1. **CALLBACK trong JAVA**

* callback là một đoạn code chạy được (thường là một hàm A) được sử dụng như tham số truyền vào của hàm B nào đó. Hàm A được gọi ngay lập tức hoặc trễ một chút sau khi hàm B được gọi. Các ngôn ngữ lập trình khác nhau hỗ trợ callback theo các cách khác nhau, thường được triển khai dưới dạng chương trình con, hàm nặc danh, chuỗi lệnh hoặc con trỏ hàm.
* Callback (gọi ngược trở lại) có thể giải thích theo một số case trong đời sống như sau:  
  1- Asynchronous callback (bất đồng bộ)  
  Bạn nghi ngờ vợ ngoại tình và muốn có chứng cớ để còn mà oánh bỏ mẹ cái thằng dám “mèo chuột” nữ nhân của mình. Nhưng mà theo dõi 3 ngày rồi mà chẳng thấy gì cả mặc dù biểu hiện ở nhà của vợ nhìn rất chi là ngứa mắt. Mà giờ xin nghỉ cả tháng đi theo dõi vợ thì ai đi làm nuôi con bây giờ. Giải pháp?
* Thuê 1 thằng thám tử, còn mình thì đi làm, khi nào có “event: vào nhà nghỉ” thì thằng thám tử sẽ nhấc phone, alo để mình phi tới xin tý “huyết”.
* **class** **Event** {
* }
* **interface** **ThamTu** {
* **void** **phoneToHusband**();
* }
* **class** **ThamTuConan** **implements** **ThamTu** {
* **private** Event **event** = **new** Event();
* **public** **ThamTuConan**(){
* monitor();
* somthingHappen(**event**);
* }
* **public** **void** **phoneToHusband**() {
* *//check event*
* **if**(**event** == **event**.VO\_HOTEL){
* System.**out**.println("Đại bàng gọi thợ săn, thỏ con chui vào hang. Mang gậy tới mần thịt nhé.");
* }
* }
* }
* **class** **Husband** {
* **public** **void** **registerEnterHotelEvent**(ThamTu thamtu) {
* thamtu.phonetoHusband();
* }
* **public** **static** **void** **main**(String[] args) {
* Husband chong = **new** Husband();
* ThamTu conan = **new** ThamTuConan();
* chong.registerEnterHotelEvent(conan); *//register thám tử còn mình thôi monitor*
* chong.continueWorkingtoFeedChildren(); *// tiếp tục đi làm kiếm tiền nuôi con*
* }
* }
* Ví dụ này bạn có thể liên hệ với EventListener trong lập trình Swing vậy. Bất cứ khi nào có sự kiện click trên Button, thì cái callback function trong EventListener sẽ được fire (trong Java không có khái niệm function callback, nên để lập trình hướng sự kiện event-driven programming buộc phải thiết kế object/interface callback. Nếu trong javascript thì khái niệm callback này sẽ dễ hiểu hơn.
* $('#btnEnterHotel').click(phoneToHusband);
* **function** phoneToHusband(){
* alert("Đại bàng gọi thợ săn, thỏ con chui vào hang. Mang gậy tới mần thịt nhé.");
* }
* Hàm click sẽ gọi một callback function khi có sự kiện bấm trên nút btnEnterHotel.

2. Synchronous callback  
Theo định nghĩa của callback thì nó là một đoạn code được truyền như là tham số tới một đoạn code khác, và sẽ được gọi tại một thời điểm nào đó. Khác với async callback, sync callback sẽ được gọi ngay khi xử lý (thường được gọi vào cuối hàm, function call at the back ^^, nhất là trong lập trình Win32API, thường thì khi cuối hàm, bạn muốn OS làm một magic task nào đó, bạn sẽ viết kiểu generic vậy để lập trình viên có thể chọn: ví dụ như clear RAM, báo thức, …).

* Bạn có thể map nó với 1 trường hợp trong thực tế như sau:  
  Bạn đi vệ sinh, trong lúc đó vì quá rảnh rỗi, bạn muốn tận dụng thời gian làm một cái gì đó nữa, ví dụ như chơi game, xem siếc, … Bạn sẽ define các function này và tuỳ vào context mà gọi nó ra (ví dụ đang chơi CoC thì buồn đi iiiii …, nếu giờ không oánh nữa thì chúng nó cướp hết resource, vậy ta vào và chơi tiếp)
* **function** diWC(i, callbackFn) {
* i++; *// đếm cừu*
* callbackFn();
* }
* **function** playCoC() {
* alert("Chơi tiếp");
* }
* **function** seeSiec() {
* alert("Xem tiếp");
* }
* *//context: sợ bị cướp tài nguyên*
* diWC(1, playCoC);
* *//context: xa vợ lâu ngày*
* diWC(1, seeSiec);
* Cái này được thiết kế khá nhiều trong Java, ví dụ như thực hiện so sánh các phần tử trong ArrayList, bạn truyền 1 callback interface Comparable vào (ArrayList được gọi là high-level layer, còn Comparable được gọi là low-level layer). Bạn thích so sánh kiểu gì thì làm, ArrayList không quan tâm, miễn là sau khi so sánh ta có một ArrayList được sắp xếp. Đây cũng là một design pattern khá thông dụng và được áp dụng nhiều.

