# Lập trình JAVA cơ bản

Tuần 5

Giảng viên: Trần Đức Minh

# Nội dung bài giảng

- Exception
- Biểu thức Lamda
- Vòng lặp forEach
- Sắp xếp trong Collection

# Exception

- **Exception** (ngoại lệ) là một sự kiện xảy ra trong quá trình thực thi chương trình, nó phá vỡ luồng xử lý bình thường của chương trình, thậm chí làm chương trình bị dừng lại đột ngột.
- Exception thường xảy ra ngoài dự tính của chương trình như:
  - Nhập dữ liệu không hợp lệ.
  - Kết nối với hệ thống mạng đi đứt.
  - Chia cho số 0.
  - …..

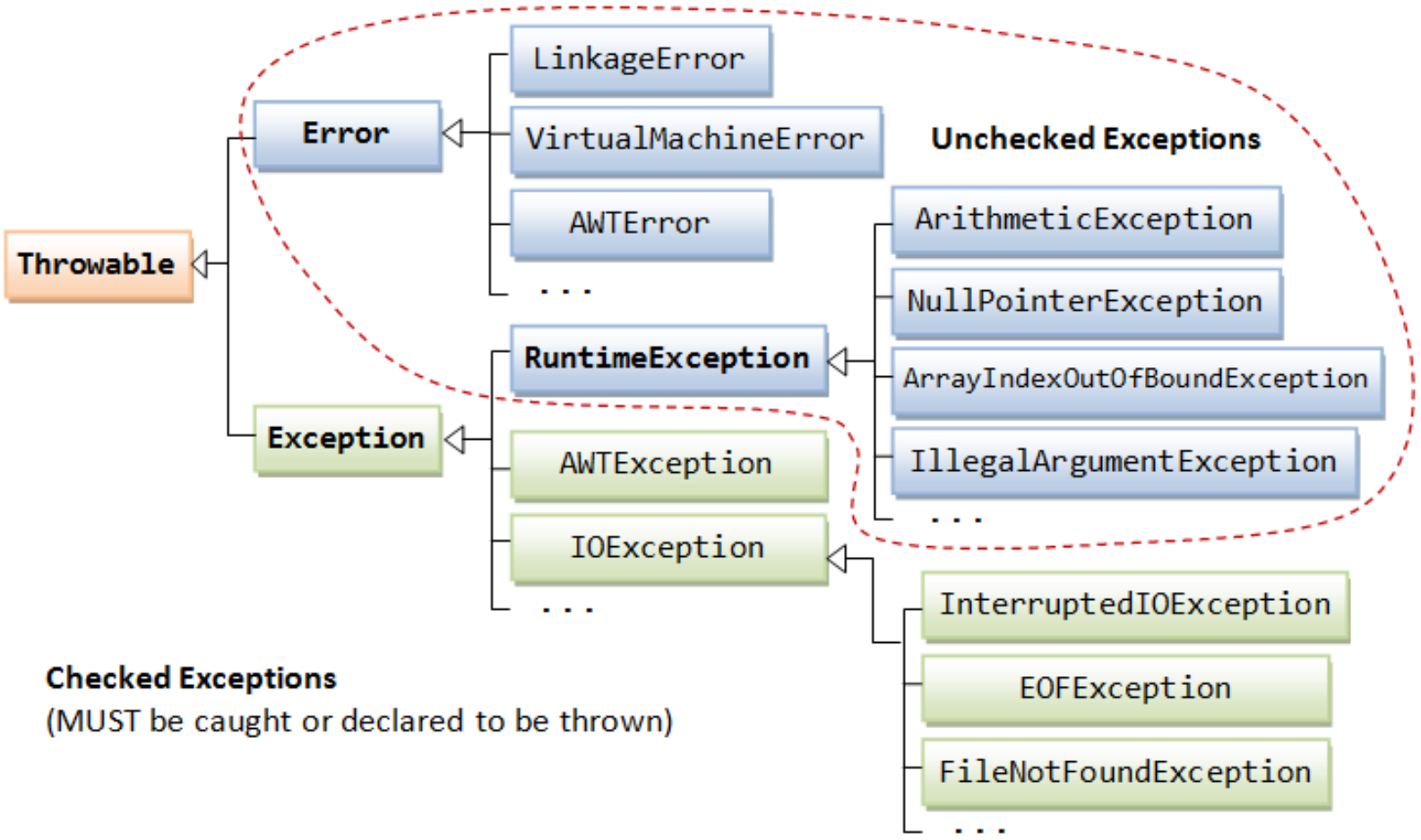
# Exception

- Chạy đoạn code với giá trị nhập vào bằng 0

```
public class MainClass {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Nhập mẫu số: ");
        int iMauSo =  scanner.nextInt();
        int iThuongSo = 10 / iMauSo;
        System.out.println("Thương số = " + iThuongSo);
    }
}
```

  - Hệ thống sẽ **tự động tạo ra một Exception** khi ta sử dụng phép chia cho 0.

# Sơ đồ phân cấp của các lớp Exception



# Exception

- Trong Java có 2 dạng Exception
  - **Checked exceptions** là các exceptions xảy ra trong quá trình **biên dịch** chương trình. Ta không được phép bỏ qua mà bắt buộc phải xử lý nó.
  - **Unchecked exceptions** là các exceptions xảy ra tại thời điểm **thực thi** chương trình. Dạng exception này được bỏ qua trong quá trình biên dịch và không yêu cầu ta phải xử lý nó.
    - **Error** (một dạng unchecked exceptions): định nghĩa những exceptions mà không thể bắt bởi chương trình và thường làm chết hệ thống.

# Exception

- Ví dụ **checked exception**:

```
import java.io.FileInputStream;

public class MainClass {
    public static void main(String[] args) {
        FileInputStream fis = new FileInputStream( name: "C:/myfile.txt");

