IEEE FCI - Helwan Student Branch

S25' Training Sessions

mobile app

SESSION 4

#create_share_innovate

Table of contenet

IEEE | FCI - Helwan Student Branch

quiz

Which collection type in Dart maintains the insertion order of elements?

A) Set

B) Map

C) List

S25' Training Sessions

IEEE | FCI - Helwan Student Branch

quiz

Answer: C) List

The List maintains the order in which elements are inserted, whereas a Set and Map do not guarantee order.

IEEE | FCI - Helwan Student Branch

quiz

In Dart, how do you create a map with initial key-value pairs?

A) Map<String, int> map = {'A', 1, 'B', 2};

B) Map<String, int> map = {'A': 1, 'B': 2};

C) Map<String, int> map = {'A' -> 1, 'B' -> 2};

D) Map<String, int> map = ['A' => 1, 'B' => 2];

IEEE | FCI - Helwan Student Branch

quiz

Answer:

B) Map<String, int> map = {'A': 1, 'B': 2};

FCI - Helwan
Student Branch

quiz

What is the best way to define a constructor in Dart?

A) Car(String make, String model) { this.make = make; this.model = model; }

B) Car(this.make, this.model);

C) constructor Car(String make, String model) { this.make = make; this.model = model; }

D) Car({String make, String model}) { this.make = make; this.model = model; }

IEEE FCI - Helwan Student Branch

quiz

Answer:

B) Car(this.make, this.model);

A is True , but B is the best practice

S25' Training Sessions

quiz

What will happen if you try to add a duplicate value to a Set in Dart?

A) It will allow the duplicate value.

B) It will throw a runtime error.

C) It will ignore the duplicate value.

D) It will replace the existing value with the new one.

quiz

Answer:

C) It will ignore the duplicate value.

quiz

Which of the following statements about constructors in Dart is true?

A) A class can have only one constructor.

B) A constructor cannot have parameters.

C) A class can have multiple named constructors.

D) Constructors cannot be inherited.

IEEE | FCI - Helwan Student Branch

quiz

Answer:

C) A class can have multiple named constructors.

S25' Training Sessions

Encapsulation

Encapsulation is the process of wrapping data (variables) and methods into a single unit (class).
It restricts direct access to data and allows controlled access through methods.

Advantages:

- Protects object integrity.
- Reduces complexity.
- Facilitates code maintenance.

Encapsulation

Encapsulation is achieved using:

Private fields: Data members are marked as **private** using **_**.

Getters and Setters: Methods to access and modify private data.

Encapsulation

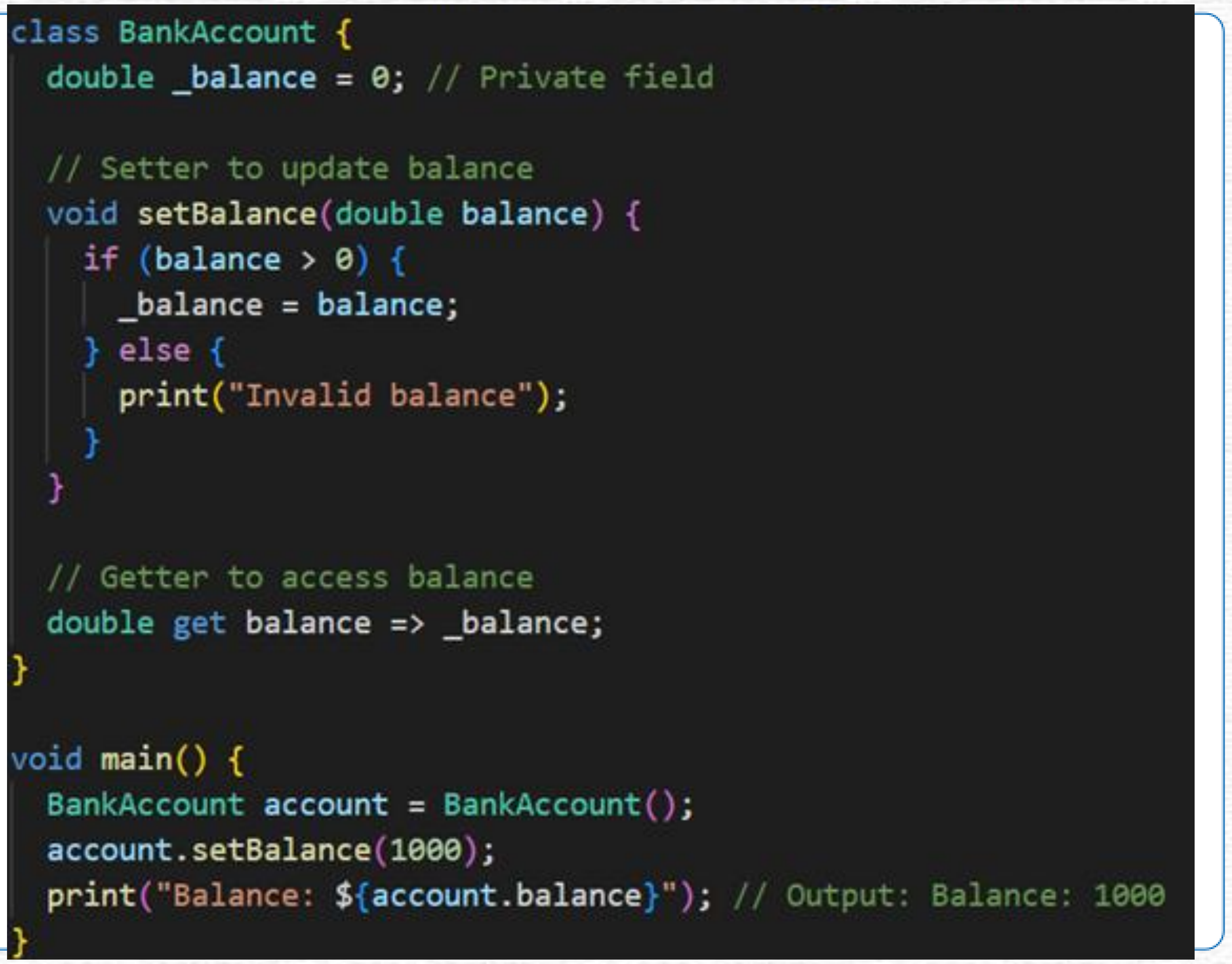
Exampl

e:

Explanation:

_balance is private → Direct access is restricted.

setBalance() and get balance control how _balance is modified and accessed.

