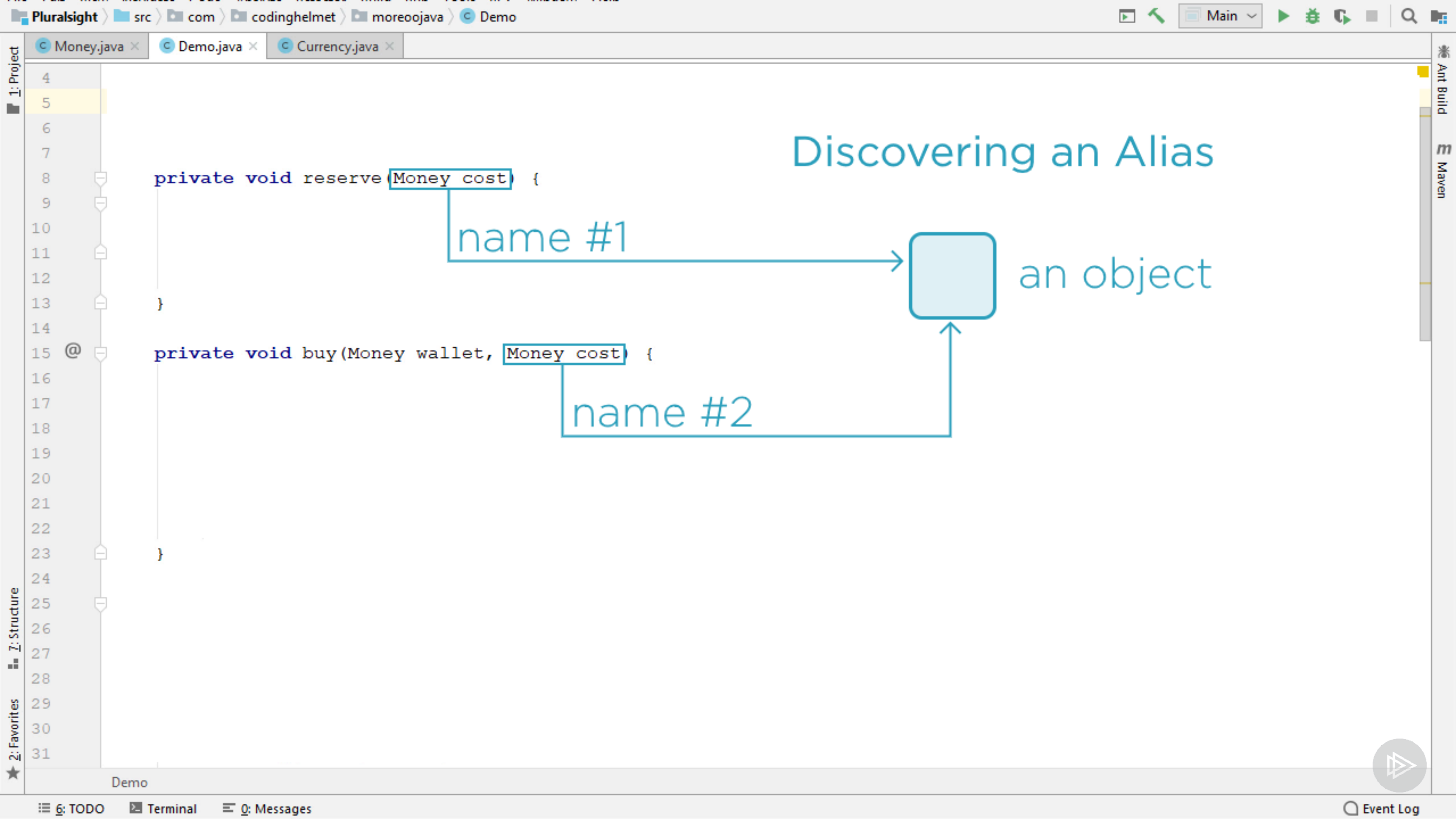
Using Immutable Objects and Value Objects

