Objectives

- Object-Oriented Programming (OOP)
- Error Handling

Object Oriented Programming

Introduction

Object-Oriented Programming (OOP) is a programming paradigm that organizes software around "objects" that contain both data (attributes) and code (methods) that operate on that data.

- Objects: Represent real-world entities like a car, a person, or a bank account. Each object has unique characteristics (attributes) and behaviors (methods).
- Classes: Blueprints or templates for creating objects. A class defines the attributes and methods that objects of that class will have.

Core Principles of OOP:

- **Encapsulation**: Combines data and the functions that operate on that data within a single unit or class. This keeps data safe from outside interference and misuse.
- **Abstraction**: Allows programmers to hide all but the relevant data about an object to reduce complexity and increase efficiency.
- **Inheritance**: Facilitates the creation of new classes based on existing ones. It promotes code reusability by allowing shared behaviors to be defined once and inherited by child classes.
- **Polymorphism**: Enables different classes to be treated through the same interface, often by overriding methods or implementing the same methods in different ways.

Creating Classes

In Dart, we create a class using the `class` keyword followed by the class name.
By convention, we use **PascalCase** (each word capitalized) for class names (e.g., `BankAccount`, `UserProfile`).
Inside the class, we define methods and variables (fields) that determine the object's behavior and state.

```
class Person {
  String name = "Ali";
  int age = 20;

  void greet() {
    print("Hello, my name is $name!");
  }
}
```

Creating an Object

To create an object (instance) from a class in Dart, we use the class name followed by parentheses.

```
class Person {
  String name = "Ali";
  int age = 20;

  void greet() {
    print("Hello, my name is $name!");
  }
}

void main() {
  Person person = Person();
  person.greet();
}
```

Constructor

A **constructor** is a special method that runs automatically when we create a new instance.
We mainly use constructors to initialize object attributes with starting values.
In Dart, we define a constructor inside the class by using the class name as a method.

```
class Person {
  String name;
  int age;
```

```
  // Constructor
  Person(this.name, this.age);

  void greet() {
    print("Hello, my name is $name!");
  }
}

void main() {
  Person person = Person("Alice", 30);
  person.greet();
}
```

Here, `this.name` and `this.age` automatically assign the provided values to the class fields.
We can add block of code that run when the contructor is called

```
class Person {
  String name;
  int age;

  // Constructor
  Person(this.name, this.age) {
    print('A new person named $name aged $age was created!');
  }
}

void main() {
  var p1 = Person('Alice', 30);
  var p2 = Person('Bob', 25);
}
```

Getter and Setter

In Dart, fields are **public by default**, but we can make them private by starting the name with an underscore `_`, and then control access using getters and setters.
If we try to access a private field directly from outside, it will result in an error.

```
class Person {
  String _name;
  int _age;
  Person(this._name, this._age);

  // Getter (read access)
  String get name => _name;

  // Setter (write access)
  set name(String newName) {
    _name = newName;
  }

  // Getter for age
  int get age => _age;
}

void main() {
  Person person = Person("Alice", 30);
  print(person.name);     // Alice
  person.name = "Bob";     // Update name
  print(person.name);      // Bob
}
```

- `get` keyword defines a getter.
- `set` keyword defines a setter.

With setters and getters, we can add more control when retrieving information from properties or when setting and updating their values.

Instance Methods and Attributes

Instance methods and variables belong to a specific object. However, sometimes we need functionality that does **not depend on a specific object**, like common constants or helper methods.
In Dart:

- **Static variables** are shared by all instances.
- **Static methods** can be called directly using the class name without creating an object.

```
class MathUtils {
  static const double pi = 3.14159;

  static double square(double number) {
    return number * number;
  }
}

void main() {
  print(MathUtils.pi);            // 3.14159
  print(MathUtils.square(5));     // 25.0
}
```

- `static const` creates a static constant.
- `static` before a method makes it callable directly by class name.