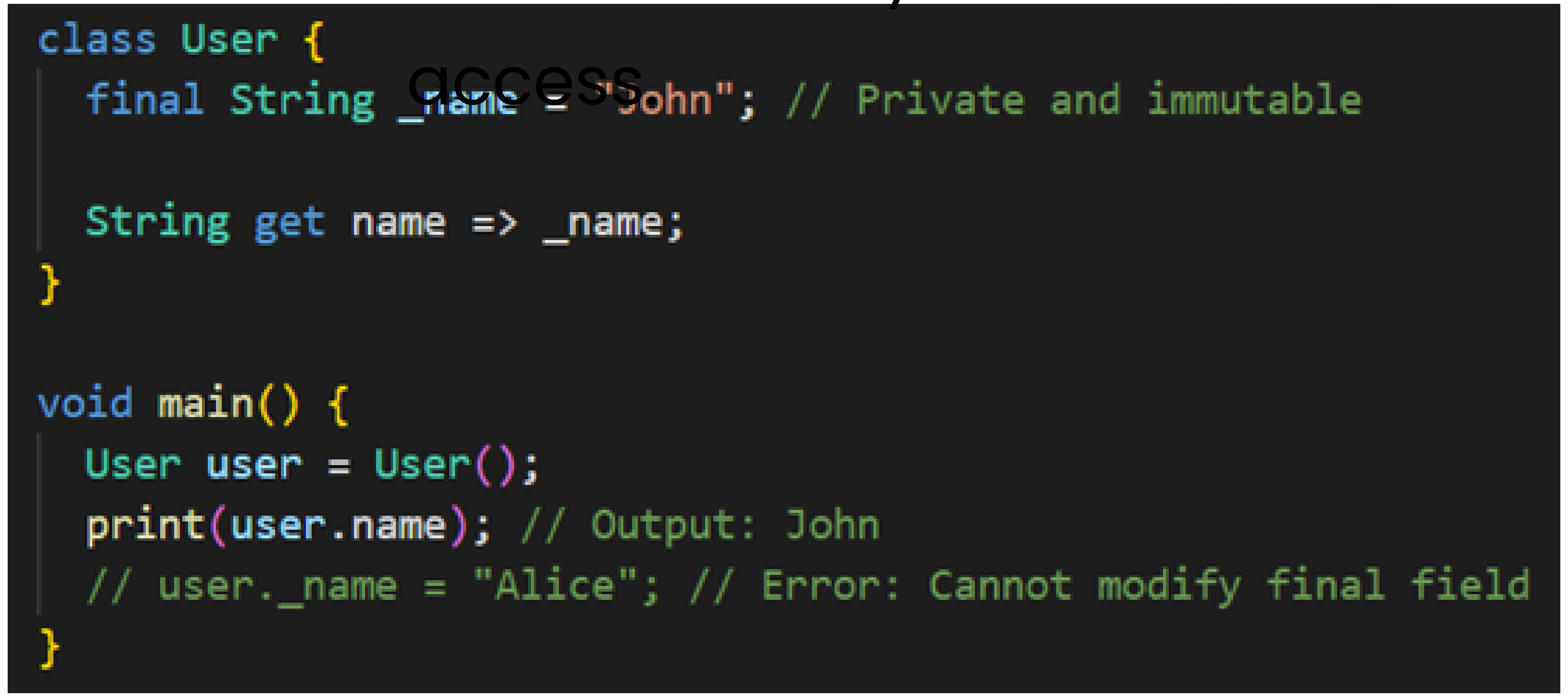


```
class BankAccount {
  double _balance = 0; // Private field

  // Setter to update balance
  void setBalance(double balance) {
    if (balance > 0) {
      _balance = balance;
    } else {
      print("Invalid balance");
    }
  }

  // Getter to access balance
  double get balance => _balance;
}

void main() {
  BankAccount account = BankAccount();
  account.setBalance(1000);
  print("Balance: ${account.balance}"); // Output: Balance: 1000
}
```