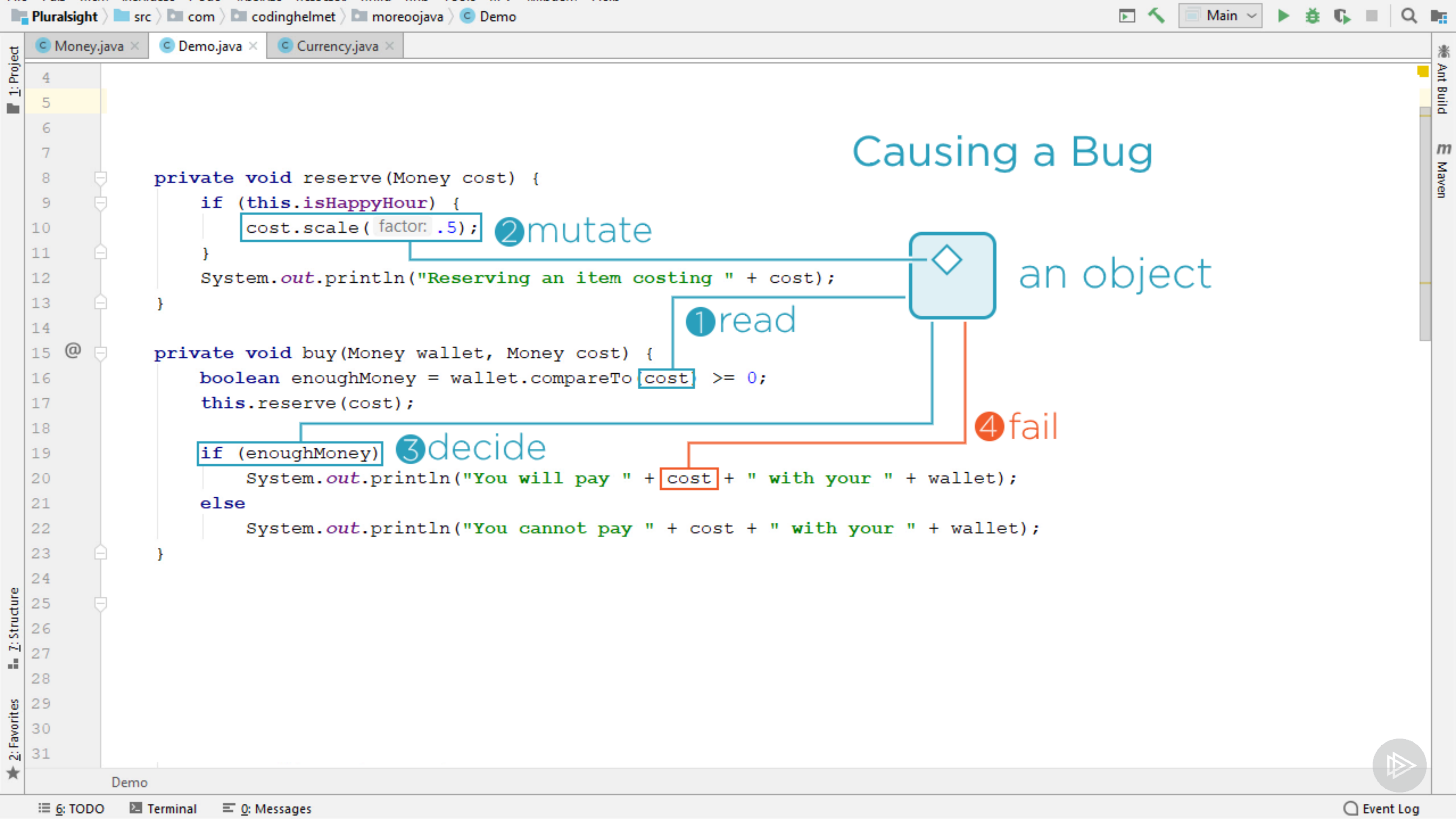
Zoran Horvat
CEO AT CODING HELMET

@zoranh75 http://codinghelmet.com

Discovering an Alias



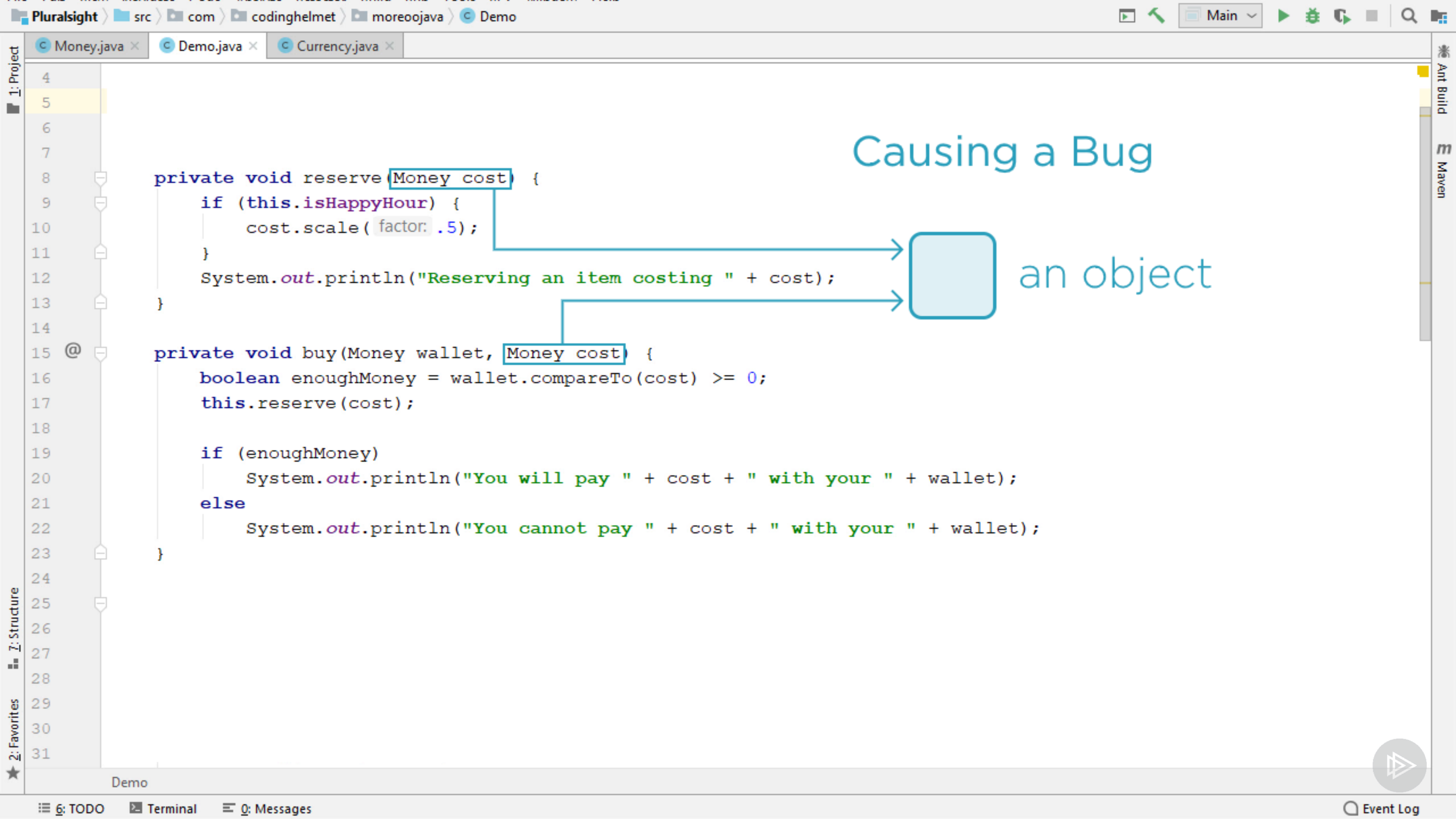
```
private void reserve(Money cost) {

}

private void buy(Money wallet, Money cost) {

}
```



