Interfaces, Composition, and System Design

Maurice Naftalin

Java Champion, JavaOne Rock Star

Author: *Mastering Lambdas, Java Generics and Collections*

@mauricenaftalin

Module Overview

Introduction to Interfaces

Demo: Paying through an Interface

The Fragile Base Class Problem

Delegation and the Decorator Pattern

The Strategy Pattern

The Interface Segregation Principle

The Dependency Inversion Principle

Introduction to Interfaces

Key Java feature

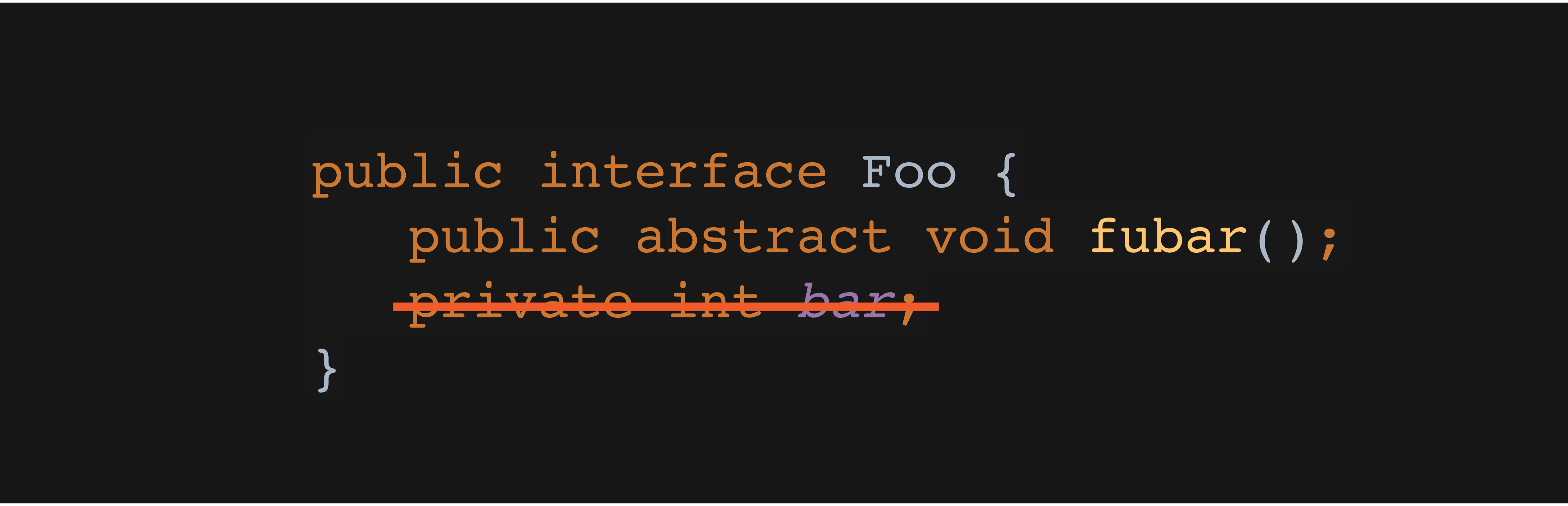
Essential to building well-engineered systems

Like abstract classes

But with one key difference…

Abstract Classes vs. Interfaces

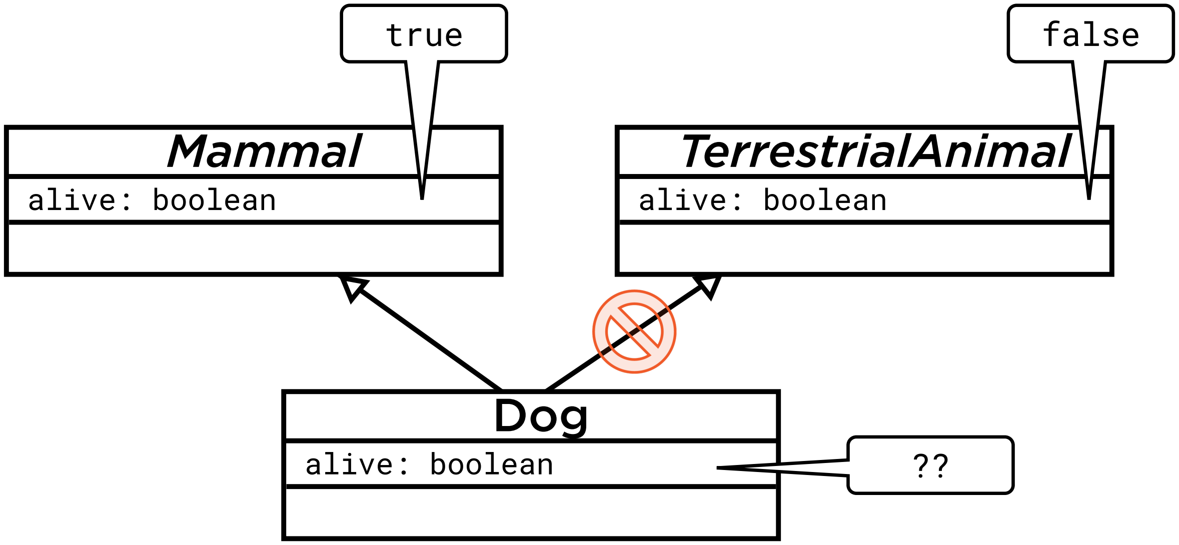
	Abstract Classes	**Interfaces**
Instantiable?	No	No
Abstract methods?	Yes	Yes
Non-abstract methods?	Yes	Yes, since Java 8 (*default methods*)
Contain state?	Yes	No



```
public interface Foo {
    public abstract void fubar();
    ~~private int bar;~~
}
```

Interfaces cannot contain state

The key difference from classes!



true

false

Mammal

alive: boolean

TerrestrialAnimal

alive: boolean

Dog

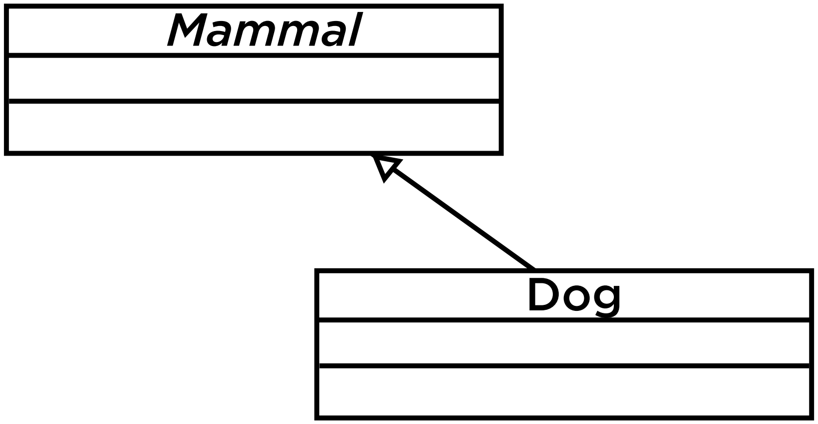
alive: boolean

??

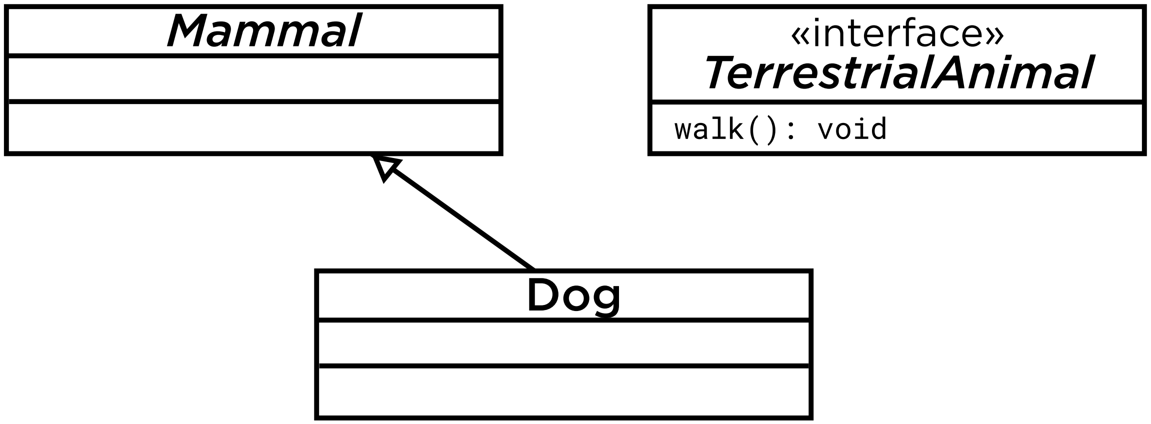
Abstract Classes vs. Interfaces

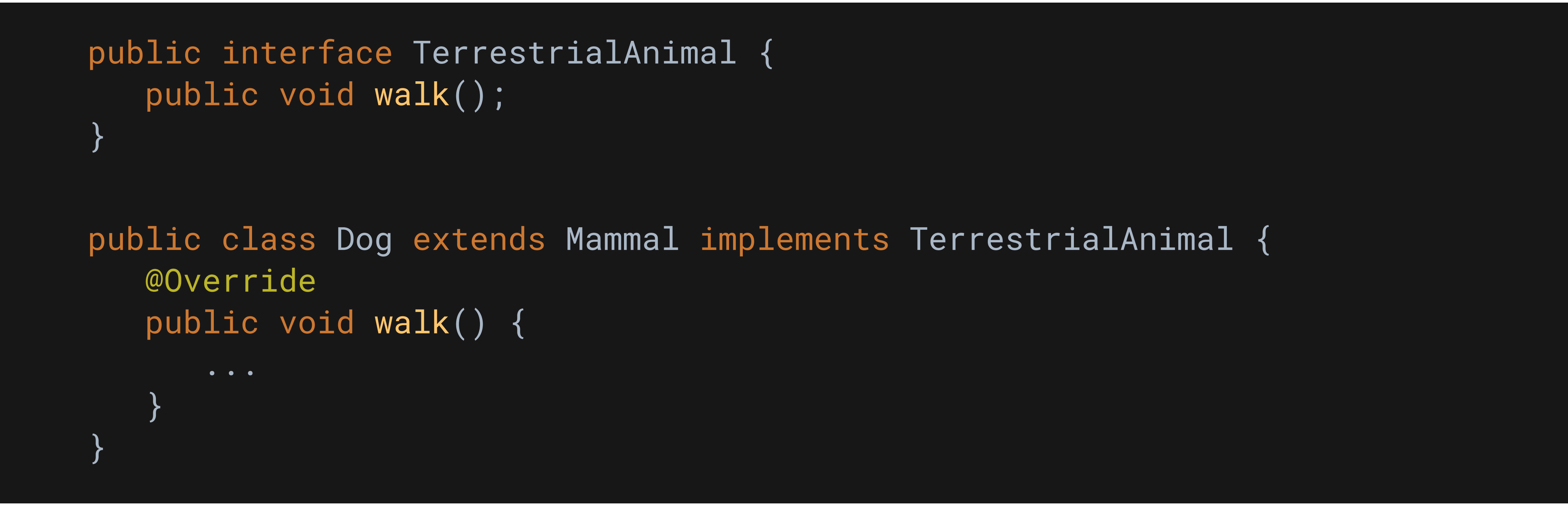
	Abstract Classes	Interfaces
Instantiable?	No	No
Abstract methods?	Yes	Yes
Non-abstract methods?	Yes	Yes, since Java 8
Contain state?	Yes	No
Multiple inheritance?	No	Yes

```java
public class Dog extends Mammal {

        ...
}
```