        int k;

        while(( k = fis.read() ) != -1)
        {
            System.out.print((char)k);
        }

        fis.close();
    }
}
```

  - Khi biên dịch, chương trình **bắt buộc** ta phải có một Exception để xử lý vấn đề file có thể không tồn tại.

# Exception

- Ví dụ **unchecked exception**:

```
public class MainClass {
    public static void main(String[] args) {
        int arr[] = { 1, 2, 3, 4, 5 };
        System.out.println(arr[5]);
    }
}
```

  - Đối với đoạn code trên, chương trình vẫn **biên dịch bình thường**. Tuy nhiên, tại **thời điểm thực thi**, **Exception cho đoạn code này mới được sinh** ra do truy cập bên ngoài phạm vi của mảng.

# Exception

- Ví dụ **Error**:

```
public class MainClass {
    public static void showInfo() {
        System.out.print("Hello");
        showInfo();
    }
    public static void main(String[] args) {
        showInfo();
    }
}
```

  - Chương trình trên vẫn biên dịch bình thường, nhưng sinh ra lỗi tràn Stack khi chương trình thực thi.

# Làm việc với Exception

- Bước 1: Nảy sinh vấn đề
  - Một vấn đề nào đó xảy ra bên trong chương trình có thể dẫn đến chương trình bị dừng lại đột ngột.
- Bước 2: Tạo ra Exception
  - Khi gặp vấn đề ở Bước 1, hệ thống hoặc người lập trình sẽ tạo ra một Exception tương ứng với vấn đề nảy sinh.
- Bước 3: Ném Exception
  - Nhảy đến vị trí xử lý Exception (nếu có)
- Bước 4: Xử lý Exception

# Xử lý Exception

- **Xử lý Exception** (Exception handling) là quá trình xử lý ngoại lệ để cố gắng duy trì luồng công việc bình thường trong chương trình.
  - Nếu không xử lý được ngoại lệ thì chương trình sẽ tự động kết thúc.
- Quá trình xử lý exception cũng được gọi là **catch exception**.

