

# Java Core #3





# Exceptions



**#P; =HLAGFK**

```
int a = 0;  
int b = 5;  
int c = b/a;  
System.out.println(c);
```

Exception in thread "main" java.lang.ArithmeticException: / by zero

# Exceptions

Ngoại lệ là một sự kiện làm gián đoạn luồng bình thường của một chương trình.

Một ngoại lệ có thể xảy ra với nhiều lý do khác nhau, nó nằm ngoài dự tính của một chương trình. Ví dụ:

- Người dùng nhập dữ liệu không hợp lệ
- Một file cần được mở nhưng không tìm thấy
- Kết nối mạng bị mất khi đang giao tiếp hoặc JVM hết bộ nhớ



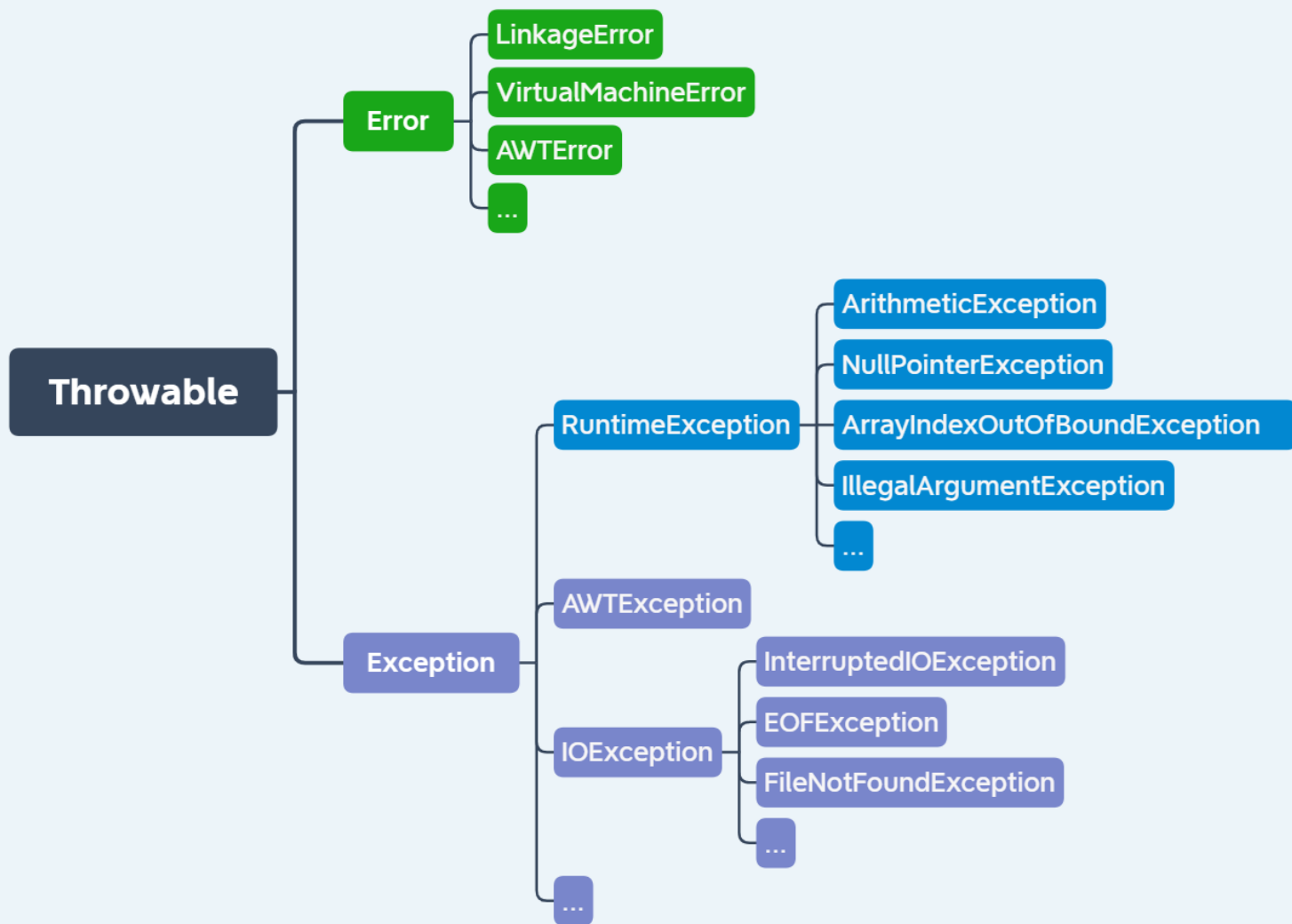


**! Č; DG° A =P; =HLAGFK**

**Checked Exception**

**Unchecked Exception**

**Error**





# ArithmeticException

Nếu ta chia bất kỳ số nào cho 0, xảy ra ngoại lệ  
ArithmeticException

Ví dụ:

```
int a = 10/0; //ArithmeticException
```

Exception in thread "main" java.lang.ArithmeticException: / by zero



, MDD. GAFL=J#P; =HLAGF

Nếu ta thực hiện bất kỳ một hành động nào với biến có giá trị null sẽ xảy ra ngoại lệ NullPointerException

Ví dụ:

```
String str = null;  
System.out.println(str.length()); //NullPointerException
```

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()" because "str" is null
```





# NumberFormatException

Một biến String có giá trị là các ký tự, chuyển đổi biến này sẽ xảy ra NumberFormatException

Ví dụ:

```
String str = "abc";  
int num = Integer.parseInt(str);
```

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "abc"
```



## JJ9Q F<=P- ML- > GMF<K#P; =HLAGF

Nếu bạn chèn bất kỳ giá trị nào vào sai index, sẽ xảy ra ngoại lệ `ArrayIndexOutOfBoundsException`

Ví dụ:

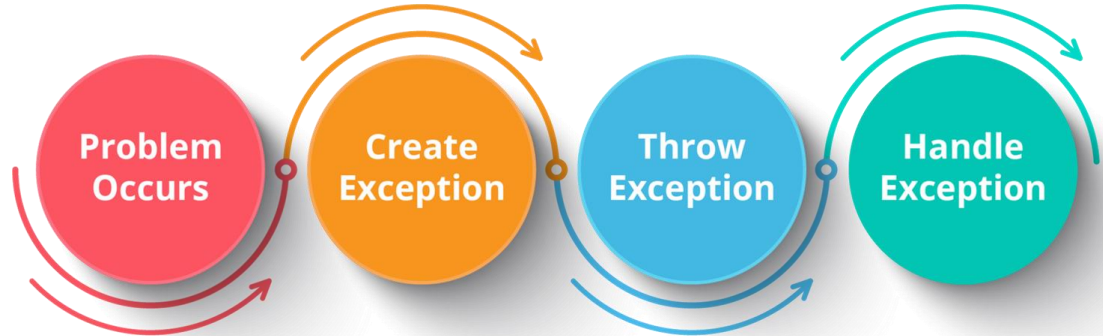
```
int arr[] = new int[5];  
arr[5] = 10;
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of  
bounds for length 5
```

# Exceptions Handling

Xử lý ngoại lệ (Exception Handling) trong java là một cơ chế xử lý các lỗi runtime để có thể duy trì luồng bình thường của ứng dụng.

Quá trình xử lý exception được gọi là catch exception, nếu Runtime System không xử lý được ngoại lệ thì chương trình sẽ kết thúc.





# Khối lệnh try - catch

! | HỒ~

```
try {  
    //Code có thể ném ra ngoại lệ  
} catch (Exception e) {  
    //Code xử lý ngoại lệ  
}
```



## Khối lệnh try - catch

```
try {  
    int arr[] = new int[5];  
    arr[5] = 10;  
} catch (Exception e) {  
    System.out.println("Something went wrong!!!");  
}
```

Something went wrong!!!