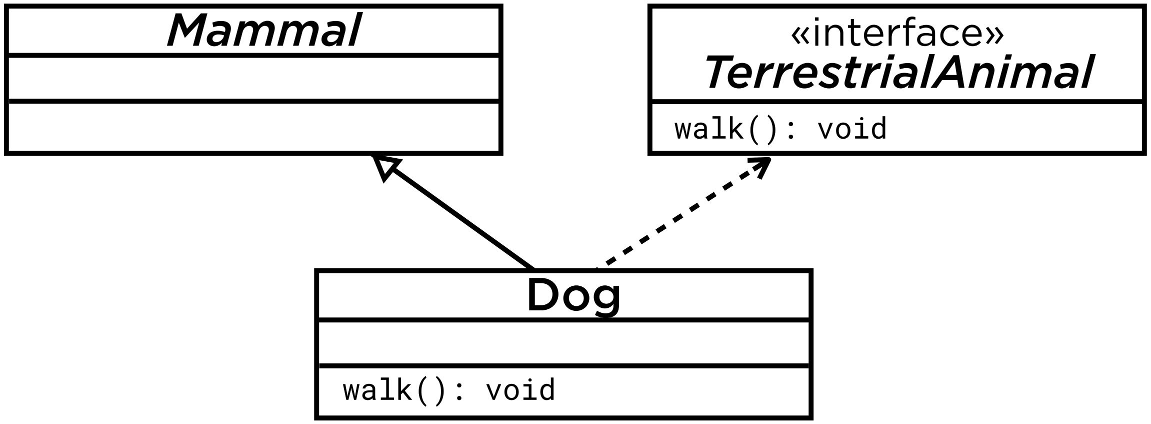


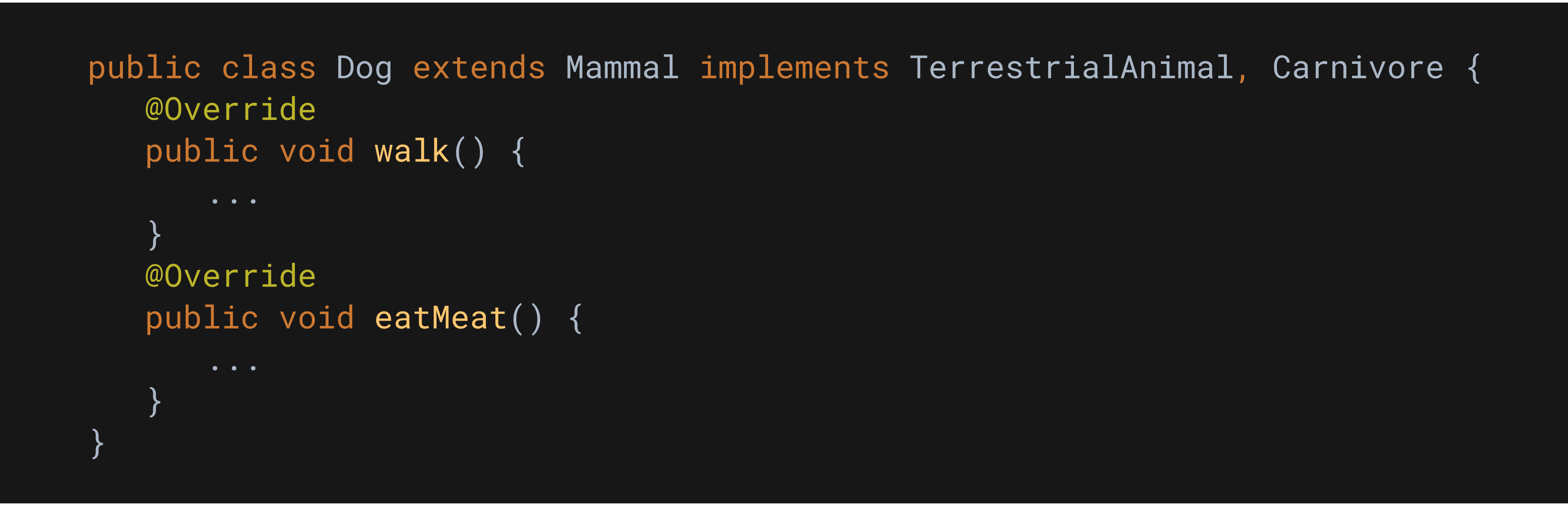
```
public interface TerrestrialAnimal {
   public void walk();
}


public class Dog extends Mammal {


      ...

}
```



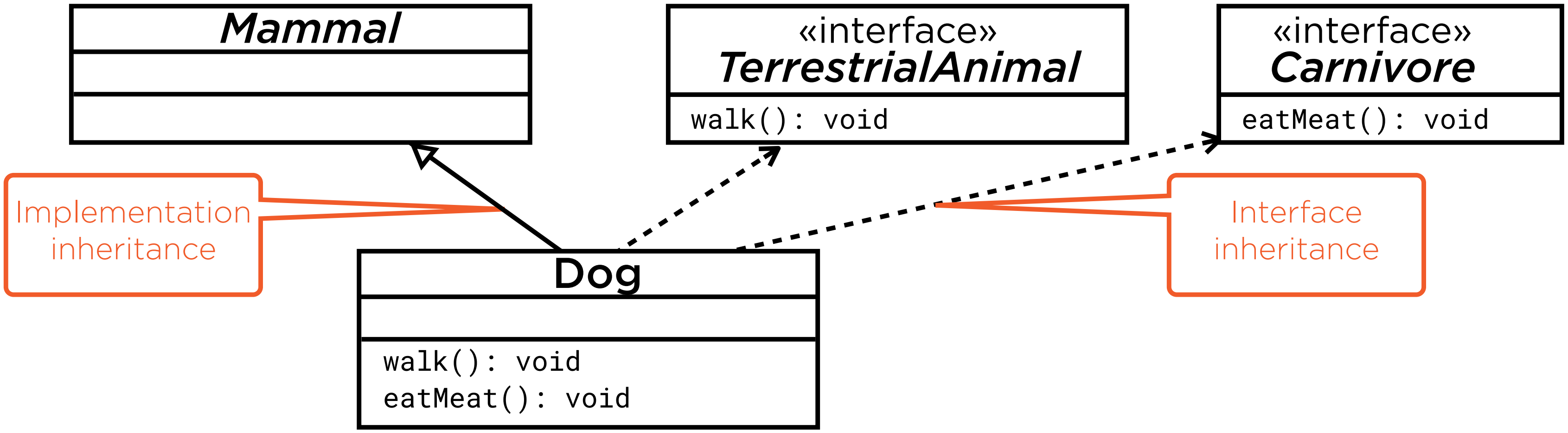


```
public interface TerrestrialAnimal {
   public void walk();
}

public class Dog extends Mammal implements TerrestrialAnimal {
   @Override
   public void walk() {
      ...
   }
}
```





```
public class Dog extends Mammal implements TerrestrialAnimal, Carnivore {
    @Override
    public void walk() {
        ...
    }
    @Override
    public void eatMeat() {
        ...
    }
}
```



Sorting a List of Products

“Electric Toothbrush”	5000	PHYSICAL
“Baby Alarm”	5000	PHYSICAL
“War and Peace (e-book)”	1000	DIGITAL
“Super Sofa”	50_000	PHYSICAL

Sorting a List of Products

“Baby Alarm”	5000	PHYSICAL
“Electric Toothbrush”	5000	PHYSICAL
“Super Sofa”	50_000	PHYSICAL
“War and Peace (e-book)”	1000	DIGITAL

Demo: Casting

```java
public static <T> void sort(List<T> list) {
    ...
    // ask pairs of list elements which one comes first
    ...
}
```

Collections.sort

- comparing two elements
- core operation of sorting

```java
public abstract class Product {
    private String name;
    ...

    public int compareTo(Product otherProduct) {
        return name.compareTo(otherProduct.name);
    }
    ...
}
```

`Product` has a Method `compareTo(Product)`

```
public static <T> void sort(List<T> list) {
    ...
    // ask pairs of list elements which one comes first
    ...
}
```

by calling the `compareTo` method

Collections.sort

- comparing two elements
- core operation of sorting

```java
public interface Comparable<T> {
   public int compareTo(T o);
}
```

```java
public class Product
                implements Comparable<Product> {
   ...

    @Override
    public int compareTo(Product otherProduct) {
       return name.compareTo(otherProduct.name);
    }

    ...
}
```

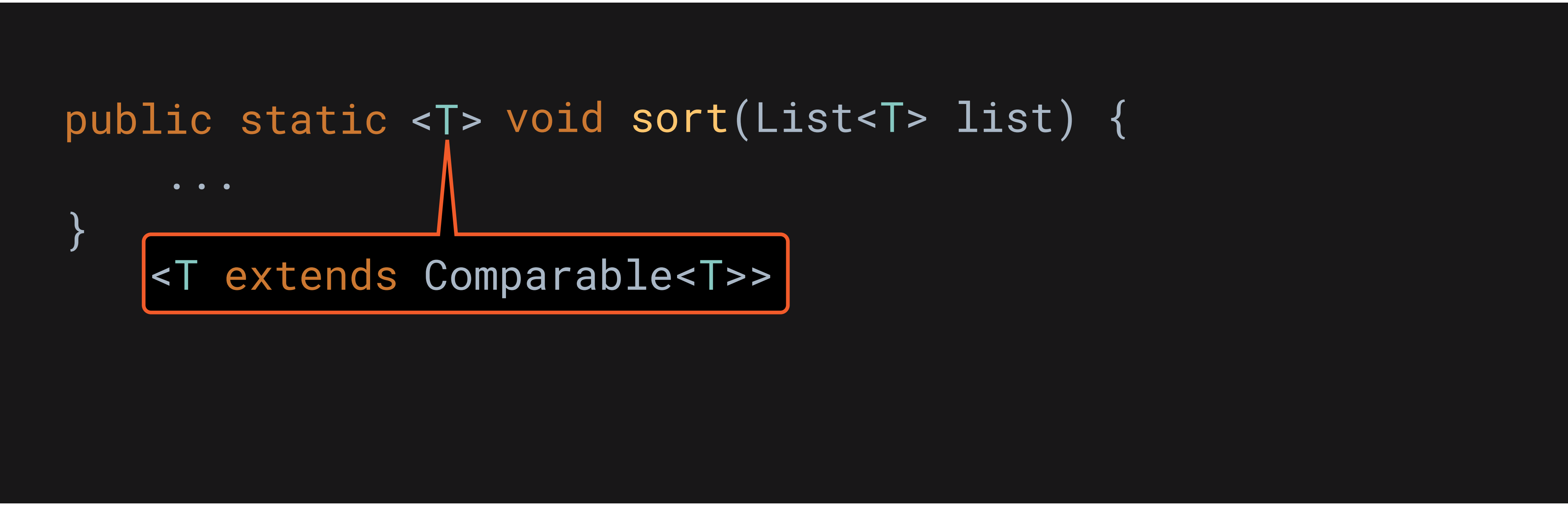
`Product` Implements `Comparable<Product>`

Which means it has to override `compareTo(Product)`

```java
public static <T> void sort(List<T> list) {
    ...
}
```

Does the actual type (eg **Product**) implement **Comparable**?

