Wrap all primitives and strings

Object Calisthenics

Colin Dexheimer

19. June 2023

Importance

- Code clarity and maintainability
- Object-oriented principles
- Better design by the use of meaningful abstractions

Benefits

- Code readability and self-documentation
- Easier modifications and updates
- Reusability by abstracting behaviour
- Facilitates design patterns and encapsulation

Disadvantages

- Increased complexity
- Performance overhead

Importance

ooo

– Code clarity and maintainability

– Object-oriented principles

– Better design by the use of meaningful abstractions

Benefits

ooo

– Code readability and self-documentation

– Easier modifications and updates

– Reusability by abstracting behaviour

– Facilitates design patterns and encapsulation

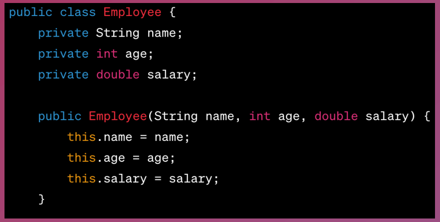
Disadvantages

ooo

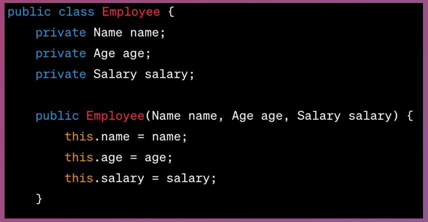
– Increased complexity

– Performance overhead

Importance of the Rule



```java
public class Employee {
    private String name;
    private int age;
    private double salary;

    public Employee(String name, int age, double salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }
```



```java
public class Employee {
    private Name name;
    private Age age;
    private Salary salary;

    public Employee(Name name, Age age, Salary salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }
```

Code clarity and maintainability

Object-oriented principles

Better design by the use of meaningful abstractions

Importance

- Code clarity and maintainability
- Object-oriented principles
- Better design by the use of meaningful abstractions

Benefits

- Code readability and self-documentation
- Easier modifications and updates
- Reusability by abstracting behaviour
- Facilitates design patterns and encapsulation

