

Task 5

FLUTTER




Introduction To OOP



Organizes code for better readability
Enhances reusability and reduces duplication
Ensures better security and error handling
Essential for building Flutter applications



Encapsulation

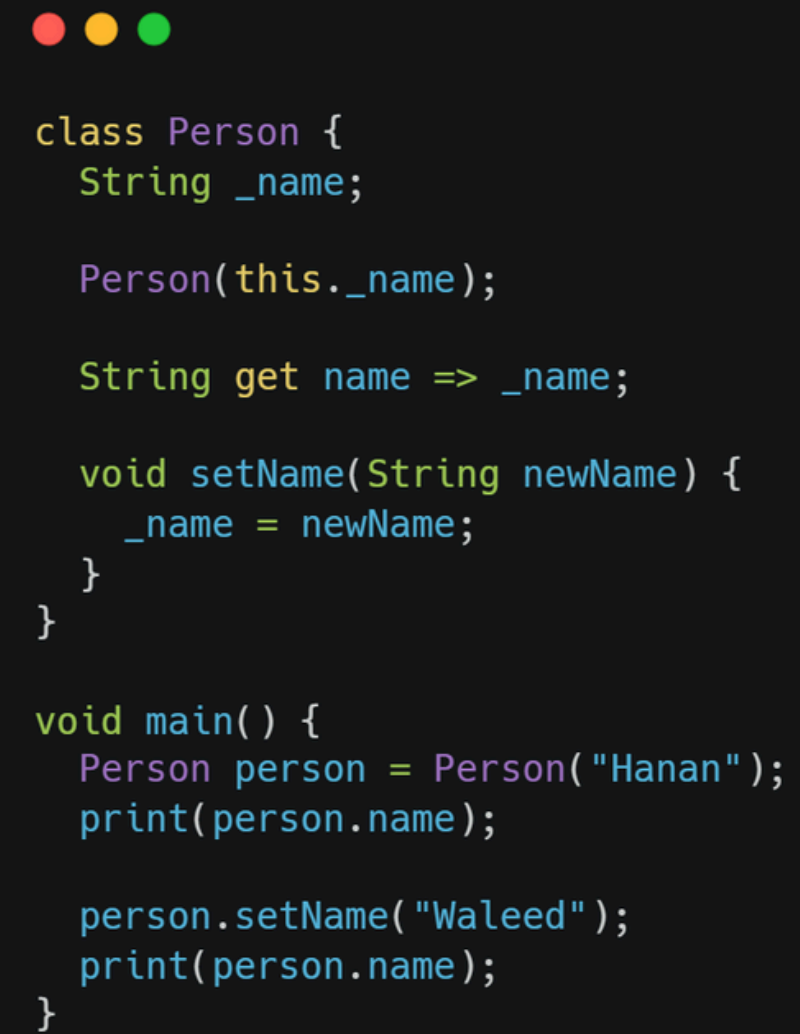


**Bundling of data and methods that operate
on that data within a single unit or object**

Protects internal data from external access

Controls how data is accessed and modified

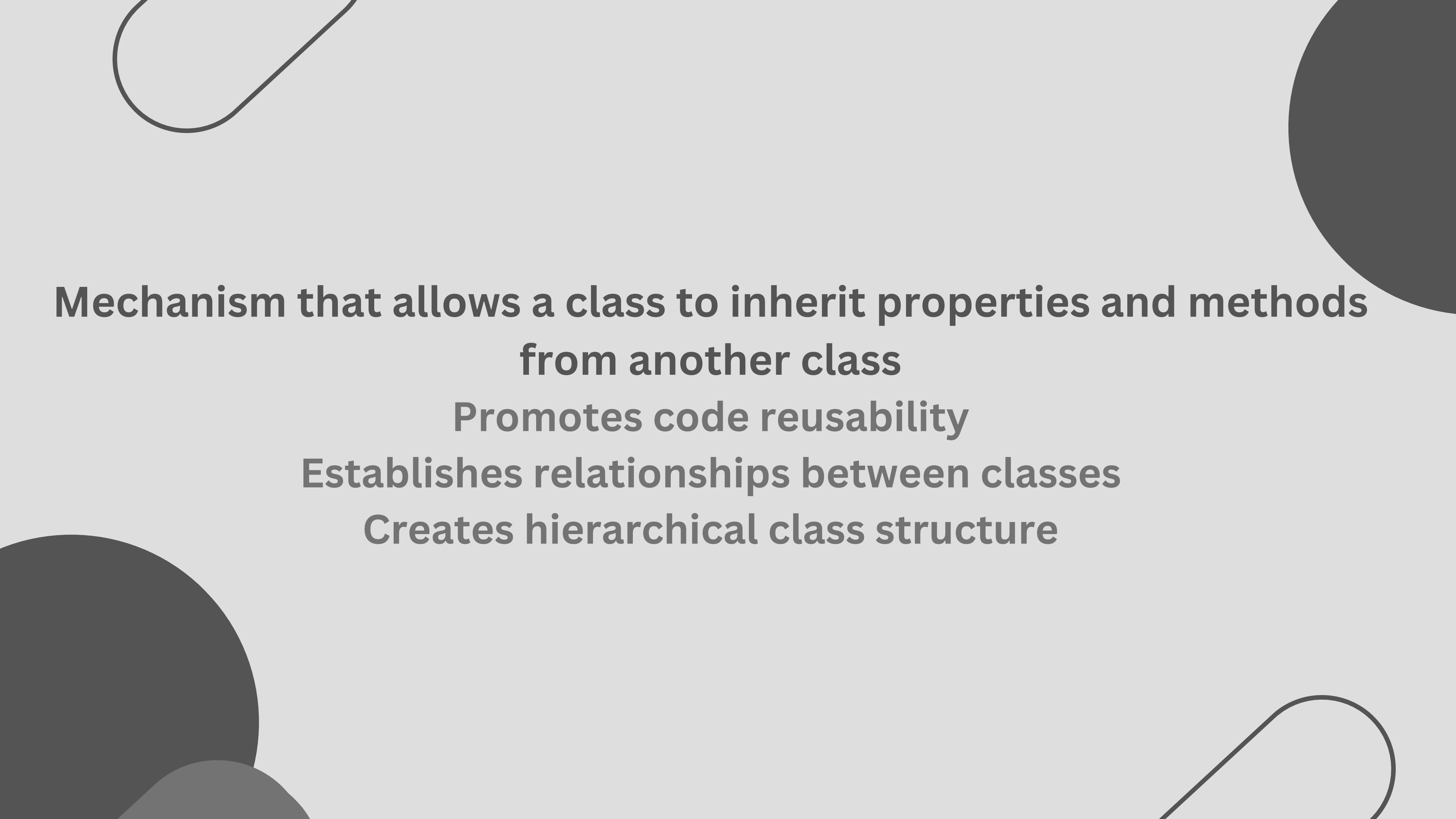
Reduces complexity and increases security



```
class Person {  
    String _name;  
  
    Person(this._name);  
  
    String get name => _name;  
  
    void setName(String newName) {  
        _name = newName;  
    }  
}  
  
void main() {  
    Person person = Person("Hanan");  
    print(person.name);  
  
    person.setName("Waleed");  
    print(person.name);  
}
```



