

---

# Bài 8

# Clean Code

Module: ADVANCED PROGRAMMING WITH JAVA

---

# Kiểm tra bài trước

Hỏi và trao đổi về các khó khăn gặp phải trong bài "Interface và Abstract Class"  
Tóm tắt lại các phần đã học từ bài "Interface và Abstract Class"

- Trình bày được Clean Code
- Trình bày được các tiêu chí cốt lõi của Clean Code
- Nhận diện được các mã nguồn bản cơ bản
- Thực hiện được kỹ thuật đổi tên biến
- Thực hiện được kỹ thuật đổi tên phương thức
- Thực hiện được kỹ thuật tách biến
- Thực hiện được kỹ thuật tách hằng
- Thực hiện được kỹ thuật tách phương thức



---

# Thảo luận

Clean Code

Các yếu tố ảnh hưởng đến chất lượng mã nguồn

Đặt tên tốt

# Clean Code – Mã sạch

---



- Clean Code (Mã sạch) là thuật ngữ để chỉ đến những mã nguồn “tốt”
- Các đặc điểm của clean code:
  - Đơn giản
  - Trực tiếp
  - Dễ đọc
  - Dễ cải tiến
  - Có unit test và acceptance test
  - Các định danh đều thể hiện rõ nghĩa
  - Có ít sự phụ thuộc
  - Không có mã bị trùng lặp (duplicate)
  - Thể hiện được ý tưởng của thiết kế