Encapsulation

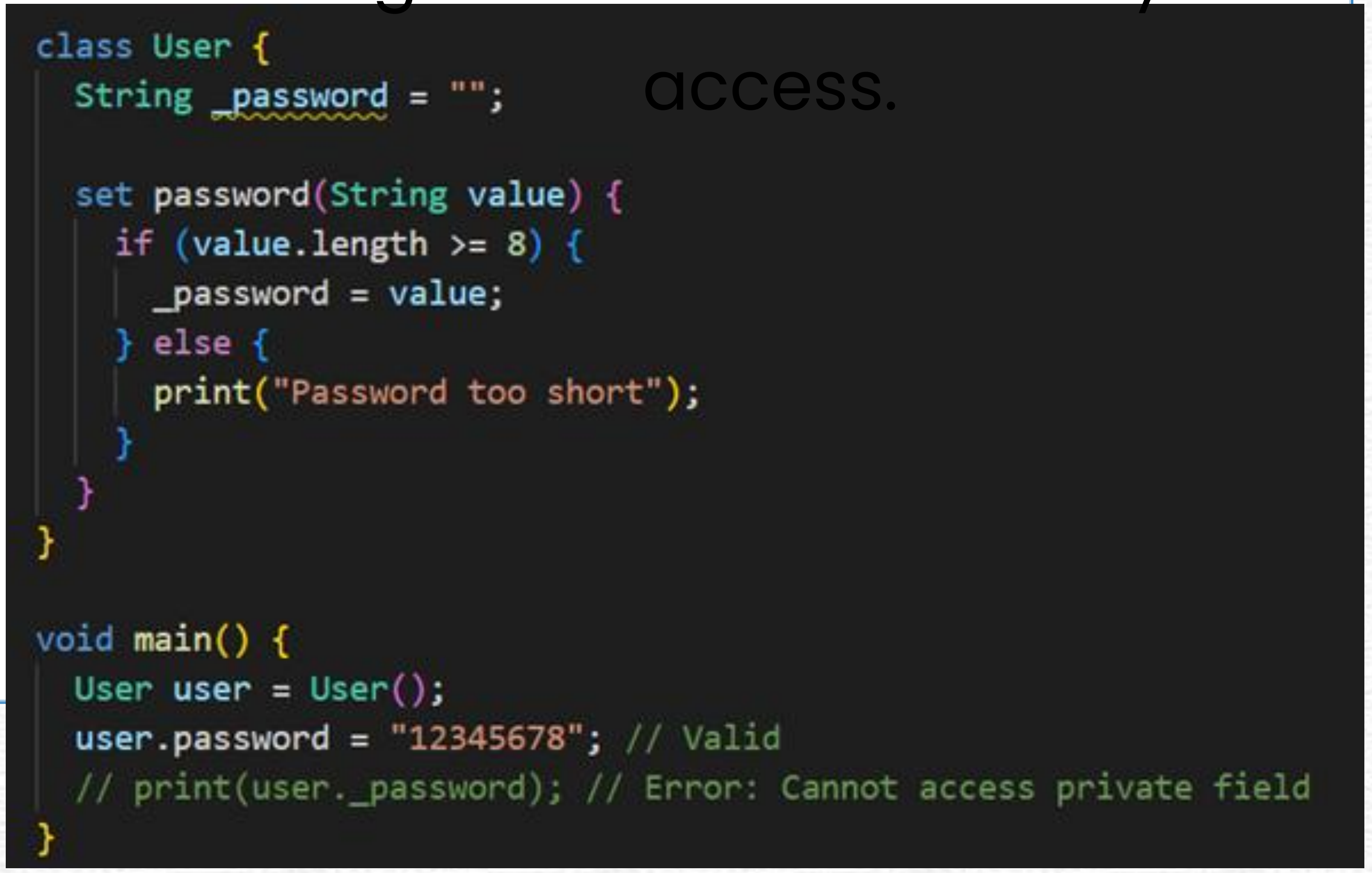
Example 2:
(Read-Only and Write-Only Fields)

Read-Only Field:
Provide only a getter; no setter allows read-only access.



```
class User {
  final String _name = "John"; // Private and immutable

  String get name => _name;
}

void main() {
  User user = User();
  print(user.name); // Output: John
  // user._name = "Alice"; // Error: Cannot modify final field
}
```

Write-Only Field:
Provide only a setter; no getter allows write-only access.



```
class User {
  String _password = "";

  set password(String value) {
    if (value.length >= 8) {
      _password = value;
    } else {
      print("Password too short");
    }
  }
}

void main() {
  User user = User();
  user.password = "12345678"; // Valid
  // print(user._password); // Error: Cannot access private field
}
```

IEEE | FCI - Helwan Student Branch

Encapsulation

Encapsulation (Best Practices):

Use private fields (_) to protect data from direct modification.

Use getters and setters to control data access and validation.

Avoid exposing internal state directly.

Ensure data consistency through validation inside setters.