# **ENUM**

* TỔNG QUAN:
* **enum**là một từ khóa trong Java, là một kiểu dữ liệu đặc biệt được sử dụng để đại diện cho hằng số cố định.
* Một enum có thể chứa các trường, phương thức và constructor.
* Bởi vì các giá trị của Enum là các hằng số, nên tên của các trường kiểu enum thường là các chữ cái hoa.
* Khai báo ENUM
  + Trong Java, enum  có thể được định nghĩa bên trong hoặc bên ngoài một lớp, vì nó tương tự như lớp trong java

public class EnumExample {

    enum WeekDay {

        MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;

    }

    public static void main(String[] args) {

        WeekDay d = WeekDay.MONDAY;

            System.out.println(d);//result: MONDAY

    }

}

Lưu ý: enum định nghĩa bên ngoài một lớp không thể dụng access modifier public, protected, private.

* Duyệt các phần tử trong enum
  + Chúng ta có thể duyệt trên tất cả các phần tử của Enum, thông qua method**values()**.
  + Trình biên dịch trong java tự động thêm phương thức values() vào enum khi nó được biên dịch. Phương thức values() trả về một mảng chứa tất cả các giá trị của enum.

for (WeekDay d : WeekDay.values()) {

            System.out.println(d);

        }

1. **SERIALIZATION và DESERIALIZATION**

3.1 Khái niệm

* Tuần tự hóa là 1 cơ chế để ghi trạng thái vào 1 đối tượng vao một byte stream mục đích cho việc lưu trữ dữ liệu vào disk, database hoặc gửi trạng thái của object qua network
* Deserialization là quá trình tái thiết lại các đối tượng từ trạng thái serialized. Đây là hoạt động ngược lại của serialization.

3.2 Đặc điểm

* Sử dụng chủ yếu để truyền trạng thái của 1 đối tượng qua mạng ( phục vụ cho việc đọc ghi các dữ liệu là đối tượng JAVA)
* một Object có thể được lưu trực tiếp vào một file và có thể đọc từ file để có được một Object hoàn chỉnh như ban đầu chỉ với 2 phương thức đơn giản writeObject() và readObject().
* Nếu không có serialization, chúng ta phải lưu từng field của đối tượng vào file như một plain text, sau đó để khôi phục lại giá trị ban đầu của đối tượng, chúng ta phải đọc dữ liệu từ file và tạo một object mới sau đó set giá trị cho từng field của object.
* Nếu một lớp con thừa kế Serializable thì tất cả các lớp con của nó cũng được thừa kế Serializable
* Nếu 1 lớp A có chứa đối tượng tham chiếu lớp B thì Class B phải được implement Serializable nếu muốn thực hiện tuần tự hóa cho các đối tượng lớp A
* Nếu 1 Class có một thuộc tính là static thì đối tượng của lớp đó không thể serializeble vì static là thành phần của lớp chứ ko phải đối tượng ( khi thực hiện deserializable đối tượng của lớp này sẽ ko xuất hiện thuộc tính static)
* Trường hợp Array hoặc Collection: tất cả các đối tượng trong đó phải được serializable nếu muốn đọc ghi Array hoặc Collection đó
* Interface **Externalizable** cung cấp khả năng viết trạng thái của một đối tượng vào một byte stream ở định dạng nén. Nó không phải là một giao diện đánh dấu.
* Nếu không muốn serializable bất kì một thuộc tính nào đó của 1 lơp ta có thể đánh dấu thuộc tính đó với từ khóa *transient*

3.3 Ưu điểm và nhược điểm

- Tại sao phải dùng Serialization:

* Communication: If you have two machines that are running the same code, and they need to communicate, an easy way is for one machine to build an object with information that it would like to transmit, and then serialize that object to the other machine. It's not the best method for communication, but it gets the job done.
* Persistence: If you want to store the state of a particular operation in a database, it can be easily serialized to a byte array, and stored in the database for later retrieval.
* Deep Copy: If you need an exact replica of an Object, and don't want to go to the trouble of writing your own specialized clone() class, simply serializing the object to a byte array, and then de-serializing it to another object achieves this goal.
* Caching: Really just an application of the above, but sometimes an object takes 10 minutes to build, but would only take 10 seconds to de-serialize. So, rather than hold onto the giant object in memory, just cache it out to a file via serialization, and read it in later when it's needed.
* Cross JVM Synchronization: Serialization works across different JVMs that may be running on different architectures.

