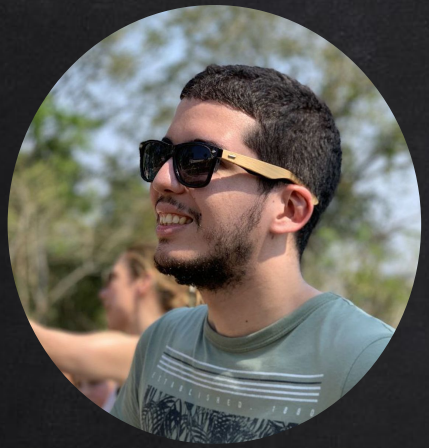
Clean code 101:

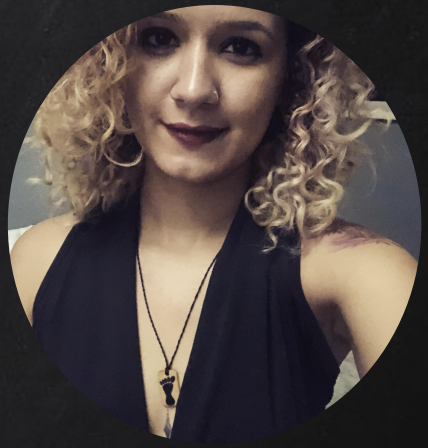
Do caos ao nirvana em poucos passos

Olá Recife!



Diogo Cabral

Consultant Developer @Thoughtworks



Gabrielly Gomes

Consultant Developer @Thoughtworks

ThoughtWorks®

https://www.thoughtworks.com/pt/careers/access

```java
/* A Naive recursive implementation of 0-1 Knapsack problem */
class Knapsack
{

    // A utility function that returns maximum of two integers
     static int max(int a, int b) { return (a > b)? a : b; }

     // Returns the maximum value that can be put in a knapsack of capacity W
     static int knapSack(int W, int wt[], int val[], int n)
     {
        // Base Case
    if (n == 0 || W == 0)
        return 0;

    // If weight of the nth item is more than Knapsack capacity W, then
    // this item cannot be included in the optimal solution
    if (wt[n-1] > W)
       return knapSack(W, wt, val, n-1);

    // Return the maximum of two cases:
    // (1) nth item included
    // (2) not included
    else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
                     knapSack(W, wt, val, n-1)
                      );
      }


   // Driver program to test above function
   public static void main(String args[])
   {
        int val[] = new int[]{60, 100, 120};
        int wt[] = new int[]{10, 20, 30};
    int  W = 50;
    int n = val.length;
    System.out.println(knapSack(W, wt, val, n));
    }
}
/*This code is contributed by Rajat Mishra */
```

E agora?

É possível realizar melhorias consideráveis
com simples (mas poderosos) ajustes!

Nomes Significativos

- ✘ Nomes que digam exatamente o que fazem
- ✘ Que façam parte do contexto do problema
- ✘ Que sejam pronunciáveis
- ✘ Não brinque ou desconte a raiva no seu código!!

```java
/* A Naive recursive implementation of 0-1 Knapsack problem */
class Knapsack
{

    // A utility function that returns maximum of two integers
     static int maximumValueBetweenTwoNumbers(int firstValue, int secondValue) {
       return (firstValue > secondValue)? firstValue : secondValue;
     }

     // Returns the maximum value that can be put in a knapsack of capacity W
     static int getMaximumValueThatCanBeStored(int capacity, int itemsWeight[], int itemValues[], int
quantityOfValues)
     {
        // Base Case
    if (quantityOfValues == 0 || capacity == 0)
        return 0;

    // If weight of the nth item is more than Knapsack capacity W, then
    // this item cannot be included in the optimal solution
    if (itemsWeight[quantityOfValues-1] > capacity)
       return knapSack(capacity, itemsWeight, itemValues, quantityOfValues-1);

    // Return the maximum of two cases:
    // (1) nth item included
    // (2) not included
     else return maximumValueBetweenTwoNumbers(itemValues[n-1] + getMaximumValueThatCanBeStored(capacity
itemsWeight[quantityOfValues-1], itemsWeight, itemValues, quantityOfValues-1),
                       getMaximumValueThatCanBeStored(capacity, itemsWeight, itemValues,
antityOfValues-1)
                     );
     }


 // Driver program to test above function
 public static void main(String args[])
 {
      int itemValues[] = new int[]{60, 100, 120};
      int itemsWeight[] = new int[]{10, 20, 30};
      int capacity = 50;
      int quantityOfValues = val.length;
      System.out.println(getMaximumValueThatCanBeStored(capacity, itemsWeight, itemValues,
ntityOfValues));
 }

/* This code is contributed by Rajat Mishra */
```