# Làm việc với Exception

- Sử dụng **khối lệnh try** để chứa đoạn code có thể xảy ra ngoại lệ.
- Sử dụng các **khối lệnh catch** để xử lý các ngoại lệ.
- Khối lệnh finally luôn được thực thi sau khi thực thi **khối lệnh try** hoặc sau khi xử lý một ngoại lệ trong **khối lệnh catch** nào đó.

```
try
{
    // statements that might cause exceptions
    // possibly including function calls
}
catch ( exception-1 ex-1 )
{
    // statements to handle this exception
}
catch ( exception-2 ex-2 )
{
    // statements to handle this exception
  .
  .
}
finally
{
    // statements to execute every time this try block
    executes
}
```

# Xử lý Exception

- Phân tích đoạn code sau:
  - Mỗi khi **phát sinh Exception bên trong khối lệnh try**, hệ thống sẽ **tự động tìm đến khối lệnh catch tương ứng** với Exception phát sinh.

```
public class MainClass {
    public static void main(String[] args) {
        try {
            int arr[] = new int[5];
            arr[5] = 4;
            System.out.println("arr[5] = " + arr[5]);

            int zero = 0;
            int average = 10 / zero;
            System.out.println("Average = " + average);

            String obj = null;
            System.out.println(obj.length());
        } catch (NullPointerException ex) {
            System.out.println(ex);
        } catch (ArithmeticException ex) {
            System.out.println(ex);
        } catch (ArrayIndexOutOfBoundsException ex) {
            System.out.println(ex);
        } finally {
            System.out.println("Kết thúc khối lệnh !");
        }

        System.out.println("Kết thúc chương trình");
    }
}
```

# Xử lý Exception

- Khối lệnh try lồng nhau
  - Phân tích đoạn code sau:
    - Sau khi thực thi xong đoạn code của khối try bên trong, chương trình vẫn tiếp tục thực thi những đoạn code bên dưới.

```
public class MainClass {
    public static void main(String[] args) {
        try {
            try {
                int zero = 0;
                int average = 10 / zero;
                System.out.println("Average = " + average);
            } catch (ArithmeticException ex) {
                System.out.println(ex);
            }

            System.out.println("Continue...");

            int arr[] = new int[5];
            arr[5] = 4;
            System.out.println("arr[5] = " + arr[5]);

        } catch (ArrayIndexOutOfBoundsException ex) {
            System.out.println(ex);
        }

        System.out.println("Finished!");
    }
}
```

# Xử lý Exception

- Một số chú ý khi làm việc với Exception
  - Khi có một khối lệnh catch được xử lý thì các khối lệnh catch còn lại sẽ không được xử lý nữa.
  - Tất cả các khối catch phải được sắp xếp theo thứ tự **exception con** trước rồi mới đến **exception cha**. (Nếu các exception kế thừa nhau nằm trong cùng một khối try … catch)
  - Khối lệnh finally luôn được thực thi dù chương trình có xảy ra ngoại lệ hay không (ngay cả khi sử dụng lệnh **return** trong khối lệnh try và catch).
  - Đối với mỗi khối try, có thể không có hoặc nhiều khối catch, nhưng chỉ có duy nhất một khối finally.
  - Khối finally chỉ không được thực thi nếu chương trình bị thoát bằng cách gọi lệnh **System.exit()** hoặc xảy ra một **Error** khiến chương trình bị dừng đột ngột.

# Xử lý Exception

- Ví dụ **System.exit()**

```
public class MainClass {
    public static void main(String[] args) {
        try {
            int arr[] = new int[5];
            arr[5] = 4;
            System.out.println("arr[5] = " + arr[5]);
        } catch (ArrayIndexOutOfBoundsException ex) {
            System.out.println(ex);
            System.exit( status: 0);
        } finally {
            System.out.println("Thực thi khối lệnh finally");
        }

        System.out.println("Finished!");
    }
}
```