1. **Concurrency**

**4.1 Thread**

Là một luồng thực thi độc lập trong mỗi chương trình

* [JVM](https://shareprogramming.net/jvm-va-su-khac-biet-giua-jvm-jdk-jre/) sẽ cung cấp cho mỗi thread một method-call stack để theo dõi quá trình thực thi của mỗi thread. Ngoài việc theo dõi đang thực thi mã code đến đâu thì method-call stack còn theo dõi các biến local, tham số truyền vào và kết quả trả về của từng method trong stack.
* Methob-call stack là một stack lưu trữ các method được gọi trong quá trình thực thi 1 thread
* Mặc định sẽ có 1 main thread, nếu có nhiều hơn 1 stack thì chương trình sẽ được gọi là đa luồng
* Bản chất của thread object: object thread không phải là 1 luồng chạy của chương trình, chúng chỉ là 1 bản mô tả tên và mã code của 1 thread
* Khi method start() được gọi, một thread sẽ được tạo mới và thực thi code bên trong mẹthod run()
* Thread name: mặc định java sẽ đặt tên cho các thread, ta có thẻ đặt tên cho thread thông qua contructor() hoặc setName()
* Thread.sleep: dừng luồng chạy trong1 khoảng thời gian, bắt buộc phải đặt trong try catch
* Timelife của thread:
* Khi chương trình gọi start() method của một thead, JVM coi như thead này bắt đầu chu kỳ sống cho đến khi run() được gọi thực thi và kết thúc. Sau khi run() method thực thi xong, JVM sẽ tiến hành dọn dẹp, quá trình này sẽ tốn một khoảng thời gian, trong khoảng thời gian này JVM xem như thead đã hết chu kỳ sống.
* Join(): chờ 1 thread thực hiện xong. Chú ý không thực hiện join() bên trong chính thread của nó
* So sánh Thread và Runable

| **Sr. No.** | **Key** | **Thread** | **Runnable** |
| --- | --- | --- | --- |
| 1 | Basic | Thread is a class. It is used to create a thread | Runnable is a functional interface which is used to create a thread |
| 2 | Methods | It has multiple methods including start() and run() | It has only abstract method run() |
| 3 |  | Each thread creates a unique object and gets associated with it | Multiple threads share the same objects. |
| 4 | Memory | More memory required | Less memory required |
| 5 | Limitation | Multiple Inheritance is not allowed in java hence after a class extends Thread class, it can not extend any other class | If a class is implementing the runnable interface then your class can extend another class. |

## 4.2. ThreadPoolExecutor

* Được implement từ Executor và ExecutorService
* Tách việc tạo và thực thi các thread thành 2 phần riêng biệt
* Khi làm việc với ThreadPoolExecutor chúng ta chỉ cần triển khai task bằng cách implement Runnable interface và gửi chúng vào executor, nó sẽ chịu trách nhiệm cho việc thực thi, khởi tạo và chạy các thread cần thiết để hoàn thành các task.
* CorepoolSize: kích thước cố định của threadPool
* maxPoolSize; thread pool có thể khởi tạo thêm một số thread tối đa lên đến **maximunPoolSize**để tăng hiệu suất làm việc. Các thread được tạo thêm này sẽ được chấm dứt khi không còn cần đến, và giữa lại số **corePoolSize**thread.
* **keepAliveTime** là khoảng thời gian cho phép các thread được thread pool tạo thêm được phép tồn tại khi ở trạng thái không hoạt động

## 4.3 ScheduledThreadPoolExecutor

* ScheduledThreadPoolExecutot extends ThreadPoolExecutor và implement ScheduledExecotorService
* Một số chức năng khác:
  + *schedule()* method cho phép thực thi một task sau một khoảng thời gian delay xác định.
  + *scheduleAtFixedRate() method*cho phép thực thi một task sau một khoảng thời gian delay xác định tính từ thời điểm khởi tạo, sau đó thực thi task lặp đi lặp lại trong một khoảng thời gian định trước, khoảng thời gian này được tính từ lúc bắt đầu task trước đó đến thời gian bắt đầu task tiếp theo.
  + *scheduleWithFixedDelay()*method tương tự *scheduleAtFixedRate()*nhưng khoảng thời gian thực thi giữa 2 task được tính đừ lúc kết thúc task trước đi đến khi bắt đầu task kế tiếp.

## 4.4 DeadLock trong java

* DeadLock trong java là một trạng thái trong đó 2 hoặc nhiều thread rơi vào tình trạng chờ đợi lẫn nhau vì mỗi thread giữ một tài nguyên và chờ đợi tài nguyên từ thread khác. Ví dụ ThreadA giữ tài nguyên A và chời đợi tài nguyên B đang bị ThreadB nắm giữ, trong lúc đó ThreadB lại chờ đợi ThreadA trả tài nguyên A để sử dụng dẫn đến ThreadA và ThreadB chờ đợi lẫn nhau mãi mãi.