Inheritance

In Dart, we use inheritance to avoid repeating code when different classes share common attributes and behaviors.
Let's imagine a simple pet store program where we have cats and dogs. Instead of duplicating code, we can create a parent class `Pet` and have `Dog` and `Cat` inherit from it.
For constructors, we can use two approaches to make the child class initialize the parent class:

- We can use `: super()` in the child class constructor to call the parent class constructor.
- We can also access parent class attributes directly inside the child constructor using `super.attribute`.

The `:` symbol in Dart is used to call the superclass constructor or initialize values before the constructor body runs.

```
class Pet {
  String name;
  int age;

  Pet(this.name, this.age);

  void eat() {
    print('$name is eating.');
  }
```

```
  void sleep() {
    print('$name is sleeping.');
  }
}

class Dog extends Pet {
  Dog(String name, int age, super.name, super.age) : super(name, age);

  void speak() {
    print('$name says Woof!');
  }
}

class Cat extends Pet {
  Cat(String name, int age) : super(name, age);

  void speak() {
    print('$name says Meow!');
  }
}

void main() {
  Dog dog = Dog('Buddy', 3);
  Cat cat = Cat('Whiskers', 2);

  dog.eat();
  dog.speak();

  cat.sleep();
  cat.speak();
}
```

Sometimes, we want the child class to override a method from the parent class, but still use part of the parent's original behavior.
In Dart, we can use the `super` keyword to call the parent class's version of a method.
Before overriding a method in Dart, we use the `@override` annotation followed by the method we want to override.

```
class Pet {
  String name;
  int age;

  Pet(this.name, this.age);

  void speak() {
    print('$name makes a sound.');
  }

  void info() {
    print('$name is $age years old.');
  }
}

class Dog extends Pet {
  Dog(String name, int age) : super(name, age);

  @override
  void speak() {
    super.speak(); // Call parent method
    print('$name says Woof!');
  }
}

class Cat extends Pet {
  Cat(String name, int age) : super(name, age);

  @override
  void speak() {
    super.speak(); // Call parent method
    print('$name says Meow!');
  }
}

void main() {
  Dog dog = Dog('Rex', 4);
  Cat cat = Cat('Luna', 2);

  dog.speak();
```

```
  cat.speak();
}
```

Encapsulation

To understand encapsulation, let's continue working with our pet store example. When building a class like `Pet`, we want to ensure that internal details such as how a pet's data is stored or updated are protected from outside access.
Encapsulation is all about controlling access to the data and behavior inside a class.
In Dart, we achieve encapsulation by using private variables and methods, which are created by prefixing their names with an underscore `_`.
By default, everything in Dart is public unless we explicitly make it private with `_`.

```
class Pet {
  String name;
  int age;

  Pet(this.name, this.age);
  void info() {
    print('$name is $age years old.');
    _checkHealth();
  }

  // Private method
  void _checkHealth() {
    print('$name\'s health is good!');
  }
}

void main() {
  Pet pet = Pet('Coco', 4);
  pet.info();
  // pet._checkHealth(); //Error: _checkHealth is private and should not be
accessed directly
}
```

- **Public** members (default): Accessible from anywhere.
- **Private** members (prefix with `_`): Accessible only within the same library (the same Dart file or package scope).

