Static & Factories, Code Reuse

Module: OOP Advanced

Class: Static, Factories & Code Reuse

Topics: Static Members | Factory Constructors | Singletons | Mixins | Extensions

1. Static Variables & Methods

The "Blueprint" Concept

Static members belong to the class itself rather than to any specific object instance.

Core Concepts

- **Class-Level vs. Instance-Level:** You do not need to create an object (using `new`) to access static members.
- **Memory Efficiency:** There is only *one* copy of a static variable in memory, shared by all instances of that class.
- **Utility Methods:** Perfect for helper functions that don't need to access the state of a specific object (e.g., `Math.sqrt()`).
- **Access:** Accessed using the class name directly: `ClassName.variable`.

Code Example

```
class Constants {
  // Shared by all instances
  static double pi = 3.14159;

  static double calculateArea(double r) {
```

```
    return pi * r * r;
  }
}

void main() {
  // No instance needed to access 'pi'
  print(Constants.pi);

  // Calling a static method
  print(Constants.calculateArea(10));
}
```

2. Factory Constructors & Singletons

Controlling Instance Creation

Unlike a normal constructor, a `factory` constructor in Dart does not necessarily create a new instance of the class.

The `factory` Keyword

- **Return Existing:** Can return an instance from a cache.
- **Return Subclass:** Can return an instance of a derived class (polymorphism).
- **Singleton:** Essential for implementing the Singleton pattern.

The Singleton Pattern

This pattern ensures a class has only **one instance** and provides a global point of access to it.

Common Use Cases:

- Database Connections
- Configuration Managers
- Logging Services
- File Systems

Code Example: Singleton

```
class Database {
  // 1. Private static instance variable
  static final Database _instance = Database._internal();

  // 2. Private named constructor
  Database._internal();

  // 3. Factory constructor returns the same static instance
  factory Database() {
    return _instance;
  }

  void connect() {
    print("Connected to DB");
  }
}

void main() {
  var db1 = Database();
  var db2 = Database();

  // Both variables point to the exact same object in memory
  print(db1 == db2); // true
}
```

3. Code Reuse: Mixins

Composition over Inheritance

Mixins allow you to reuse a class's code in multiple class hierarchies without requiring a parent-child relationship.

Understanding Mixins

1. **No Inheritance:** Mixins solve the limitations of single inheritance.
2. **"Has-A" Ability:** Think of Mixins as abilities (e.g., `Flyable`, `Swimmable`) that you can layer onto any class.
3. **The `with` Keyword:** Used to apply a mixin to a class.

Code Example

```
mixin Flyable {
  void fly() => print("I'm flying!");
}

class Animal {}

// Composing behavior: Duck IS-A Animal, HAS-A Flyable ability
class Duck extends Animal with Flyable {
}

void main() {
  var donald = Duck();
  donald.fly(); // Output: I'm flying!
}
```

4. Dart Extensions

Enhancing Existing Libraries

Extensions allow you to add functionality to existing libraries that you don't own (like the `String`, `int`, or `List` classes) without creating a subclass.

Code Example

```
extension NumberParsing on String {
  int parseInt() {
    return int.parse(this);
  }
```

```
}

void main() {
  // Usage: The method appears as if it belongs to the String class
  print("42".parseInt());
}
```

5. Key Takeaways

- **Static:** Use for global constants and utility methods that don't require instance state.
- **Factory:** Use to control the instantiation process (caching, polymorphism).
- **Singleton:** A pattern to restrict a class to exactly one instance.
- **Mixins:** A way to reuse code across multiple class hierarchies (Capabilities).
- **Extensions:** Add new methods to existing types (even built-in ones) keeping code clean.
- **Design:** Prefer composition (Mixins) over deep inheritance trees.