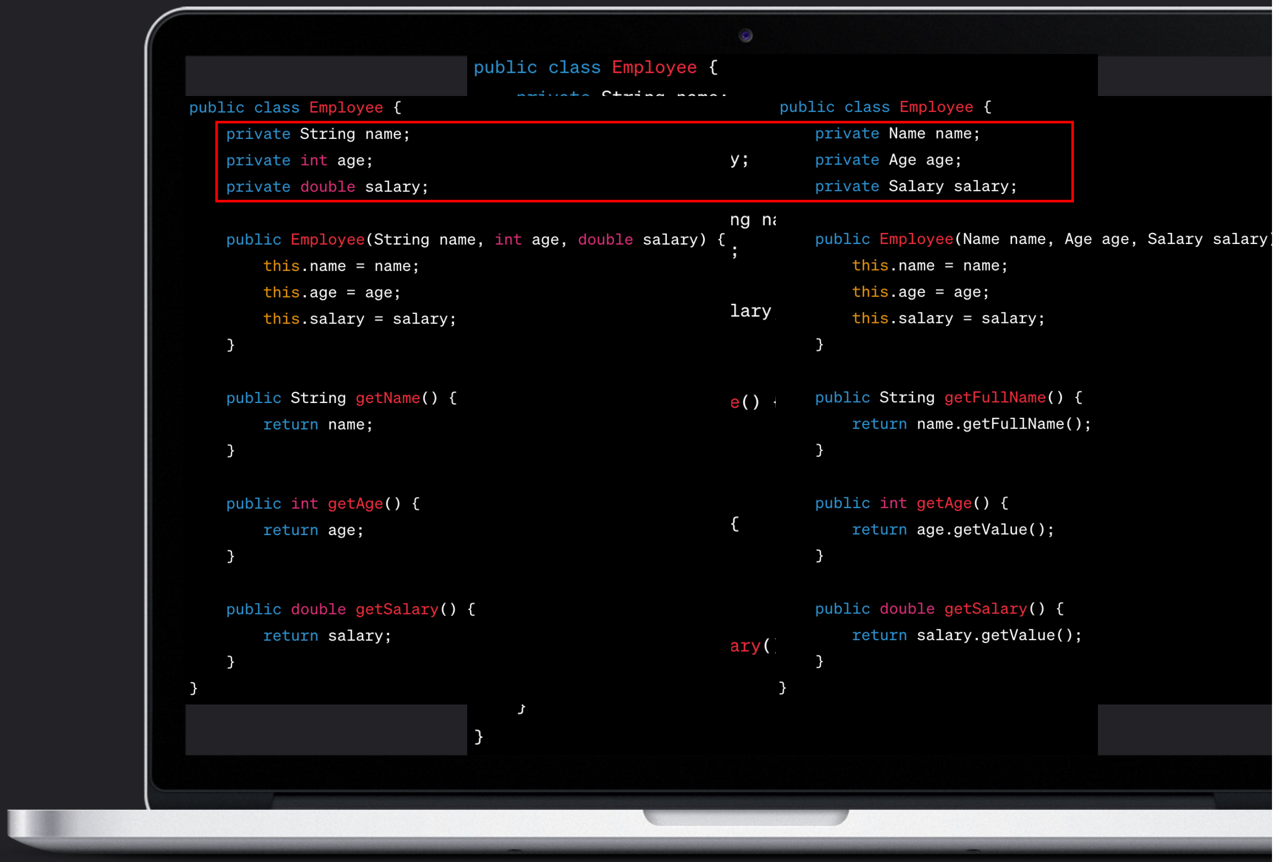
Disadvantages

- Increased complexity
- Performance overhead

Code readability and self-documentation

Readability

Self-documentation

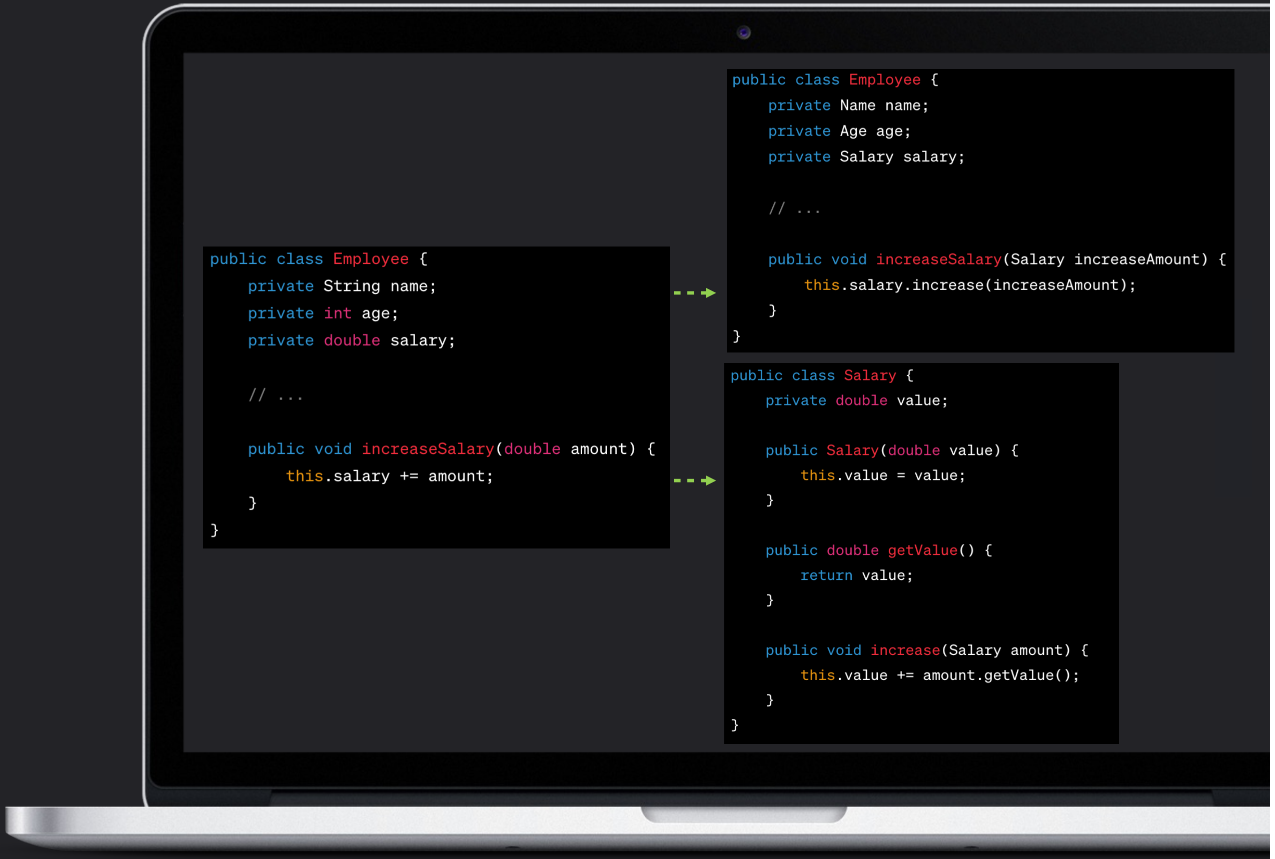


```java
public class Employee {
    private String name;
    private int age;
    private double salary;

    public Employee(String name, int age, double salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public double getSalary() {
        return salary;
    }
}
```

```java
public class Employee {
    private Name name;
    private Age age;
    private Salary salary;

    public Employee(Name name, Age age, Salary salary)
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    public String getFullName() {
        return name.getFullName();
    }

    public int getAge() {
        return age.getValue();
    }

    public double getSalary() {
        return salary.getValue();
    }
}
```