Polymorphism

Polymorphism means **"many forms."** , In programming, it allows us to use the same method name to perform different behaviors depending on the object that’s calling it.
This makes our code more flexible and easier to extend.
Let’s go back to our pet store example. Suppose we want each pet to "speak." ,we can define a `speak` method in both the `Dog` and `Cat` classes. Even though the method name is the same, each class can implement it differently, so when we call `speak` on a dog, it might bark, and when we call it on a cat, it might meow.
Thanks to polymorphism, we can treat different objects in a uniform way, we can loop through a list of pets and call `speak` on each one, without worrying about whether it’s a dog or a cat.
Dart will automatically run the correct version of the method based on the object's class.

```
class Pet {
  String name;

  Pet(this.name);
}

class Dog extends Pet {
  Dog(String name) : super(name);

  void speak() {
    print('$name says Woof!');
  }
}

class Cat extends Pet {
  Cat(String name) : super(name);

  void speak() {
    print('$name says Meow!');
  }
}

void main() {
  List<Pet> pets = [
    Dog('Rex'),
    Cat('Luna'),
  ];
```

```
  for (var pet in pets) {
    if (pet is Dog) {
      pet.speak();
    } else if (pet is Cat) {
      pet.speak();
    }
  }
}
```

Mixins and Interfaces

In Dart, mixins and interfaces allow us to share behavior across multiple classes and enforce structure without traditional inheritance.

Mixins

A mixin is a class that provides functionality that can be reused in multiple classes without being their parent class.
Instead of extending a class, we use the `with` keyword to apply a mixin.

- We create a mixin using the `mixin` keyword.
- A mixin cannot be instantiated by itself (you can't create an object from a mixin).
- A class can use multiple mixins at once, separated by commas.

```
mixin Logger {
  void log(String message) {
    print('Log: $message');
  }
}

class Car with Logger {
  String model;

  Car(this.model);

  void drive() {
    log('Driving $model');
  }
}
```

```
void main() {
  Car car = Car('Tesla Model 3');
  car.drive();
}
```

Interfaces

In Dart, every class is an interface by default, we can create an interface by defining a normal class, and then implement it using the implements keyword.

- When a class **implements** another class, it must **override** all of its methods and properties.
- Interfaces allow us to **force** certain structure or behavior across unrelated classes.

```
class Animal {
  void move() {}
}

class Bird implements Animal {
  @override
  void move() {
    print('Flying in the sky.');
  }
}

class Fish implements Animal {
  @override
  void move() {
    print('Swimming in the water.');
  }
}

void main() {
  Bird bird = Bird();
  Fish fish = Fish();

  bird.move();
  fish.move();
}
```

Abstract Classes

An abstract class is a class that cannot be instantiated directly, It is used to define a common structure for other classes to follow.

- We define an abstract class with the `abstract` keyword.
- Abstract classes can have both abstract methods (methods without a body) and normal methods.

```
abstract class Shape {
  void draw();
}

class Circle extends Shape {
  @override
  void draw() {
    print('Drawing a circle.');
  }
}

class Square extends Shape {
  @override
  void draw() {
    print('Drawing a square.');
  }
}

void main() {
  Circle circle = Circle();
  Square square = Square();

  circle.draw();
  square.draw();
}
```

- Abstract classes let us enforce a structure (like an interface), but also allow us to include common code if needed.
- Use `extends` to inherit from an abstract class.

Operator Overloading

Operator overloading is the act of overriding basic operators like `+`, `-`, `*`, `==`, and others to work with our own classes.
By default, Dart doesn't know how to use these operators with custom objects, but we can definine special operator methods inside our class, that tell Dart how the operators should behave.
Let’s say we’re working with a `FoodPortion` class in our pet store, we want to be able to add two portions together or compare if they are equal.
Instead of creating separate methods like `add` or `equals`, we can overload the `+` and `==` operators to keep our code clean and natural.

```
class FoodPortion {
  final int grams;

  FoodPortion(this.grams);

  FoodPortion operator +(FoodPortion other) {
    return FoodPortion(this.grams + other.grams);
  }

  @override
  bool operator ==(Object other) {
    if (other is FoodPortion) {
      return grams == other.grams;
    }
    return false;
  }
  @override
  int get hashCode => grams.hashCode;

  @override
  String toString() {
    return '${grams}g';
  }
}

void main() {
  var portion1 = FoodPortion(50);
  var portion2 = FoodPortion(30);
```

```
  var total = portion1 + portion2;

  print('Total: $total');
  print(portion1 == portion2);
}
```

In Dart, the `operator` keyword is used to overload operators. This allows you to define how custom objects should behave when used with operators like `+`, `-`, `*`, and `==`.
When overriding the `==` operator, it is important to also override the `hashCode` property. This ensures that your objects behave correctly when used in collections such as `Set` and `Map`, where proper hashing and equality checking are required.
In operator overloading, the `other` variable represents the second operand that the operation is being applied to. It is passed as a parameter to the operator method and allows you to define how two objects should interact.