# Code Smell – Mã bẩn

---

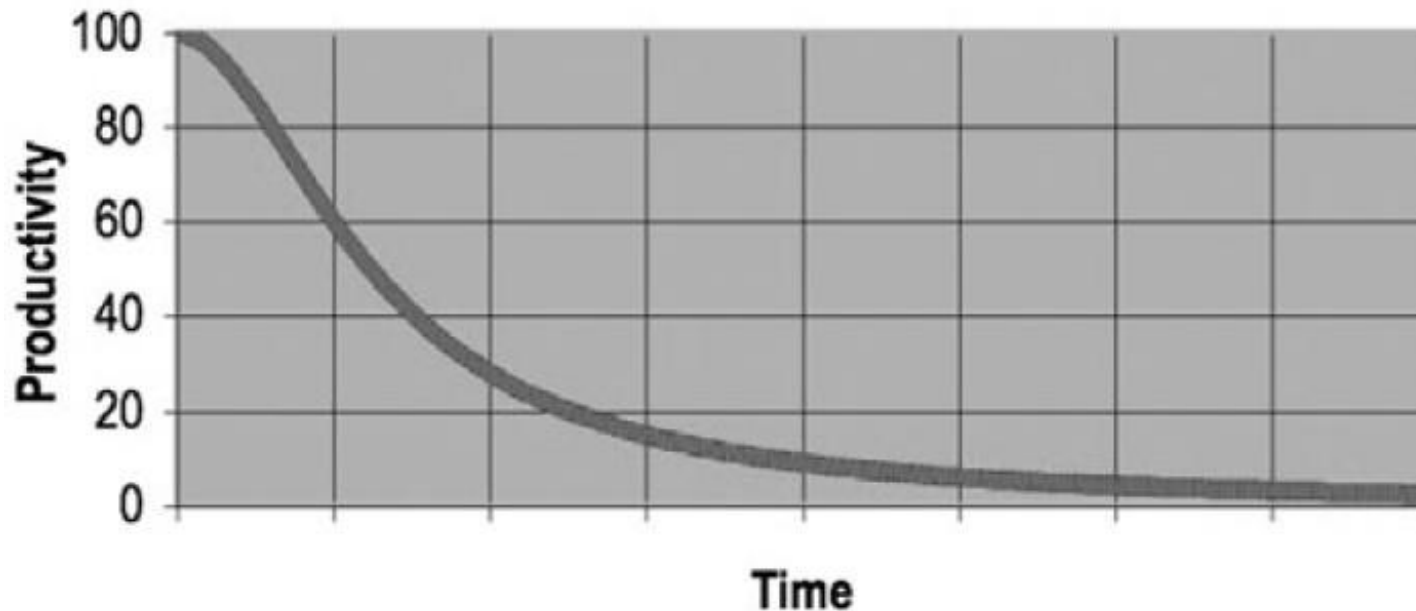


- Code Smell là thuật ngữ để chỉ đến những trường hợp mã nguồn gây khó khăn cho việc đọc, duy trì và mở rộng
- Một số trường hợp thông dụng của mã bẩn:
  - Đặt tên không tốt
  - Phương thức quá dài
  - Lớp quá dài
  - Phương thức xử lý quá nhiều việc
  - Phương thức có quá nhiều tham số
  - Lạm dụng quá nhiều ghi chú (comment) trong mã nguồn
  - Mã nguồn bị trùng lặp
  - Sử dụng các giá trị magic

# Tại sao cần Clean Code



- Năng suất sẽ giảm nhanh theo thời gian nếu làm việc với mã nguồn không tốt



**Figure 1-1**  
Productivity vs. time

# Các yếu tố ảnh hưởng đến chất lượng mã nguồn

---



- Định danh
- Phương thức
- Ghi chú
- Định dạng(format) của mã nguồn
- Thiết kế kiến trúc
- Xử lý lỗi và ngoại lệ
- Test



# Đặt tên: Cần thể hiện rõ ý nghĩa

---



- Tên gọi cần thể hiện rõ ý nghĩa của đối tượng mà nó đại diện (chẳng hạn như biến, phương thức, đối tượng...)
- Ví dụ:
  - Tên gọi không thể hiện được ý nghĩa:

`int d; // elapsed time in days`

- Tên gọi tốt:

```
int elapsedTimeInDays;  
int daysSinceCreation;  
int daysSinceModification;  
int fileAgeInDays;
```

# Ví dụ: tên gọi không tốt



- Nhiều tên gọi không tốt:

```
public List<int[]> getThem() {  
    List<int[]> list1 = new ArrayList<int[]>();  
    for (int[] x : theList)  
        if (x[0] == 4)  
            list1.add(x);  
    return list1;  
}
```

- Có thể sửa thành:

```
public List<int[]> getFlaggedCells() {  
    List<int[]> flaggedCells = new ArrayList<int[]>();  
    for (int[] cell : gameBoard)  
        if (cell[STATUS_VALUE] == FLAGGED)  
            flaggedCells.add(cell);  
    return flaggedCells;  
}
```

# Đặt tên: Tránh gây hiểu nhầm

---



- Chữ O và số 0 rất dễ nhầm lẫn với nhau
- Chữ l và số 1 rất dễ nhầm lẫn với nhau
- Ví dụ:

```
int a = l;  
if (O == l) a = O1;  
else  
    l = 01;
```

# Đặt tên: Có sự khác biệt rõ ràng giữa các tên



- Ví dụ, a1 và a2 trong đoạn code sau không phân biệt rõ ràng:

```
public static void copyChars(char a1[], char a2[]) {  
    for (int i = 0; i < a1.length; i++) {  
        a2[i] = a1[i];  
    }  
}
```

- Ví dụ, tên của các phương thức sau không giúp phân biệt được mục đích thực sự của từng phương thức:

```
getActiveAccount();  
getActiveAccounts();  
getActiveAccountInfo();  
getActiveAccountData();
```

# Đặt tên: Tên phát âm được



- Nên *đặt* các tên gọi có thể phát âm được
- Ví dụ, các tên gọi sau gây khó khăn cho phát âm:

```
class DtaRcrd102 {  
    private Date genymdhms;  
    private Date modymdhms;  
    private final String pszqint = "102";  
    /* ... */  
};
```

- Có thể *sửa* thành:

```
class Customer {  
    private Date generationTimestamp; private  
    Date modificationTimestamp;; private final  
    String recordId = "102";  
    /* ... */  
};
```

# Đặt tên: Tên gọi có thể tìm kiếm được



- Nên *đặt* các tên gọi có thể dễ dàng tìm kiếm bằng công cụ IDE
- Ví dụ, tên gọi rất khó tìm kiếm và phân biệt:

```
for (int j=0; j<34; j++) { s
    += (t[j]*4)/5;
}
```