Constraining the Type Parameter of `Collections.sort`



```java
public static <T> void sort(List<T> list) {
    ...
}
```

```java
<T extends Comparable<T>>
```

Constraining the Type Parameter of `Collections.sort`

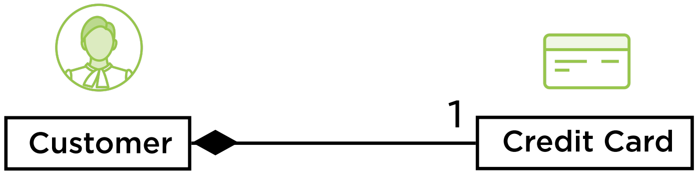
```java
public static <T extends Comparable<T>> void sort(List<T> list) {
    ...
}
```

Constraining the Type Parameter of `Collections.sort`

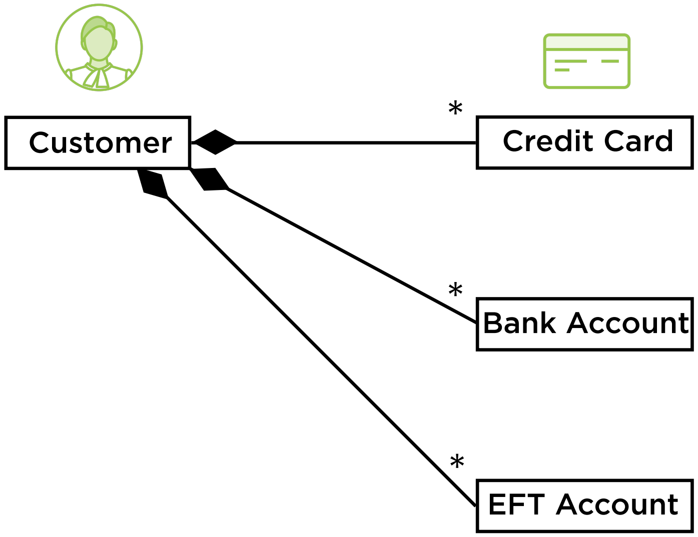
Demo

Paying through an Interface

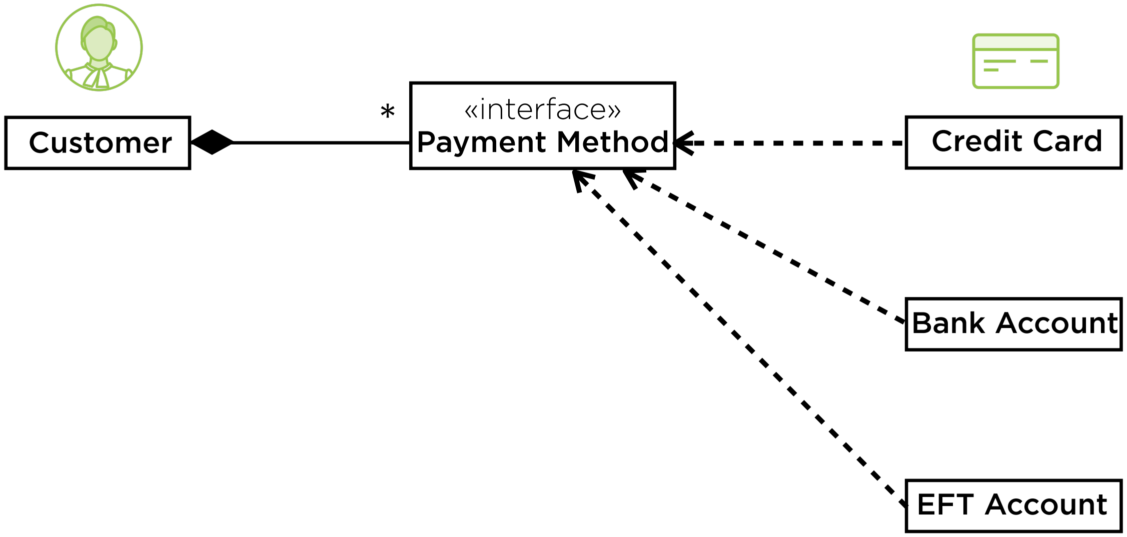
More Ways to Pay



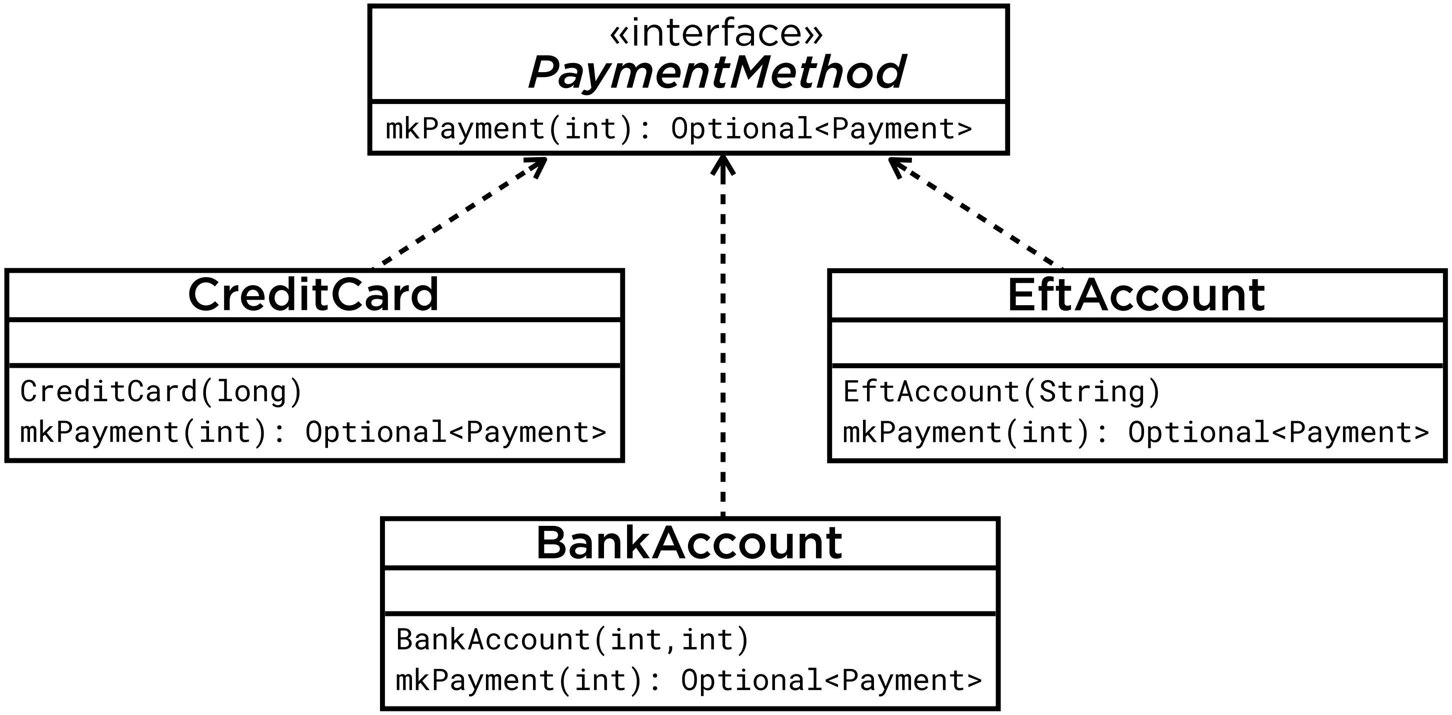
More Ways to Pay



More Ways to Pay



More Ways to Pay


«interface»
PaymentMethod
mkPayment(int): Optional<Payment>
CreditCard
CreditCard(long)
mkPayment(int): Optional<Payment>
EftAccount
EftAccount(String)
mkPayment(int): Optional<Payment>
BankAccount
BankAccount(int,int)
mkPayment(int): Optional<Payment>

Demo

The Fragile Base Class Problem

What’s Wrong with Implementation Inheritance?

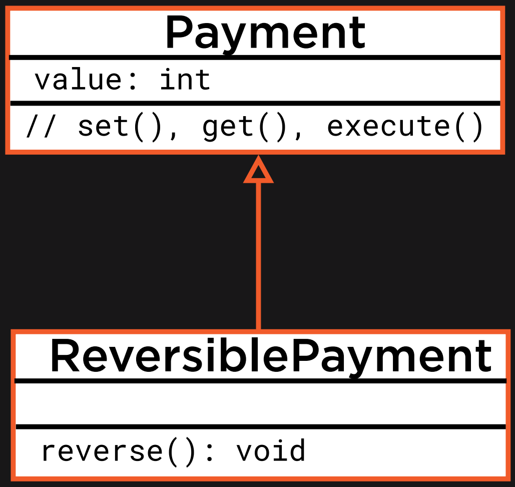
Working in one package

Implementations can safely depend on one another

Published libraries

Must be able to evolve independently from client code

```
public class Payment {
    public void execute() {...}
}


public class ReversiblePayment extends Payment {
    public void reverse() {
        ...
        execute();
    }
}
```



Payment

value: int

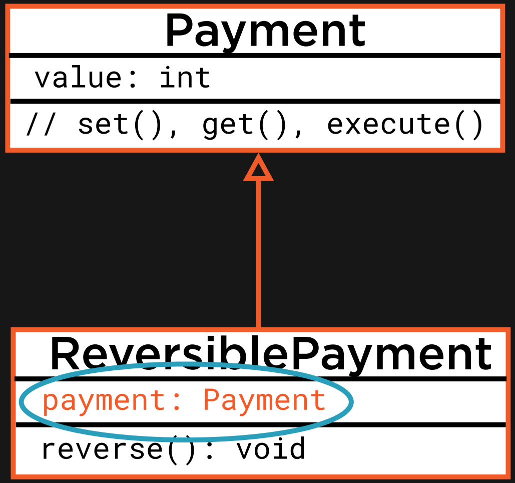
// set(), get(), execute()