## Khối lệnh try - finally

```
try{  
    //Code nem ra ngoai le  
}finally{  
    //Code trong khoi lenh nay luon duoc thuc thi  
}
```



## Khối lệnh try - catch - finally

```
try{  
    //Code nem ra ngoai le  
}catch(Exception e){  
    //Code xu ly ngoai le  
}finally{  
    //Code trong khoi lenh nay lun duoc thuc thi  
}
```



) @A D F@ LJQ Ö ; 9L; @ Ö > AF9DDQ

```
try {  
    int arr[] = new int[5];  
    arr[5] = 10;  
} catch (Exception e) {  
    System.out.println("Something went wrong!!!");  
} finally {  
    System.out.println("Cau lenh nay luon duoc thuc thi");  
}
```

Something went wrong!!!  
Cau lenh nay luon duoc thuc thi





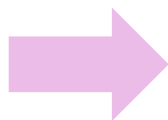
+L K H@!!F? L@; ; 9 D-H  
#P; =HLAGF

- **`getMessage()`**: trả về một message cụ thể về exception đã xảy ra
- **`getCause()`**: Trả về nguyên nhân xảy ra exception
- **`toString()`**: Trả về tên của lớp và kết hợp với kết quả từ phương thức `getMessage()`
- **`printStackTrace()`**: In ra kết quả của phương thức `toString` cùng với stack trace đến `System.err`
- **`fillInStackTrace()`**: Làm đầy stack trace của đối tượng `Throwable` bằng stack trace hiện tại

# Throw



Throw  
Exception



Từ khóa throw trong java được sử dụng để ném ra một ngoại lệ (exception) cụ thể

```
public class ThrowExample {  
    static void checkAge(int age) {  
        if (age < 18){  
            throw new ArithmeticException("Access denied -  
You must be at least 18 years old.");  
        }else{  
            System.out.println("Access granted -  
You are old enough!");  
        }  
    }  
    public static void main(String args[]) {  
        checkAge(15);  
        System.out.println("rest of the code...");  
    }  
}
```



2@GOK

Từ khóa **throws** trong java được sử dụng để khai báo một ngoại lệ.

**Throws** được dùng khi bạn không muốn phải xây dựng try catch bên trong một phương thức nào đó, bạn “đẩy trách nhiệm” phải try catch này cho phương thức nào đó bên ngoài có gọi đến nó phải try catch giúp cho bạn.

throw	throws
Từ khóa throw trong java được sử dụng để ném ra một ngoại lệ rõ ràng.	Từ khóa throws trong java được sử dụng để khai báo một ngoại lệ.
Ngoại lệ checked không được truyền ra nếu chỉ sử dụng từ khóa throw.	Ngoại lệ checked được truyền ra ngay cả khi chỉ sử dụng từ khóa throws.
Sau throw là một <b>instance</b> .	Sau throws là một hoặc nhiều <b>class</b> .
Throw được sử dụng trong phương thức có thể quăng ra Exception ở bất kỳ dòng nào trong phương thức (sau đó dùng try-catch để bắt hoặc throws cho thằng khác xử lý)	Throws được khai báo ngay sau dấu đóng ngoặc đơn của phương thức. Khi một phương thức có throw bên trong mà không bắt lại (try – catch) thì phải ném đi (throws) cho thằng khác xử lý.
Không thể throw nhiều exceptions.	Có thể khai báo nhiều exceptions, Ví dụ: <pre>public void method() throws IOException, SQLException { }</pre>

## Custom exception

!MKLGE =P; =HLAGE D· F?G A D□ <G F?¬ξA <¬F? L<sup>ne</sup>  
!MKLGE #P; =HLAGE LJGF? B9N9 L<sub>7</sub>□; K□  
<□F? L□ L<sub>7</sub>Q : A□F F?G A D□ L@G QLM ; °M ; □9  
F?¬ξA <¬F?} , @ K<sup>ne</sup> ?A|H L□ ; □9 F?G A D□ F9Q  
F?¬ξA <¬F? ; ε L@□ ; ε JALF? CA□M N· L@F? LA□H  
F?G A D□ JALF? ; @G EŤF@

Exception thuộc loại  
checked, nên thừa kế lớp  
Exception

```
public class CustomException extends Exception {  
    CustomException(String s){  
        super(s);  
    }  
}
```

*Thông thường, để tạo custom exception thuộc loại checked sẽ kế thừa từ lớp Exception. Để tạo custom exception thuộc loại unchecked sẽ kế thừa lớp RuntimeException*

```
public class ExceptionExample {  
    static void checkAge(int age) throws CustomException{  
        if(age < 18){  
            throw new CustomException("Not valid!!!");  
        }else{  
            System.out.println("Correct!!");  
        }  
    }  
    public static void main(String[] args) {  
        try {  
            checkAge(15);  
        } catch (CustomException e) {  
            e.printStackTrace();  
        }  
    }  
}
```





**! @AF=< =P; =HLAGF**

Chained exception cho phép bạn liên hệ một ngoại lệ với một ngoại lệ khác, tức là một ngoại lệ mô tả nguyên nhân của ngoại lệ khác

Mục đích chính của Chained exception là để bảo toàn ngoại lệ khi nó truyền qua nhiều lớp trong một chương trình.

```
public class ChainedException {  
    public static void main(String[] args) throws Exception {  
        int n = 20, result = 0;  
        try{  
            result = n / 0;  
            System.out.println("Result: " + result);  
        }catch(ArithmeticException eArith){  
            System.out.println("Arithmetic exception : "+eArith);  
            try{  
                throw new NumberFormatException();  
            }catch(NumberFormatException eNum){  
                System.out.println("Chained exception: "+eNum);  
            }  
        }  
    }  
}
```

Kết quả nhận được :

```
Arithmetic exception : java.lang.ArithmeticException: / by zero  
Chained exception: java.lang.NumberFormatException  
Press any key to continue . . .
```



# Generic





# Generic

Generics → 20E K @ 9 CA M < DA M

```
ArrayList<Integer> arr = new ArrayList<Integer>();  
arr.add(5);  
arr.add(35);  
arr.add("Java"); //error  
arr.add(true);   //error
```

**Generic**



## Một số quy ước đặt tên kiểu tham số generic

$E, K, V, T, N$	Ý nghĩa
E	Element (Phần tử)
K	Key (Khóa)
V	Value (Giá trị)
T	Type (Kiểu dữ liệu)
N	Number (Số)

# Lớp generic

Một lớp có thể tham chiếu bất kỳ kiểu đối tượng nào được gọi là lớp generic

```
public class MyGeneric<T> {  
    public T obj;  
    public T getObj() {  
        return obj;  
    }  
    public void add(T obj){  
        this.obj = obj;  
    }  
}
```

Lớp generic

# Lớp generic

Sử dụng kiểu  
Integer

```
public static void main(String[] args) {  
    //Use Integer  
    MyGeneric<Integer> myGeneric1 = new MyGeneric<Integer>();  
    myGeneric1.add(3);  
    System.out.println(myGeneric1.getObj());  
  
    //Use String  
    MyGeneric<String> myGeneric2 = new MyGeneric<String>();  
    myGeneric2.add("java");  
    System.out.println(myGeneric2.getObj());  
}
```


Sử dụng kiểu  
String



# Phương thức generic

Một phương thức trong class hoặc interface đều có thể sử dụng generic

Phương thức generic



```
public static <E> void printArray(E[] elements) {  
    for (E element : elements) {  
        System.out.print(element + " ");  
    }  
    System.out.println();  
}
```

```
public static void main(String[] args) {  
    Integer[] intArray = { 10, 20, 30, 40, 50 };  
    Character[] charArray = { 'J', 'A', 'V', 'A' };  
  
    System.out.print("Mang so nguyen: ");  
    printArray(intArray);  
  
    System.out.print("Mang ky tu: ");  
    printArray(charArray);  
}
```

Gọi tới phương thức  
generic



## Mảng generic

Có thể khai báo một mảng generic nhưng không thể khởi tạo mảng generic vì kiểu generic không tồn tại tại thời điểm chạy. Generic chỉ có tác dụng với trình biên dịch để kiểm soát code.