- Ví dụ, các tên gọi dễ tìm kiếm:

```
int realDaysPerIdealDay = 4;
final int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
    int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
    int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
    sum += realTaskWeeks;
}
```

# Đặt tên: Không viết tắt hoặc mã hoá



- Các tên gọi viết tắt hoặc mã hoá gây khó khăn cho việc đọc và hiểu mã nguồn
- Ví dụ, sử dụng tên viết tắt:

```
public class Part {  
    private String m_dsc; // The textual description  
    void setName(String name) {  
        m_dsc = name;  
    }  
}
```

- Ví dụ, tên gọi tốt:

```
public class Part { String  
    description;  
    void setDescription(String description){  
        this.description = description;  
    }  
}
```

# Đặt tên: Sử dụng đúng ngữ nghĩa

---



- Sử dụng cùng một từ cho một khái niệm, không sử dụng các từ bừa bãi
  - Ví dụ, các từ *fetch*, *retrieve* hoặc *get* có thể tương đương nhau, do đó nên lựa chọn sử dụng một từ duy nhất
- Lưu ý đến ngữ nghĩa của từng từ, ví dụ cả 3 từ là *add*, *insert* và *append* đều có nghĩa là thêm vào có sự phân biệt:
  - *add*: thêm vào - thường là thêm vào ở cuối hoặc là một vị trí nào đó
  - *insert*: chèn vào - thường là chèn vào ở giữa
  - *append*: thêm vào - thường là thêm vào ở cuối



# Đặt tên lớp

---



- Tên của lớp nên là danh từ
  - Ví dụ: Customer, WikiPage, Account, AddressParser
- Tránh dùng những từ như Manager, Processor, Data, Info
  - Ví dụ: CustomerData, AccountInfo, PageProcessor
- Tên lớp nên bắt đầu bằng chữ viết hoa
- Tên lớp nên tuân theo quy tắc CamelCase
  - Ví dụ: ActiveRecordRepository

# Đặt tên phương thức

---



- Tên phương thức nên là một *động từ* hoặc bắt đầu bằng một *động từ*
  - Ví dụ: `postPayment`, `deletePage`, `save`
- Tên phương thức nên bắt đầu bằng chữ viết thường
- Tên phương thức nên tuân theo quy tắc CamelCase

---

# Thảo luận

Kỹ thuật đổi tên biến

Kỹ thuật đổi tên phương thức

Kỹ thuật tách biến

Kỹ thuật tách hằng

Kỹ thuật tách phương thức

# Tái cấu trúc mã nguồn

---



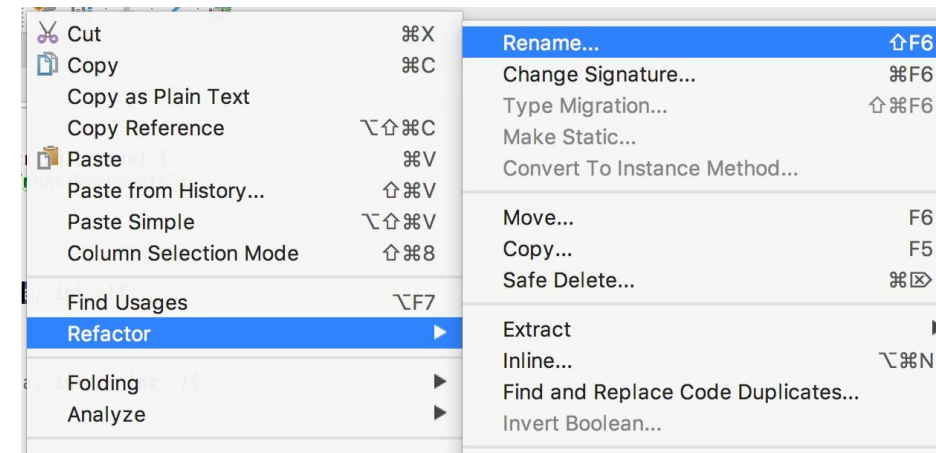
- Tái cấu trúc mã nguồn là các kỹ thuật cho phép chỉnh sửa mã nguồn nội bộ mà không làm thay đổi hành vi của hệ thống đối với bên ngoài
- Tái cấu trúc mã nguồn nhằm mục đích chính:
  - Mã nguồn dễ duy trì hơn
  - Mã nguồn dễ mở rộng hơn