Inheritance

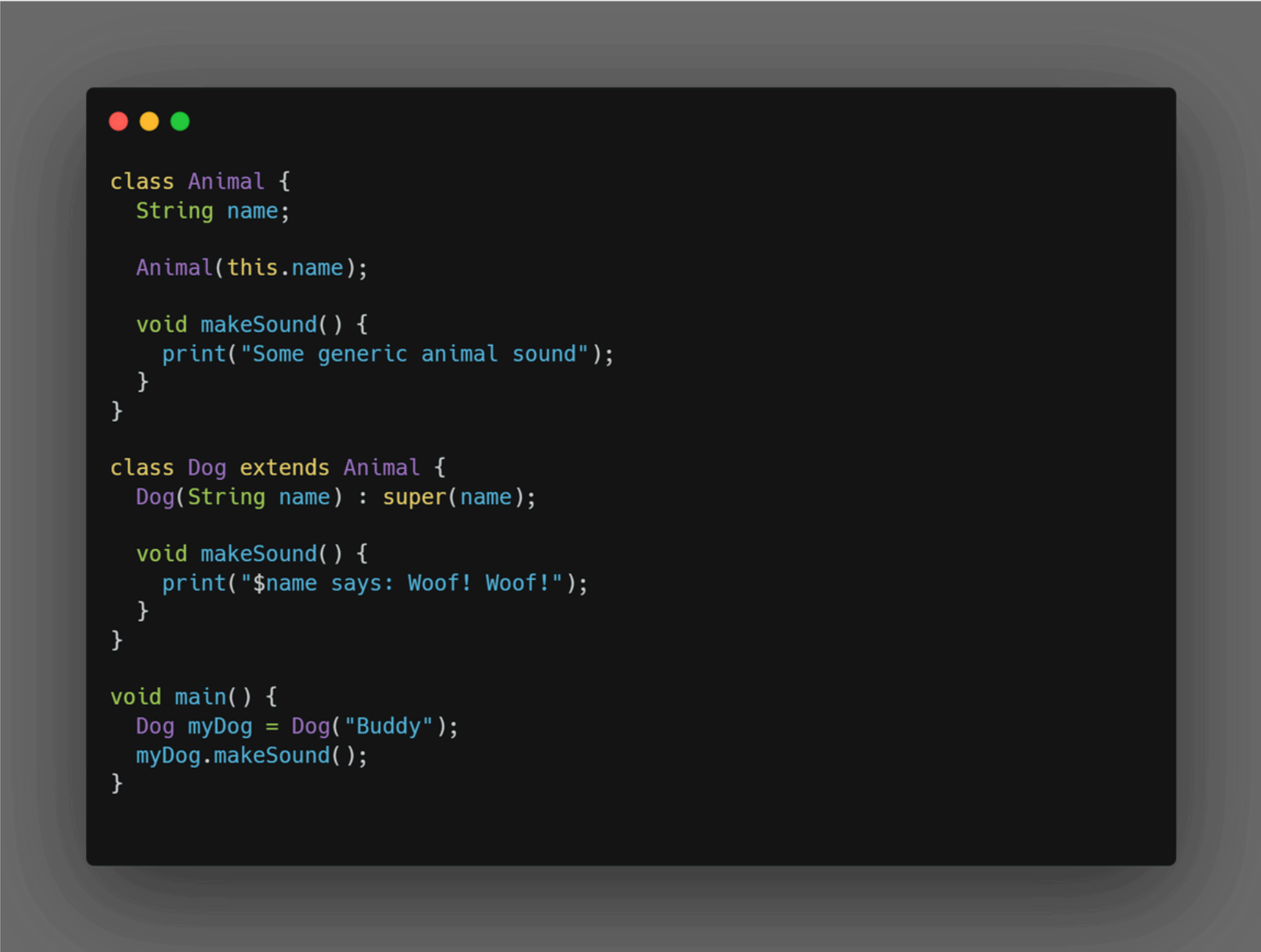


**Mechanism that allows a class to inherit properties and methods
from another class**

Promotes code reusability

Establishes relationships between classes

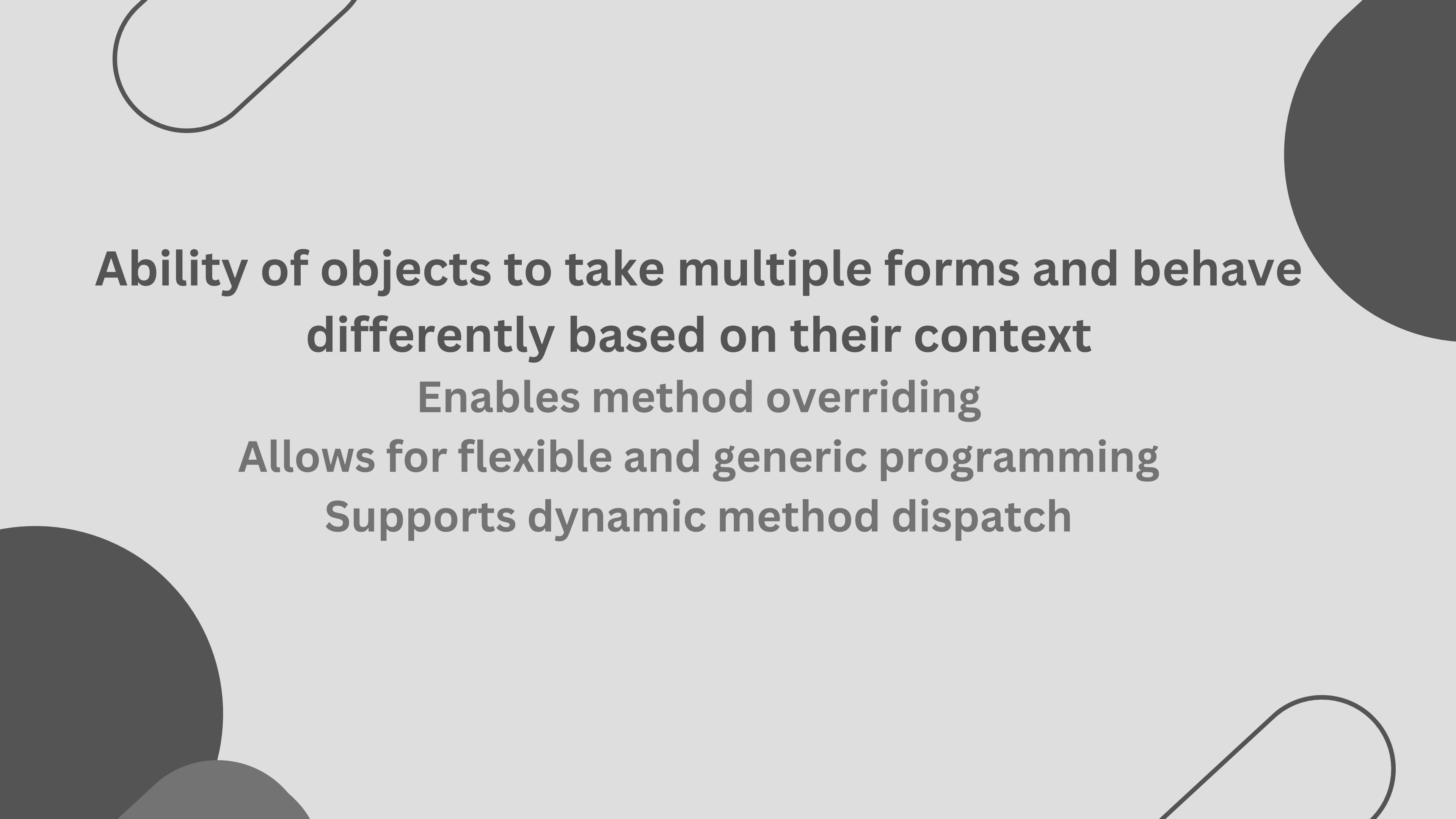
Creates hierarchical class structure



```
class Animal {  
    String name;  
  
    Animal(this.name);  
  
    void makeSound() {  
        print("Some generic animal sound");  
    }  
}  
  
class Dog extends Animal {  
    Dog(String name) : super(name);  
  
    void makeSound() {  
        print("$name says: Woof! Woof!");  
    }  
}  
  
void main() {  
    Dog myDog = Dog("Buddy");  
    myDog.makeSound();  
}
```