Khai báo

```
T[] arr; // Ok
```

```
T[] arr2 = new T[5]; // Error
```

Khởi tạo



## Thừa kế lớp generic

+sL ; D9KK E<sub>u</sub> J<sub>s</sub>F? L□ E<sub>s</sub>L ; D9KK ?=F=JA; K| Fε ; ε L@□ ; @<sub>u</sub> L<sub>u</sub> F@  
J→ CA□M ; @G L@E K<sub>u</sub> %=F=JA; K| ?A□ F?MQ<sub>u</sub>F ; Č; L@E K<sub>u</sub>  
%=F=JA; K @G□; L@E ; Č; L@E K<sub>u</sub> %=F=JA; K

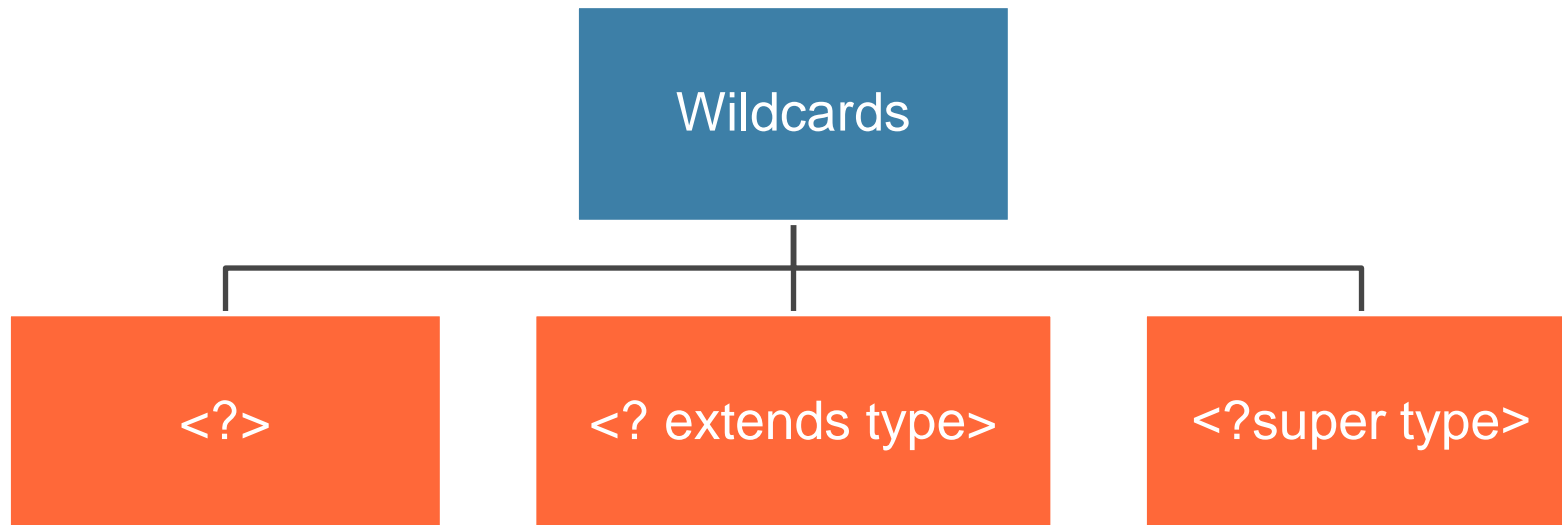
```
public abstract class AbstractParam<T> {  
    protected T value;  
    protected abstract void printValue();  
}
```

```
public class Email extends AbstractParam<String>{
    @Override
    protected void printValue() {
        System.out.println("My email is:"+value);
    }
}
```

```
public static void main(String[] args) {
    Email email = new Email();
    email.value = "ngoc@techmaster.vn";
    email.printValue();// My email is:ngoc@techmaster.vn
    email.value = 10; // Lỗi
}
```



## Các ký tự đại diện generic



# Ưu điểm của generics

01

Kiểu dữ liệu an toàn

02

Kiểm tra dữ liệu chặt chẽ ở Compile-time mà không phải là Runtime-error

03

Hạn chế việc ép kiểu (cast) thủ công mà không an toàn.

04

%A|H ; @f? L9 NA□L ; Ć; L@M L LGĈF Í7□; K□  
<□F? F@A□M <□ <· F? L@Q ÍA| 9F LG F <□ DA□M  
N· <□ L□; @f



## Hạn chế của generics

- Không thể gọi Generics bằng kiểu dữ liệu nguyên thủy (Primitive type: int, long, double, ...), thay vào đó sử dụng các kiểu dữ liệu Object (wrapper class thay thế: Integer, Long, Double, ...).
- Không thể tạo instances của kiểu dữ liệu Generics, thay vào đó sử dụng reflection từ class.
- Không thể sử dụng static cho Generics.
- Không thể ép kiểu hoặc sử dụng instanceof.
- Không thể tạo mảng với parameterized types.
- Không thể tạo, catch, throw đối tượng của parameterized types (Generic Throwable)
- Không thể overload các hàm trong một lớp





## Exercise

Tạo class có tên là `MyArray` và thực hiện các công việc sau:

- Thêm vào mảng một số nguyên
- Thêm vào mảng một số thực
- Thêm vào mảng một giá trị Boolean
- Thêm vào mảng một chuỗi
- In ra màn hình 4 giá trị trên



## Exercise

- 1, Tạo class tên Student có các thuộc tính như id, name, age. Viết các phương thức constructor, setter, getter, toString.
- 2, Tạo class tên Employee có các thuộc tính như id, name, salary. Viết các phương thức constructor, setter, getter, toString.

Tạo class PersonModel và thực hiện các công việc sau:

```
public class PersonModel<T> {  
    private ArrayList<T> al = new ArrayList<T>();  
  
    public void add(T obj) {  
        al.add(obj);  
    }  
  
    public void display() {  
        for (T object : al) {  
            System.out.println(object);  
        }  
    }  
}
```

Tạo đối tượng **PersonModel<Student>**

- Gọi phương thức **add** để nhập vào 2 sinh viên
- Gọi phương thức **display** để hiển thị thông tin của 2 sinh viên vừa nhập

Tạo đối tượng **PersonModel<Employee>**

- Gọi phương thức **add** để nhập vào 2 nhân viên
- Gọi phương thức **display** để hiển thị thông tin của 2 nhân viên vừa nhập



# Collections



# Collection là gì?



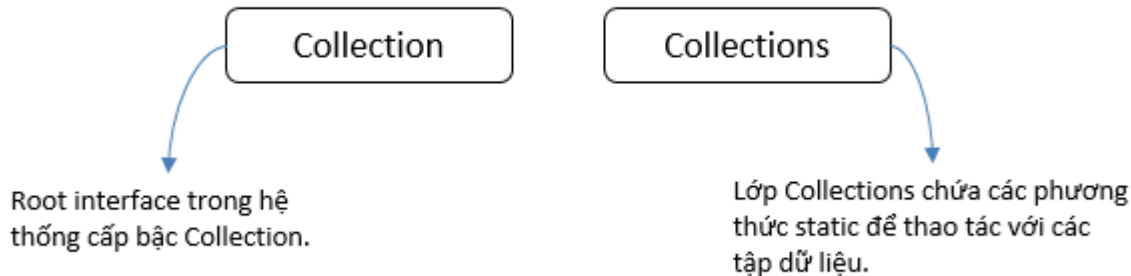
Collection là một framework cung cấp một kiến trúc để lưu trữ và thao tác với nhóm các đối tượng