Dart Operator Overloading Table:

Operator	Method to define	Description
`+`	`operator +(other)`	Addition
`-`	`operator -(other)`	Subtraction
`*`	`operator *(other)`	Multiplication
`/`	`operator /(other)`	Division
`%`	`operator %(other)`	Modulo (remainder)
`==`	`operator ==(other)`	Equality comparison
`<`	`operator <(other)`	Less than
`<=`	`operator <=(other)`	Less than or equal to
`>`	`operator >(other)`	Greater than
`>=`	`operator >=(other)`	Greater than or equal to

Error Handling

Introduction

When writing scripts and code, errors are inevitable. These errors can be syntax errors, such as misspelling a function or variable name, or **logical errors**, like mistakes in `if` conditions or loops that lead to unexpected behavior. Such errors can cause our program to fail, requiring

fixes before the script can run properly.
Another category is **runtime errors**, which occur during script execution. These can be caused by invalid user input or unpredictable conditions. While we can't always prevent runtime errors, JavaScript provides tools to handle them gracefully.

Types of Errors

- **Syntax Errors:** These occur when we make mistakes in writing our script, such as misspelling a function or variable name, or forgetting parentheses or curly braces.
- **Runtime Errors:** These errors occur while our program is running and cause it to stop. They are often caused by user input, such as entering invalid values.
- **Logic Errors:** These errors happen when the code does not work as intended due to incorrect formatting or logic in conditions. They don't crash the program but cause it to behave in unexpected ways.
- **Arithmetic Errors:** These are similar to logic errors but specifically involve mathematical operations, such as dividing by zero, or improper use of operator precedence (e.g., `+`, `-`, `/`, `*`).
- **Resource Errors:** These errors occur when we exceed the resources available on our machine, such as running out of memory or creating an infinite loop.

Dart Errors

Error Type	Description	Example
Compile-time Error	Invalid code structure	`int x = ;` (missing value)
NoSuchMethodError	Accessing undefined methods or properties	`nonexistentMethod();`
TypeError	Wrong data type usage	`int x = "hello";` (assigning String to int)
RangeError	Index or value out of valid range	`var list = [1,2]; print(list[5]);`
Custom Exceptions	Developer-defined exceptions	`throw Exception('Invalid input');`

Handling Runtime Errors

Some errors happen because of unexpected user input or external factors. Even if the code is correct, these runtime errors can cause the program to crash. Dart allows us to catch and handle such errors using `try`, `catch`, `on`, and `finally` blocks.

- **try Block**: Code that might throw an error is placed inside a `try` block.
- **catch Block**: If an error occurs, the `catch` block handles it. We can also access the error and its stack trace.
- **finally Block (Optional)**: The `finally` block always runs whether an error occurred or not.

```
void main() {
  try {
    print('Enter a number:');
    String? input = 'abc'; // Simulating bad input
    int num = int.parse(input);
    print('You entered: $num');
  } on FormatException catch (e) {
    print('Invalid input! Please enter a valid number.');
    print('Error details: ${e.message}');
  } catch (e) {
    print('An unexpected error occurred: $e');
  } finally {
    print('Execution complete.');
  }
}
```

We can specify the type of errors to handle before the `catch` block by using the `on` keyword. This allows us to write different code for each specific error type.
If we don't specify an error type, we can use a general `catch` block to handle any kind of error.