Comentários

- ✘ Seu código está legível?
- ✘ Comentários são ruins. Ponto.
- ✘ Se sentir a necessidade de comentar, pense antes em refatorar
- ✘ Comentários são úteis quando algo que foi feito não pode ficar no código por muito tempo, ou quando algo deve ser feito com urgência
- ✘ Não deixe códigos comentados
- ✘ Evite fazer comentários //TO DO, provavelmente você nunca mais vai lembrar porque ele está lá!! Mas, se fizer, descreva o que deve feito

```java
/*This code is contributed by Rajat Mishra */
class Knapsack
{

     static int maximumValueBetweenTwoNumbers(int firstValue, int secondValue) {
       return (firstValue > secondValue)? firstValue : secondValue;
     }

     static int getMaximumValueThatCanBeStored(int weigthCapacity, int itemsWeight[], int itemValues[],
int quantityOfItems)
     {
        if (quantityOfValues == 0 || weigthCapacity == 0)
            return 0;

        if (itemsWeight[quantityOfItems-1] > weigthCapacity)
            return getMaximumValueThatCanBeStored(weigthCapacity, itemsWeight, itemValues,
quantityOfItems-1);

        else {
          int includedValue = getMaximumValueThatCanBeStored(weigthCapacity -
itemsWeight[quantityOfItems-1], itemsWeight, itemValues, quantityOfItems-1);
          int valueAfterIncludeItem = itemValues[quantityOfItems-1] + includedValue;

          int lastValueStored = getMaximumValueThatCanBeStored(weigthCapacity, itemsWeight, itemValues,
quantityOfItems-1);

          return maximumValueBetweenTwoNumbers(valueAfterIncludeItem, lastValueStored);
        }


   public static void main(String args[])
   {
        int itemValues[] = new int[]{60, 100, 120};
        int itemsWeight[] = new int[]{10, 20, 30};
        int weigthCapacity = 50;
        int quantityOfItems = itemValues.length;
        System.out.println(getMaximumValueThatCanBeStored(weigthCapacity, itemsWeight, itemValues,
quantityOfItems));
    }
}
```

```java
class DifferentWaysToAddParentesis
{
    public List<Integer> diffWaysToCompute(String input) {
        List<Integer> result = new ArrayList<>();

        if(input == null || input.length() == 0) {
            return result;
        }

        for(int i = 0; i < input.length(); i++) {
            if(input.charAt(i) == '+' || input.charAt(i) == '*' || input.charAt(i) == '-') {
                List<Integer> left = diffWaysToCompute(input.substring(0, i));
                List<Integer> right = diffWaysToCompute(input.substring(i + 1));

                for (int firstNumber : left) {
                    for (int secondNumber : right) {
                        int value = 0;
                        if(input.charAt(i) == '+') value = firstNumber + secondNumber;
                        else if(input.charAt(i) == '-') value = firstNumber - secondNumber;
                        else if(input.charAt(i) == '*') value = firstNumber * secondNumber;
                        result.add(value);
                    }
                }
            }
        }

        if (result.isEmpty()) {
            result.add(Integer.parseInt(input));
        }

        return result;
    }
```

Funções

- ✘ Tenha responsabilidade única
- ✘ O contrato da sua função deve contar o que faz
- ✘ Tenha a menor quantidade de parâmetros possível
- ✘ Separe comando e consulta

```java
class DifferentWaysToAddParentesis

    public List<Integer> diffWaysToCompute(String mathExpression) {
        List<Integer> result = new ArrayList<>();

        if(mathExpression == null || mathExpression.length() == 0) {
            return result;
        }

        for(int i = 0; i < mathExpression.length(); i++) {
            if(isAnOperator(mathExpression.charAt(i))) {
                List<Integer> left = diffWaysToCompute(mathExpression.substring(0, i));
                List<Integer> right = diffWaysToCompute(mathExpression.substring(i + 1));

                for (int firstNumber : left) {
                    for (int secondNumber : right) {
                        int value = calculate(firstNumber, secondNumber, mathExpression.charAt(i));
                        result.add(value);
                    }
                }
            }
        }

        if (result.isEmpty()) {
            result.add(Integer.parseInt(mathExpression));
        }

        return result;
    }

    private boolean isAnOperator(char c) {
        if(c == '+' || c == '*' || c == '-')
            return true;
        return false;
    }

    private int calculate(int firstNumber, int secondNumber, char operator) {
        if(operator == '+') return firstNumber + secondNumber;
        else if(operator == '-') return firstNumber - secondNumber;
        else if(operator == '*') return firstNumber * secondNumber;
        return 0;
    }
}
```