Easier modifications and updates

Modifications and Updates



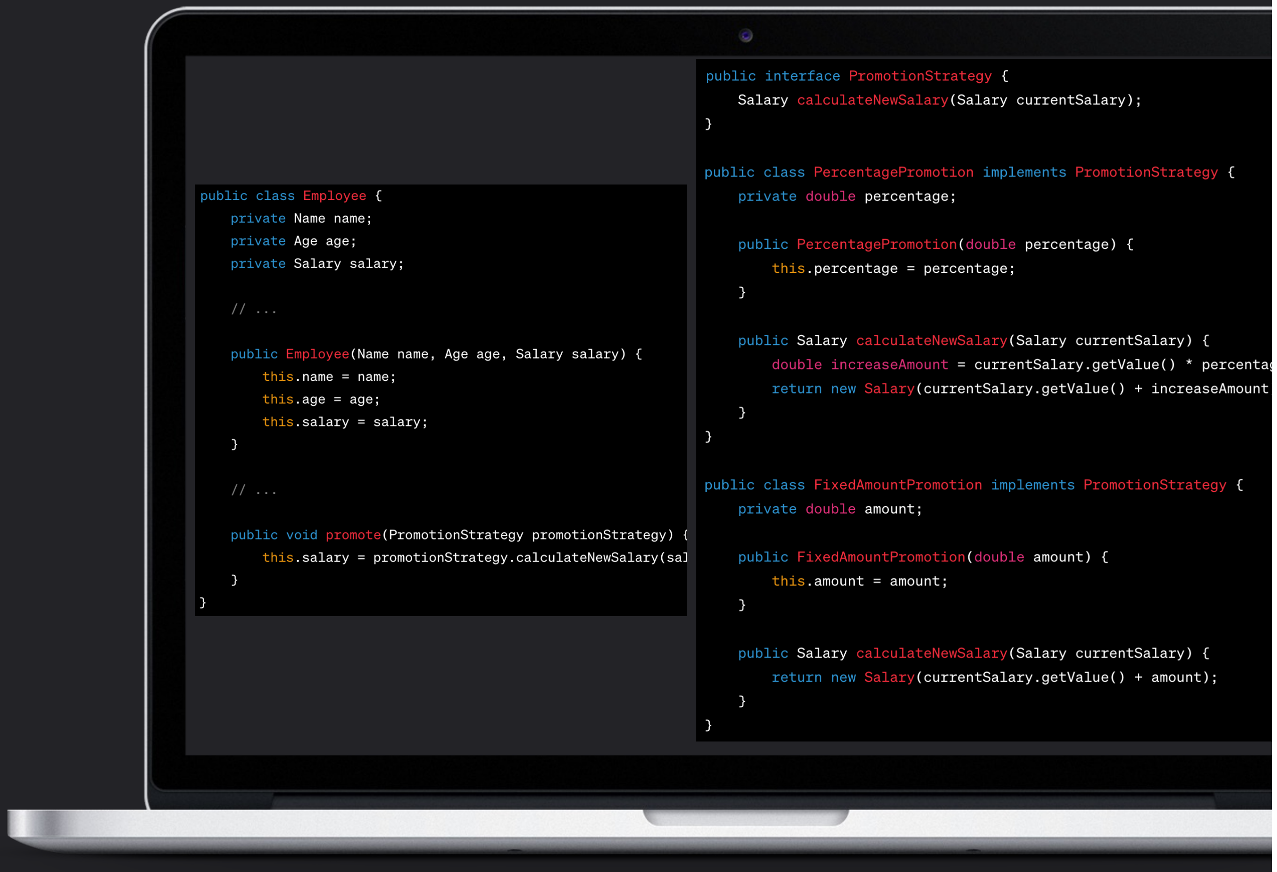
```java
public class Employee {
    private String name;
    private int age;
    private double salary;

    // ...

    public void increaseSalary(double amount) {
        this.salary += amount;
    }
}
```

```java
public class Employee {
    private Name name;
    private Age age;
    private Salary salary;

    // ...

    public void increaseSalary(Salary increaseAmount) {
        this.salary.increase(increaseAmount);
    }
}
```

```java
public class Salary {
    private double value;

    public Salary(double value) {
        this.value = value;
    }

    public double getValue() {
        return value;
    }

    public void increase(Salary amount) {
        this.value += amount.getValue();
    }
}
```