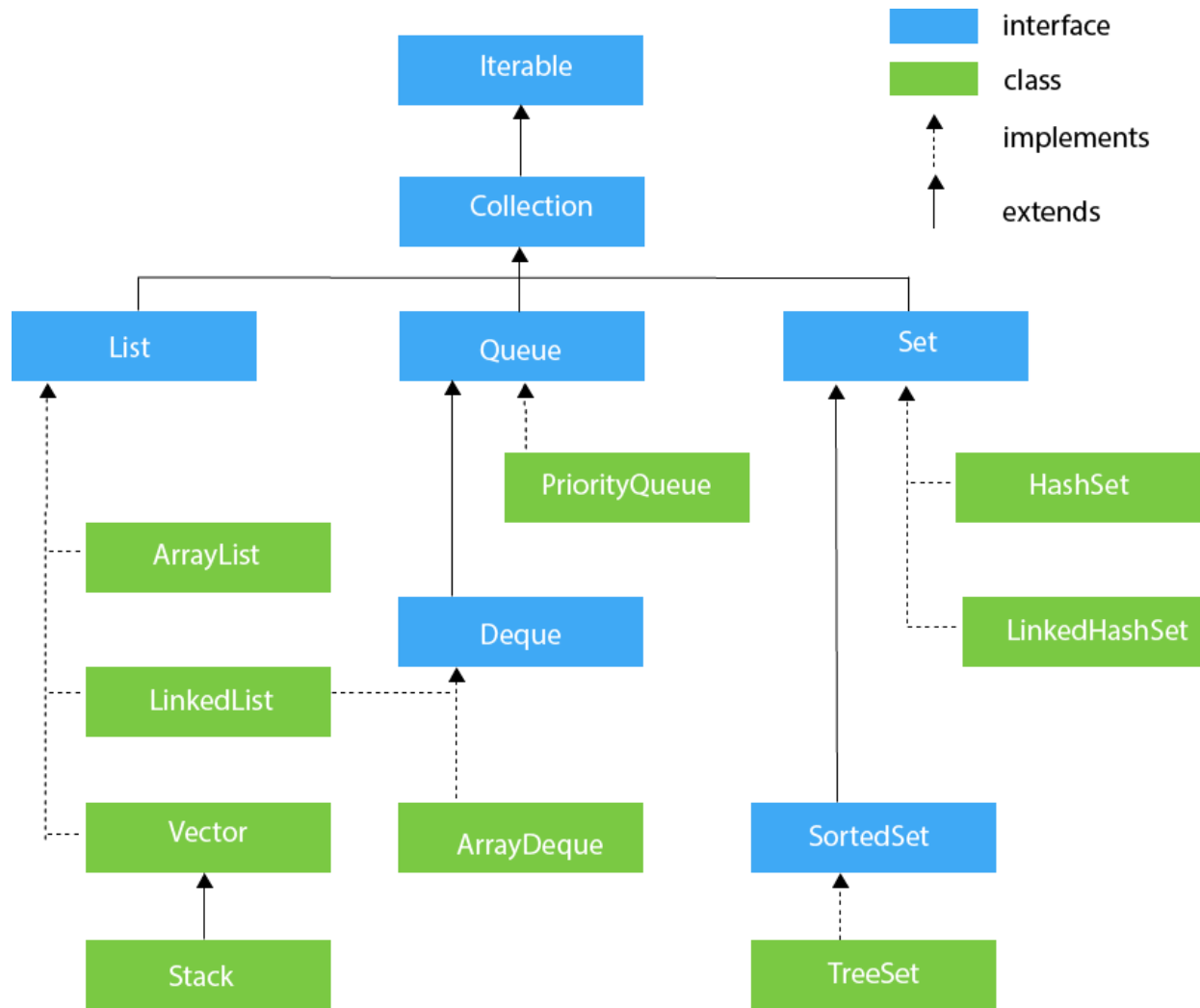
Java collections có thể đạt được tất cả các thao tác mà bạn thực hiện trên dữ liệu như tìm kiếm, sắp xếp, chèn, xóa

# Collection là gì?

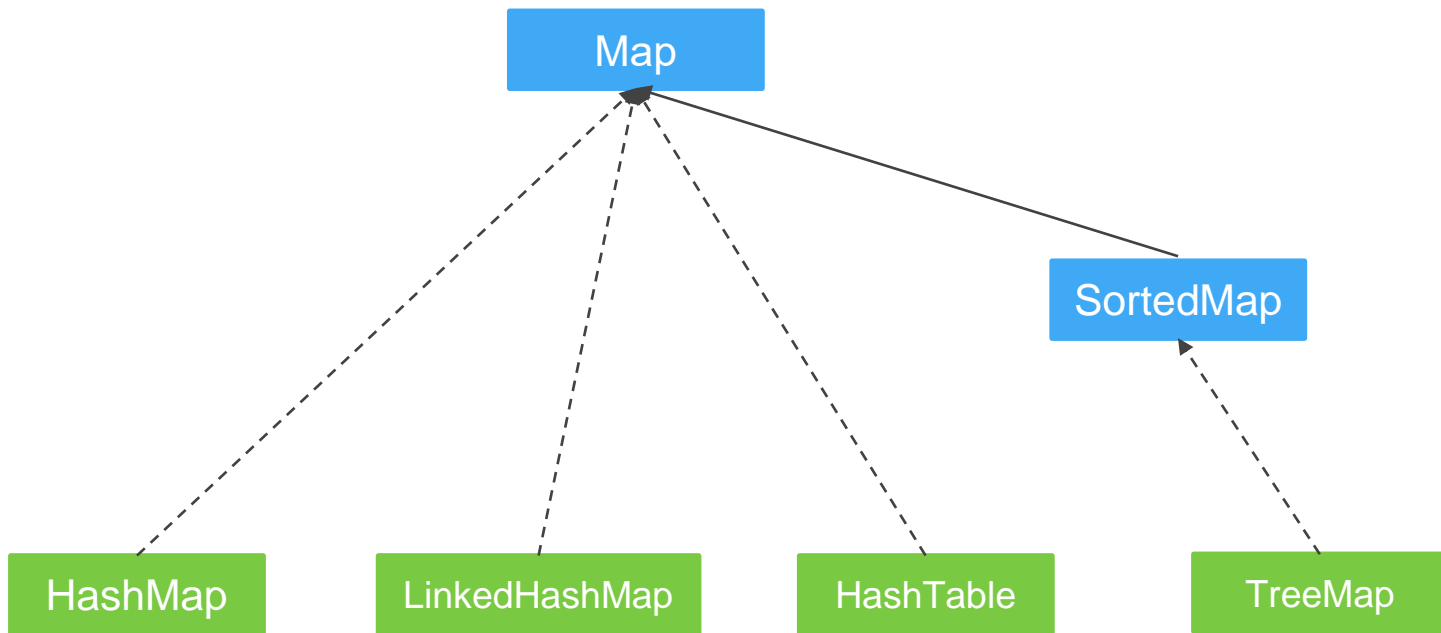


Java collection cung cấp nhiều interface (Set, List, Queue, Deque vv) và các lớp (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet,...)











# Iterable interface và iterator interface



**Iterable interface** chứa dữ liệu thành viên iterator interface



**Iterator interface** cung cấp phương tiện để lặp đi lặp lại các thành phần từ đầu đến cuối của một collection

# Các phương thức của iterator interface



Phương thức	Mô tả
public boolean hasNext()	Trả về giá trị true nếu iterator còn phần tử kế tiếp đang duyệt
public object next()	Trả về phần tử hiện tại và di chuyển con trỏ tới phần tử tiếp theo
public void remove()	Loại bỏ phần tử cuối được trả về bởi iterator

# Collection interface



Collection interface được thực hiện bởi tất cả các lớp trong Collection Framework. Nói cách khác, Collection interface là nền tảng mà Collection Framework phụ thuộc vào nó.

# List Interface

List Interface là giao diện con của Collection Interface. Nó ngăn cách cấu trúc dữ liệu kiểu danh sách trong đó chúng ta có thể lưu trữ tập hợp các đối tượng có thứ tự.