ReversiblePayment

reverse(): void

Implementation Inheritance

```java
public class Payment {
    public void execute() {...}
}


public class ReversiblePayment extends Payment {
    public void reverse() {
        ...
        execute();
    }
}
```



Payment

value: int

// set(), get(), execute()

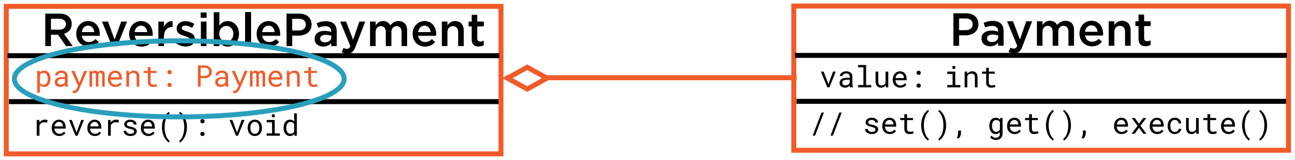
ReversiblePayment

payment: Payment

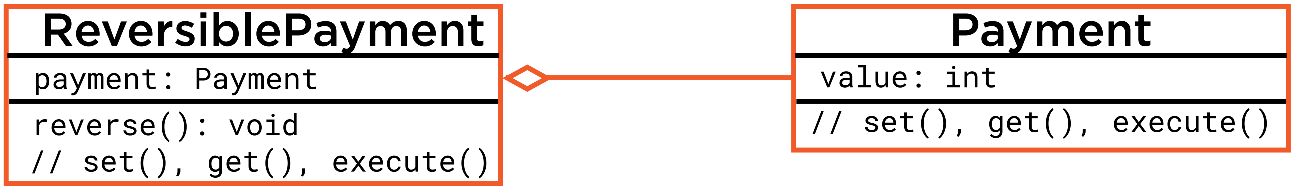
reverse(): void

Implementation Inheritance

Call Forwarding



Call Forwarding



ReversiblePayment

payment: Payment

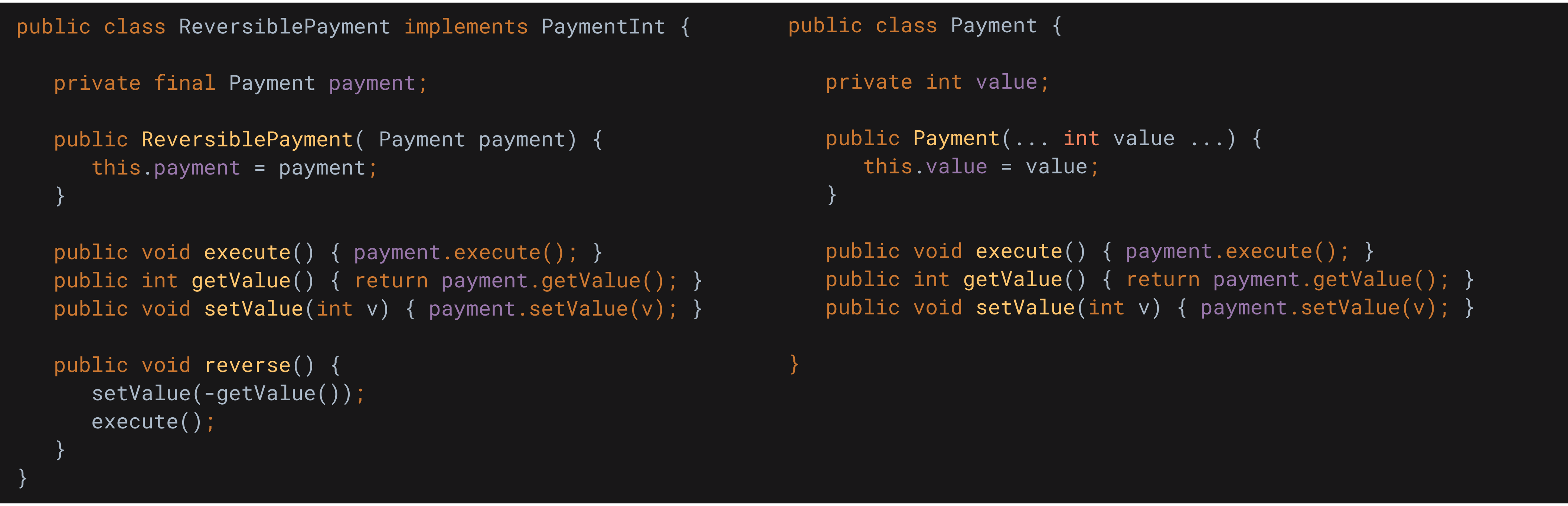
reverse(): void

// set(), get(), execute()

Payment

value: int

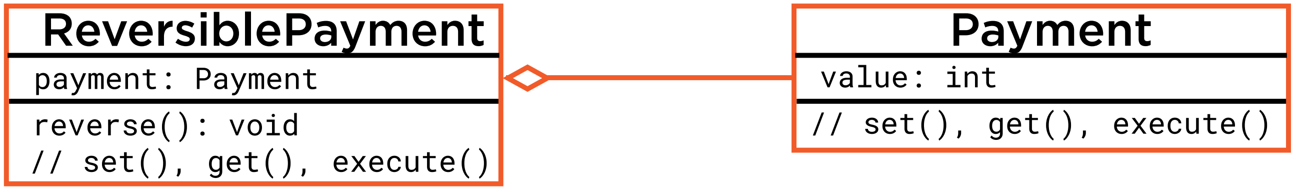
// set(), get(), execute()

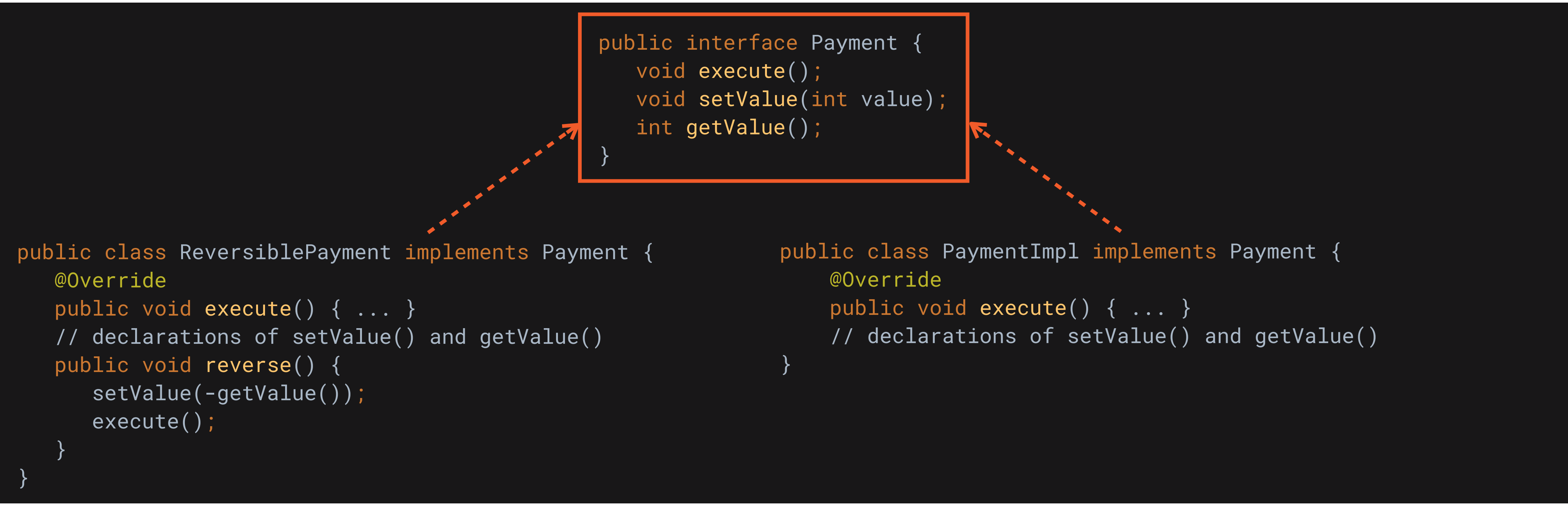


```java
public class ReversiblePayment implements PaymentInt {

   private final Payment payment;

   public ReversiblePayment( Payment payment) {
      this.payment = payment;
   }

   public void execute() { payment.execute(); }
   public int getValue() { return payment.getValue(); }
   public void setValue(int v) { payment.setValue(v); }

   public void reverse() {
      setValue(-getValue());
      execute();
   }
}
```

```java
public class Payment {

   private int value;

   public Payment(... int value ...) {
      this.value = value;
   }

   public void execute() { payment.execute(); }
   public int getValue() { return payment.getValue(); }
   public void setValue(int v) { payment.setValue(v); }

}
```

Call Forwarding

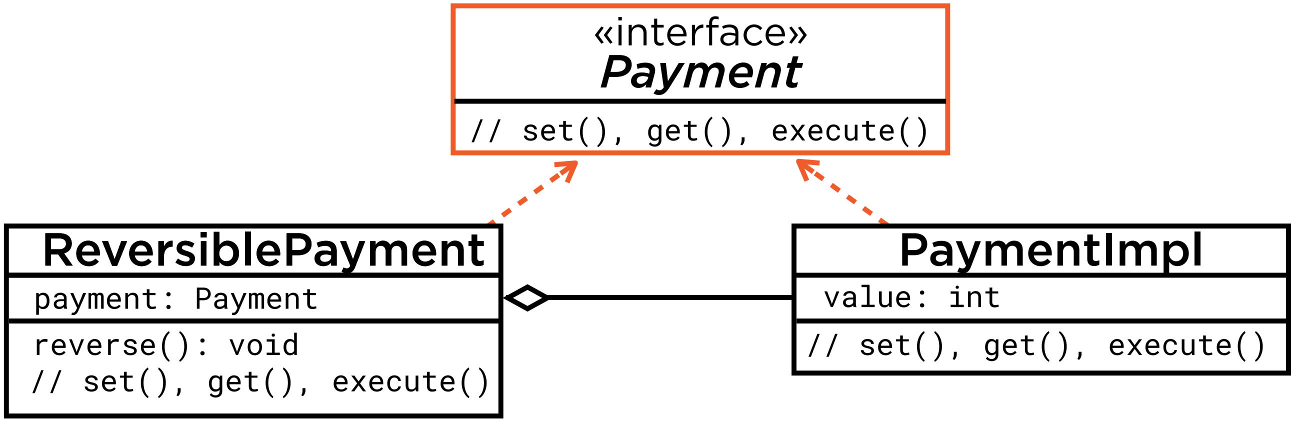




```
public interface Payment {
    void execute();
    void setValue(int value);
    int getValue();
}
```

```
public class ReversiblePayment implements Payment {
    @Override
    public void execute() { ... }
    // declarations of setValue() and getValue()
    public void reverse() {
        setValue(-getValue());
        execute();
    }
}
```