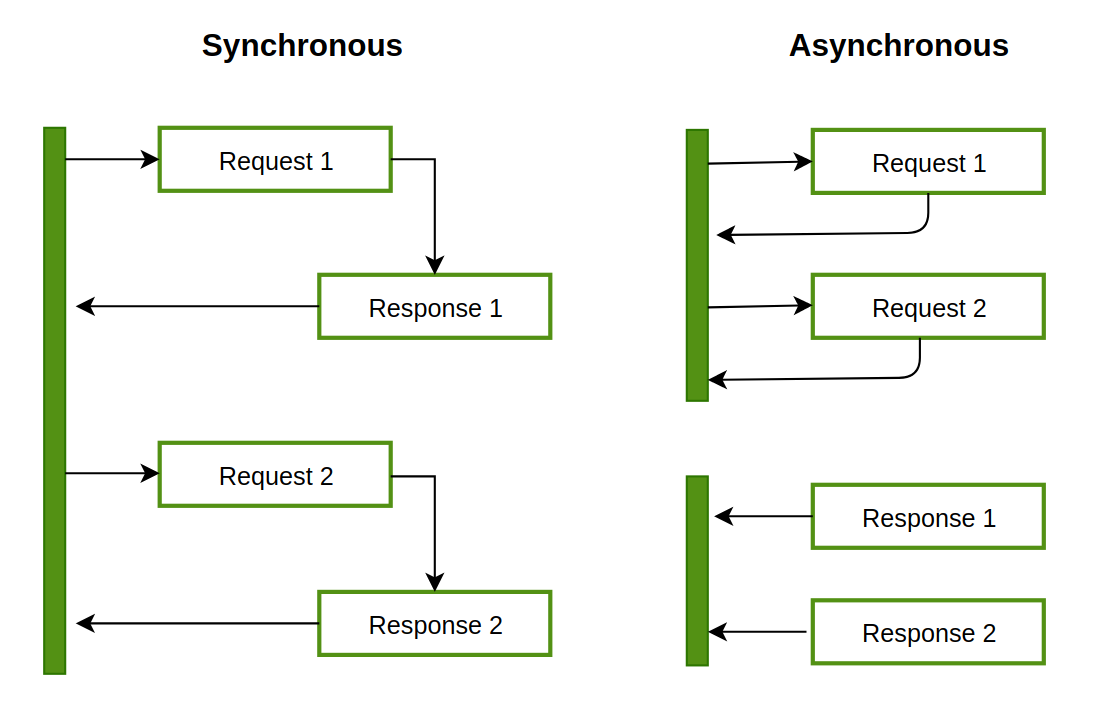
## Deamon thread trong JAVA

* Khái niệm: Deamon Thread là các luông được JVM liệt kê vào luông có độ ưu tiên thấp, chạy trên nền ứng dụng và không làm ảnh hưởng đến các luồng khác, làm nhiệm vụ dọn rác …
* Đặc điểm: Sự khac nhau giữa deamon Thread và user Thread là ở cách thức chúng ngưng hoạt động: Khi các luồng thông thường kết thúc thì tất cả các luồng deamon cũng sẽ kết thúc bất kể chúng đang làm gì
* Cách tạo luông deamon:
  + Các luông con sẽ thừa hưởng đặc tính deamon từ luông cha ( khi luông cha là deamon thì luông còn cũng sẽ là deamon và ngược lại)
  + Thiết lập 1 luông là deamon bằng phương thức setDeamon(boolean), chỉ gọi hàm này khi thread chưa được chạy