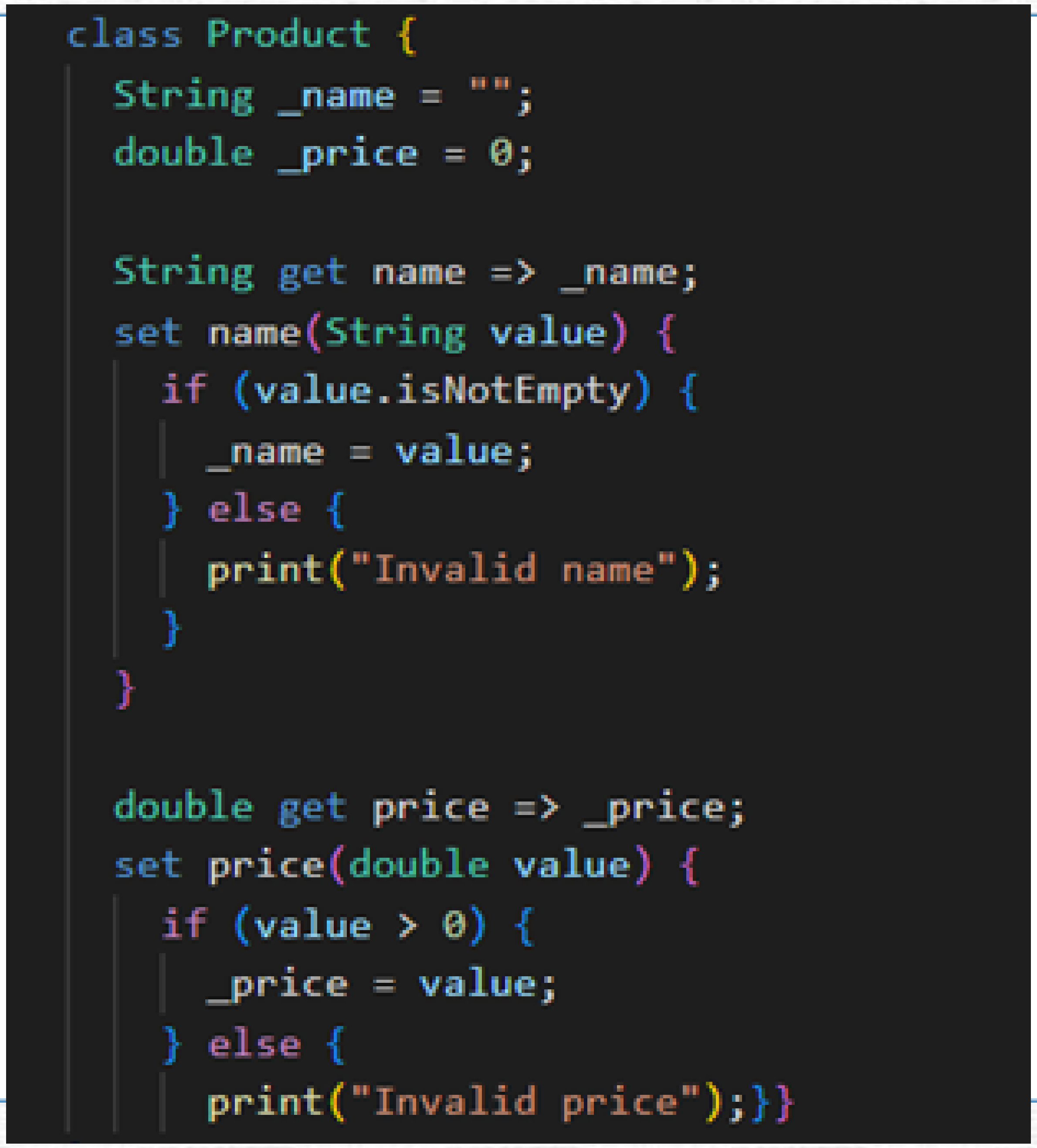
S25' Training Sessions

IEEE | FCI - Helwan Student Branch

Encapsulation

Example:

Encapsulation ensures that the object's state remains valid and consistent.



```
class Product {
  String _name = "";
  double _price = 0;

  String get name => _name;
  set name(String value) {
    if (value.isNotEmpty) {
      _name = value;
    } else {
      print("Invalid name");
    }
  }

  double get price => _price;
  set price(double value) {
    if (value > 0) {
      _price = value;
    } else {
      print("Invalid price");}}
```

Inheritance

Inheritance allows a class to inherit properties and methods from another class.

Establishes a parent-child relationship.

Why Use Inheritance?

Using (extends) Keyword

- **Code Reusability:** Write once, use multiple times.
- **Extensibility:** New functionality can be added to existing classes without modifying them.
- **Polymorphism:** Enables dynamic method binding.

Inheritance

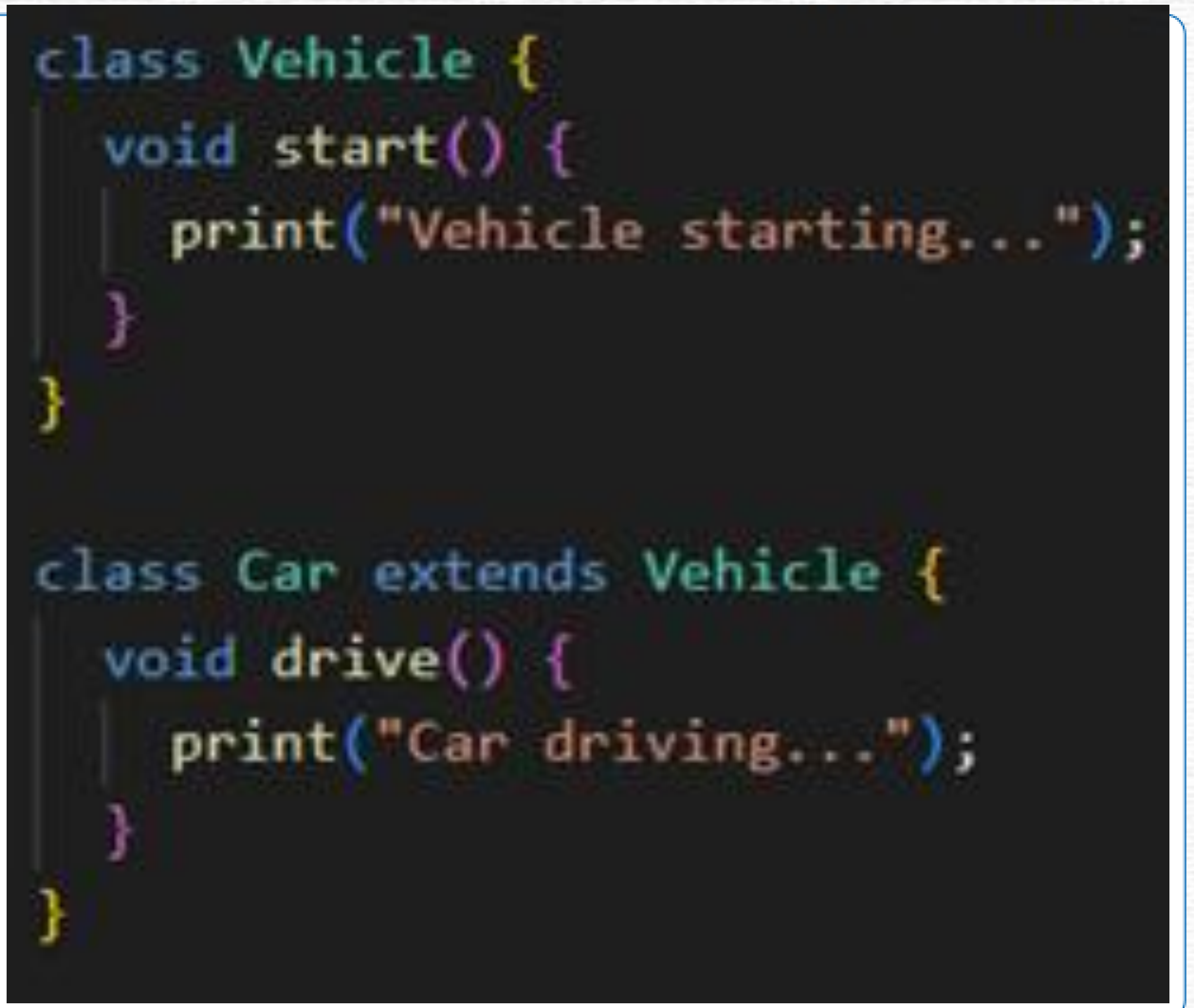
Types of Inheritance:

- Single Inheritance : One parent, one child.
- Multilevel Inheritance : Child class inherits from parent, which also has a parent.
- Hierarchical Inheritance : One parent, multiple child classes.