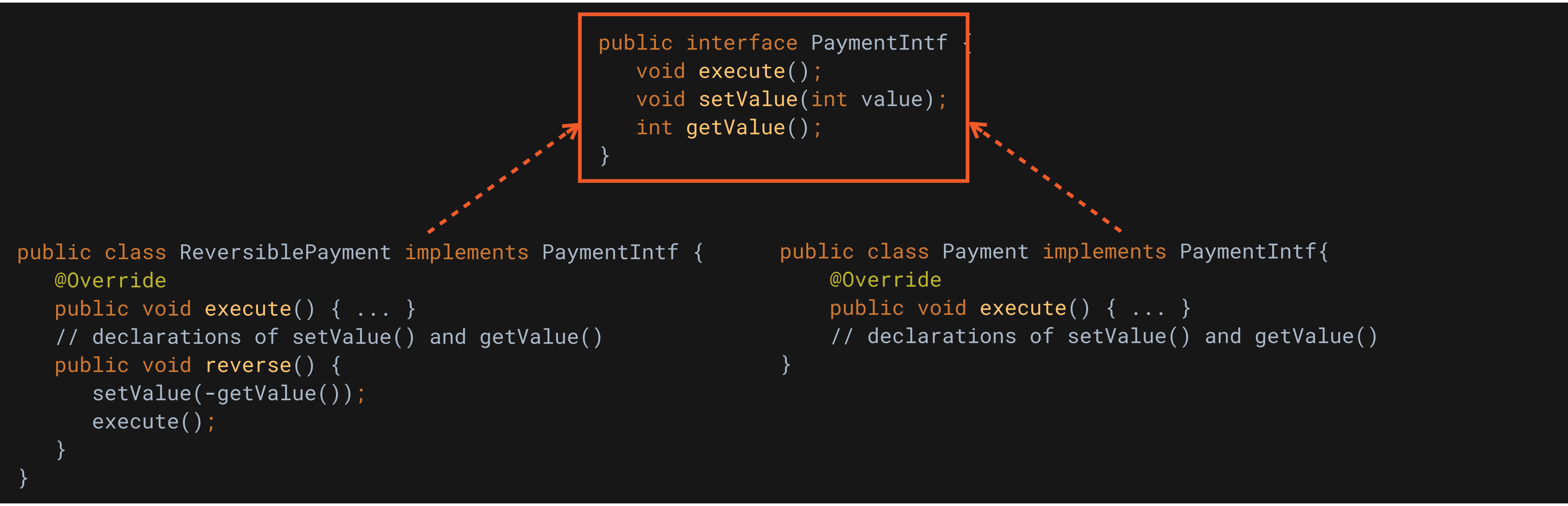
```
public class PaymentImpl implements Payment {
    @Override
    public void execute() { ... }
    // declarations of setValue() and getValue()
}
```



«interface»
Payment
// set(), get(), execute()

ReversiblePayment
payment: Payment
reverse(): void
// set(), get(), execute()

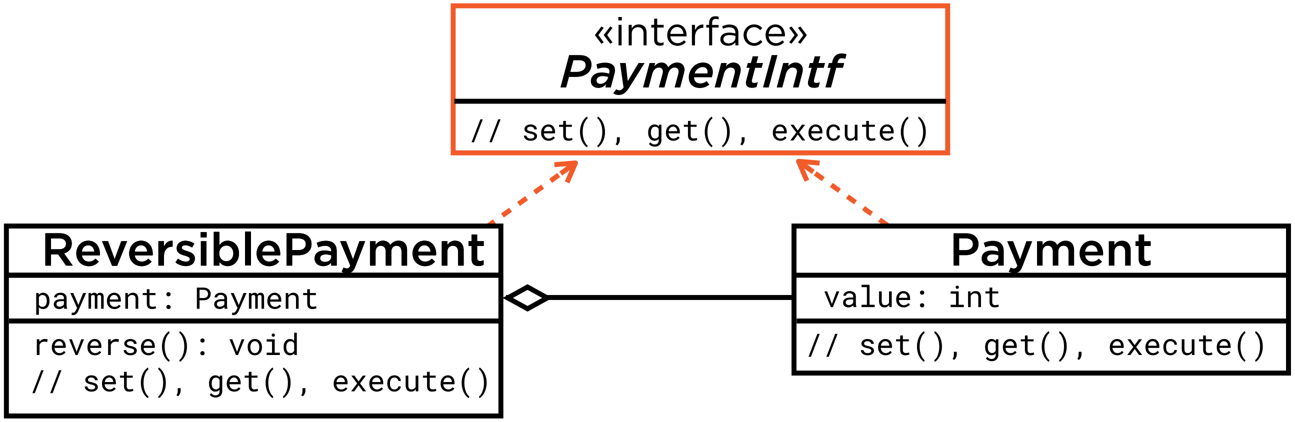
PaymentImpl
value: int
// set(), get(), execute()



```java
public interface PaymentIntf {
   void execute();
   void setValue(int value);
   int getValue();
}
```

```java
public class ReversiblePayment implements PaymentIntf {
   @Override
   public void execute() { ... }
   // declarations of setValue() and getValue()
   public void reverse() {
      setValue(-getValue());
      execute();
   }
}
```

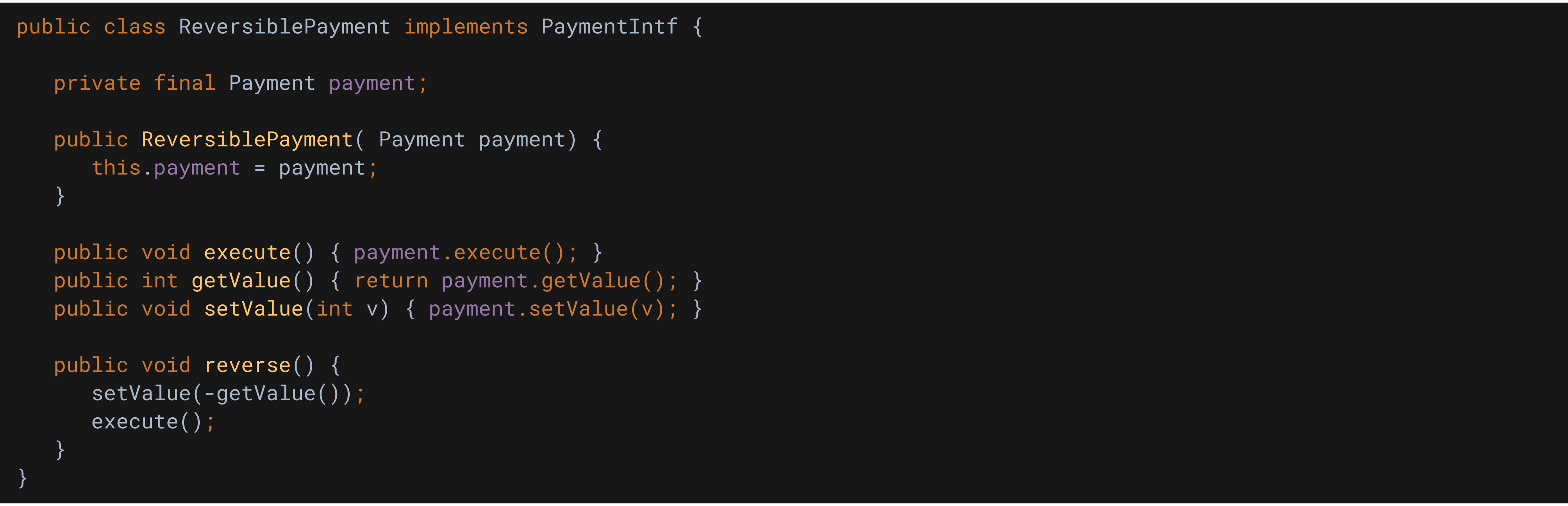
```java
public class Payment implements PaymentIntf{
     @Override
     public void execute() { ... }
     // declarations of setValue() and getValue()
}
```



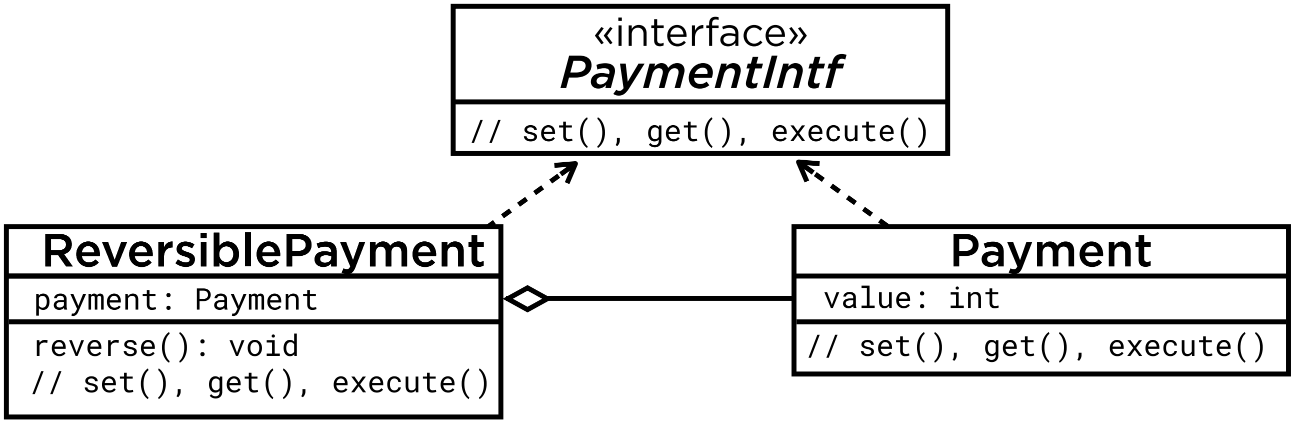
«interface»
PaymentIntf
// set(), get(), execute()

ReversiblePayment
payment: Payment
reverse(): void
// set(), get(), execute()

Payment
value: int
// set(), get(), execute()