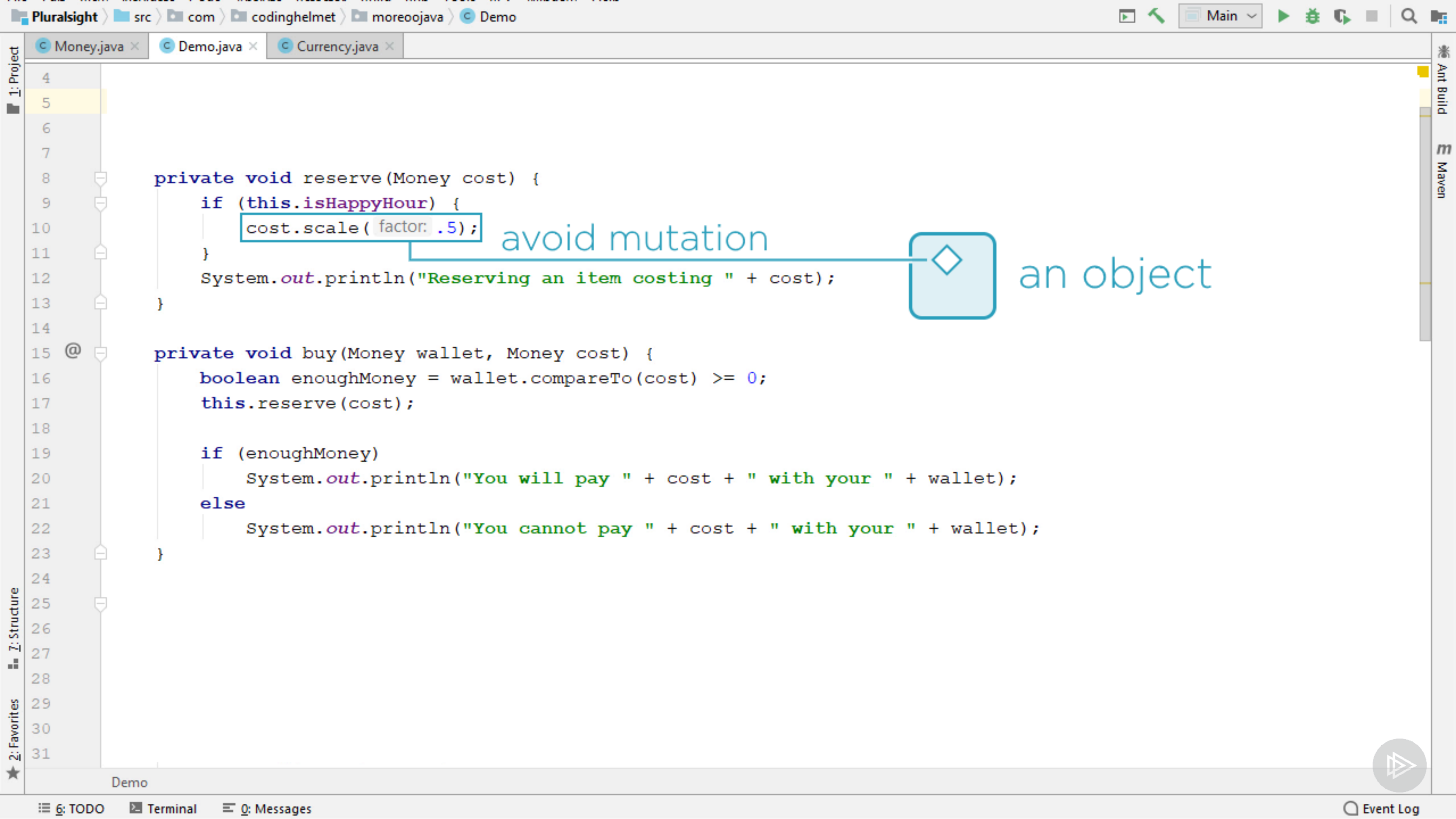
Causing a Bug

```java
private void reserve(Money cost) {
    if (this.isHappyHour) {
        cost.scale( factor: .5);
    }
    System.out.println("Reserving an item costing " + cost);
}

private void buy(Money wallet, Money cost) {
    boolean enoughMoney = wallet.compareTo(cost) >= 0;
    this.reserve(cost);

    if (enoughMoney)
        System.out.println("You will pay " + cost + " with your " + wallet);
    else
        System.out.println("You cannot pay " + cost + " with your " + wallet);
}
```

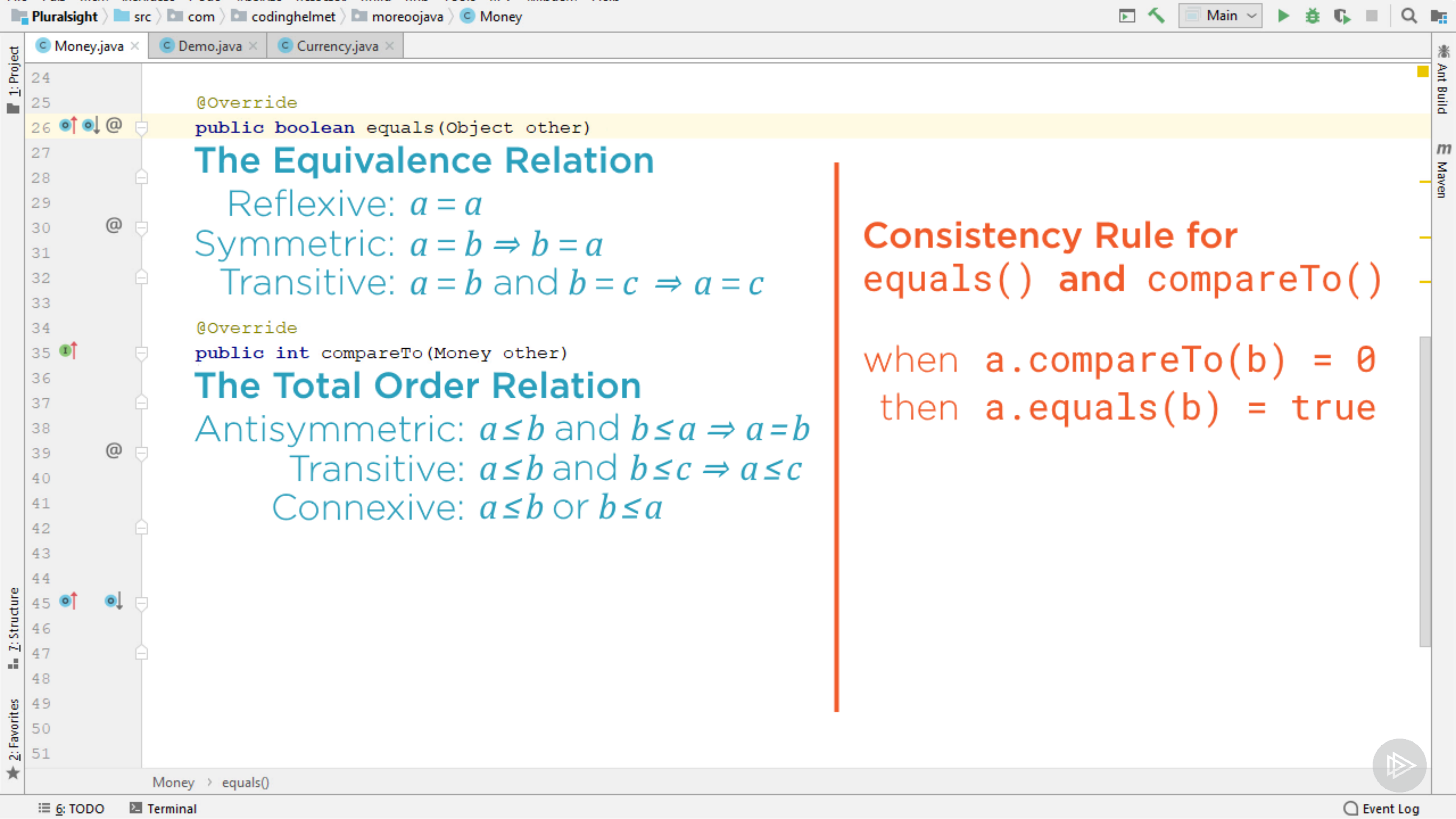
Causing a Bug



```java
    private void reserve(Money cost) {
        if (this.isHappyHour) {
            cost.scale( factor: .5);
        }
        System.out.println("Reserving an item costing " + cost);
    }

    private void buy(Money wallet, Money cost) {
        boolean enoughMoney = wallet.compareTo(cost) >= 0;
        this.reserve(cost);

        if (enoughMoney)
            System.out.println("You will pay " + cost + " with your " + wallet);
        else
            System.out.println("You cannot pay " + cost + " with your " + wallet);
    }
```



```java
    private void reserve(Money cost) {
        if (this.isHappyHour) {
            cost.scale( factor: .5);
        }
        System.out.println("Reserving an item costing " + cost);
    }

    private void buy(Money wallet, Money cost) {
        boolean enoughMoney = wallet.compareTo(cost) >= 0;
        this.reserve(cost);

        if (enoughMoney)
            System.out.println("You will pay " + cost + " with your " + wallet);
        else
            System.out.println("You cannot pay " + cost + " with your " + wallet);
    }
```



```
    @Override
    public boolean equals(Object other)
```