# Xử lý Exception

- Ví dụ **return**

```
public class MainClass {
    public static int getValue() {
        try {
            int arr[] = new int[5];
            arr[5] = 4;
            System.out.println("arr[5] = " + arr[5]);
            return 1;
        } catch (ArrayIndexOutOfBoundsException ex) {
            System.out.println(ex);
            return 0;
        } finally {
            System.out.println("Thực thi khối lệnh finally !!! ");
        }
    }

    public static void main(String[] args) {
        System.out.println("Giá trị nhận được là = " + getValue());
    }
}
```

# Exception

- Từ khóa **throw** được sử dụng để ném ra một **exception** cụ thể.
- Ví dụ:

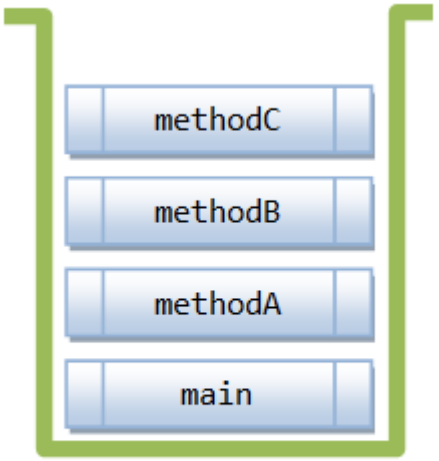
```
public class MainClass {
    public static void main(String args[]) {
        int number = -5;

        try {
            if (number < 0)
                throw new ArithmeticException("Số nguyên không hợp lệ");
            else
                System.out.println("Số nguyên hợp lệ");
        } catch (ArithmeticException ex) {
            System.out.println(ex);
        }

        System.out.println("Kết thúc chương trình");
    }
}
```

# Ngăn xếp gọi phương thức

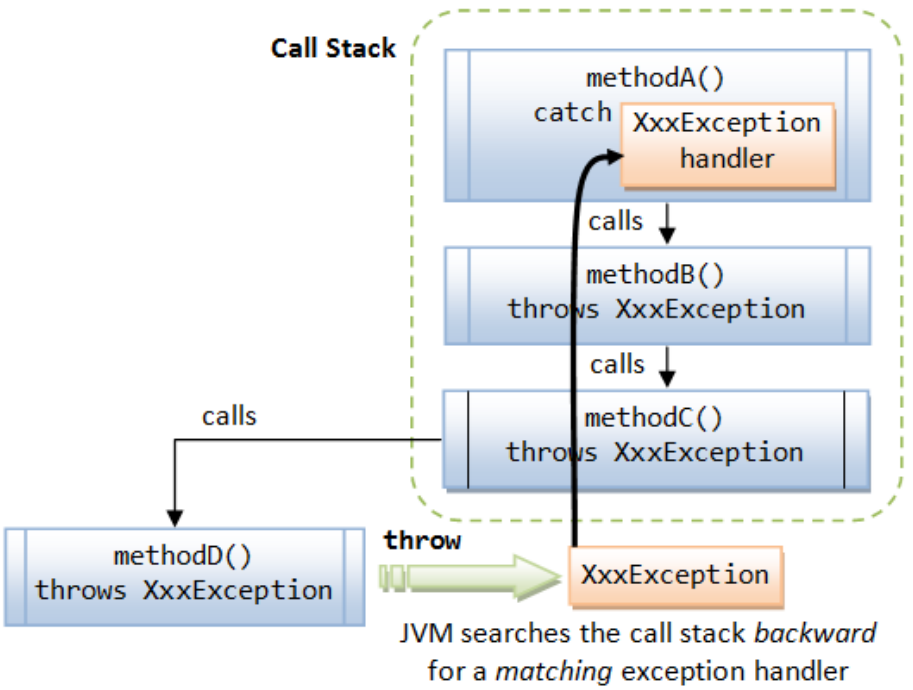
- Trong Java, khi các phương thức gọi đến nhau, chúng sẽ được đưa vào một vùng nhớ gọi là **ngăn xếp gọi phương thức** (method call stack). Khi phương thức thực hiện xong, nó sẽ bị xóa khỏi ngăn xếp.
- Ví dụ: Hàm **main** gọi **phương thức A**, A gọi **phương thức B** và B gọi **phương thức C**.