List Interface được thực hiện bởi các lớp ArrayList, LinkedList, Vector và Stack

Để khởi tạo List interface chúng ta sử dụng:

```
List <Kiểu dữ liệu> <Tên>= new ArrayList();  
List <Kiểu dữ liệu> <Tên> = new LinkedList();  
List <Kiểu dữ liệu> <Tên> = new Vector();  
List <Kiểu dữ liệu> <Tên> = new Stack();
```

# Set Interface

Set là kiểu dữ liệu mà bên trong nó mỗi phần tử chỉ xuất hiện duy nhất một lần và Set interface cung cấp các phương thức để thao tác với set

Set interface được kế thừa từ Collection Interface nên nó được cung cấp đầy đủ các phương thức của Collection Interface



# Set Interface



Một số class thực thi Set Interface thường gặp:

- **TreeSet**: là 1 class thực thi giao diện Set Interface, trong đó các phần tử trong set đã được sắp xếp.
- **HashSet**: là 1 class implement Set Interface, mà các phần tử được lưu trữ dưới dạng bảng băm (hash table).
- **EnumSet**: là 1 class dạng set như 2 class ở trên, tuy nhiên khác với 2 class trên là các phần tử trong set là các enum chứ không phải object.

# Queue Interface



Queue(Hàng đợi) là kiểu dữ liệu nổi tiếng với kiểu vào ra FIFO, tuy nhiên với Queue Interface thì queue không chỉ còn dừng lại ở mức đơn giản như vậy mà nó cung cấp cho bạn các phương thức để xây dựng các queue phức tạp hơn nhiều như priority queue, deque. Queue Interface cũng kế thừa và mang đầy đủ các phương thức từ Collection Interface.

- **LinkedList**: chính là LinkedList mình đã nói ở phần List
- **PriorityQueue**: là 1 dạng queue mà trong đó các phần tử trong queue sẽ được sắp xếp.
- **ArrayDeque**: là 1 dạng deque (queue 2 chiều) được implement dựa trên mảng



# Comparable Interface



Comparable interface được sử dụng để sắp xếp các đối tượng của lớp do người dùng định nghĩa. Interface này thuộc package `java.lang` và chỉ chứa một phương thức có tên `compareTo(Object)`

**`public int compareTo (Object obj)`**: được sử dụng để so sánh đối tượng hiện tại với đối tượng được chỉ định.

# Comparator Interface



Comparator interface được sử dụng để sắp xếp các đối tượng của lớp do người dùng định nghĩa. Interface này thuộc package `java.util` và chứa hai phương thức `compare(Object obj1, Object obj2)` và `equals(Object element)`.

**`public int compare(Object obj1, Object obj2)`**: so sánh đối tượng đầu tiên với đối tượng thứ hai.

# Map Interface



Map (đồ thị/ánh xạ) là kiểu dữ liệu cho phép ta quản lý dữ liệu theo dạng cặp key-value, trong đó key là duy nhất và tương ứng với 1 key là một giá trị value. Không giống như các interface ở trên, Map Interface không kế thừa từ Collection Interface mà đây là 1 interface độc lập với các phương thức của riêng mình.

# Class về Map

TreeMap

EnumMap

HashMap

WeakHashMap

Map

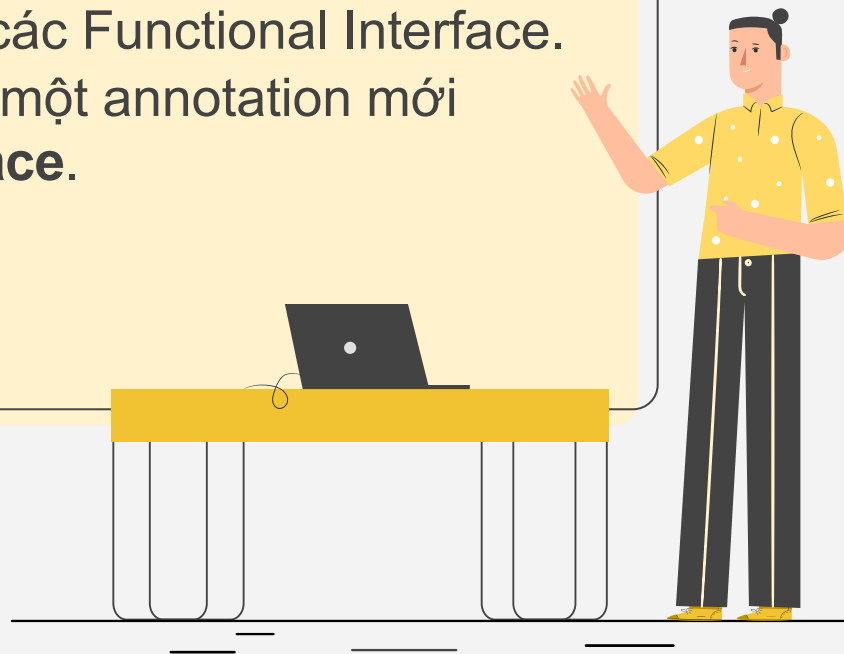


# **Lambda Expression & Functional Interface**



# FUNCTIONAL INTERFACE

- Java 8 gọi các interface có duy nhất một method trừu tượng là các Functional Interface.
- Java 8 cũng giới thiệu một annotation mới là **@FunctionalInterface**.



# Functional Interface

Ví dụ:

```
@FunctionalInterface
public interface DemoFunctionalInterface {
    void doSomething();
}
```

Annotation

Duy nhất một phương  
thức trừu tượng

Có thể thêm các phương thức không trừu tượng bằng từ khóa default và static

```
@FunctionalInterface
public interface DemoFunctionalInterface {
    void doSomething();
    default void defaultMethod1() {
    }
    default void defaultMethod2() {
    }
    static void staticMethod() {
    }
}
```

Từ khóa default

Từ khóa static



Sử dụng khi: Consumer thường được dùng với `InputStream` để xử lý các dữ liệu bên trong.

```
@FunctionalInterface
public interface Consumer<T> {

    /**
     * Performs this operation on the given argument.
     *
     * @param t the input argument
     */
    void accept(T t);
}
```

là một phương thức trừu tượng có thể được sử dụng với lambda expression hoặc method reference cho một mục đích cụ thể nào đó

# java.util.function.Predicate

Sử dụng khi:  Kiểm tra từng phần tử lúc xóa, lọc,... dùng với list, stream...

```
@FunctionalInterface
public interface Predicate<T> {

    /**
     * Evaluates this predicate on the given argument.
     *
     * @param t the input argument
     * @return {@code true} if the input argument matches the predicate,
     * otherwise {@code false}
     */
    boolean test(T t);
}
```



Kiểm tra một tham số đầu vào và trả về true hoặc false.

# java.util.function.Function

Sử dụng khi Function thường được dùng với Stream khi muốn thay đổi giá trị cho từng phần tử trong stream

```
@FunctionalInterface
public interface Function<T, R> {

    /**
     * Applies this function to the given argument.
     *
     * @param t the function argument
     * @return the function result
     */
    R apply(T t);
}
```

Method này nhận đầu vào là 1 tham số và trả về một giá trị.

# java.util.function Supplier

Supplier<T> được sử dụng với lambda expression hoặc method reference cho một mục đích cụ thể nào đó.

```
@FunctionalInterface
public interface Supplier<T> {

    /**
     * Gets a result.
     *
     * @return a result
     */
    T get();
}
```

method này không có tham số  
nhưng trả về một kết quả.



# Inner class

- **Lớp lồng nhau (inner class) trong java** là một lớp được khai báo trong lớp (class) hoặc interface khác.
- Chúng ta sử dụng inner class để nhóm các lớp và các interface một cách logic lại với nhau ở một nơi để giúp cho code dễ đọc và dễ bảo trì hơn.
- Thêm vào đó, nó có thể truy cập tất cả các thành viên của lớp bên ngoài (outer class) bao gồm các thành viên dữ liệu private và phương thức.