The Equivalence Relation

Reflexive: $a = a$

Symmetric: $a = b \Rightarrow b = a$

Transitive: $a = b$ and $b = c \Rightarrow a = c$

```
    @Override
    public int compareTo(Money other)
```

The Total Order Relation

Antisymmetric: $a \leq b$ and $b \leq a \Rightarrow a = b$

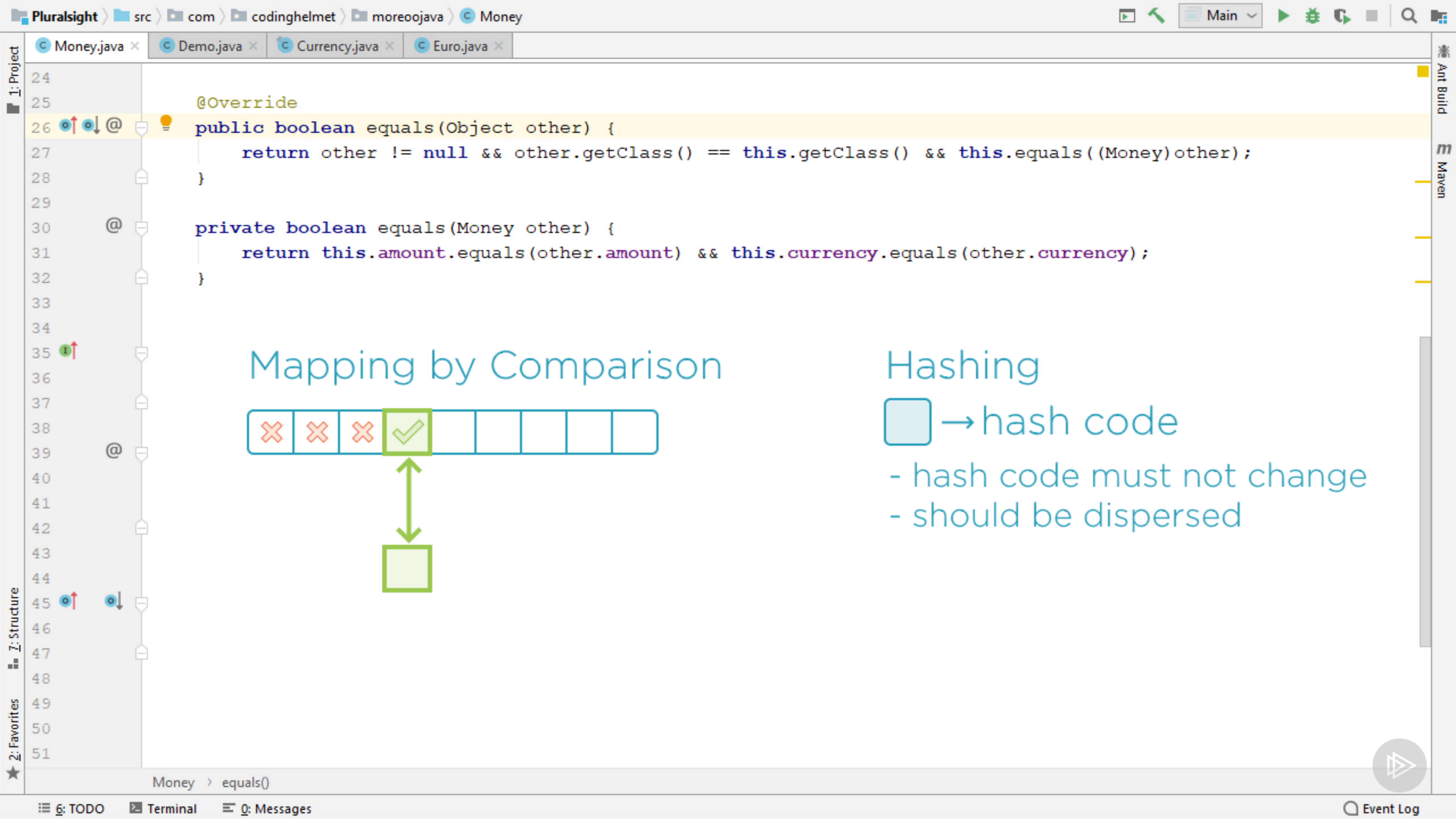
Transitive: $a \leq b$ and $b \leq c \Rightarrow a \leq c$

Connexive: $a \leq b$ or $b \leq a$

Consistency Rule for equals() and compareTo()

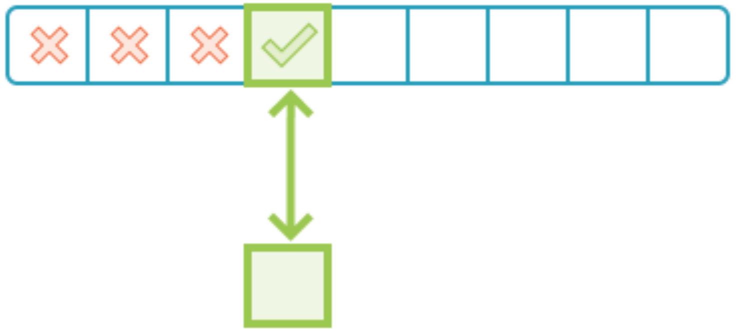
when `a.compareTo(b) = 0`

then `a.equals(b) = true`



```java

    @Override
    public boolean equals(Object other) {
        return other != null && other.getClass() == this.getClass() && this.equals((Money)other);
    }

    private boolean equals(Money other) {
        return this.amount.equals(other.amount) && this.currency.equals(other.currency);
    }
```