Polymorphism



Ability of objects to take multiple forms and behave differently based on their context

Enables method overriding

Allows for flexible and generic programming

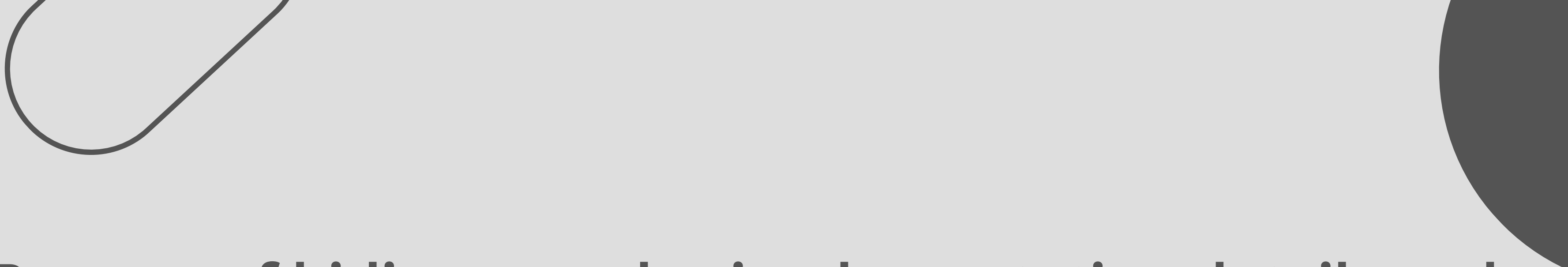
Supports dynamic method dispatch



```
class Animal {  
    void makeSound() {  
        print("Some generic animal sound");  
    }  
}  
  
class Dog extends Animal {  
    @override  
    void makeSound() {  
        print("Woof! Woof!");  
    }  
}  
  
class Cat extends Animal {  
    @override  
    void makeSound() {  
        print("Meow! Meow!");  
    }  
}  
  
void main() {  
    Animal myAnimal = Animal();  
    Animal myDog = Dog();  
    Animal myCat = Cat();  
  
    myAnimal.makeSound();  
    myDog.makeSound();  
    myCat.makeSound();  
}
```

The background is a light gray color. It features several abstract dark gray elements: a large circle in the top right corner, a large circle in the bottom left corner, and two curved lines, one in the top left and one in the bottom right, resembling stylized parentheses or swooshes.

Abstraction



**Process of hiding complex implementation details and
showing only necessary features**

Reduces complexity

Focuses on what an object does rather than how it does it

Provides a simple interface for complex systems





```
abstract class Animal {  
    void makeSound();  
}  
  
class Dog extends Animal {  
    @override  
    void makeSound() {  
        print("Woof! Woof!");  
    }  
}  
  
class Cat extends Animal {  
    @override  
    void makeSound() {  
        print("Meow! Meow!");  
    }  
}  
  
void main() {  
    Animal myDog = Dog();  
    Animal myCat = Cat();  
  
    myDog.makeSound();  
    myCat.makeSound();  
}
```



Benefits of Using OOP Principles



Code Organization

Better structure

Easier maintenance

Clear dependencies




Reusability

Less duplicate code

Modular design

Time-efficient



Scalability
Easy to extend
Simple to modify
Future-proof



Summary



Object-Oriented Programming

is a programming paradigm that
organizes code into classes and objects,
making it more modular, reusable
and maintainable



**THANK
YOU**