# Cú pháp

```
public class Main {  
    class InnerClass{  
  
    }  
}
```

```
public class OuterClass {  
    int x = 10;  
    class InnerClass{  
        int y = 5;  
    }  
}
```

Tạo đối tượng outerClass

```
public class Main {  
    public static void main(String[] args) {  
        OuterClass outerClass = new OuterClass();  
        OuterClass.InnerClass innerClass = outerClass.new InnerClass();  
        System.out.println(outerClass.x + innerClass.y);  
    }  
}
```

Tạo đối tượng innerClass

```
public class OuterClass {  
    static class InnerClass2{  
        int y = 20;  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        OuterClass.InnerClass2 InnerClass2 = new OuterClass.InnerClass2();  
        System.out.println(InnerClass2.y);  
    }  
}
```



# Anonymous Inner Class



Một lớp không có tên được gọi là lớp vô danh hay anonymous inner class. Nó nên được sử dụng nếu bạn phải ghi đè phương thức của lớp hoặc interface.

Anonymous inner class có thể được tạo bằng hai cách:

01

Class

02

Interface

# Khi nào nên sử dụng lớp vô danh



Lớp vô danh thường được sử dụng khi bạn không muốn phải khai báo cụ thể lớp con của một lớp nào đó, kể cả khi bạn không muốn khai báo cụ thể lớp triển khai của một interface nào đó , mà vẫn muốn sử dụng các đối tượng của chúng

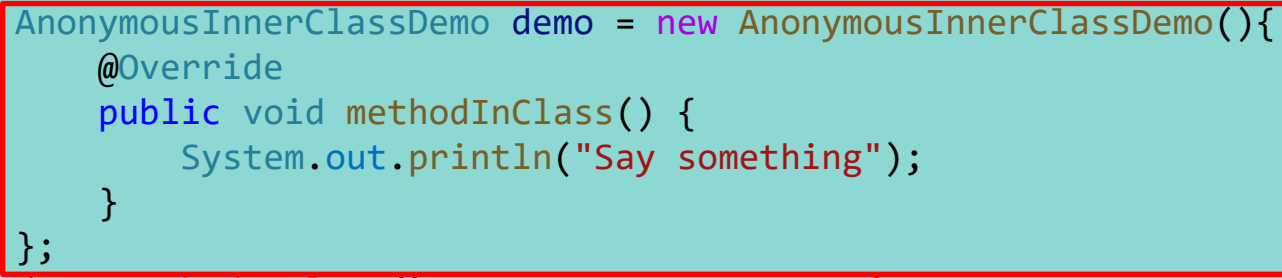
# Đặc điểm của lớp vô danh



- Lớp vô danh chỉ có thể triển khai từ duy nhất một interface
- Lớp vô danh chỉ có thể kế thừa hoặc triển khai một lớp khác hoặc một interface khác
- Lớp vô danh không có constructor

```
public abstract class AnonymousInnerClassDemo {  
    public abstract void methodInClass();  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        AnonymousInnerClassDemo demo = new AnonymousInnerClassDemo(){  
            @Override  
            public void methodInClass() {  
                System.out.println("Say something");  
            }  
        };  
        demo.methodInClass();  
    }  
}
```



Anonymous inner class with class

# Lambda Expression



Lambda Expression là một hàm không có tên với các tham số và nội dung thực thi. Nội dung thực thi của LE có thể là một khối lệnh hoặc 1 biểu thức

```
// Không có tham số, 1 câu lệnh  
( ) -> expression
```

```
// 1 tham số, 1 câu lệnh  
(parameters) -> expression
```

```
// các tham số và nội dung khối  
(arg1, arg2, ...) -> {  
    body-block  
}
```

```
// các tham số, nội dung khối, dữ liệu trả về  
(arg1, arg2, ...) -> {  
    body-block;  
    return return-value;  
}
```

```
public class Main {  
  
    public static void main(String[] args) {  
        ArrayList<Integer> numbers = new ArrayList<Integer>();  
        numbers.add(5);  
        numbers.add(9);  
        numbers.add(8);  
        numbers.add(1);  
  
        for (Integer integers : numbers) {  
            System.out.println(integers);  
        }  
    }  
}
```



Sử dụng for each thông thường

Biểu thức lambda thường được truyền tham số dưới dạng tham số cho một hàm

```
numbers.forEach((n) -> {  
    System.out.println(n);  
});
```

Sử dụng Lambda  
Expressions



```
//Sử dụng Comparator  
list.sort(new Comparator<String>(){  
    public int compare(String o1, String o2) {  
        return o2.compareTo(o1);  
    };  
});
```

```
//Sử dụng lambda  
list.sort((String o1, String o2) -> o1.compareTo(o2));
```

```
@FunctionalInterface
public interface AddNumber {
    public int add2Number(int a, int b);
}
```

```
public static void main(String[] args) {
    //Using anonymous inner class
    AddNumber addNumber = new AddNumber(){
        @Override
        public int add2Number(int a, int b) {
            return a+b;
        }
    };
}
```

```
public static void main(String[] args) {  
    //Using lambda  
    AddNumber addNumberUsingLambda = (a, b) -> {  
        return a+b;  
    };  
}
```

# Method Reference

Method References (Phương thức tham chiếu) cung cấp các cú pháp hữu ích để truy cập trực tiếp tới constructor hoặc method đã tồn tại của các lớp hoặc đối tượng trong java mà không cần thực thi chúng



# Method Reference

Method references là cú pháp viết tắt của biểu thức lambda để gọi phương thức. Ví dụ, nếu biểu thức lambda được viết như sau:

```
str -> System.out.println(str);
```

Có thể viết lại theo các của Method reference như sau:

```
System.out::println;
```

# Các loại Method Reference



- Tham chiếu đến một static method – `Class::staticMethod`
- Tham chiếu đến một instance method của một đối tượng cụ thể – `object::instanceMethod`
- Tham chiếu đến một instance method của một đối tượng tùy ý của một kiểu cụ thể – `Class::instanceMethod`
- Tham chiếu đến một constructor – `Class::new`

```
ArrayList<Person> list = new ArrayList<>();  
list.add(new Person("Ngoc", 25));  
list.add(new Person("Hoang", 30));  
list.add(new Person("Tuan", 27));  
list.add(new Person("Hoa", 20));
```

```
//Sử dụng lambda  
list.forEach(n -> System.out.println(n));
```

```
//Sử dụng method reference  
list.forEach(System.out::println);
```

```
public static int compareByAge(Person p1, Person p2){  
    return p1.getAge() - p2.getAge();  
}
```

```
//Sử dụng lambda  
Collections.sort(list, (o1, o2) -> Person.compareByAge(o1, o2));
```

```
//Sử dụng method reference  
Collections.sort(list, Person::compareByAge);
```





# Stream

# Stream là gì?



**Stream** (luồng) là một đối tượng mới của Java được giới thiệu từ phiên bản Java 8, giúp cho việc thao tác trên collection và array trở nên dễ dàng và tối ưu hơn.

Một Stream đại diện cho một chuỗi các phần tử hỗ trợ các hoạt động tổng hợp tuần tự (sequential) và song song (parallel).