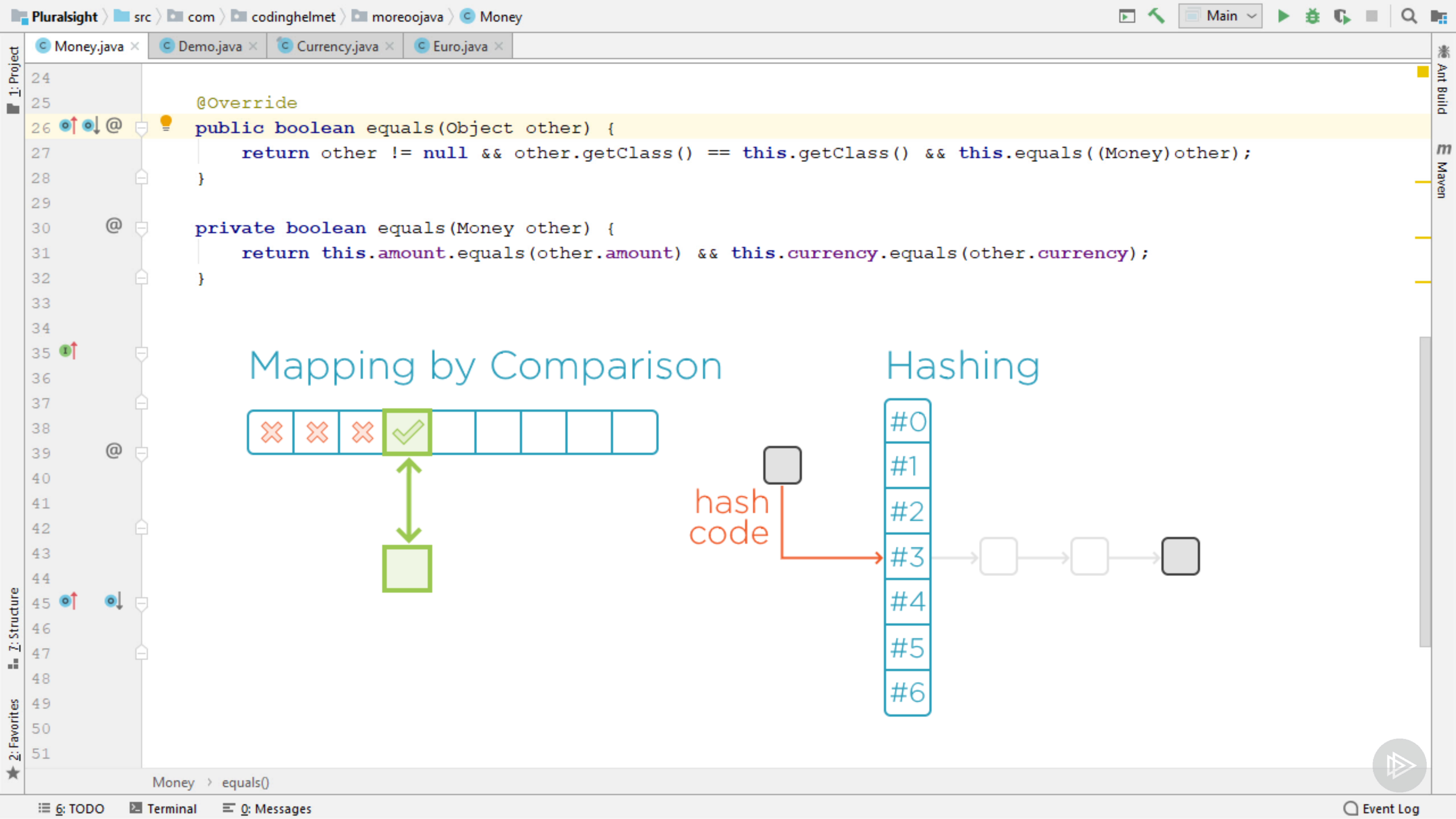
Mapping by Comparison



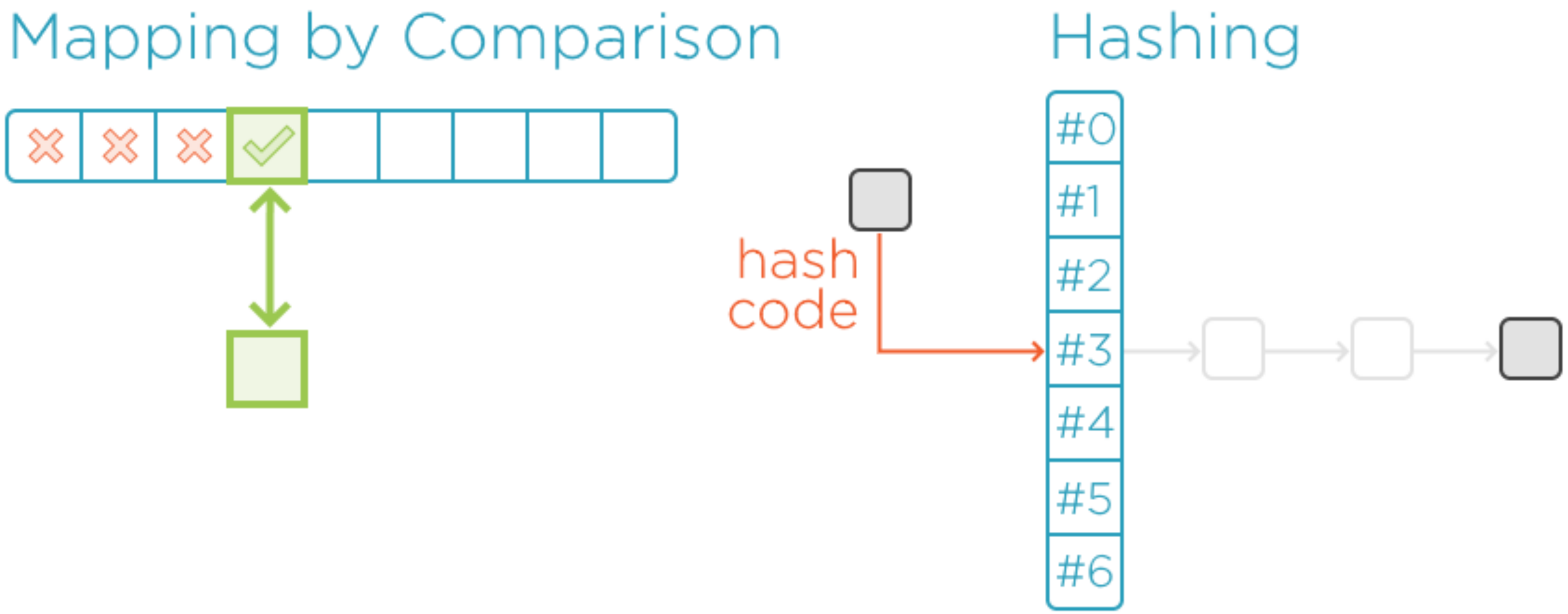
Hashing

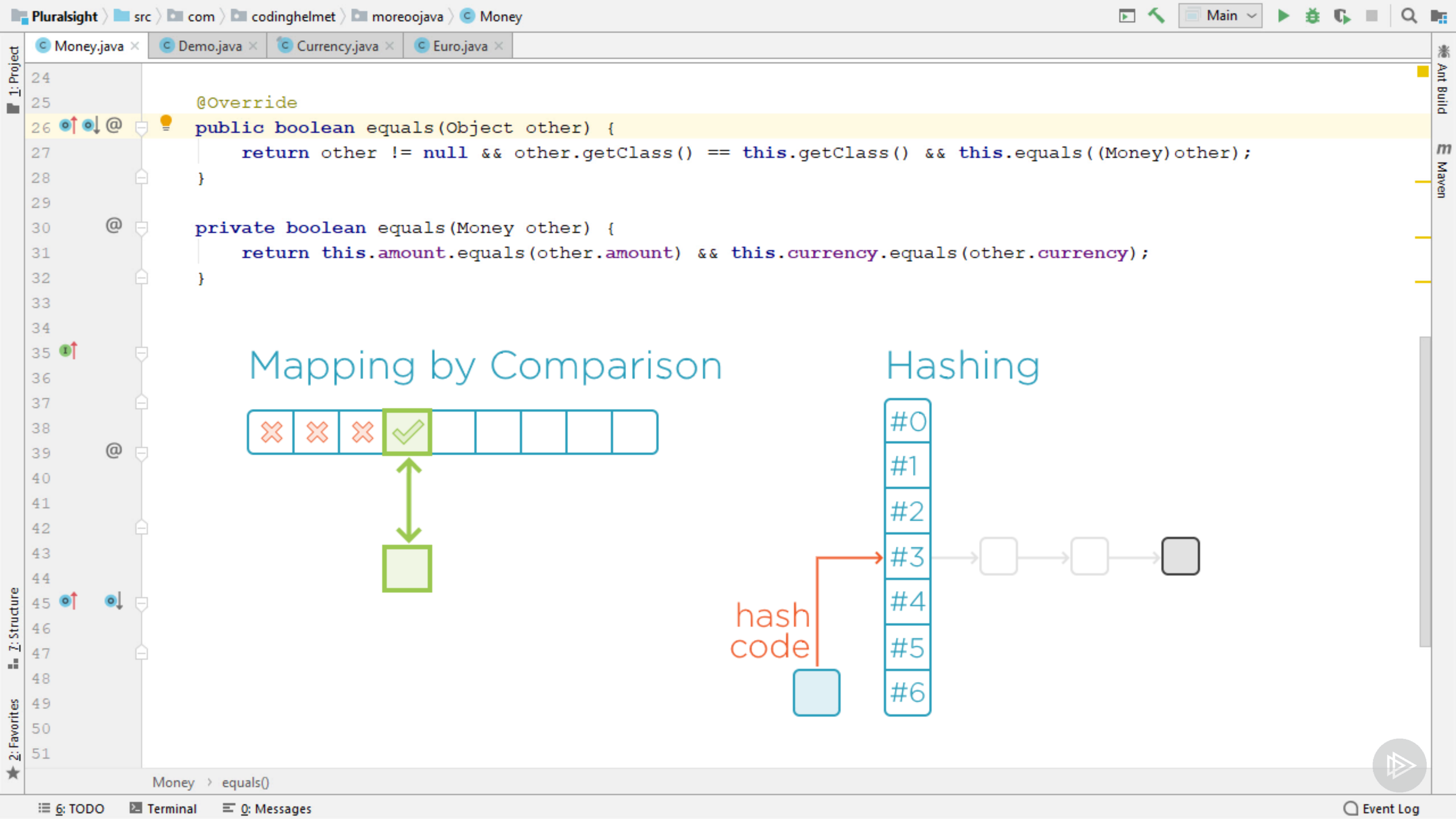
→ hash code

- hash code must not change
- should be dispersed



```java
    @Override
    public boolean equals(Object other) {
        return other != null && other.getClass() == this.getClass() && this.equals((Money)other);
    }

    private boolean equals(Money other) {
        return this.amount.equals(other.amount) && this.currency.equals(other.currency);
    }
```