## Asynchronous And Multi-Threading

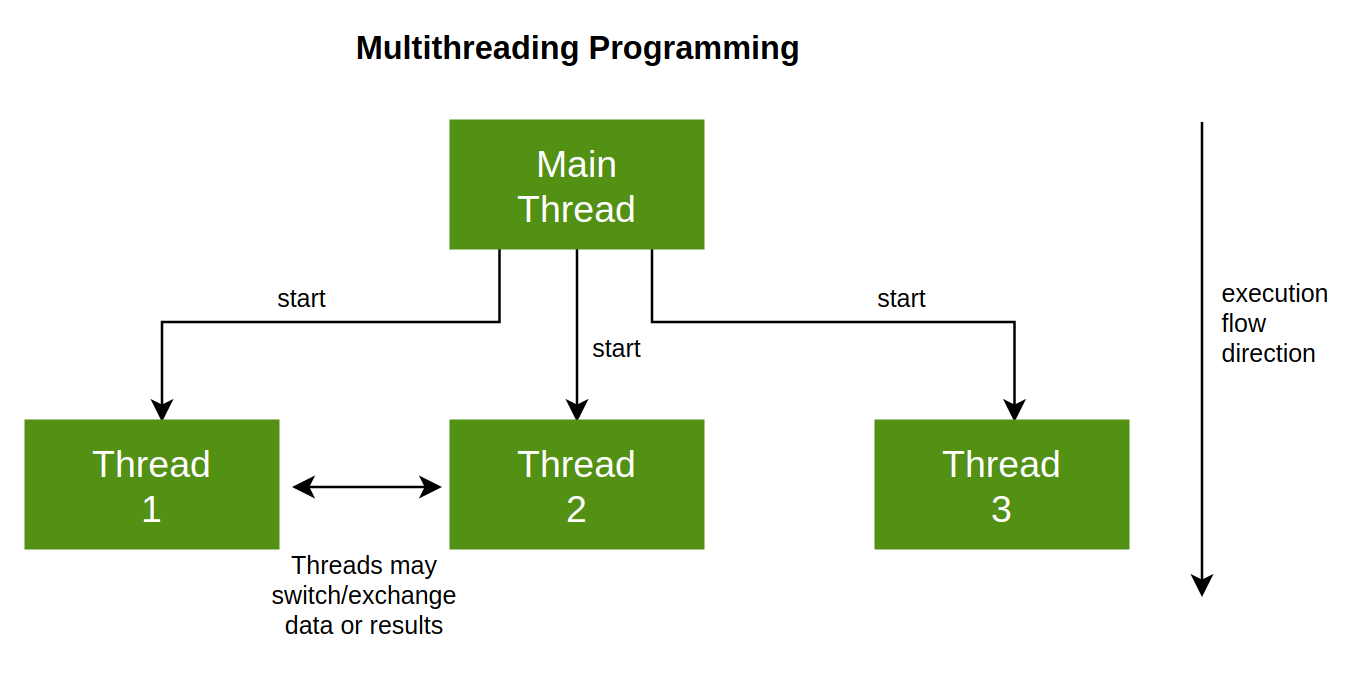
1. **ASYNCHRONOUS**

* An Asynchronous model allow mutiply thing happen at the same time. When progame run a long-runnung function, it doesn’t block the execurte flow and your progame continues to run



* In an asynchronous system, the solution is to start an **additional thread of control**. The first thread fetches the first file, and the second thread fetches the second file **without waiting for the first thread to finish**, and then both threads wait for their results to come back, after which they resynchronize to combine their results.

1. **Multithreading**

* refers to **the concurrent/parallel execution** of more than one sequential set (thread) of instructions.
* On a single processor, multithreading gives the illusion of running in parallel. In reality, the processor is switching by using a scheduling algorithm
* On multiple processor cores, threads are truly parallel. Individual microprocessors work together to achieve the result more efficiently. There are multiple parallel, concurrent tasks happening at once.
* 

C. Asynchronous vs Multithreading

- multithreading programming is all about concurrent execution of different functions

- Async programming is about non-blocking execution between functions, we can apply async with single-threaded or multithreaded programming.

- multithreading is one form of asynchronous programming.

 - **Multithreading is about workers, Asynchronous is about tasks.**

**-** In a nutshell, for large scale applications with a lot of I/O operations and different computations, using asynchronous multithreading programming flow, will utilize the computation resources, and take care of non-blocking functions.

## Asynchornous và Synchonous Callbacks trong java

* Hàm Callback là hàm được truyển vào hàm khác như là một tham số và sẽ thực hiện sau một loại sự kiện, mục đích để thông báo tới Sync/Async class nếu một công việc của một class khác được hoàn thành
* Sử trong tiểu biểu trong Observer Design Pattern
* Cấu trúc tiếu biểu trong java:
  + 1. Define the methods in an interface that we want to invoke after callback.
  + 2. Define a class that will implement the callback methods of the interface.
  + 3. Define a reference in other class to register the callback interface.
  + 4. Use that reference to invoke the callback method.
* **Synchronous Callback:**
  + Cho đến khi event trả về response, chương trình sẽ không thực hiện. hàm call back thực hiện tất cả các tác vụ trước khi trả về kết quả cho hàm gọi đên nó
* **Asynchronous Callback**

An Asynchronous call does not block the program from the code execution. When the call returns from the event, the call returns back to the callback function. So in the context of Java, we have to Create a new thread and invoke the callback method inside that thread. The callback function may be invoked from a thread but is not a requirement. A Callback may also start a new thread, thus making themselves asynchronous.

# **So sánh Abtract class và Interface**

6.1 Abtract Class: 

6.2 Inteface

- Interface có thể có các thuộc tính ( mặc định là static final) và các method



* Từ java8, inteface có thể có default vá static method

6.3 Sự khác biệt giữa Inteface và Abtract Class

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) An **abstract class** can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |
| 7) An **abstract class** can be extended using keyword "extends". | An **interface** can be implemented using keyword "implements". |
| 8) A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)**Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

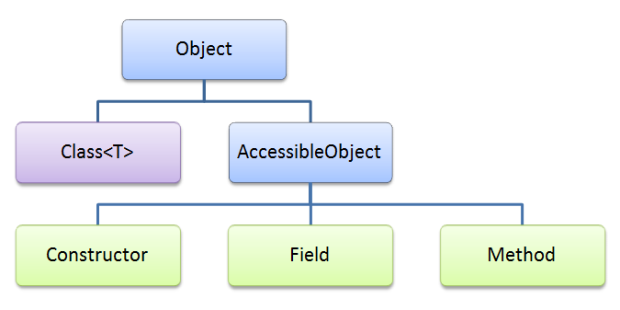
# **JAVA REFLECTION**

7.1. khái niêm cơ bản

* Java là ngôn ngữ hướng đối tượng, thông thường phải tạo ra một đối tượng để có thể truy cập vào các thuộc tính hoặc method qua toán tử dot(.)
* Java Reflection là một cách tiếp cận khác: có thể truy cập vào thuộc tính của một đối tượng nếu biết tên trường đó, hoặc truy cập vào phương thức nếu biết tên phương thức và tham số truyền vào
* cho phép bạn truy cập, sửa đổi cấu trúc và hành vi của một đối tượng tại thời gian chạy (runtime) của chương trình, cho phép truy cập vào các thành viên ***private***  tại mọi nơi trong ứng dụng
* Java Reflection khá mạnh mẽ và rất hữu ích đối với những ai hiểu rõ về nó. Ví dụ, bạn có thể ánh xạ (mapping) đối tượng (object) thành table dưới database tại thời điểm runtime. Kỹ thuật này các bạn có thể thấy rõ nhất ở JPA và Hibernate.