# Các tính chất của Stream



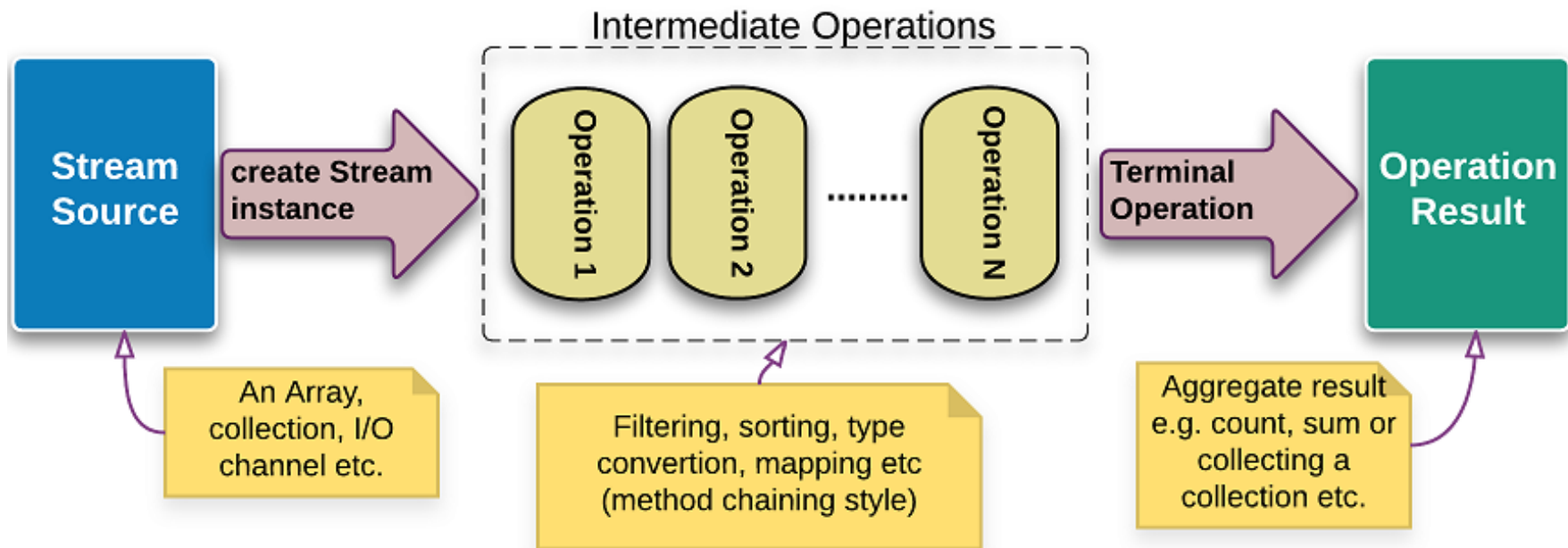
- Stream không phải một cấu trúc dữ liệu, đầu vào của Stream có thể là các collections(ArrayList, Set, LinkedList,...), Arrays và các kênh input/output
- Stream không làm thay đổi dữ liệu gốc mà chỉ trả về kết quả thông qua các methods
- Về cơ bản các method của Stream được chia thành 2 loại là hoạt động trung gian (intermediate Operation) và hoạt động đầu cuối (Terminal Operation)

Ví dụ:

```
Collection<Integer> collection = Arrays.asList(1,2,3);  
Stream<Integer> streamOfCollection = collection.stream();
```

```
public class StreamExample {
    List<Integer> numbers = Arrays.asList(7, 2, 5, 4, 2, 1);
    public void withoutStream() {
        long count = 0;
        for (Integer number : numbers) {
            if (number % 2 == 0) {
                count++;
            }
        }
        System.out.printf("There are %d elements that are even", count);
    }
    public void withStream() {
        long count = numbers.stream().filter(num -> num % 2 == 0).count();
        System.out.printf("There are %d elements that are even", count);
    }
}
```

# Java Streams





# Terminal Operations

Phương thức `collect()`: Dùng để trả về kết quả của Stream dưới dạng List hoặc Set

```
List<String> strings = Arrays.asList("args", "", "code", "learn", "...");  
List<String> filter = strings.stream().collect(Collectors.toList());  
System.out.println(filter);
```

Kết quả trả về:

```
[args, , code, learn, ...]
```

forEach(): dùng để duyệt mọi phần tử trong Stream

```
List<String> strings = Arrays.asList("args", "", "code", "learn", "...");  
strings.stream().forEach(s -> System.out.println(s));
```

Kết quả trả về:

args

code

learn

...



Reduce(): reduce() method với 1 trong 2 tham số truyền vào là method reference, dùng để kết hợp các phần tử thành một giá trị đơn cùng kiểu với dữ liệu ban đầu

```
List<String> strings = Arrays.asList("args", "", "code", "learn", "...");  
String result = strings.stream().reduce("-", String::concat);  
System.out.println(result);
```

Tham số đầu tiên là chỉ giá trị ban đầu, tham số thứ 2 là một method reference String::concat nhằm mục đích ghép các phần tử của Stream với nhau

```
-argscodelearn...
```

max(), min(): Trả về giá trị lớn nhất hoặc bé nhất trong các phần tử

```
List<Integer> list = Arrays.asList(1,2,3,4,5,6,7,8,9,10);  
Integer maxx = list.stream().max(Integer::compare).get();  
Integer minn = list.stream().min(Integer::compare).get();  
System.out.println("Max: "+maxx);  
System.out.println("Min: "+minn);
```

Kết quả trả về:

```
Max: 10  
Min: 1
```



# Intermediate Operations

`distinct()`: dùng để loại bỏ các phần tử trùng lặp

```
List<Integer> list = Arrays.asList(1,2,2,2,2,3,4,5);  
list.stream().distinct().forEach(System.out::println);
```

Kết quả trả về:

```
1  
2  
3  
4  
5
```

Map(): dùng để trả về một Stream mà ở đó các phần tử đã được thay đổi theo cách người dùng tự định nghĩa

```
List<Integer> list = Arrays.asList(1,2,2,2,2,3,4,5);  
list.stream()  
.distinct()  
.map(i -> i*i)  
.forEach(System.out::println);
```

Kết quả trả về:

```
1  
4  
9  
16  
25
```

filter(): dùng để lọc và xóa bỏ các phần tử với điều kiện do người dùng định nghĩa

```
List<Integer> list = Arrays.asList(1,2,2,2,2,3,4,5);  
list.stream()  
  .distinct()  
  .filter(i -> i > 2)  
  .forEach(System.out::println);
```

Kết quả trả về:

```
3  
4  
5
```

sorted(): dùng để sắp xếp các phần tử

```
List<Integer> list = Arrays.asList(4,3,2,1,0,3,4,5);  
list.stream().sorted().forEach(System.out::println);
```

Kết quả trả về:

```
0  
1  
2  
3  
3  
4  
4  
5
```

limit(n): với tham số truyền vào là số nguyên không âm n, nó sẽ trả về một stream chứa n phần tử đầu tiên

```
List<Integer> list = Arrays.asList(4,3,2,1,0,3,4,5);  
list.stream().limit(3).forEach(System.out::println);
```

Kết quả trả về:

```
4  
3  
2
```

skip(n): với tham số truyền vào là một số không âm n, nó sẽ trả về các phần tử còn lại đằng sau n phần tử đầu tiên

```
List<Integer> list = Arrays.asList(4,3,2,1,0,3,4,5);  
list.stream().skip(3).forEach(System.out::println);
```

Kết quả trả về:

```
1  
0  
3  
4  
5
```



# Tạo Stream cho kiểu primitive

```
public class StreamExample {  
    public static void main(String[] args) {  
        IntStream.range(1, 4).forEach(System.out::println);  
        IntStream.of(1, 2, 3).forEach(System.out::println);  
        DoubleStream.of(1, 2, 3).forEach(System.out::println);  
        LongStream.range(1, 4).forEach(System.out::println);  
        LongStream.of(1, 2, 3).forEach(System.out::println);  
    }  
}
```

# Tạo Stream từ các cấu trúc dữ liệu khác

```
public class StreamExample {  
    public static void main(String[] args) {  
        String[] languages = { "Java", "C#", "C++", "PHP", "Javascript" };  
  
        // Get Stream using the Arrays.stream  
        Stream<String> testStream = Arrays.stream(languages);  
        testStream.forEach(x -> System.out.println(x));  
    }  
}
```

# Đọc ghi file với Buffered Stream

Buffered Stream được dùng để tăng tốc độ hoạt động I/O, bằng cách đơn giản là tạo ra khoảng nhớ đệm với kích thước cụ thể nào đó.