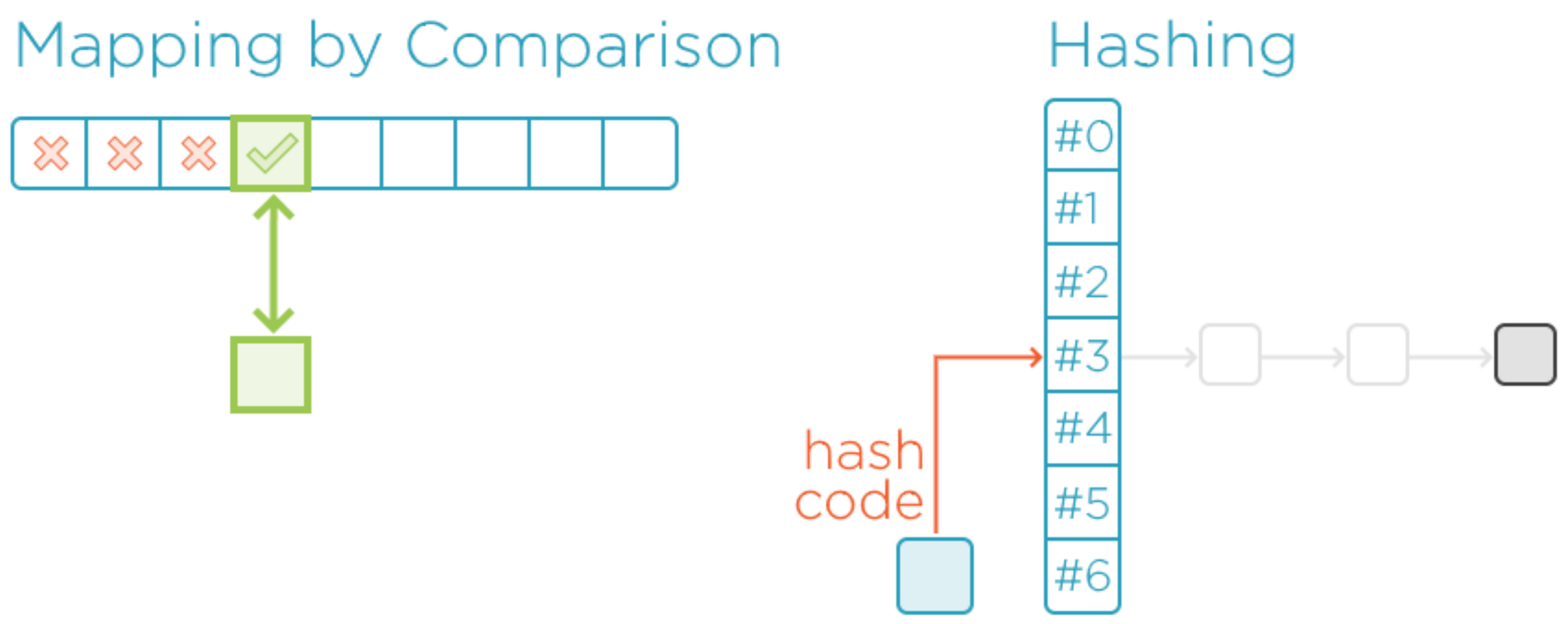




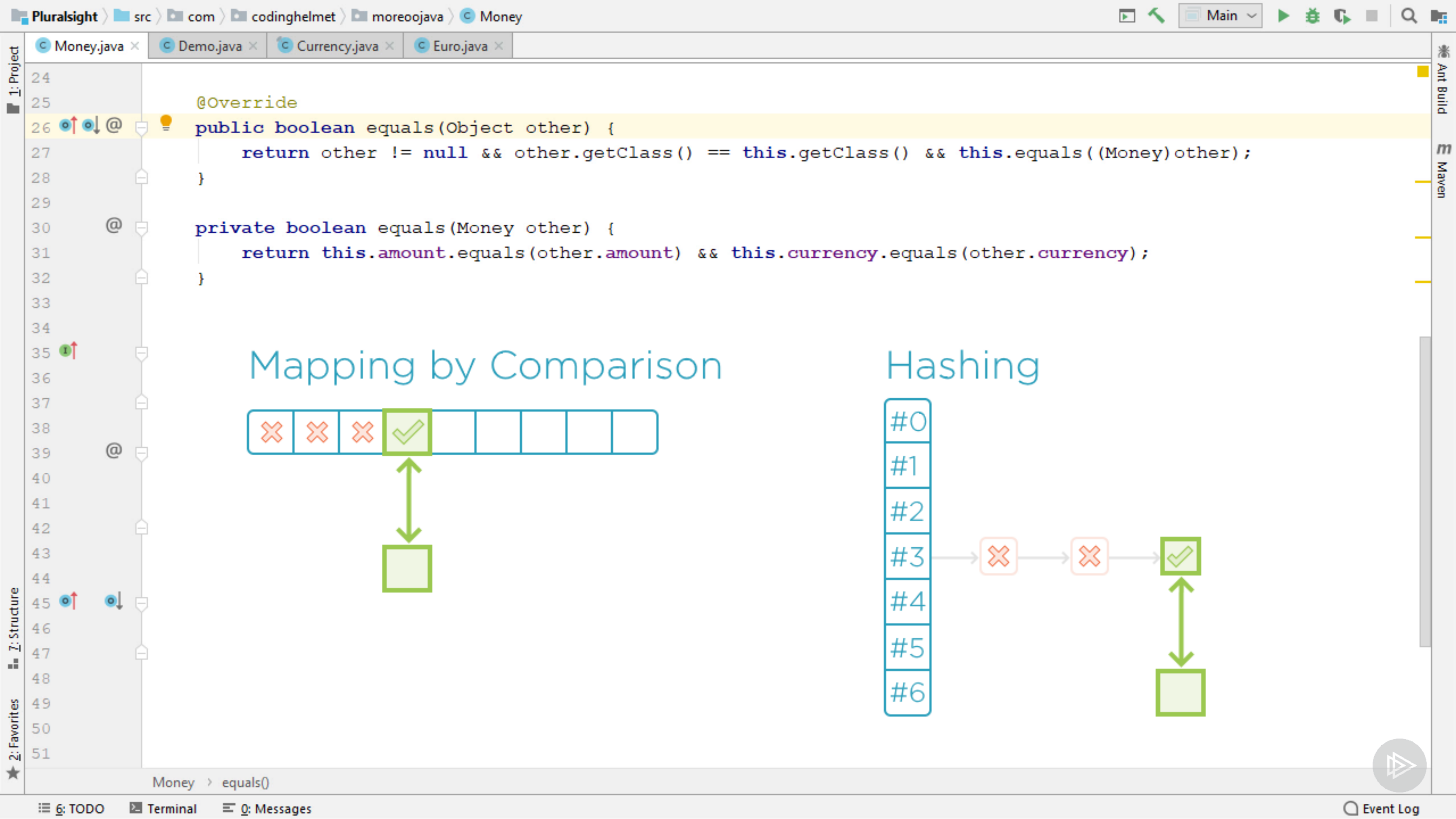
Pluralsight › src › com › codinghelmet › moreoojava › Money

Money.java | Demo.java | Currency.java | Euro.java

```java
    @Override
    public boolean equals(Object other) {
        return other != null && other.getClass() == this.getClass() && this.equals((Money)other);
    }

    private boolean equals(Money other) {
        return this.amount.equals(other.amount) && this.currency.equals(other.currency);
    }
```



Money › equals()