7.2 Kiến trúc của một JAVA REFLECTION API

* 3 lớp quan trọng bao gồm: Constructor, Field, Method

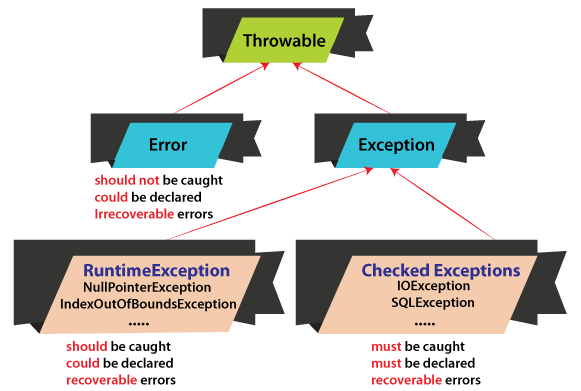


* + Class<T>: lớp này đại diện cho các lớp, interface và chứa các phương thức dùng để lấy các đối tượng kiểu Constructor, Field, Method,…
  + AccessibleObject: các kiểm tra về phạm vi truy xuất (public, private, protected) của field, method, constructor sẽ được bỏ qua. Nhờ đó bạn có thể dùng reflection để thay đổi, thực thi các thành phần này mà không cần quan tâm đến phạm vi truy xuất của nó.
  + Constructor: chứa các thông tin về một constructor của lớp.
  + Field: chứa các thông tin về một field của lớp, interface.
  + Method: chứa các thông tin về một phương thức của lớp, interface.
* Lớp (CLASS): Để có thể duyệt qua 1 class thì việc đầu tiền là phải lấy được đối tượng thuộc class đó, sau đó ta có thể lấy được các thông tin: Class name, class modifies, package info, super class…
* Tạo đối tượng Class<T>: được tạo ra bằng cách sử dụng phương thực static Class.forName(), cách này được sử dụng khi chỉ biết được tên lớp lúc thực thi
  + Nếu biết chính xác tên lớp ta có thể sử dụng cách:
    - Class c1 = Cat.class;// khi biết tên class
    - Class c4 = Integer.TYPE;// với kiểu nguyên thủy
    - Cat cat = new Cat();/\*\* khai báo 1 đối tượng
    - Class c = cat.getClass();/\* lấy tên class

**8. THREAD VÀ RUNNABLE TRONG JAVA**

# **EXCEPTION AND ERROR**

* <https://stackoverflow.com/questions/5813614/what-is-difference-between-errors-and-exceptions>



* **EXCEPTION**
* Khái niệm: an event that occurs during the execution of the program and interrupts the normal flow of program instructions 🡪 là 1 sự kiện xảy ra trong quá trình thực hiện chương trình và gây gián đoạn luồng xử lý của chương trình
* Lỗi xảy ra tại thời điểm biên dịch và chạy chương trình, thường là lỗi code
* Có thẻ xử lý bằng cách try-catch hoặc throw
* Gồm 2 loại: **CHECKED và UNCHECKED**
* Một số điêm cần lưu ý:
  + Khi có lỗi xảy ra, sẽ ném ra một exception
  + Bất cứ ngoại lệ nào ném ra cũng cần được xử lý
  + Hệ thống sẽ bắt ngoại lệ nếu ngoại lệ không được dev xử lý
  + Ngoại lệ có thể được ném lại nếu xử lý ngoại lệ lỗi
* **ERROR**
* **Errors** are problems that mainly occur due to the lack of system resources: Lỗi nghiêm trọng xảy ra do thiếu tài nguyên hệ thống
* Là lỗi nghiêm trọng, xảy ra tại thời điểm chạy chương trình, không thể bắt và xử lý
* Ví dụ : **OutOfMemoryError, LinkageError, AssertionError**

|  |  |  |
| --- | --- | --- |
| **Basis of Comparison** | **Exception** | **Error** |
| **Recoverable/ Irrecoverable** | Exception can be recovered by using the try-catch block. An error cannot be recovered. |  |
| **Type** | It can be classified into two categories i.e. checked and unchecked. | All errors in Java are unchecked. |
| **Occurrence** | It occurs at compile time or run time. | It occurs at run time. |
| **Package** | It belongs to java.lang.Exception package. | It belongs to java.lang.Error package. |
| **Known or unknown** | Only checked exceptions are known to the compiler. | Errors will not be known to the compiler. |
| **Causes** | It is mainly caused by the application itself. | It is mostly caused by the environment in which the application is running. |
| **Example** | **Checked Exceptions:**  SQLException,IOException **Unchecked Exceptions:** ArrayIndexOutOfBoundException, NullPointerException, ArithmaticException | Java.lang.StackOverFlow, java.lang.OutOfMemoryError |