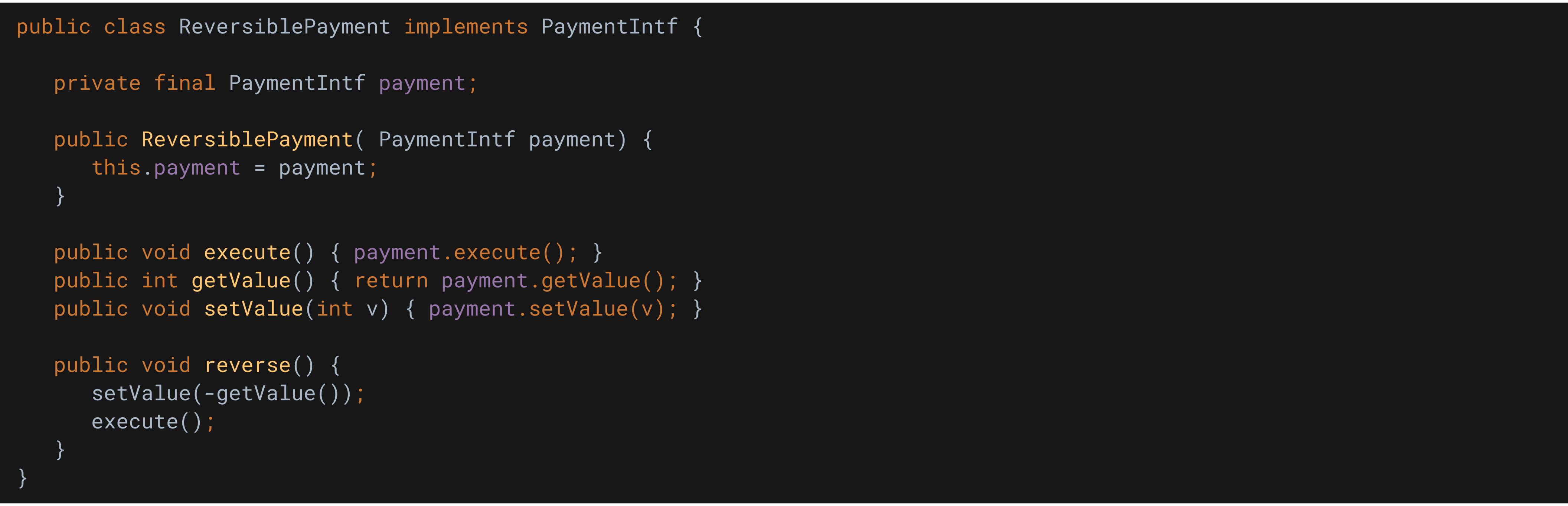
```java
public class ReversiblePayment implements PaymentIntf {

   private final Payment payment;

   public ReversiblePayment( Payment payment) {
      this.payment = payment;
   }

   public void execute() { payment.execute(); }
   public int getValue() { return payment.getValue(); }
   public void setValue(int v) { payment.setValue(v); }

   public void reverse() {
      setValue(-getValue());
      execute();
   }
}
```



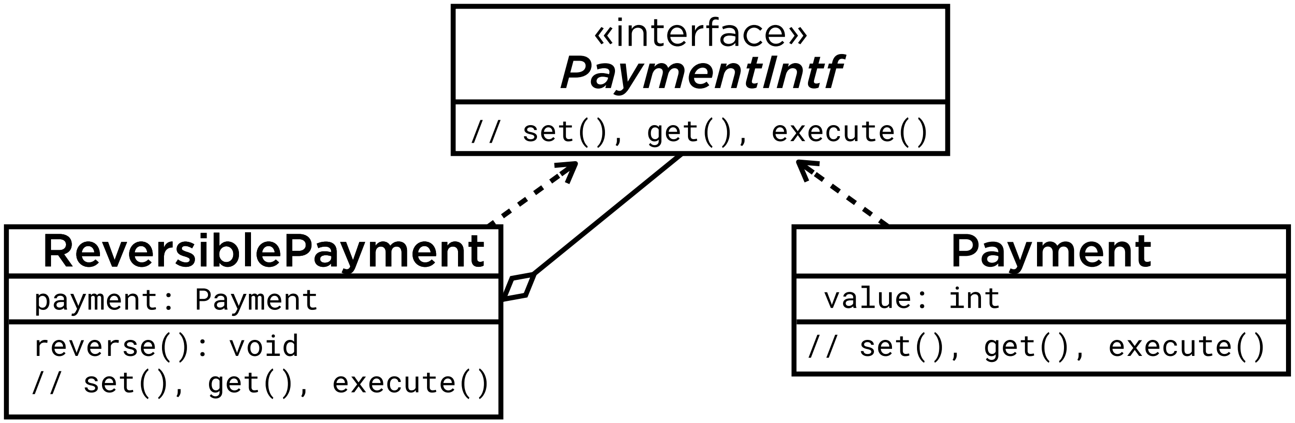
«interface»
PaymentIntf
// set(), get(), execute()

ReversiblePayment
payment: Payment
reverse(): void
// set(), get(), execute()

Payment
value: int
// set(), get(), execute()



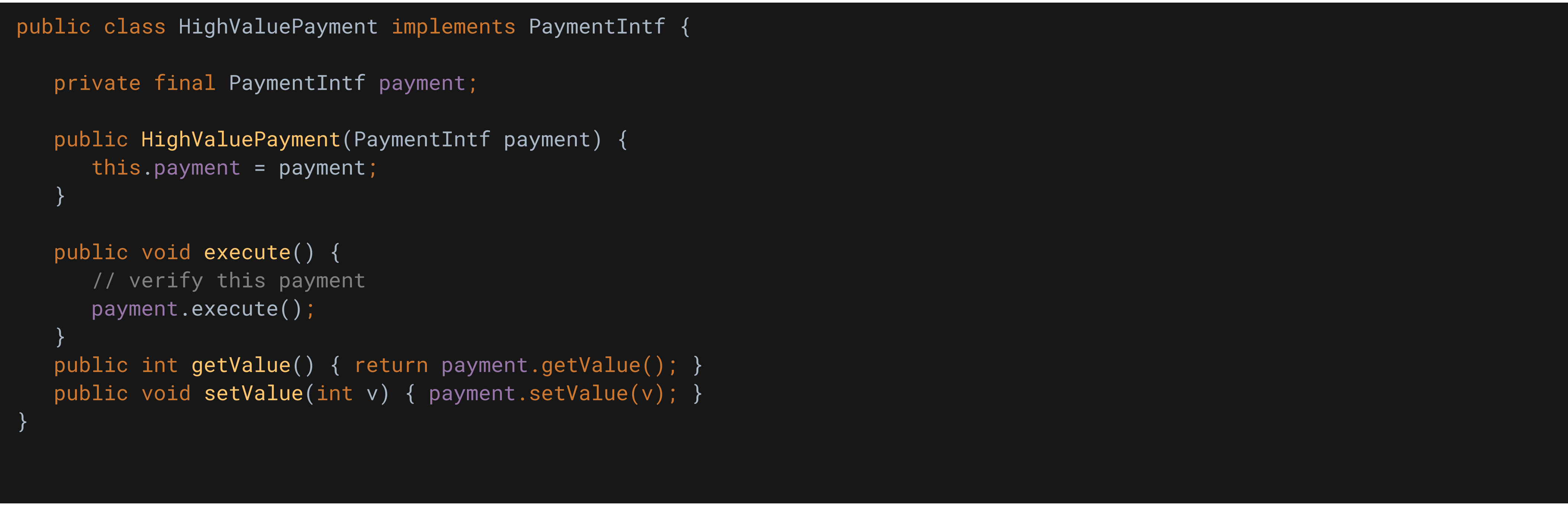
```java
public class ReversiblePayment implements PaymentIntf {

   private final PaymentIntf payment;

   public ReversiblePayment( PaymentIntf payment) {
      this.payment = payment;
   }

   public void execute() { payment.execute(); }
   public int getValue() { return payment.getValue(); }
   public void setValue(int v) { payment.setValue(v); }

   public void reverse() {
      setValue(-getValue());
      execute();
   }
}
```



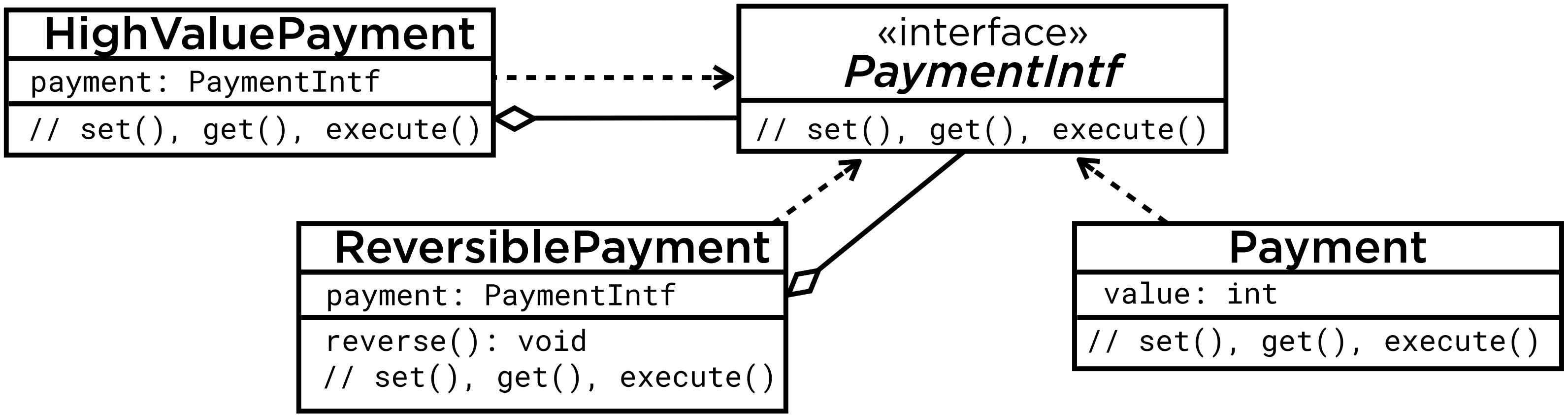
«interface»
PaymentIntf
// set(), get(), execute()

ReversiblePayment
payment: Payment
reverse(): void
// set(), get(), execute()

Payment
value: int
// set(), get(), execute()



```java
public class HighValuePayment implements PaymentIntf {

   private final PaymentIntf payment;

   public HighValuePayment(PaymentIntf payment) {
      this.payment = payment;
   }

   public void execute() {
      // verify this payment
      payment.execute();
   }
   public int getValue() { return payment.getValue(); }
   public void setValue(int v) { payment.setValue(v); }
}
```



HighValuePayment

payment: PaymentIntf

// set(), get(), execute()

«interface»
PaymentIntf

// set(), get(), execute()