**Method Call Stack**
**(Last-in-First-out Queue)**

# Lan truyền exception

- Khi một exception xuất hiện, nếu phương thức ở trên đỉnh stack không chứa khối lệnh xử lý catch và phương thức này bị loại bỏ khỏi stack, hệ thống sẽ tự động tìm khối lệnh xử lý catch trong phương thức kế tiếp ở stack, quá trình cứ như vậy cho đến khi stack rỗng. Đây được gọi là quá trình **lan truyền exception** (exception propagation).



# Lan truyền exception

- Unchecked exception được phép lan truyền.

```java
public class MainClass {
    void exceptionHere() {
        int data = 50 / 0;
    }

    void tempMethod() {
        exceptionHere();
    }

    void exceptionHandle() {
        try {
            tempMethod();
        } catch (ArithmeticException e) {
            System.out.println("Xử lý Exception");
        }
    }

    public static void main(String args[]) {
        MainClass obj = new MainClass();
        obj.exceptionHandle();
        System.out.println("Kết thúc chương trình");
    }
}
```

# Lan truyền exception

- Checked exception không được phép lan truyền.

```java
public class MainClass {
    void exceptionHere() {
        throw new java.io.IOException("Thiết bị vào ra lỗi");
    }

    void tempMethod() {
        exceptionHere();
    }

    void exceptionHandle() {
        try {
            tempMethod();
        } catch (IOException e) {
            System.out.println("Xử lý Exception");
        }
    }

    public static void main(String args[]) {
        MainClass obj = new MainClass();
        obj.exceptionHandle();
        System.out.println("Kết thúc chương trình");
    }
}
```

# Exception

- Từ khóa **throws** được sử dụng để ném exception cho phương thức khác xử lý.
- Ta sử dụng throws khi nghi ngờ trong phương thức có thể xảy ra exception.
- throws nên sử dụng chỉ với checked exception do
  - Ta thường kiểm soát được các unchecked exception.
  - Ta thường không thể làm được bất kỳ điều gì khi gặp error (VirtualMachineError, StackOverflowError, ...)

# Exception

```java
import java.io.IOException;

public class MainClass {
    void exceptionHere() throws IOException {
        throw new java.io.IOException("Thiết bị vào ra lỗi");
    }

    void tempMethod() throws IOException {
        exceptionHere();
    }

    void exceptionHandle() {
        try {
            tempMethod();
        } catch (IOException e) {
            System.out.println("Xử lý Exception");
        }
    }

    public static void main(String args[]) {
        MainClass obj = new MainClass();
        obj.exceptionHandle();
        System.out.println("Kết thúc chương trình");
    }
}
```

# Sự khác nhau giữa throw và throws

| throw | throws |
|---|---|
| throw được sử dụng để ném ra một exception cụ thể | throws được sử dụng để khai báo một exception |
| checked exception không được lan truyền | checked exception được lan truyền |
| throw ném ra một đối tượng exception | throws khai báo một hoặc nhiều lớp exception |
| throw được sử dụng bên trong phương thức | throws được khai báo sau khai báo tên và các đối số của phương thức |
| Một throw không thể ném nhiều exceptions cùng một lúc | Có thể khai báo nhiều exceptions cho một phương thức |

# Functional Interface

- **Functional Interface** là một **interface** chỉ chứa **duy nhất** một phương thức trừu tượng do người dùng tự định nghĩa.

```java
@FunctionalInterface
public interface MyInterface {
    void showData();
}
```

- Ví dụ: interface sau không phải là một Functional Interface

```
@FunctionalInterface
public interface MyInterface {
    void showData();
    int calculateValue();
}
```

# Functional Interface

- Functional Interface có thể chứa các phương thức **public** của lớp **java.lang.Object**

```
@FunctionalInterface
public interface MyInterface {
    void showData();

    boolean equals(Object obj);
    int hashCode();
}
```