* **Java generates two types of exceptions. These are:**

1. Checked exception
2. Unchecked exception

Difference Between Checked and Unchecked Exceptions in Java

|  |  |
| --- | --- |
| **Checked Exception** | **Unchecked Exception** |
| Checked exceptions occur at compile time. | Unchecked exceptions occur at runtime. |
| The compiler checks a checked exception. | The compiler does not check these types of exceptions. |
| These types of exceptions can be handled at the time of compilation. | These types of exceptions cannot be a catch or handle at the time of compilation, because they get generated by the mistakes in the program. |
| They are the sub-class of the exception class. | They are runtime exceptions and hence are not a part of the Exception class. |
| Here, the JVM needs the exception to catch and handle. | Here, the JVM does not require the exception to catch and handle. |
| Examples of Checked exceptions:   * File Not Found Exception * No Such Field Exception * Interrupted Exception * No Such Method Exception * Class Not Found Exception | Examples of Unchecked Exceptions:   * No Such Element Exception * Undeclared Throwable Exception * Empty Stack Exception * Arithmetic Exception * Null Pointer Exception * Array Index Out of Bounds Exception * Security Exception |

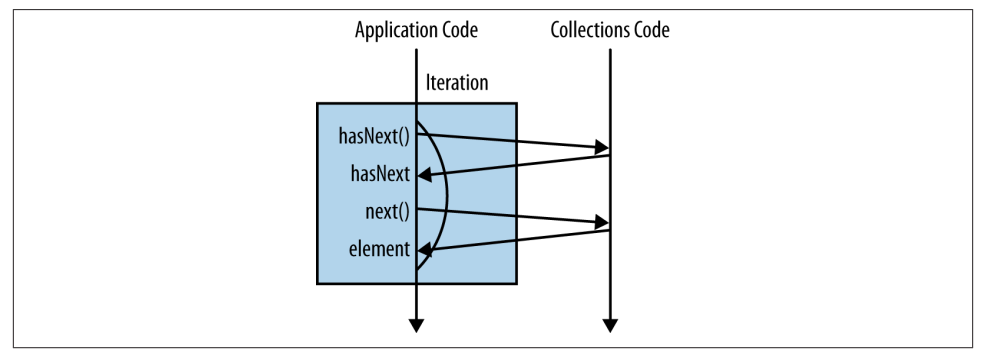
# **JAVA8 ADVANCE**

1. **Lambdas Expression**

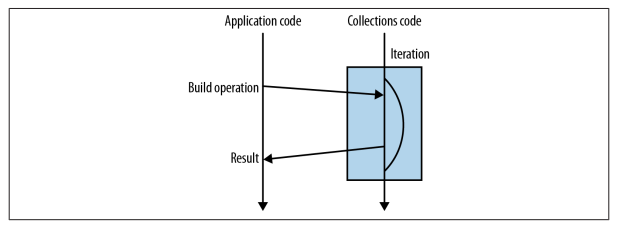
* Cú pháp của lambdas expression
  + Không có tham số : Runnable noArguments = () -> System.out.println("Hello World");
  + Có tham số : ActionListener oneArgument = event -> System.out.println("button clicked");
  + Hàm nằm trong khối code: Runnable multiStatement = () -> {  
    System.out.print("Hello");  
    System.out.println(" World");  
    };
  + Có nhiều tham số : BinaryOperator<Long> add = (x, y) -> x + y;
  + Tham số có kiểu rõ ràng: BinaryOperator<Long> addExplicit = (Long x, Long y) -> x + y;
* Sử dụng giá trị: các biến được sử dụng trong lambdas mặc có tính chất final (không thể gán lại giá trị cho các biến này) nếu ta thay đổi giá trị của nó sẽ báo lỗi khi compiler

1. **Stream**

* Khi duyệt qua một collection với vòng lặp thông thường sẽ có mọt số nhược điểm:
  + Lặp lại code với mỗi lần cần duyệt qua toàn bộ collection.
  + Khó khăn khi muốn duyệt song song
  + Khó diễn đạt ý của lập trình viên một cachs trôi chảy
* Bản chất của vòng lặp này là tạo ra một iteration và ẩn chúng đi được gọi là external iteratol. Các bước để thực hiện bao gồm : gọi đến iteratol method để tạo ra một Iteratol Object và sau đó call đến các method hasnext() và next () của chúng



* Một cách thay thế là sử dụng internal iteration, thay vì gọi đến method *iterator*() ta sẽ gọi đến method *stream()*  sẽ trả ra một interface tương đương là *STREAM:*
  + **long** count = allArtists.stream()  
    .filter(artist -> artist.isFrom("London"))  
    .count();