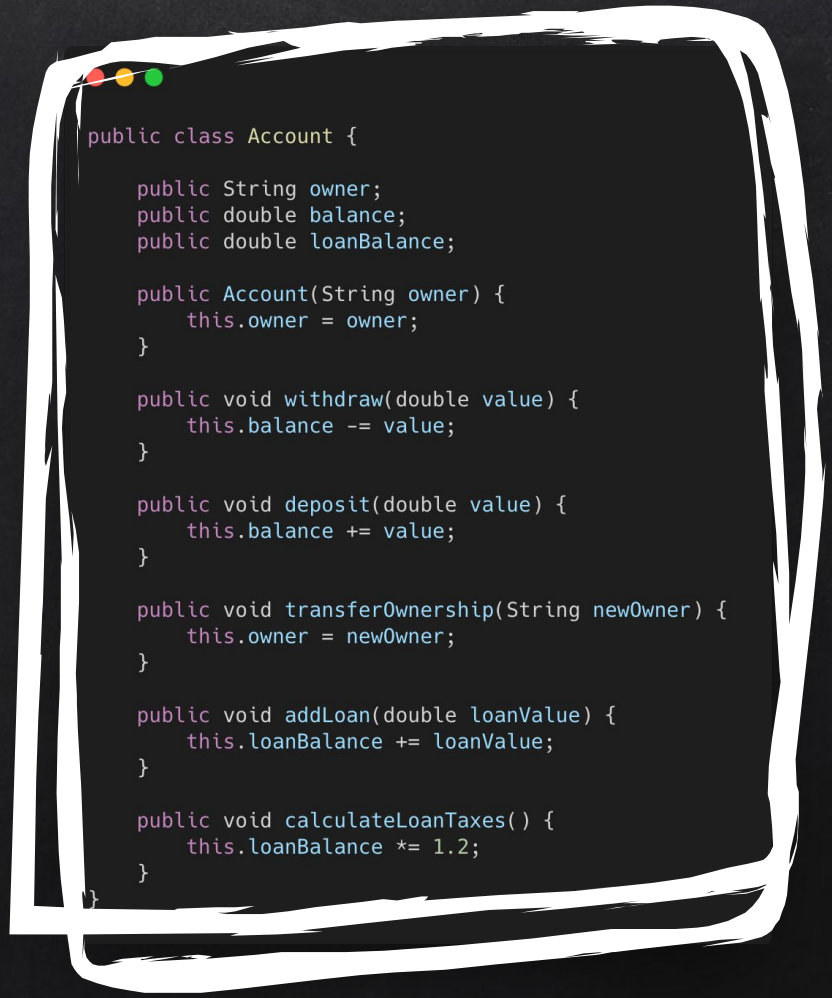
```
public class Account { public String owner;public double balance;public double loanBalance;
    public Account(String owner) { this.owner = owner; }
    public void withdraw(double value) { this.balance -= value; }
    public void deposit(double value) { this.balance += value; }
    public void transferOwnership(String newOwner) { this.owner = newOwner; }
    public void addLoan(double loanValue) { this.loanBalance += loanValue; }
    public void calculateLoanTaxes( ) { this.loanBalance *= 1.2; }
```

Já está tratado!?

Existem diferentes tipos de code smells
além dos citados anteriormente!

Formatação

- ✘ Indentação padronizada facilita a leitura do código como um todo
- ✘ Tenha em mente os limites verticais e horizontais
- ✘ Utilização de linters como ferramenta de auxílio



```java
public class Account {

    public String owner;
    public double balance;
    public double loanBalance;

    public Account(String owner) {
        this.owner = owner;
    }

    public void withdraw(double value) {
        this.balance -= value;
    }

    public void deposit(double value) {
        this.balance += value;
    }

    public void transferOwnership(String newOwner) {
        this.owner = newOwner;
    }

    public void addLoan(double loanValue) {
        this.loanBalance += loanValue;
    }

    public void calculateLoanTaxes() {
        this.loanBalance *= 1.2;
    }
}
```