Raising Our Own Errors

We can throw our own exceptions in Dart using the `throw` keyword. This is useful to enforce certain rules in the program.

```
void withdraw(double balance, double amount) {
  if (amount > balance) {
    throw Exception('Insufficient funds');
  } else {
    print('Withdrawing \$${amount}');
  }
}

void main() {
  try {
```

```
    withdraw(100, 50);
    withdraw(100, 150);
  } catch (e) {
    print('Error: $e');
  }
}
```

Custom Exception Classes

In Dart, we can define custom exceptions by creating a class that implements `Exception`.

```
class OverdrawError implements Exception {
  final String message;
  OverdrawError(this.message);

  @override
  String toString() => 'OverdrawError: $message';
}

void withdraw(double balance, double amount) {
  if (amount > balance) {
    throw OverdrawError('Not enough money!');
  }
  print('Withdrawal successful');
}

void main() {
  try {
    withdraw(100, 200);
  } on OverdrawError catch (e) {
    print('Custom error: $e');
  }
}
```

Tasks

Task 1:

Define a `BankAccount` class with:

Attributes:

- `accountNumber` (String): A unique account identifier.
- `ownerName` (String): The account holder's name.
- `balance` (double): The current balance (default: 0).
- `transactions` (List of Strings): Stores transaction history.

Methods:

- **Constructor** `BankAccount(accountNumber, ownerName, [initialBalance = 0])`:
  - Initializes the account with given details.
  - Throws an exception if the initial balance is negative.
- **deposit(amount)**:
  - Adds `amount` to the balance.
  - Throws an exception if `amount` is negative.
  - Logs the transaction in `transactions`.
- **withdraw(amount)**:
  - Deducts `amount` from the balance.
  - Throws an exception if:
    - `amount` is negative.
    - Insufficient funds (`balance < amount`).
  - Logs the transaction in `transactions`.
- **getBalance()**:
  - Returns the current balance.
- **getStatement()**:
  - Prints all transactions in a formatted way.
- **static transfer(sourceAccount, targetAccount, amount)**:
  - Transfers `amount` from `sourceAccount` to `targetAccount`.
  - Throws an exception if the transfer fails.
  - Use `try-catch` blocks to handle:
    - Negative deposits/withdrawals.
    - Insufficient funds.
    - Invalid transfers.

Task 2:

Create a Dart program to represent the functionality of a robot.
The program should have a parent class that represents a basic robot with the following:

Attributes:

- `id` (String): The ID of the robot.
- `x` (int): The robot's position along the x-axis.
- `y` (int): The robot's position along the y-axis.
- `orientation` (String): The current orientation (e.g., "North", "East", etc.).
- `step` (int): How many steps the robot moves when walking (default: 1).

Methods:

- **Constructor**:
  - Initializes the robot's `id`, `x`, `y`, and `orientation`.
  - `step` should default to 1 and does not need to be passed as a parameter.
- **turnClockWise()**:
  - Turns the robot 90 degrees clockwise.
- **turnAntiClockWise()**:
  - Turns the robot 90 degrees anti-clockwise.
- **walk()**:
  - Moves the robot forward by `step` units depending on its current orientation.
- **getPosition()**:
  - Prints the current position and orientation of the robot.

Then, create a child class that represents a new generation of robot which inherits from the basic robot and adds these additional features:

Attributes:

- `charge` (int): The amount of energy the robot has.
- `turboState` (bool): Whether the robot is in turbo mode or not.

Methods:

- **Constructor**:
  - Calls the parent class constructor.
  - Initializes the `charge` attribute.
  - Sets `turboState` to `false` by default.
- **turbo()**:
  - Activates turbo mode if `charge > 0`.
- **walk()** (override):
  - If `turboState` is `true`, set `step` to 2 and decrease `charge` by 1.

- If `charge` drops to 0, reset `step` to 1 and turn off `turboState`.
- Then perform the normal walk behavior from the parent class.