Inheritance

1. Single Inheritance

- One parent class and one child class.
- Child inherits properties and behaviors of the parent.



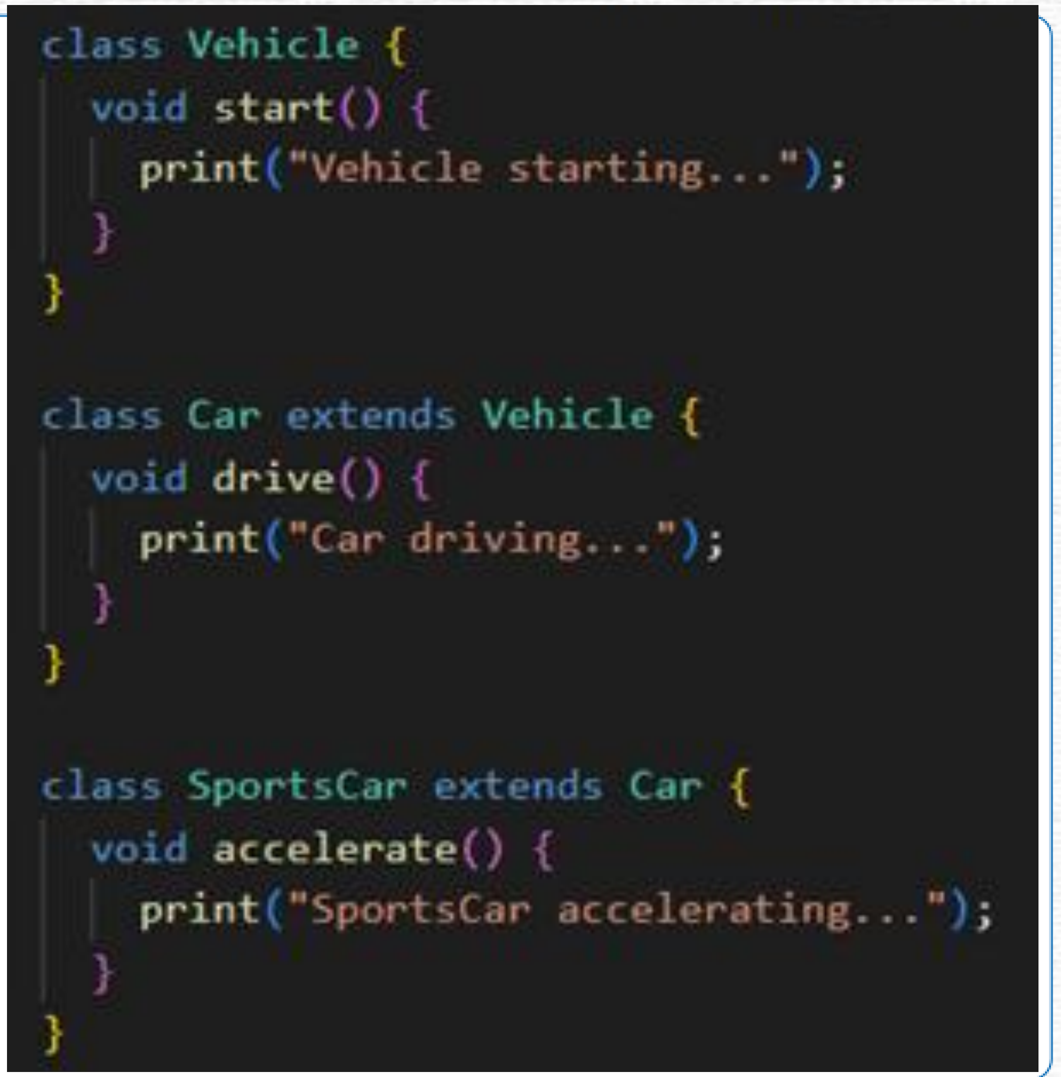
```
class Vehicle {
  void start() {
    print("Vehicle starting...");
  }
}

class Car extends Vehicle {
  void drive() {
    print("Car driving...");
  }
}
```

IEEE | FCI - Helwan Student Branch

Inheritance

2. Multilevel Inheritance

- A child class inherits from a parent class, and that parent class itself is a child of another class.



```
class Vehicle {
  void start() {
    print("Vehicle starting...");
  }
}

class Car extends Vehicle {
  void drive() {
    print("Car driving...");
  }
}

class SportsCar extends Car {
  void accelerate() {
    print("SportsCar accelerating...");
  }
}
```