* Một số hàm ví dụ như *filter()* xây dựng công thức Stream nhưng không tạo ra giá trị mới khi kết thúc được gọi là *lazy.* Ngược lại các hàm ví dụ như .count() tạo là một giá trị khi kết thúc luồng stream được gọi là *eager.*

1. Advance Collection and Collector

* Các hàm tham chiếu:
  + trong trường hợp lambdas sử mà trong đó sử dụng các hàm có sẵn của class , ta có thể sử dụng một cú pháp khác ngắn gọn hơn được họi là *method reference: Classname : methodName*

artist -> artist.getName() có thể chuyển thành Artist::getName

* + *Với trường hợp hàm contructor có thể sử dụng* ClassName::new
* Sắp xêp các thành phần
  + Stream tự biểu diễn một sự sắp xếp vì mỗi phần tử sẽ được vận hành khi đến lượt, được gọi là *encounter order (trật tự gặp gỡ)*
  + Việc sắp xếp trong stream có thể rất tốn kém vì thế nên hạn chế sử dụng hàm order

1. Data Parallelism

* Hiệu năng: có 5 nhân tố ảnh hưởng đến hiệu năng bao gồm:
  + Data size: sử dụng xử lý song song chỉ hiêu quả nếu số lượng data đủ lớn để thực hiện đẩy vào các luồng pipeline trong một khoảng thời gian
  + Cấu trúc dữ liệu nguồn: Thông thường các dữ liệu nguồn là các collector và cần cân nhắc đến hiệu năng khi chia dữ liệu và gộp các kết quả
  + Packing (Đóng gói): các kiểu dữ liệu nguyên thủy thường sẽ tối ưu hơn kiểu boxed
  + Số lượng core: Càng nhiều core thì khả năng xử lý song song càng cao, tuy nhiên cần lưu ý đến số lượng core có thể dùng tại thời điểm runtime
  + Cost at element: Thời gian xử lý càng lớn so với thời gian chia dữ liệu và ghép dữ liệu thì việc chạy song song càng hiệu quả

# JAVA MEMORY MODEL

1. Heap vs Stack

* **Java Heap Space**
  + được sử dụng bởi java runtime để cấp phát bộ nhớ cho các đối tượng và các lớp JRE.
  + Khi 1 đối tượng được tạo ra, nó luôn luôn nằm trong Heap Space.
  + Garbage Collection chạy trên bộ nhớ heap để giải phóng bộ nhớ được sử dụng bởi các đối tượng không còn được tham chiếu đến nữa.
  + Bất cứ đối tượng nào được tạo trên heap có thể truy cập từ mọi nơi trên ứng dụng
* **Java Stack Memory**
  + Được sử dụng để thực thi 1 luồng. No có chứa giá trị xác định hàm (có đời sống ngắn và tham chiếu đến đối tượng khác trong heap)
  + Luôn theo cơ chế LIFO (Last In First Out)
  + Khi môt hàm được triệu hồi thì một khối sẽ được tạo trong stack dùng để chứ các biến nguyên thủy cục bộ và các tham chiếu đến các đối tượng khác trong hàm. Khi hàm kết thúc, khối này sẽ được giải phóng và sẵn sàng sử dụng cho hàm kế tiếp.
  + Stack memory có kích thước nhỏ hơn nhiều so với Heap
* **Sự khác nhau giữa Heap Space và Stack Memory:**

# **DATA STRUCTURE**

1. Different QUEUE and DEQUEUE

|  |  |  |
| --- | --- | --- |
| **Sr. No** | **Deque** | **Queue** |
| 1 | As the deque is a sequence container, so the insertion of elements takes place from both the ends, i.e. front and rear. | As the general behavior of Queue, insertion of the elements takes place from the one end, i.e. from the rear. |
| 2 | In deque, deletion of elements takes place from both the ends, i.e. front and rear. | In the normal queue, deletion of elements takes place from the front only. |
| 3 | As the deque has the ability to expand and contract from both the ends, it is implemented as dynamic arrays in the programs. | Talking about the implementation of Queue in a programming language, it is implemented as container adaptors. |
| 4 | Elements can easily be accessed in the deque using the iterators. | It is not possible to access the elements of the Queue using the iterators. |
| 5 | push\_back(x) and push\_front(x) functions are used in order to insert the elements in the Queue where push\_back(x) will insert the element at the rear and push\_Front(x) will insert the element at the front. | push(x) function is used in order to push the element in the Queue. Elements will be automatically inserted at the rear of the queue. |
| 6 | Similar to the insertion, for deletion of the elements in the deque, we have 2 functions, i.e. pop\_back() and pop\_front(), where pop\_back() will delete the element from the deque from the back and pop\_front() will remove the element from the queue from the front end. | In Queue, only one function is present in order to remove the element from the queue, i.e. pop(). It will automatically remove the element from the queue from the front end. |
| 7 | Deque is basically a hybrid data structure which provides the facility of both the stack and queue in a single data structure. | A queue is a single data structure providing the facility of only the Queue data structure. |
| 8 | Insertion and deletion of elements are more efficient as the resources are completely utilized in the case of the deque. | Insertion and deletion of elements are not much efficient in Queue as compared to deque. |