# Biểu thức Lamda

- **Biểu thức Lamda** là một **hàm ẩn danh** (anonymous function) và có những đặc điểm sau:
  - Cung cấp việc cài đặt cụ thể cho **phương thức trừu tượng được định nghĩa bên trong Functional Interface**.
  - Có đầy đủ các đặc điểm như một hàm
    - Tham số đầu vào (có thể có hoặc không)
    - Nội dung lập trình
    - Giá trị trả về (có thể có hoặc không)
  - Ưu điểm lớn: Giảm thiểu mã nguồn khi lập trình

# Biểu thức Lamda

- Cú pháp:

  **(<danh sách tham số>) -> { … }**

- Ví dụ

Không dùng biểu thức Lamda,
sử dụng Inner class vô danh

```java
@FunctionalInterface
interface MyInterface {
    void showData();
}

public class MainClass {
    public static void main(String args[]) {
        MyInterface mi = new MyInterface() {
            @Override
            public void showData() {
                System.out.println("Xin chao !!!");
            }
        };
        mi.showData();
    }
```

Sử dụng biểu thức Lamda

```java
@FunctionalInterface
interface MyInterface {
    void showData();
}

public class MainClass {
    public static void main(String args[]) {
        MyInterface mi = () -> {
            System.out.println("Xin chao !!!");
        };
        mi.showData();
    }
}
```

# Biểu thức Lamda

- Các trường hợp đặc biệt khi sử dụng biểu thức Lamda
  - Ta có thể **bỏ cặp dấu ngoặc đơn ()** trong trường hợp chỉ truyền vào một tham số duy nhất
    - Ví dụ:

```java
@FunctionalInterface
interface MyInterface {
    void showData(String name);
}

public class MainClass {
    public static void main(String args[]) {
        MyInterface mi = name -> {
            System.out.println("Xin chao " + name + " !!!");
        };
        mi.showData( name: "Java");
    }
}
```

# Biểu thức Lamda

- Các trường hợp đặc biệt khi sử dụng biểu thức Lamda
  - Ta có thể **bỏ cặp dấu ngoặc nhọn {}** trong trường hợp phần lập trình chỉ có một lệnh duy nhất.
    - Ví dụ:

```
@FunctionalInterface
interface MyInterface {
    void showData(String name);
}

public class MainClass {
    public static void main(String args[]) {
        MyInterface mi = name -> System.out.println("Xin chao " + name + " !!!");

        mi.showData( name: "Java");
    }
}
```

# Biểu thức Lamda

- Các trường hợp đặc biệt khi sử dụng biểu thức Lamda
  - Ta có thể **bỏ từ khóa return** trong trường hợp phần lập trình chỉ có một lệnh duy nhất là lệnh trả về giá trị cho hàm
    - Ví dụ:

```
@FunctionalInterface
interface MyInterface {
    int calculateValue(int number);
}

public class MainClass {
    public static void main(String args[]) {
        MyInterface mi = n -> n * (n + 8);

        System.out.println(mi.calculateValue( number: 5));
    }
}
```

# Vòng lặp forEach

- **Vòng lặp forEach** cung cấp một cách viết mới ngắn gọn để **lặp trên một collection** và **thực hiện một hành động** nào đó trên mỗi phần tử.
  - void forEach(Consumer<? super T> action)
- Hành động trên mỗi phần tử được đặt bên trong một lớp implements đến một **functional interface** có tên là **Consumer**.
  - @FunctionalInterface

    public interface Consumer<T> {

        void accept(T t);

    }

# Cách sử dụng vòng lặp forEach

- Sử dụng Inner Class vô danh
  - Khởi tạo một **đối tượng của lớp vô danh** mà implements đến interface Consumer rồi sử dụng đối tượng đó như tham số của phương thức forEach.

```
public class MainClass {
    public static void main(String args[]) {
        Integer arrNumber[] = {3, 9, 8, 6, 5};

        ArrayList<Integer> arrList = new ArrayList<~>(Arrays.asList(arrNumber));

        Consumer<Integer> printArrNumber = new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) {
                System.out.println(integer);
            }
        };

        arrList.forEach(printArrNumber);

    }
}
```