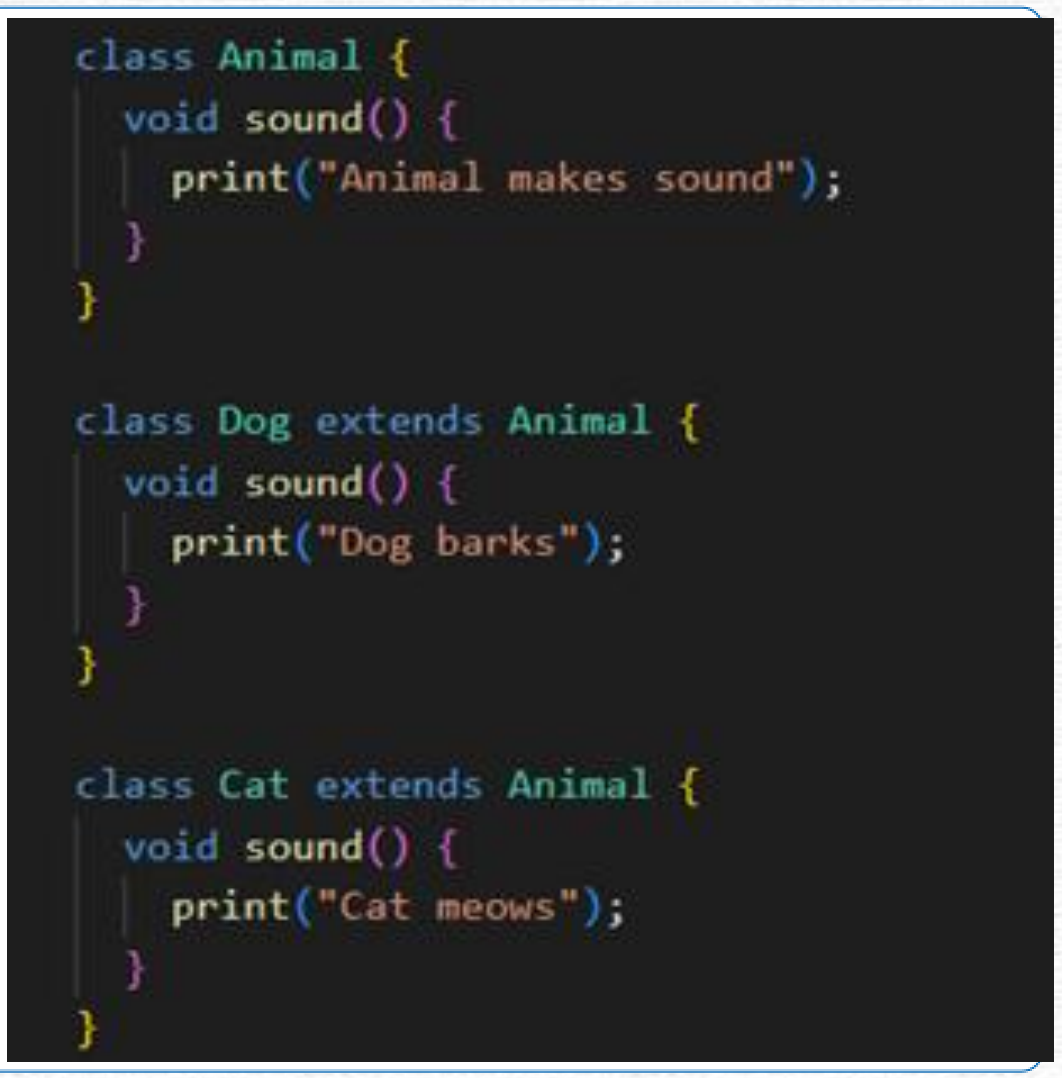
S25' Training Sessions

IEEE FCI - Helwan Student Branch

Inheritance

3. Hierarchical Inheritance

- One parent class and multiple child classes.
- Each child inherits properties and behaviors from the parent.


```
class Animal {
  void sound() {
    print("Animal makes sound");
  }
}

class Dog extends Animal {
  void sound() {
    print("Dog barks");
  }
}

class Cat extends Animal {
  void sound() {
    print("Cat meows");
  }
}
```