Reusability by abstracting behaviour

Reusability

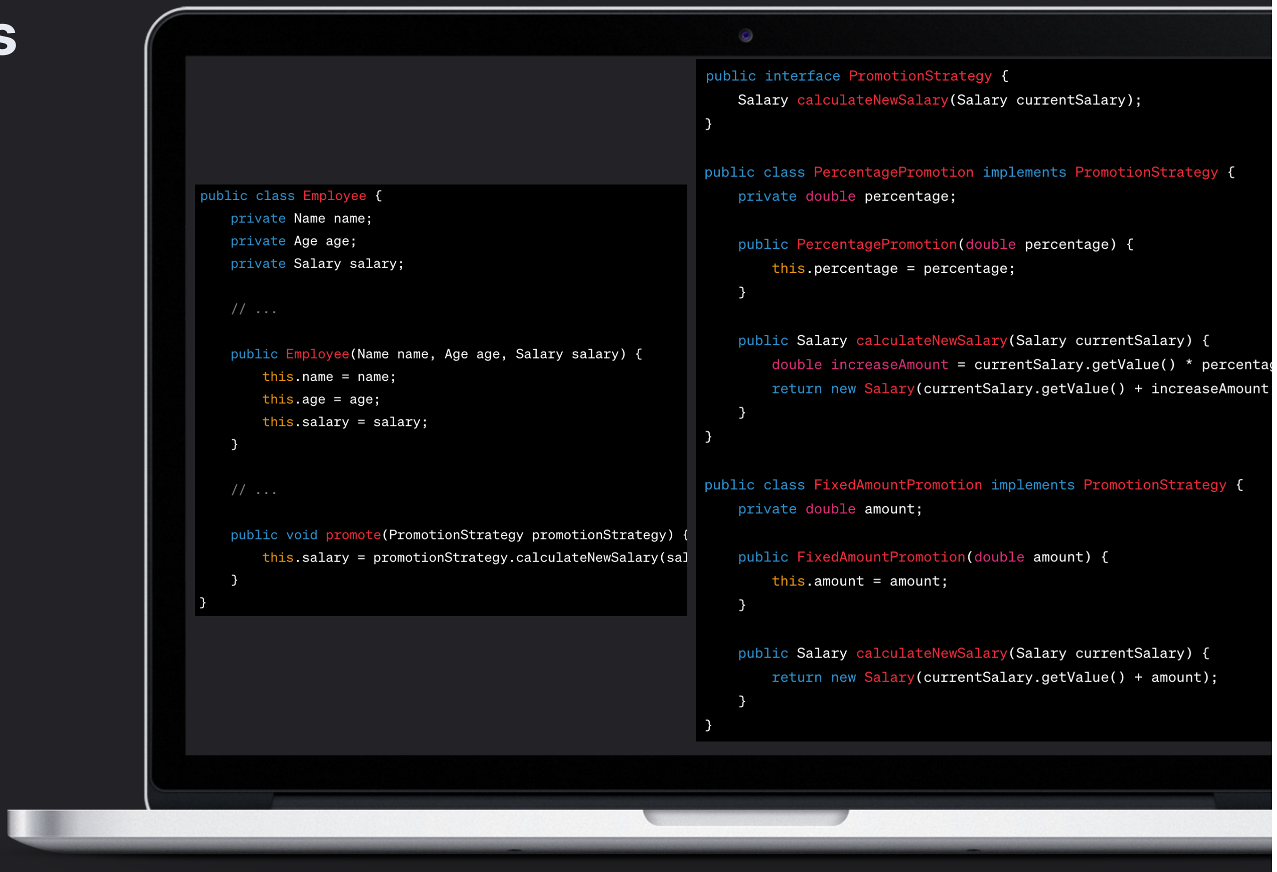


```java
public class Employee {
    private Name name;
    private Age age;
    private Salary salary;

    // ...

    public Employee(Name name, Age age, Salary salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    // ...

    public void promote(PromotionStrategy promotionStrategy) {
        this.salary = promotionStrategy.calculateNewSalary(sal
    }
}
```

```java
public interface PromotionStrategy {
    Salary calculateNewSalary(Salary currentSalary);
}

public class PercentagePromotion implements PromotionStrategy {
    private double percentage;

    public PercentagePromotion(double percentage) {
        this.percentage = percentage;
    }

    public Salary calculateNewSalary(Salary currentSalary) {
        double increaseAmount = currentSalary.getValue() * percenta
        return new Salary(currentSalary.getValue() + increaseAmount
    }
}

public class FixedAmountPromotion implements PromotionStrategy {
    private double amount;

    public FixedAmountPromotion(double amount) {
        this.amount = amount;
    }

    public Salary calculateNewSalary(Salary currentSalary) {
        return new Salary(currentSalary.getValue() + amount);
    }
}
```

Facilitates design patterns and encapsulation

Design patterns and encapsulation



```java
public class Employee {
    private Name name;
    private Age age;
    private Salary salary;

    // ...

    public Employee(Name name, Age age, Salary salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    // ...

    public void promote(PromotionStrategy promotionStrategy) {
        this.salary = promotionStrategy.calculateNewSalary(sal
    }
}
```

```java
public interface PromotionStrategy {
    Salary calculateNewSalary(Salary currentSalary);
}

public class PercentagePromotion implements PromotionStrategy {
    private double percentage;

    public PercentagePromotion(double percentage) {
        this.percentage = percentage;
    }

    public Salary calculateNewSalary(Salary currentSalary) {
        double increaseAmount = currentSalary.getValue() * percenta
        return new Salary(currentSalary.getValue() + increaseAmount
    }
}

public class FixedAmountPromotion implements PromotionStrategy {
    private double amount;

    public FixedAmountPromotion(double amount) {
        this.amount = amount;
    }

    public Salary calculateNewSalary(Salary currentSalary) {
        return new Salary(currentSalary.getValue() + amount);
    }
}
```

Importance

○ ○ ○

– Code clarity and maintainability

– Object-oriented principles

– Better design by the use of meaningful abstractions

Benefits

○ ○ ○

– Code readability and self-documentation

– Easier modifications and updates

– Reusability by abstracting behaviour

– Facilitates design patterns and encapsulation

Disadvantages

○ ○ ○

– Increased complexity fdfadsfasdfadsf

– Performance overhead

Disadvantages of the Rule

Every advantage has its disadvantage

Increased complexity

Performance overhead