ReversiblePayment

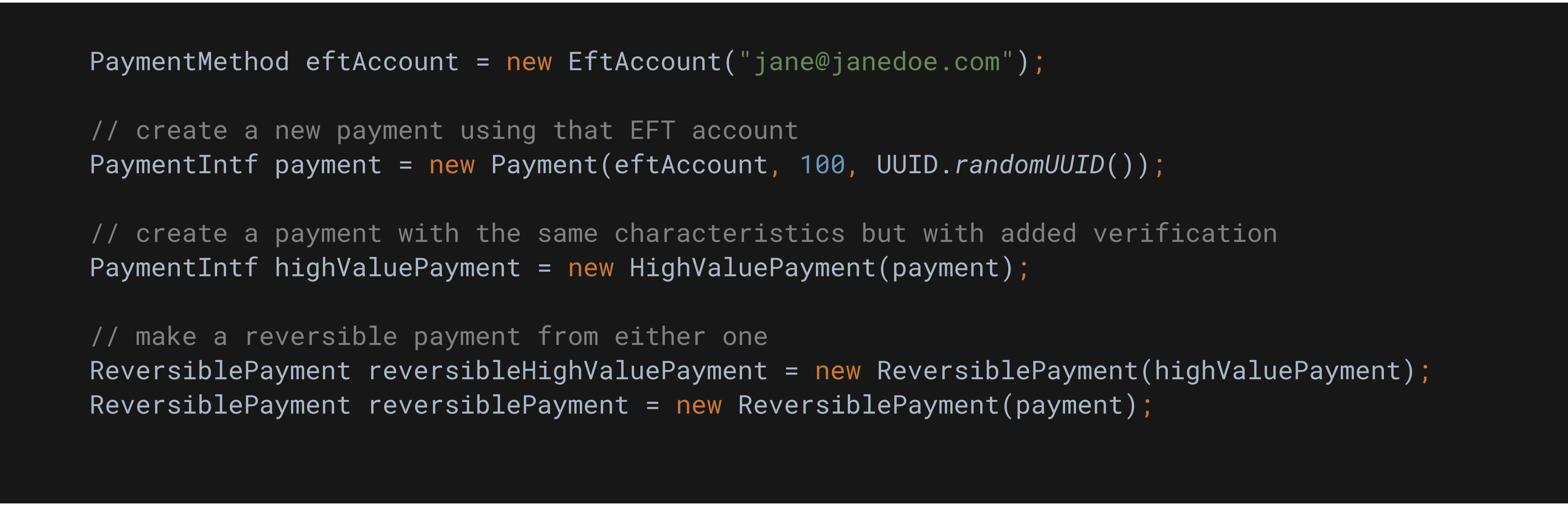
payment: PaymentIntf

reverse(): void
// set(), get(), execute()

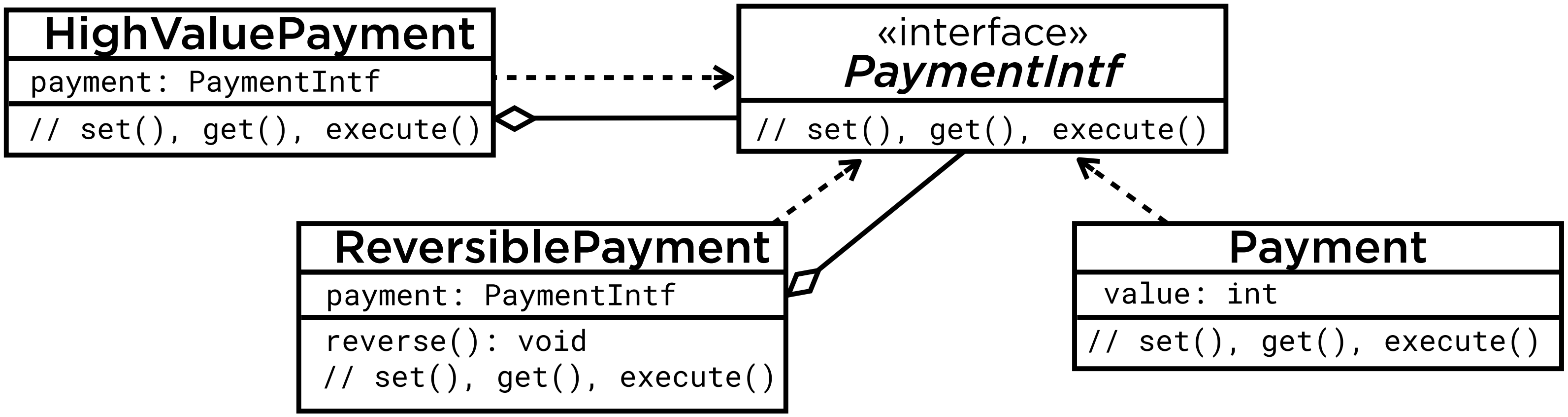
Payment

value: int

// set(), get(), execute()



```java
PaymentMethod eftAccount = new EftAccount("jane@janedoe.com");

// create a new payment using that EFT account
PaymentIntf payment = new Payment(eftAccount, 100, UUID.randomUUID());

// create a payment with the same characteristics but with added verification
PaymentIntf highValuePayment = new HighValuePayment(payment);

// make a reversible payment from either one
ReversiblePayment reversibleHighValuePayment = new ReversiblePayment(highValuePayment);
ReversiblePayment reversiblePayment = new ReversiblePayment(payment);
```



HighValuePayment

payment: PaymentIntf

// set(), get(), execute()

«interface»
PaymentIntf

// set(), get(), execute()

ReversiblePayment

payment: PaymentIntf

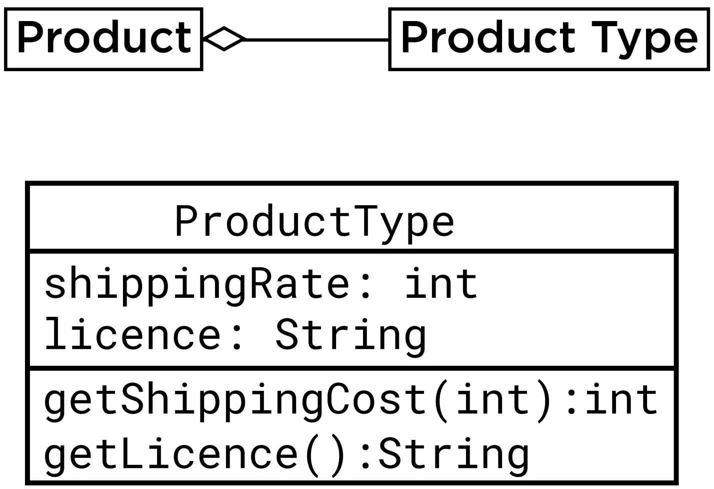
reverse(): void
// set(), get(), execute()

Payment

value: int

// set(), get(), execute()

The **Product** Class Revisited



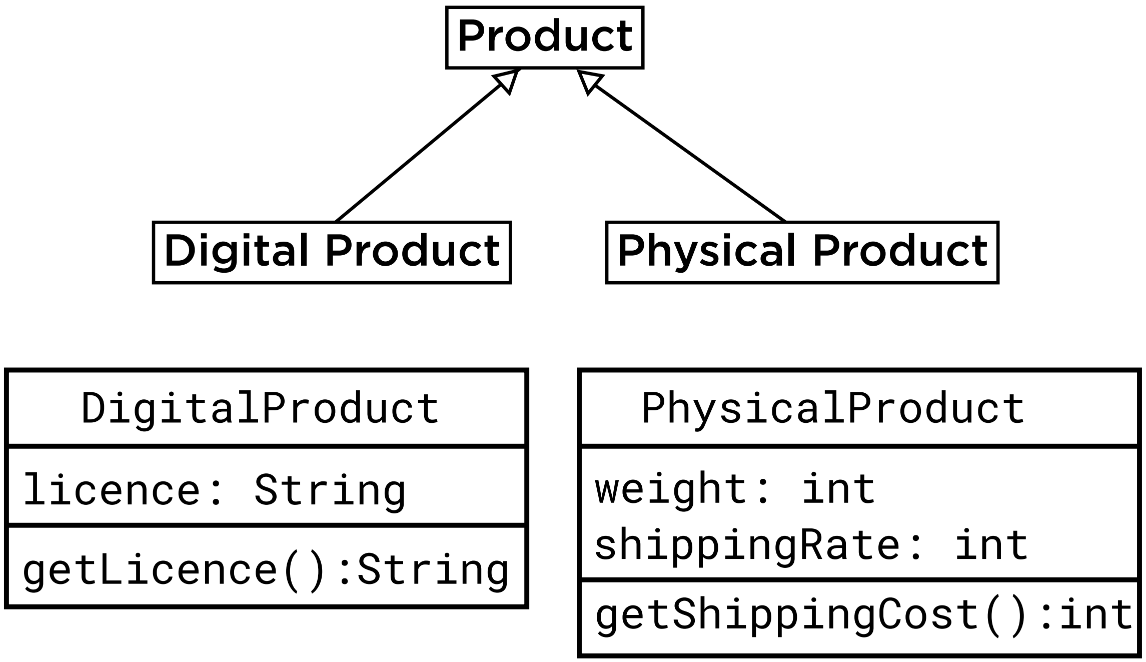
Product

Product Type

ProductType

shippingRate: int
licence: String

getShippingCost(int):int
getLicence():String



Product

Digital Product

Physical Product

DigitalProduct

licence: String

getLicence():String

PhysicalProduct

weight: int
shippingRate: int

getShippingCost():int

```
public class LineItem {

    public int getPrice() {
        // where should LineItem go for a PriceCalculator?
        PriceCalculator priceCalculator = ...

        // supply calculatePrice method with data from LineItem
        return priceCalculator.calculatePrice(product, quantity);
    }

    ...
}

public interface PriceCalculator {
   public int calculatePrice(Product product, int quantity);
}
```

The PriceCalculator Interface

- **Obtaining and using an instance**
- **The interface definition**

Who Should Create a `PriceCalculator`?

Answer*: a class that –

- **tracks instances of** `PriceCalculator`
- **closely uses instances of** `PriceCalculator`
- **contains instances of** `PriceCalculator`
- **has information that** `PriceCalculator` **will need**

* according to the principles of GRASP (General Responsibility Assignment Software Patterns)

```java
public class LineItem {

    public int getPrice() {
        // where should LineItem go for a PriceCalculator?
        PriceCalculator priceCalculator = ...

        // supply calculatePrice method with data from LineItem
        return priceCalculator.calculatePrice(product, quantity);
    }

    ...
}

public interface PriceCalculator {
   public int calculatePrice(Product product, int quantity);
}
```

The PriceCalculator Interface

- **Obtaining and using an instance**
- **The interface definition**

```java
public class LineItem {

    public int getPrice() {
        // where should LineItem go for a PriceCalculator?
        PriceCalculator priceCalculator = product.createPriceCalculator();

        // supply calculatePrice method with data from LineItem
        return priceCalculator.calculatePrice(product, quantity);
    }

    ...
}

public interface PriceCalculator {
   public int calculatePrice(Product product, int quantity);
}
```

The PriceCalculator Interface

- **Obtaining and using an instance**
- **The interface definition**

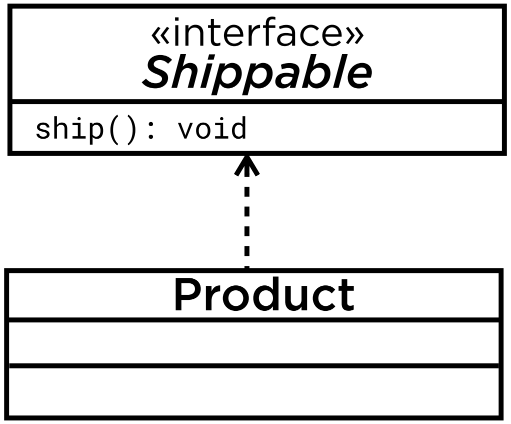
```java
public class LineItem {

    public int getPrice() {
        // where should LineItem go for a PriceCalculator?
        PriceCalculator priceCalculator = product.createPriceCalculator();

        // supply calculatePrice method with data from LineItem
        return priceCalculator.calculatePrice(quantity);
    }

    ...
}

public interface PriceCalculator {
   public int calculatePrice(int quantity);
}
```

The PriceCalculator Interface

- **Obtaining and using an instance**
- **The interface definition**

2. Fulfill order

```java
public interface Shippable {
   public void ship();
}


public abstract class Product implements Comparable<Product> {

      ...

}
```



«interface»
Shippable
ship(): void

Product

The Interface Segregation Principle

Clients should not be forced to depend upon interfaces that they do not use

Cohesion

How strongly the methods and data of a class belong together: interdependence *within* a class.