S25' Training Sessions

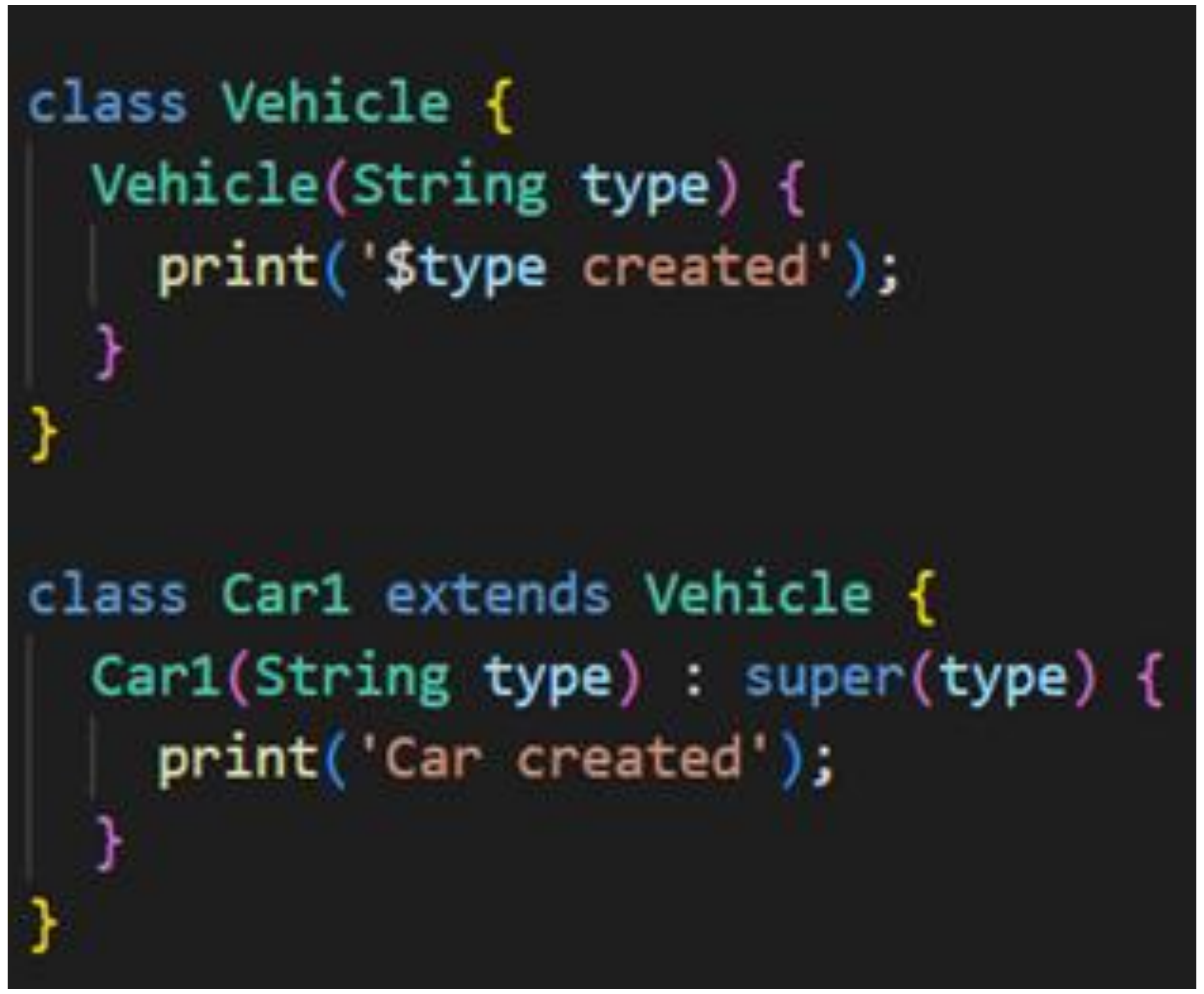
Inheritance

Constructor Chaining and (super) Keyword

- **super()** calls the parent class constructor.
- Used to initialize parent class attributes.

Rules:

- **super()** must be the first statement in the constructor.
- If a parent class has a



```
class Vehicle {
  Vehicle(String type) {
    print('$type created');
  }
}

class Car1 extends Vehicle {
  Car1(String type) : super(type) {
    print('Car created');
  }
}
```

IEEE | FCI - Helwan Student Branch

Polymorphism

Ability to present the same interface for different forms (data types).

Behavior is determined at **runtime** or **compile time**.

Why Use Polymorphism?

- **Code Flexibility:** Handle different objects uniformly.
- **Extensibility:** Add new functionality with minimal code change

Types of Polymorphism:

1. Compile-Time Polymorphism **(Static)**
2. Run-Time Polymorphism **(Dynamic)**

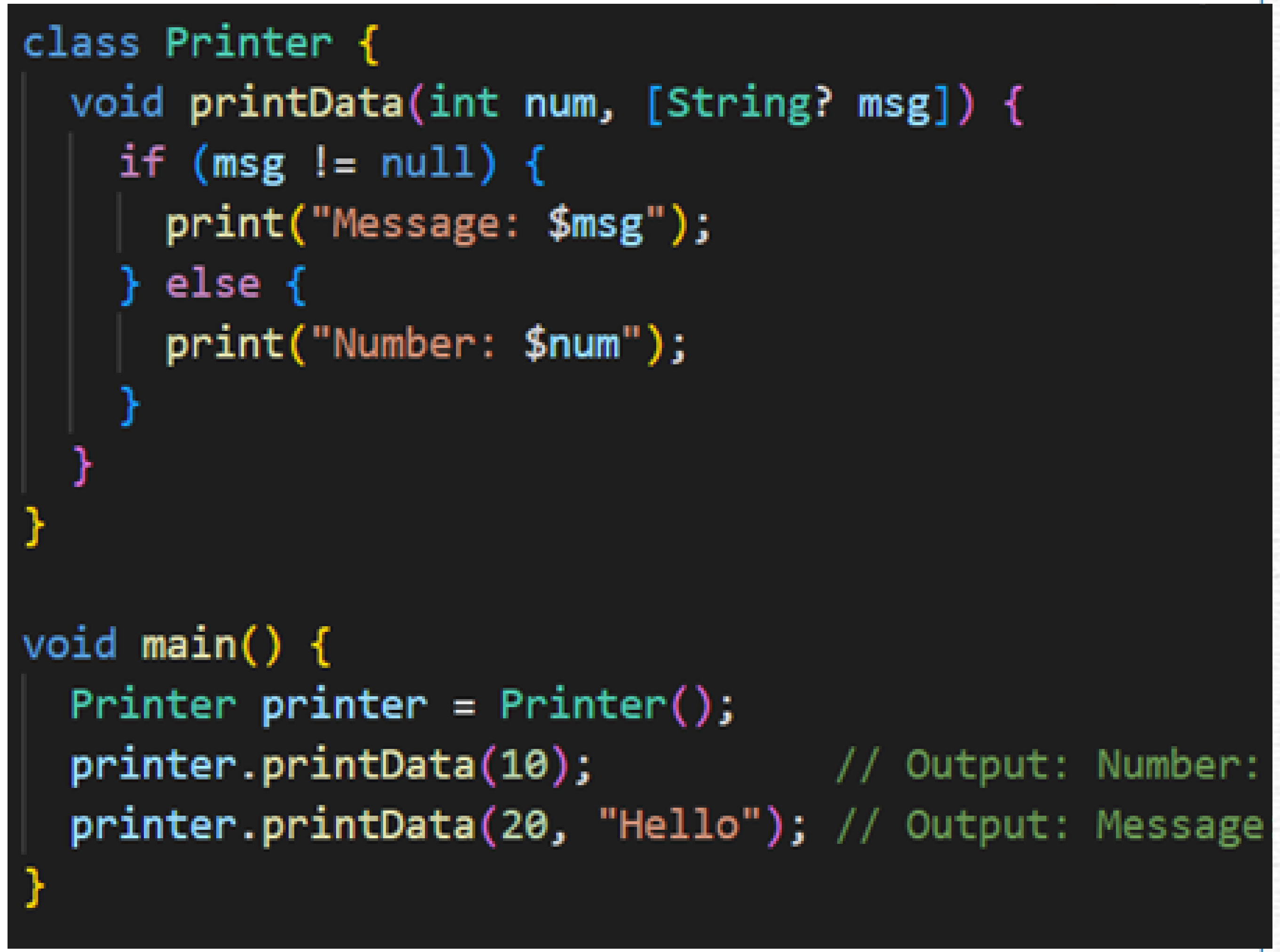
S25' Training Sessions

IEEE FCI - Helwan Student Branch

Polymorphism

Compile-Time Polymorphism (Static Binding)

