequences

Name: must be self-explanatory

- Must describe the problem, solution and consequences in a word or two
- Allows developers to discuss without having to describe details of implementation
- Becomes part of the common language for developers

Description: Describes when the pattern should be used

  - It can describe specific design problems, give class or object structures and offer a list of conditions that must be met before it makes sense to apply the pattern

Consequences: The consequences will list the trade-offs (e.g. space, time) of applying the pattern

Pattern Solution

- The solution describes the elements that make up the design
- This will offer information on the relationships and responsibilities the elements may have
- The solution is more like a template that can be applied in many different situations
- The pattern will give an abstract description of a design problem and a general solution to it

(Gamma et al. 1994)

Design Pattern Classification

The GoF divided design patterns into 3 categories:

Creational

- Used to construct objects such that they can be decoupled from the implementing system

Structural Patterns

- Used to form large object structures from many disparate objects

Behavioural Patterns

- Used to manage algorithms, relationships, and responsibilities between objects

Creational Design Patterns

Strategies for creating Objects

Pattern Name	Description
Singleton	The singleton pattern restricts the initialization of a class to ensure that only one instance of the class can be created.
Factory	The factory pattern takes out the responsibility of instantiating a object from the class to a Factory class.

Creational Design Patterns

Pattern Name	Description
Abstract Factory	Allows us to create a Factory for factory classes.
Builder	Creating an object step by step and a method to finally get the object instance.
Prototype	Creating a new object instance from another similar instance and then modify according to our requirements.

Example - Singleton Design Pattern

Define a class that has only one instance and provides a global point of access to it.

- The singleton pattern defines a class that **can only have one instance** of itself and provides global access to that one instance, which means it can be accessed in all parts of your system.
- The rationale behind this pattern is that there may be only one unique object of a class in your application space
  - e.g. a GUI window or a database connection manager and it does not make sense to instantiate new instances of that class every time you need access to it.

How to create a singleton pattern

Create a private constructor of the class to restrict object creation outside of the class.
Create a private attribute of the class type that refers to the single object.
Create a public static method that allows us to create and access the object we created.

Inside the method, we will create a condition that restricts us from creating more than one object.

- A private static instance of the class
- A private constructor
- A public static method to return the object

- Example – connection to a database

How to create a singleton pattern

```
class SingletonExample {
    // private field that refers to the object of the class
    private static SingletonExample singleObject;

    private SingletonExample() {
        // private constructor of the SingletonExample class
        //that restricts creation of objs outside the class
    }
    public static SingletonExample getInstance() {
        // write code that allows us to create only one object
        //we access the object as per our need
        //this method returns the reference to the only object of the class.
        //Since the method static, it can be accessed using the class name.
    }
}
```

Structural Design Patterns

The design patterns in this category deals with the class structure such as Inheritance and Composition.

Pattern Name	Description
Adapter	Provides an interface between two unrelated entities so that they can work together.
Composite	Used when we have to implement a part-whole hierarchy. For example, a diagram made of other pieces such as circle, square, triangle, etc.
Proxy	Provide a surrogate or placeholder for another object to control access to it.

Structural Design Patterns

Pattern Name	Description
Flyweight	Caching and reusing object instances, used with immutable objects. For example, string pool.
Facade	Creating a wrapper interfaces on top of existing interfaces to help client applications.
Bridge	The bridge design pattern is used to decouple the interfaces from implementation and hiding the implementation details from the client program.
Decorator	The decorator design pattern is used to modify the functionality of an object at runtime.

Example - Decorator Pattern

- The decorator pattern is used to **attach additional responsibilities** to an object **dynamically**. Decorators provide a flexible **alternative to sub-classing** for extending functionality.
  - Also known as the Wrapper Pattern
- In an application a developer may wish to add responsibilities to individual objects and not specifically to an entire class
- Example - Using a GUI it allows the developer to add properties like borders and scroll bars to any user interface component

Example - Decorator Pattern

One way to add responsibilities is using Inheritance.

We could allow a class to inherit a border from another class, and this would put a border around every subclass instance.

- The main problem with this solution is the choice of border is made statically
- A client can’t control how and when to decorate the component with a border

Example - Decorator Pattern

A more flexible approach would be to enclose the component in another object that adds the border. This enclosing object is called a **decorator**.

- The decorator conforms to the interface of the component it decorates so that its presence is transparent to the component's clients.
- Transparency allows decorators to be nested recursively, this will facilitate an unlimited number of added responsibilities.

When to use?

The Decorator pattern can be used:

- To add responsibilities to individual objects dynamically and transparently
- To implement responsibilities that can be withdrawn
- When concrete implementations should be decoupled from responsibilities and behaviours
- When extension by sub-classing is impractical
- When a lot of little objects surrounding a concrete implementation is acceptable

Behavioural Design Patterns

This type of design patterns provide solution for the better interaction between objects, how to provide lose coupling, and flexibility to extend easily in future.

Pattern Name	Description
Template Method	used to create a template method stub and defer some of the steps of implementation to the subclasses.
Mediator	used to provide a centralized communication medium between different objects in a system.
Chain of Responsibility	used to achieve loose coupling in software design where a request from the client is passed to a chain of objects to process them.

Behavioural Design Patterns

Pattern Name	Description
Observer	useful when you are interested in the state of an object and want to get notified whenever there is any change.
Strategy	Strategy pattern is used when we have multiple algorithm for a specific task and client decides the actual implementation to be used at runtime.
Command	Command Pattern is used to implement lose coupling in a request-response model.
State	State design pattern is used when an Object change it’s behavior based on it’s internal state.

Behavioural Design Patterns

Pattern Name	Description
Visitor	Visitor pattern is used when we have to perform an operation on a group of similar kind of Objects.
Interpreter	defines a grammatical representation for a language and provides an interpreter to deal with this grammar.
Iterator	used to provide a standard way to traverse through a group of Objects.

Example - Template Pattern

- Used when the behaviour of an algorithm can vary and this is handled via subclasses and method overriding
- Used to avoid code duplication
- In effect, an abstract superclass defines a template of methods for its subclasses
- Subclasses can vary the implementation of the superclass using method overriding
- Example - Sorting


Questions?