Coupling

How strongly different classes depend on one another: interdependence *between classes.*

`java.util.List`

Nearly 30 abstract methods

Reasonably cohesive

- `size(), isEmpty()`

Java Collections Framework

- Few interfaces, so easier to learn

Compromises

- `UnsupportedOperationException`

The SOLID Principles of Object-Oriented Design

Single Responsibility Principle

Open-Closed Principle

Liskov Substitution Principle

I

D

The SOLID Principles of Object-Oriented Design

Single Responsibility Principle

Open-Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

D

The SOLID Principles of Object-Oriented Design

Single Responsibility Principle

Open-Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

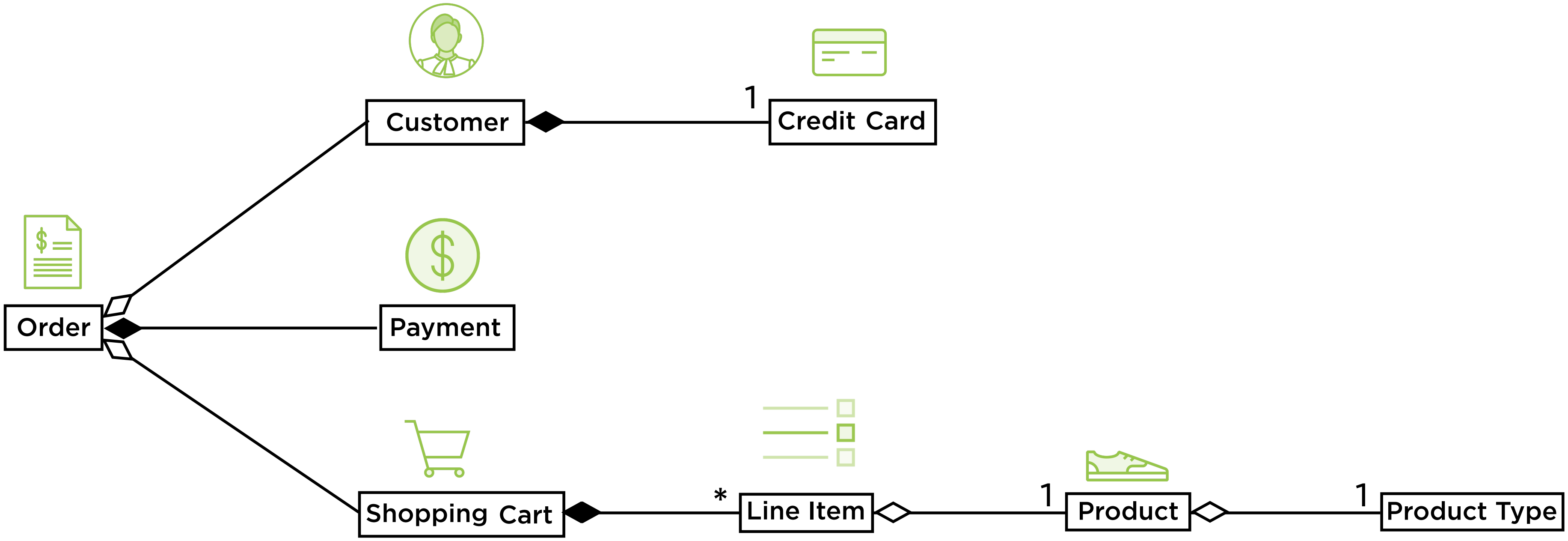
Dependency Inversion Principle

The Dependency Inversion Principle

The title assumes an expectation

That high-level components will depend on low-level ones

The Order Processing System



The Dependency Inversion Principle

The title assumes an expectation

That high-level components will depend on low-level ones

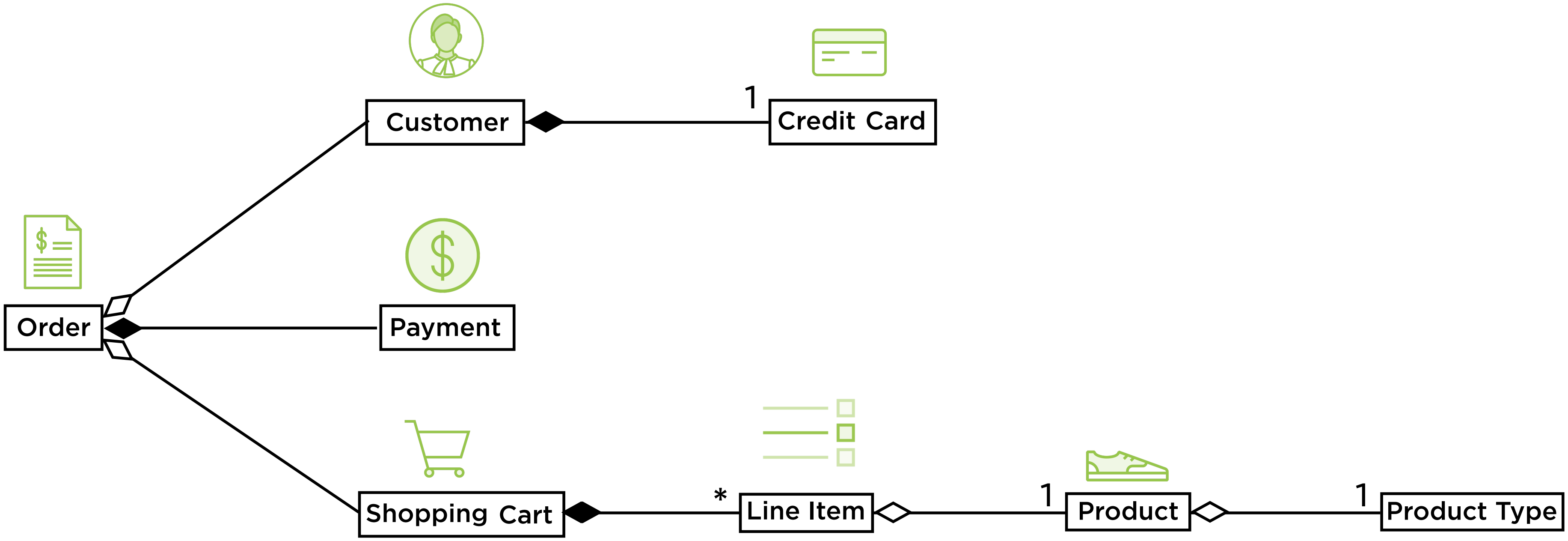
Inverts that expectation

All components should depend on abstractions

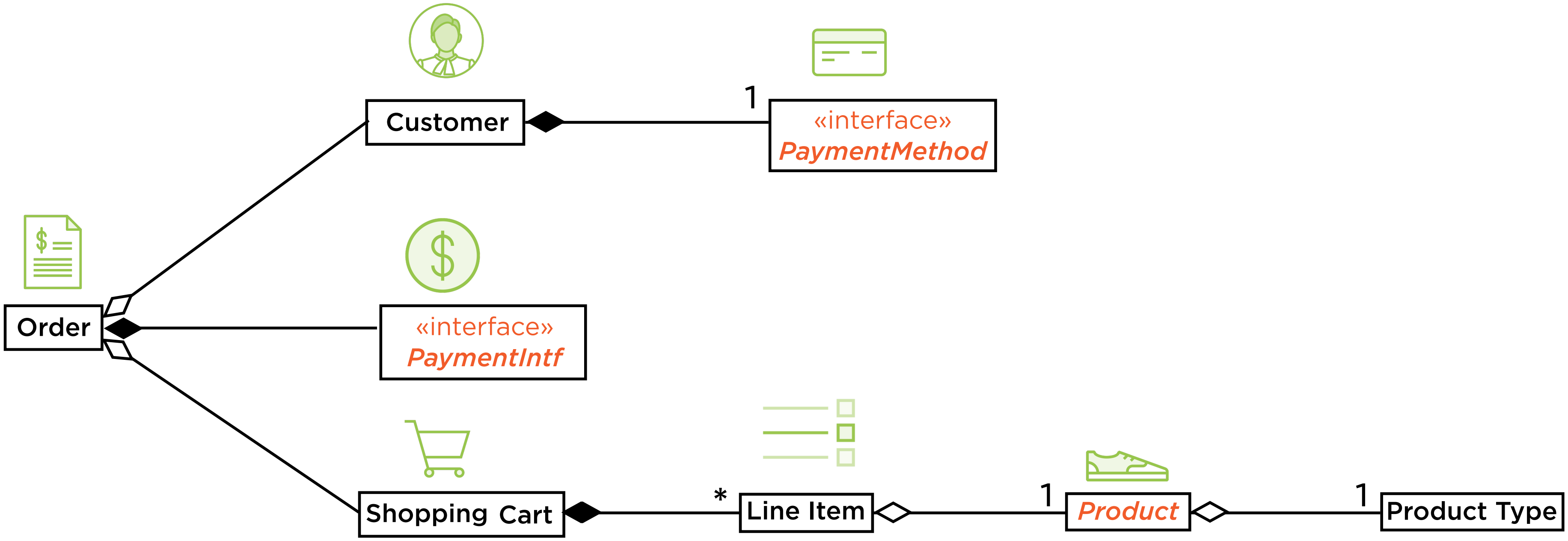
The Dependency Inversion Principle

Depend upon abstractions rather than upon concrete classes

The Order Processing System



The Order Processing System



“Program to an Interface”

Wider meaning to “interface”

Application Program Interface

- Published, stable
- A contract

`Set`, `List`, `Map` **are contracts**

`List myList;` **not** `ArrayList myList;`

Advantages

- Freedom to improve implementation
- Freedom to replace implementation

Summary

Introduction to Interfaces

Demo: Paying through an Interface

Delegation and the Decorator Pattern

The Strategy Pattern

The Interface Segregation Principle

The Dependency Inversion Principle

Principles of the Course

DRY (Don’t Repeat Yourself)

Encapsulate What Varies

The SOLID Principles

The Decorator and Strategy Patterns

High Cohesion, Low Coupling

Program to an Interface

Where Next?

Pluralsight learning path

- Design Patterns in Java

Books

- *Head-First Design Patterns*
- *Effective Java*
- *Applying UML and Patterns*

Practice!