**Lưu ý:** Tái cấu trúc mã nguồn không phải luôn luôn giúp tăng hiệu năng (performance) của thuật toán. Trong một số trường hợp, có thể cần hy sinh hiệu năng để có được mã nguồn tốt hơn

# Đổi tên biến và phương thức



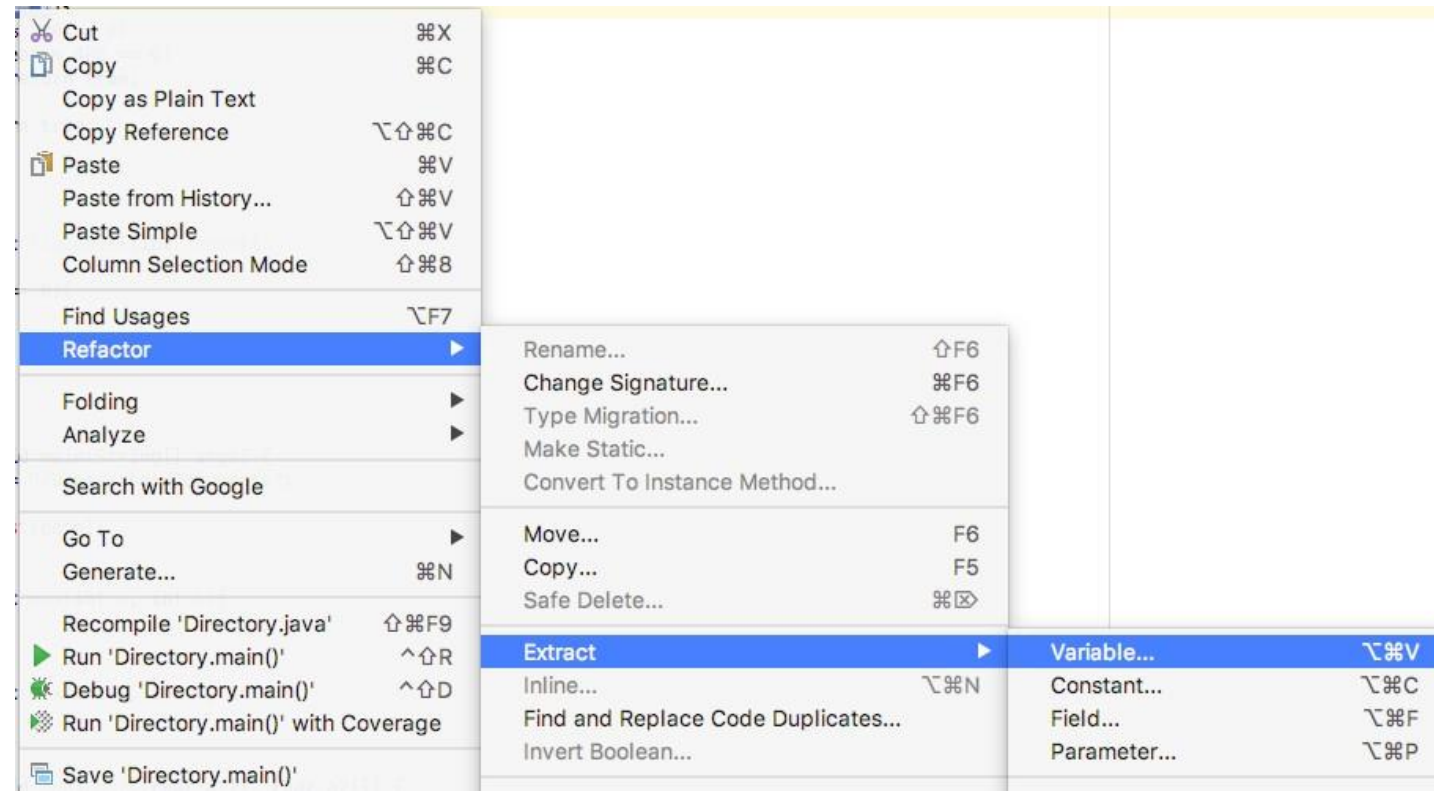
- Thay đổi tên biến hoặc phương thức để trở nên tốt hơn: dễ đọc, có ý nghĩa, thể hiện được ý nghĩa, tuân thủ coding convention
- Khi đổi tên biến hoặc phương thức cần lưu ý:
  - Đổi tên tại vị trí khai báo
  - Đổi tên tại tất cả các vị trí có sử dụng biến hoặc phương thức
- Nên sử dụng tính năng của IDE để đổi tên biến hoặc phương thức
- Phím tắt đổi tên của IntelliJ IDEA là:



# Tách biến



- Trong nhiều trường hợp, các biểu thức phức tạp sẽ gây khó hiểu
- Tách biến (Variable Extraction) là kỹ thuật giúp đơn giản hoá các biểu thức và giúp dễ hiểu hơn
- Có thể sử dụng phím tắt hoặc menu trên IntelliJ IDEA để tách biến



# Tách biến: Ví dụ



- Ví dụ, kiểm tra năm nhuận:

```
private static boolean isLeapYear(int year) {  
    if(year % 4 == 0){  
        if (year % 100 == 0){  
            if(year % 400 == 0)  
                return true;  
        } else {  
            return true;  
        }  
    }  
    return false;  
}
```

```
private static boolean isLeapYear(int year) {  
    boolean isDivisibleBy4 = year % 4 == 0;  
    if(isDivisibleBy4){  
        boolean isDivisibleBy100 = year % 100 == 0;  
        if (isDivisibleBy100){  
            boolean isDivisibleBy400 = year % 400 == 0;  
            if(isDivisibleBy400) return  
                true;  
        } else {  
            return true;  
        }  
    }  
    return false;  
}
```

# Tách hằng

---



- Trong nhiều trường hợp, các giá trị “thần kỳ” (magic value) sẽ gây khó khăn cho việc đọc hiểu mã nguồn
- Tách hằng giúp mang lại ý nghĩa cho các giá trị “thần kỳ” và mã nguồn dễ hiểu hơn
- Có thể sử dụng phím tắt hoặc menu của IntelliJ IDEA để tách hằng



# Tách hằng: Ví dụ



- Ví dụ, kiểm tra quyền dựa vào role:

```
public static boolean isAuthorized(int role){  
    if(role == 1){  
        return true;  
    }  
    return false;  
}
```

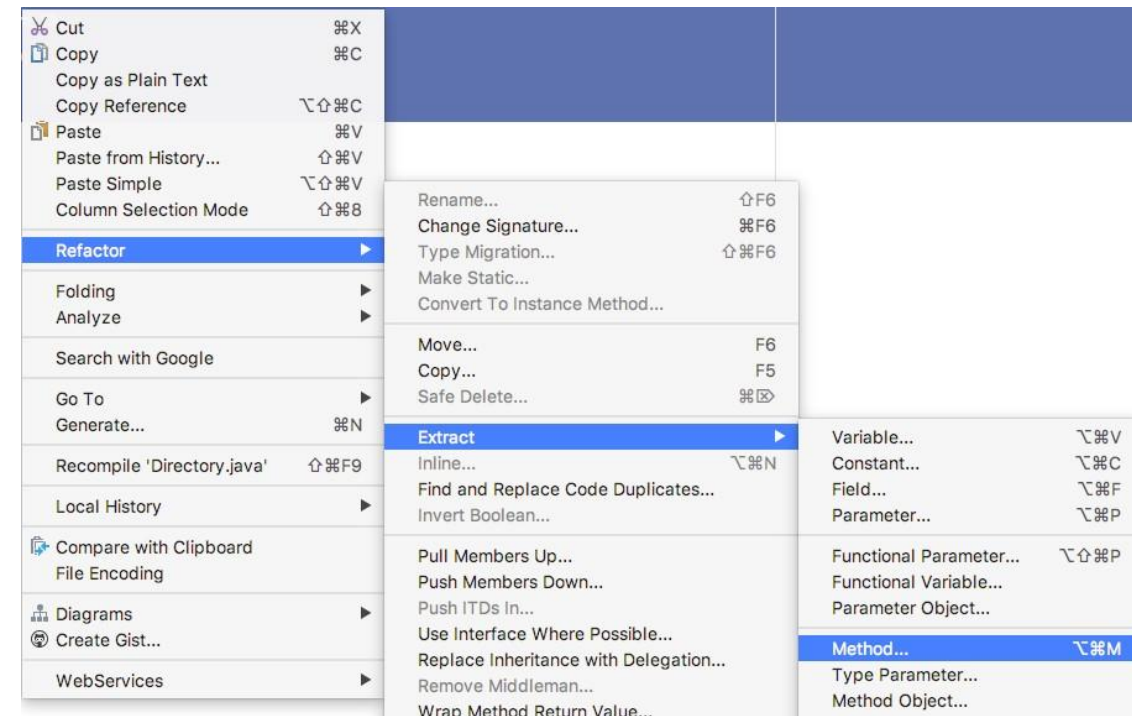
```
public static final int ROLE_ADMIN = 1;
```

```
public static boolean isAuthorized(int role){  
    if(role == ROLE_ADMIN){  
        return true;  
    }  
    return false;  
}
```