1. Method binding is determined at compile time.
2. Achieved through method overloading and operator overloading **(not supported in Dart)**.



```
class Printer {
  void printData(int num, [String? msg]) {
    if (msg != null) {
      print("Message: $msg");
    } else {
      print("Number: $num");
    }
  }
}

void main() {
  Printer printer = Printer();
  printer.printData(10);          // Output: Number:
  printer.printData(20, "Hello"); // Output: Message
}
```

S25' Training Sessions

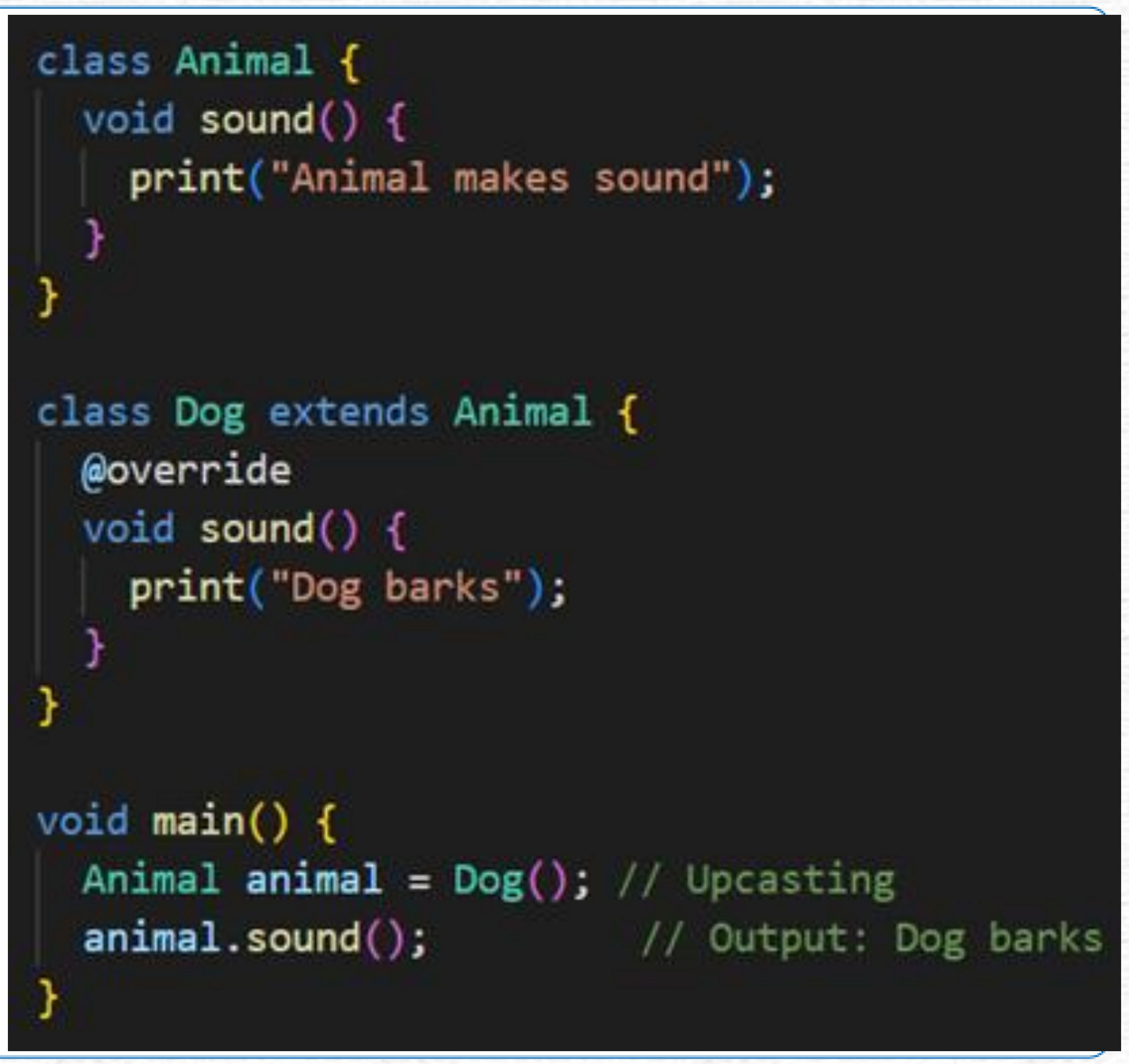
Polymorphism

Run-Time Polymorphism (Dynamic Binding)

- Method binding is determined at runtime.
- Achieved through method overriding.

Why Dynamic?

The method to be executed is determined at runtime based on the actual object type (Dog).



```
class Animal {
  void sound() {
    print("Animal makes sound");
  }
}

class Dog extends Animal {
  @override
  void sound() {
    print("Dog barks");
  }
}

void main() {
  Animal animal = Dog(); // Upcasting
  animal.sound();          // Output: Dog barks
}
```

Exercise

1. Create a Shape class with a color and name property.
2. Make color private and use getters and setters to access it.
3. Create two classes Circle and Rectangle that inherit from Shape.
4. Circle should have a radius and a method to calculate its area.
5. Rectangle should have width and height and a method to calculate its area.
6. Add a displayDetails() method in Shape to print the shape's details.
7. Override this method in Circle and Rectangle to show their

IEEE FCI - Helwan Student Branch

task

Software solutions company system

FCI - Helwan
Student Branch

tHANK YOU

SEE YOU NEXT TIME

S25' Training Sessions