# Cách sử dụng vòng lặp forEach

- Sử dụng biểu thức Lamda
  - Mã nguồn lập trình sẽ được **rút gọn hơn** so với sử dụng Inner Class vô danh
  - Ví dụ sử dụng biểu thức Lamda với List

```
public class MainClass {
    public static void main(String args[]) {
        Integer arrNumber[] = {3, 9, 8, 6, 5};

        ArrayList<Integer> arrList = new ArrayList<Integer>(Arrays.asList(arrNumber));

        arrList.forEach(songuyen -> System.out.println(songuyen));

    }
}
```

# Cách sử dụng vòng lặp forEach

- Sử dụng biểu thức Lamda
  - Ví dụ sử dụng biểu thức Lamda với Map

```
public class MainClass {
    public static void main(String args[]) {
        Map<Integer, String> namesMap = new HashMap<>();
        namesMap.put(1, "Nguyen Phuong Anh");
        namesMap.put(2, "Tran Van Toan");
        namesMap.put(3, "Luong Thu Hoai");

        namesMap.forEach((key, value) -> System.out.println(key + " : " + value));

    }
}
```

# Sắp xếp trong Collection

- Để sắp xếp **các phần tử trong Collection**, Java cung cấp phương thức tĩnh có tên là **sort()** để làm công việc này.
- Ví dụ: Sắp xếp danh sách các số nguyên theo chiều tăng dần (***mặc định***)

```
public class MainClass {
    public static void main(String args[]) {
        Integer arrNumber[] = {3, 9, 8, 6, 5};

        ArrayList<Integer> arrList = new ArrayList<Integer>(Arrays.asList(arrNumber));

        Collections.sort(arrList);

        arrList.forEach(i -> System.out.println(i));

    }
}
```

# Sắp xếp trong Collection

- Sắp xếp danh sách các số nguyên theo chiều giảm dần
  - Sử dụng phương thức quá tải sort() trong Collection với 2 tham số đầu vào.

    **static <T> void sort(List<T> list, Comparator<? super T> c)**

    - Giải thích: Sắp xếp **list** theo thứ tự được xác định bởi **Comparator**
      - **Comparator** là một **functional interface**, trong đó có chứa phương thức hai tham số đầu vào dùng để hỗ trợ cho việc sắp xếp.

        ```
        @FunctionalInterface
        public interface Comparator<T> {
            int compare(T o1, T o2);
        }
        ```
      - Người lập trình cần viết code cho phương thức **compare()** và thông qua giá trị trả về để so sánh hai giá trị đầu vào o1 và o2 .
        - Giá trị trả về là -1: Phương thức compare() coi o1 < o2
        - Giá trị trả về là 0: Phương thức compare() coi o1 = o2
        - Giá trị trả về là 1: Phương thức compare() coi o1 > o2

# Sắp xếp trong Collection

- Sắp xếp danh sách các số nguyên theo chiều giảm dần
  - Cách 1: Sử dụng Inner Class vô danh

```
public class MainClass {
    public static void main(String args[]) {
        Integer arrNumber[] = {3, 9, 8, 6, 5};

        ArrayList<Integer> arrList = new ArrayList<~>(Arrays.asList(arrNumber));

        Collections.sort(arrList, new Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
                return o1 > o2 ? -1 : 1;
            }
        });

        arrList.forEach(i -> System.out.println(i));
    }
}
```

# Sắp xếp trong Collection

- Sắp xếp danh sách các số nguyên theo chiều giảm dần
  - Cách 2: Sử dụng biểu thức Lamda

```
public class MainClass {
    public static void main(String args[]) {
        Integer arrNumber[] = {3, 9, 8, 6, 5};

        ArrayList<Integer> arrList = new ArrayList<~>(Arrays.asList(arrNumber));

        Collections.sort(arrList, (number1, number2) -> number1 > number2 ? -1 : 1);

        arrList.forEach(i -> System.out.println(i));
    }
}
```

# Hết Tuần 5

Cảm ơn các bạn đã chú ý lắng nghe !!!