Một chương trình có thể chuyển đổi từ không sử dụng Buffered Stream sang sử dụng Buffered Stream bằng việc sử dụng ý tưởng “Wrapping”

# Lớp wrapper

Lớp wrapper cho byte stream

- BufferedInputStream
- BufferedOutputStream

Lớp wrapper cho character stream

- BufferedReader
- BufferedWriter

## Đọc file với BufferedReader

```
import java.io.BufferedReader;
import java.io.FileReader;

public class ReadFile {
    public static void main(String[] args) throws Exception {
        FileReader fr = new FileReader("input.txt");
        BufferedReader br = new BufferedReader(fr);

        int i;
        while ((i = br.read()) != -1) {
            System.out.print((char) i);
        }
        br.close();
        fr.close();
    }
}
```

## Ghi file với BufferedWriter

```
import java.io.BufferedWriter;
import java.io.FileWriter;

public class WriteFile {
    public static void main(String[] args) throws Exception {
        FileWriter writer = new FileWriter("D:\\testout.txt");
        BufferedWriter buffer = new BufferedWriter(writer);
        buffer.write("Welcome to java.");
        buffer.close();
        System.out.println("Success...");
    }
}
```



# Multi-threading



# Processes & threads

Đầu tiên khi nói về lập trình đa luồng, chúng ta cần phân biệt được 2 khái niệm Process (tiến trình) và Thread (luồng) trong Java:

## Process

Một chương trình đang chạy được gọi là một **process**.

## Thread

Một chương trình chạy có thể có nhiều **thread**, Cho phép chương trình đó chạy trên nhiều luồng một cách "đồng thời".





# Multi - thread

Multi-thread (đa luồng) là một tiến trình thực hiện nhiều luồng đồng thời. Một ứng dụng ngoài luồng chính có thể có các luồng khác thực thi đồng thời làm ứng dụng chạy nhanh và hiệu quả.



## Ưu điểm

- Ta có thể thực hiện nhiều công việc cùng một lúc với luồng
- Mỗi luồng có thể dùng chung và chia sẻ nguồn tài nguyên trong quá trình chạy
- Luồng là độc lập vì vậy nó không ảnh hưởng đến luồng khác nếu ngoại lệ xảy ra trong một luồng duy nhất
- Giúp tiết kiệm thời gian



## Nhược điểm

- Càng nhiều luồng xử lý càng phức tạp
- Xử lý vấn đề về tranh chấp bộ nhớ, đồng bộ dữ liệu khá phức tạp
- Cần phát hiện tránh các luồng chết (dead lock), luồng chạy mà không làm gì trong ứng dụng cả



# Tạo thread

Có hai cách để tạo một thread:

- Cách 1 – Kế thừa từ lớp Thread
- Cách 2 – Triển khai từ Interface Runnable

## Kế thừa từ lớp Thread

```
public class CreateThread extends Thread {  
    //Nội dung  
}
```

## Triển khai từ Interface Runnable

```
public class CreateThread implements Runnable {  
    //Nội dung  
}
```

# Một số phương thức thường được sử dụng

Public void run() : Sử dụng để thực hiện hành động cho một luồng

```
public class CreateThread extends Thread {  
    @Override  
    public void run() {  
        System.out.println("This code is running in a thread");  
    }  
}
```

```
public class CreateThread implements Runnable {  
    @Override  
    public void run() {  
        System.out.println("This code is running in a thread");  
    }  
}
```

public void start() : Bắt đầu thực thi luồng. JVM gọi phương thức run() trên luồng

Kế thừa lớp Thread

```
public class CreateThread extends Thread {  
    @Override  
    public void run() {  
        System.out.println("This code is running in a thread");  
    }  
    public static void main(String[] args) {  
        CreateThread createThread = new CreateThread();  
        createThread.start();  
    }  
}
```

Tạo đối tượng  
createThread

Bắt đầu khởi  
chạy Thread



## Triển khai từ Interface Runnable

```
public class CreateThreads implements Runnable {  
    @Override  
    public void run() {  
        System.out.println("This code is running in a thread");  
    }  
    public static void main(String[] args) {  
        CreateThreads createThread = new CreateThreads();  
        Thread thread = new Thread(createThread);  
        thread.start();  
    }  
}
```

Luồng được chạy bằng cách chuyển một thể hiện của lớp tới constructor của Thread và gọi phương thức start() của luồng



## Tạm dừng thread


Phương thức `sleep()` của lớp `Thread` được sử dụng để tạm ngưng một thread cho một khoảng thời gian nhất định.

Lớp `Thread` cung cấp hai phương thức để tạm ngưng một thread:

- `public static void sleep(long milliseconds) throws InterruptedException`
- `public static void sleep(long milliseconds, int nanos) throws InterruptedException`

Hãy chạy thử đoạn code này

```
@Override
public void run() {
    for(int i = 0; i < 5; i++){
        System.out.println(i + " This code is running in a thread");
        try {
            Thread.sleep(1000);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    System.out.println("Done!!!");
}
```



Truyền 1000 mili giây

# Join các thread



Phương thức `join()` chờ một thread chết. Nói cách khác, nó làm cho các thread đang chạy ngừng hoạt động cho đến khi luồng mà nó tham gia hoàn thành nhiệm vụ của nó.

- `public void join()throws InterruptedException`
- `public void join(long milliseconds)throws InterruptedException`

```
public static void main(String[] args) {  
    CreateThread createThread = new CreateThread();  
    CreateThread createThread2 = new CreateThread();  
    createThread.start();  
    try {  
        createThread.join();  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
  
    createThread2.start();  
    try {  
        createThread2.join();  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}
```



## Gián đoạn thread

Interrupt một thread trong java hay làm gián đoạn một luồng trong java. Nếu thread nằm trong trạng thái sleep hoặc wait (nghĩa là sleep() hoặc wait() được gọi ra), việc gọi phương thức interrupt() trên thread đó sẽ phá vỡ trạng thái sleep hoặc wait và ném ra ngoại lệ InterruptedException.

- `public void interrupt()`
- `public static boolean interrupted()`
- `public boolean isInterrupted()`

Ví dụ về interrupt một thread không làm thread ngừng hoạt động

```
public class CreateThread extends Thread {  
    @Override  
    public void run() {  
        try{  
            for(int i = 0; i < 5; i++){  
                System.out.println("Child Thread executing");  
  
                Thread.sleep(1000);  
            }  
        }catch (InterruptedException e){  
            System.out.println("InterruptedException occur");  
        }  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) throws InterruptedException {  
        CreateThread thread = new CreateThread();  
  
        thread.start();  
        thread.interrupt();  
    }  
}
```

## Output

```
Child Thread executing  
InterruptedException occur
```



Ví dụ về interrupt một thread khiến thread ngừng hoạt động

```
@Override
public void run() {
    try {
        Thread.sleep(1000);
        System.out.println("Test!");
    } catch (InterruptedException e) {
        throw new RuntimeException("Thread interrupt ...\n" + e);
    }
    System.out.println("Thread is running ...");
}
```

```
public static void main(String[] args) {  
    CreateThreads createThread = new CreateThreads();  
    Thread thread = new Thread(createThread);  
    thread.start();  
    try {  
        thread.interrupt();  
    } catch (Exception e) {  
        System.out.println("Exception handled \n"+e);  
    }  
}
```

Exception in thread "Thread-0" java.lang.RuntimeException: Thread interrupt ...  
java.lang.InterruptedIOException: sleep interrupted  
at CreateThreads.run(CreateThreads.java:8) CreateThreads.java:8  
at java.base/java.lang.Thread.run(Thread.java:832) Thread.java:832

## Ví dụ về interrupt một thread mà thread hoạt động bình thường

```
public class CreateThreads implements Runnable {
    @Override
    public void run() {
        for(int i = 0; i < 5; i++){
            System.out.println(i);
        }
    }

    public static void main(String[] args) {
        CreateThreads createThread = new CreateThreads();
        Thread thread = new Thread(createThread);
        thread.start();
        thread.interrupt();
    }
}
```

Viết chương trình tạo một thread đếm ngược 10 giây. Khi start Thread sẽ in ra console giá trị 10, sau mỗi giây Nó sẽ giảm 1 đơn vị cho đến khi bằng 0 thì in ra hết giờ.