# Tách phương thức



- Trong nhiều trường hợp, một phương thức quá dài, quá phức tạp hoặc xử lý quá nhiều tác vụ sẽ dẫn đến khó hiểu, khó kiểm soát
- Tách phương thức giúp cho các phương thức dễ đọc hiểu hơn, dễ kiểm soát hơn
- Có thể sử dụng phím tắt hoặc menu của IntelliJ IDEA để tách phương thức



# Tách phương thức: Ví dụ(1)



- Ví dụ, tính số ngày của tháng:

case 2:

```
public static int getDaysOfMonth(int month, int year){  
    switch (month){  
        case 1:  
        case 3:  
        case 5:  
        case 7:  
        case 8:  
        case 10:  
        case 12:  
            return 31;  
        case 4:  
        case 6:  
        case 9:  
        case 11:  
            return 30;  
        ...  
    }
```

```
        boolean isLeapYear = false; if(year  
        % 4 == 0){  
            if (year % 100 == 0){  
                if(year % 400 == 0)  
                    isLeapYear = true;  
            } else {  
                isLeapYear = true;  
            }  
        }  
        if(isLeapYear){ return  
            29;  
        } else {  
            return 28;  
        }  
        default:  
            return 0;  
    }  
}
```

# Tách phương thức: Ví dụ(2)



```
public static int getDaysOfMonth(int month, int year){  
    switch (month){  
        .....  
        case 4:  
        case 6:  
        case 9:  
        case 11:  
            return 30;  
        case 2:  
            boolean isLeapYear = isLeapYear(year);  
            if(isLeapYear){  
                return 29;  
            } else {  
                return 28;  
            }  
        default:  
            return 0;  
    }  
}
```

```
private static boolean isLeapYear(int year) {  
    boolean isLeapYear = false;  
    if(year % 4 == 0){  
        if (year % 100 == 0){  
            if(year % 400 == 0)  
                isLeapYear = true;  
        } else {  
            isLeapYear = true;  
        }  
    }  
    return isLeapYear;  
}
```

# Tóm tắt bài học

---



- Clean Code là thuật ngữ để chỉ đến những mã nguồn “tốt”: Dễ đọc, dễ hiểu, dễ bảo trì, dễ mở rộng
- Code Smell là thuật ngữ để chỉ đến những mã nguồn “có vấn đề”
- Một số Code Smell phổ biến đó là: code duplicate, đặt tên vô nghĩa, phương thức quá dài, phương thức làm nhiều việc, sử dụng các magic value
- Sử dụng các kỹ thuật refactoring để loại bỏ Code Smell
- Refactoring là chỉnh sửa mã nguồn để trở nên “clean” hơn nhưng không làm thay đổi hành vi của hệ thống
- Một số kỹ thuật refactoring cơ bản: thay đổi tên biến, tách biến, tách hằng, tách phương thức



---

# Hướng dẫn

Hướng dẫn làm bài thực hành và bài tập

Chuẩn bị bài tiếp theo: Test First -TDD