Inheritance

Class	Inheritance
Topcis	- Inheritance with `extends` - `super` keyword for parent class access - Method overriding

1. Lesson Objectives

By the end of this class, you will be able to:

- **Explain** the concept of inheritance and its "is-a" relationship.
- **Use** the `extends` keyword to create a subclass (child class).
- **Understand** the difference between a superclass (parent) and a subclass (child).
- **Call** the parent class constructor using the `super` keyword.
- **Override** parent class methods using the `@override` annotation.
- **Access** parent class methods from a child class using `super`.

---

2. Core Concepts

The central theme of this lesson is **code reusability**. We want to avoid duplicating code, and inheritance is a primary way to achieve this.

Analogy: Think of a general blueprint for an "Animal." This blueprint defines that all animals have an `age` and can `eat()`. Instead of creating separate, new blueprints for "Dog" and "Cat" from scratch, we can just *extend* the "Animal" blueprint. The "Dog" blueprint *inherits* everything from "Animal" and adds a specific `bark()` method.

A `Dog` **"is-a"** `Animal`. This "is-a" relationship is the test for when to use inheritance.

Part 1: The `extends` Keyword (Creating the "is-a" link)

- **Inheritance** allows one class (the **subclass** or **child class**) to inherit the properties and methods of another class (the **superclass** or **parent class**).
- We use the `extends` keyword in Dart.
- **Example (Parent Class):**

```
// The Superclass (Parent)
class Animal {
  String name;

  Animal(this.name); // Constructor

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

  void makeSound() {
    print('Animal makes a sound.');
  }
}
```

- **Example (Child Class):**

```
// The Subclass (Child)
class Dog extends Animal {
  // 'Dog' now has 'name', 'eat()', and 'makeSound()' automatically!

  // We'll fix this constructor error in the next step.
}

// In our main function:
var myDog = Dog('Buddy'); // Error: 'Animal' has a constructor, so 'Dog' m
ust call it.
myDog.eat(); // This *would* work if we could create the object.
```

Part 2: The `super` Keyword (Calling the Parent Constructor)

- **The Problem:** The `Animal` class needs a `name` to be created. When we create a `Dog`, we are *also* creating an `Animal` inside it. We must tell Dart how to initialize that parent `Animal` part.
- **The Solution:** We use `super()` in the child's constructor initializer list to pass the required data up to the parent's constructor.
- **Example (Fixed):**

```
class Dog extends Animal {

  // This constructor takes a 'name'
  // and passes it 'up' to the Animal constructor using 'super(name)'.
  Dog(String name) : super(name);

}

// In our main function:
var myDog = Dog('Buddy');
myDog.eat(); // Output: Buddy is eating.
```

- Now, `myDog` is an object that is both a `Dog` and an `Animal`. It has access to all public methods from `Animal`.

Part 3: Method Overriding (Changing Parent Behavior)

- **The Problem:** `myDog.makeSound()` would print "Animal makes a sound." This isn't specific enough. We want a `Dog` to "Woof."
- **The Solution:** We **override** the `makeSound` method. This means we provide a new implementation for a method that already exists in the parent.
- In Dart, we **must** use the `@override` annotation. This is a safety check—it tells the compiler "I am *intentionally* replacing a parent method." If you misspell the method, the compiler will give you an error.
- **Example (Code):**

```
class Dog extends Animal {

  Dog(String name) : super(name);

  // We are providing a new version of 'makeSound'
  @override
  void makeSound() {
    print('Woof! Woof!');
  }
}

class Cat extends Animal {

  Cat(String name) : super(name);

  @override
  void makeSound() {
    print('Meow.');
  }
}

// In our main function:
var myDog = Dog('Buddy');
var myCat = Cat('Whiskers');

myDog.makeSound(); // Output: Woof! Woof!
myCat.makeSound(); // Output: Meow.
```

Part 4: Accessing the Parent Method (Using `super.` somewhere else)

- **The Problem:** What if we don't want to *replace* the parent's method, but just *add to it*?
- **Example:** Let's say `Animal.eat()` handles the logic for hunger. We want the `Dog` to do that, *and also* wag its tail.

- **The Solution:** We can use `super.methodName()` to call the parent's version of the method from *inside* the child's overridden method.
- **Example (Code):**

```
class Dog extends Animal {

  Dog(String name) : super(name);

  @override
  void makeSound() {
    print('Woof! Woof!');
  }

  // Let's override eat()
  @override
  void eat() {
    super.eat(); // This calls the 'eat()' method from the Animal class
    print('$name wags its tail.');
  }
}

// In our main function:
var myDog = Dog('Buddy');
myDog.eat();
// Output:
// Buddy is eating.
// Buddy wags its tail.
```

---

3. In-Class Exercise (15 minutes)

Goal: Build a simple hierarchy for `Vehicle` and `Car`.

1. **Create the Superclass (Parent):**
   - Create a class named `Vehicle`.
   - Give it two properties: `String make` and `String model`.

- Create a constructor that initializes these properties: `Vehicle(this.make, this.model)`.
- Create a method: `void drive()` that prints "Driving the $make $model."

2. **Create the Subclass (Child):**
   - Create a class named `Car` that `extends Vehicle`.
   - Create a constructor for `Car` that takes `make` and `model`.
   - Use `super()` to pass these values to the `Vehicle` constructor.
3. **Test It:**
   - In `main()`, create a `Car` object: `var myCar = Car('Toyota', 'Corolla');`
   - Call `myCar.drive()`. It should print "Driving the Toyota Corolla."
4. **Override a Method:**
   - In the `Car` class, add a new property: `int numberOfDoors = 4`.
   - `@override` the `drive()` method.
   - Make the new `drive()` method call the *parent's* version first (using `super.drive()`).
   - After that, make it print "This car has $numberOfDoors doors."
5. **Final Test:**
   - Run `myCar.drive()` again. The output should now be:

     ```
     Driving the Toyota Corolla.
     This car has 4 doors.
     ```

---

4. Key Takeaways & Review

- **Inheritance:** Creates an **"is-a"** relationship (a `Car` *is a* `Vehicle`).
- **`extends`:** The keyword used to create a subclass.
- **`super()`:** Used in the child's **constructor** to call the parent's constructor.
- **`@override`:** An annotation that *must* be used when replacing a parent's method. It's a vital safety check.
- **`super.method()`:** Used *inside* a method to call the parent's version of that method.

- **Main Benefit: Code Reusability**. We write common code once in the parent and reuse it in all child classes.