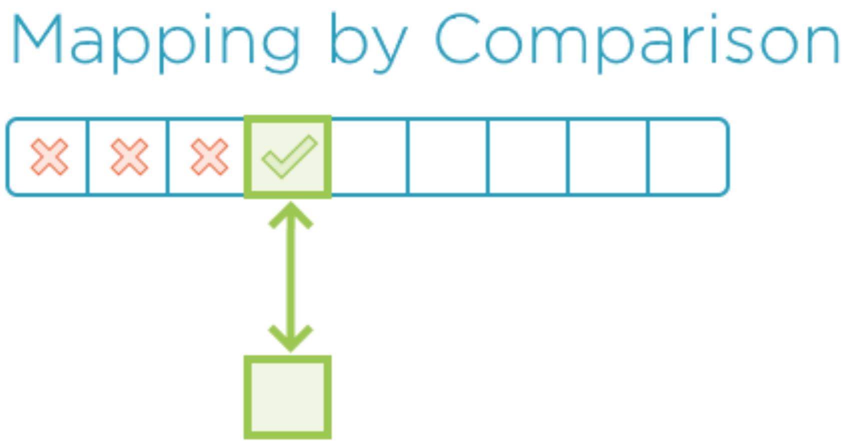


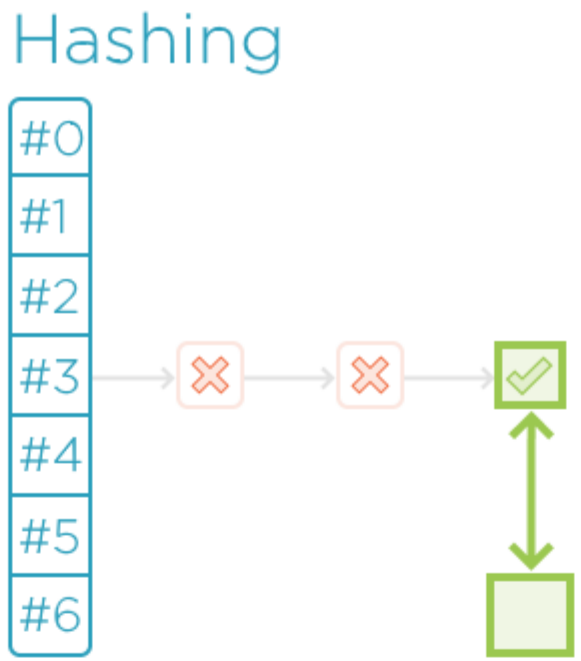
Pluralsight › src › com › codinghelmet › moreoojava › Money

Money.java | Demo.java | Currency.java | Euro.java

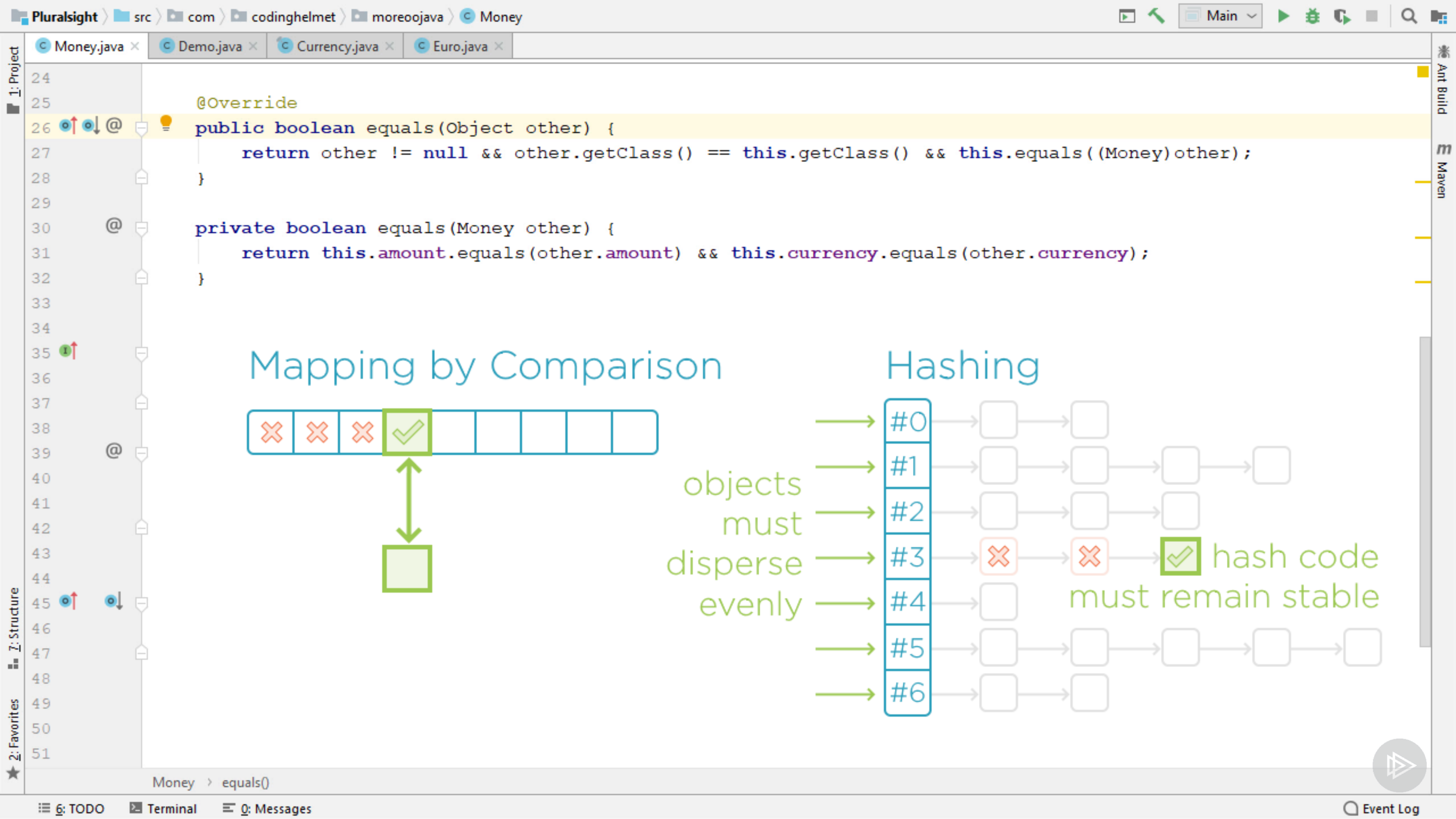
```java
    @Override
    public boolean equals(Object other) {
        return other != null && other.getClass() == this.getClass() && this.equals((Money)other);
    }

    private boolean equals(Money other) {
        return this.amount.equals(other.amount) && this.currency.equals(other.currency);
    }
```

Money › equals()





6: TODO | Terminal | 0: Messages | Event Log



```
    @Override
    public boolean equals(Object other) {
        return other != null && other.getClass() == this.getClass() && this.equals((Money)other);
    }

    private boolean equals(Money other) {
        return this.amount.equals(other.amount) && this.currency.equals(other.currency);
    }
```

Summary

Immutable objects and values

- Immutability is simple to implement
- Saves us from bugs
- Rules out aliasing bugs

Summary

Immutable objects can behave as values

- Value objects behave as plain values
- No different than `int` or a string
- Makes code easy to maintain

Summary

Implementing value-typed semantic

- Value objects must be immutable
- They must override the `equals` method
- `equals` is reflexive, symmetric, transitive
- They must override `hashCode`
- Hash code must be stable and uniform

Summary

Pitfalls of equivalence

- `equals` implements equivalence relation
- Base and derived objects are not equivalent
- Otherwise, they would violate symmetry
- Objects of the same type are equal if their components are equal
- Value object only consists of values

Summary

Next module:
Leveraging Special Case Objects to Remove Null Checks