Classes

- ✘ Manter seus atributos bem encapsulados*
- ✘ Seguir o princípio da responsabilidade única
- ✘ Funções internas da classe podem necessitar ser refatoradas, e isso pode gerar novas classes!

```java
public class Account {

    private String owner;
    private double balance;

    public Account(String owner) {
        this.owner = owner;
    }

    public void withdraw(double value) {
        this.balance -= value;
    }

    public void deposit(double value) {
        this.balance += value;
    }

    public void transferOwnership(String newOwner) {
        this.owner = newOwner;
    }

    public String getOwner() {
        return owner;
    }

    public void setOwner(String owner) {
        this.owner = owner;
    }

    public double getBalance() {
        return balance;
    }

    public void setBalance(double balance) {
        this.balance = balance;
    }
```

```java
public class Bank {

    private List<Account> accounts;

    public void addLoan(int accountId, double loanValue) {
        Account currentAccount = this.accounts.get(accountId);
        double newBalance = currentAccount.getBalance() + loanValue;
        currentAccount.setBalance(newBalance);
    }

    public void calculateLoanTaxes(int accountId) {
        Account currentAccount = this.accounts.get(accountId);
        double newBalance = currentAccount.getBalance() * 0.8;
        currentAccount.setBalance(newBalance);
    }
}
```

Tratamento de erros

- ✘ Use exceções
- ✘ Forneça exceções com contexto
- ✘ Retorne uma exceção ao invés de *null*

O código já está ok?

Dicas gerais de como sempre continuar melhorando seu código!

Reuso de código

- ✘ Refatorações podem gerar códigos similares, faça bom proveito disso!
- ✘ Quanto menos linhas, menor sua aplicação
- ✘ Alterações em códigos reutilizados tornam mais simples a manutenção do sistema
- ✘ A boa utilização dos conceitos anteriores deixa fácil para demais pessoas da sua equipe encontrarem códigos reutilizáveis

```
public String generateExpirationDateMessage(Date expirationDate) {
    SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd");
    return "The expiration date is" + format.format(expirationDate);
}

public boolean validateExpirationDate(Date expirationDate) {
    SimpleDateFormat format = new SimpleDateFormat("dd-MM-yyyy");
    String parsedDate = format.format(expirationDate);
    return parsedDate.contains("2019");
}
```

```java
public String generateExpirationDateMessage(Date expirationDate) {
    return "The expiration date is" + formatDateToString(expirationDate, "yyyy-MM-dd");
}

public boolean validateExpirationDate(Date expirationDate) {
    String parsedDate = formatDateToString(expirationDate, "dd-MM-yyyy");
    return parsedDate.contains("2019");
}

private String formatDateToString(Date date, String datePattern) {
    SimpleDateFormat format = new SimpleDateFormat(datePattern);
    return format.format(date);
}
```

Testes

- ✘ Triple A: Arrange, Act & Assert
- ✘ Tenha um único conceito por teste
- ✘ Aproveite bem a etapa de refatoração

```java
@Test
    public void shouldGoUpAfterHittingTheBottom() throws Exception {
        int theBottomEdge = BallWorld.BOX_HEIGHT - Ball.DEFAULT_RADIUS;
        Ball bouncingBall = BallFactory.bouncingBall(0, theBottomEdge, Bouncing.DOWN);

        bouncingBall.update();

        assertCenterYCoordinateIs(oneStepUpFrom(theBottomEdge), bouncingBall);
    }

    @Test
    public void shouldGoUp() throws Exception {
        Ball bouncingBall = BallFactory.bouncingBall(0, 100, Bouncing.UP);

        bouncingBall.update();

        assertCenterYCoordinateIs(oneStepUpFrom(100), bouncingBall);
    }
```

- ✘ Single Responsibility Principle
- ✘ Open-Close Principle
- ✘ Liskov Substitution Principle
- ✘ Interface Segregation Principle
- ✘ Dependency Inversion Principle

Um código limpo é simples e direto. Ele é tão bem legível quanto uma prosa bem escrita. Ele jamais torna confuso o objetivo do desenvolvedor, em vez disso, ele está repleto de abstrações claras e linhas de controle objetivas.

Grady Booch

Referências

- ✘ Livro “Código Limpo” por Robert C. Martin
- ✘ https://sourcemaking.com/refactoring/smells

Obrigado!

Dúvidas?

Linkedin: Gabrielly Gomes
gabrielly.a.gomes@gmail.com

Linkedin: Diogo Cabral
diogo.cabral.dev@gmail.com