**MODULE 2**

1. **JAVA**
2. *OOP* (Object Oriented Programming) là kỹ thuật hướng đối tượng trong thực tế vào lập trình
3. *Java* là ngôn ngữ lập trình hướng đối tượng bậc cao, hiện đại, linh hoạt
4. Java có 8 kiểu dữ liệu nguyên thủy:

* Byte: 1 byte 0
* Short: 2 byte 0
* Int: 4 byte 0
* Long: 8 byte 0
* Float: 4 byte 0.0f
* Double: 8 byte 0.0d
* Boolean: true false false
* Char : 2 byte “\u0000”

1. *JDK* (Java Development Kit): Bộ công cụ giúp LTV phát triển ứng dụng qua: JRE, Compiler(javax), Debugger

* *JRE* (Java runtime enviroment): Môi trường thực thi ứng dụng java: libraries, JVM...
* *JVM* (Java vitual machine): Máy ảo java

1. *“Write one, run anywhere”* : file.java thông qua quá trình biên dịch sẽ trở thành file.class . File.class chứa các mã bytecode. Những mã bytecode này độc lập nền tảng. Chỉ cần cung cấp JRE thì có thể thực thi được thông qua quá trình thông dịch ra ngôn ngữ máy.

* Biên dịch: là dịch toàn bộ mã nguồn 1 lần và tái sử dụng kết quả (chỉ dịch lại khi có sự thay đổi)
* Thông dịch: là quá trình dịch từng dòng lệnh. Sẽ được hoạt động mỗi khi chạy chương trình.

1. **LOOP & ARRAY IN JAVA**

|  |  |
| --- | --- |
| **Array JS** | **Array Java** |
| * Có thể thay đổi độ dài * Chứa nhiều kiểu dữ liệu cùng lúc * Giá trị mặc định của phần tử trong mảng khi chưa khởi tạo là undefined * Cung cấp các hàm hỗ trợ tương tác với mảng * Không cần khơi báo độ dài ban đầu | * Không thay đổi độ dài * Chỉ chưa kiểu dữ liệu giống kiểu dữ liệu đã khai báo * Theo kiểu dữ liệu của mảng * Chỉ cung cấp thuộc tính length * Bắt buộc phải có độ dài ban đầu |

|  |  |
| --- | --- |
| **Method** | **Function** |
| * Có kiểu dữ liệu trả về * Cần AM để biết phạm vi truy cập * Return đúng kiểu dữ liệu đã khai báo, có thể return nhiều lần trong cùng 1 method những phải trong câu điều kiện | * Có thể có hoặc không có kiểu dữ liệu trả về * Không cần AM * Return nhiều KDL |

|  |  |
| --- | --- |
| **Tham trị** | **Tham chiếu** |
| * Biến và giá trị đều nằm ở stack | * Biến ở stack * Giá trị ở bộ nhớ heap * Biến ở stack tham chiếu vào giá trị trong bộ nhớ heap |

1. Java được coi là ngôn ngữ truyền tham trị vì khi một phương thức được gọi và truyền 1 biến vào, giá trị của biến đọ được sao chép và truyền phương thức, chứ không phải tham chiếu đến biến ban đầu. Điều màu có nghĩa là bất kì thay đổi nào được thực hiện trên biến bên trong phương thức không ảnh hưởng đến giá trị biến ban đầu.
2. **CLASS & OBJECT IN JAVA**
3. *Constructor* là một phương thức đặc biệt được sử dụng để khởi tạo các đối tượng. Đặc biệt vì gọi ngầm định thông qua từ khóa new và được sử dụng với ý nghĩa khởi tạo đối tượng (khác với thể hiện hành vi,thuộc tính)

* *Đặc điểm:*
* Phải trùng tên với class
* Có thể có nhiều constructor, với điều kiện:
* Phải khác số lượng truyền vào. Nếu cùng số lượng thì phải khác kiểu dữ liệu
* Không có kiểu dữ liệu trả về
* Nếu không khai báo constructor thì java cung cấp ngầm định 1 constructor không có tham số. **Lưu ý:** mất đi khi trong class tồn tại bất kì constructor nào
* Có thể sử dụng this() để gọi lại constructor đã được khai báo trước đó. **Lưu ý:**  this() bắt buộc phải được khai báo ở dòng lệnh đầu tiên trong body của constructor

|  |  |
| --- | --- |
| **JavaScript** | **Java** |
| * Có 1 constructor * Dùng từ khóa Constructor | * Nhiều constructor * Dùng từ khóa new |

1. *Class & Object*

|  |  |
| --- | --- |
| **Class** | **Object** |
| * Là một nhóm các đối tượng có cùng thuộc tính và hành vi * Là khuôn mẫu để tạo ra các đối tượng * Các class trong cùng 1 package/class thì không được đặt trùng tên * Khai báo class thông qua từ khóa class | * Là thể hiện/ sản phẩm cụ thể của class * Mỗi đối tượng có giá trị của thuộc tính và hành vi khác nhau để phân biệt với nhau * Các đối tượng có thể đặt trùng tên nhưng phải khác file.java * Khởi tạo thông qua từ khóa new |

1. *So sánh Method & Contructor*

|  |  |
| --- | --- |
| **Method** | **Contructor** |
| * Không cần trùng tên với class * Có thể return or không return * Java không cung cấp 1 method ngầm định không tham số * Dùng để mô tả hành vi của đối tượng * Có thể gọi được nhiều lần và gọi bằng tên method * Đi kèm với các từ khóa | * Phải trùng tên với class * Không có kiểu dữ liệu trả về * Java cung cấp ngầm định 1 constructor không có tham số * Dùng để khởi tạo đối tượng * Dùng từ khóa new để gọi và chỉ gọi được 1 lần * Không đi kèm abstract, final, static |

1. **ACCESS MODIFIER, STATIC METHOD, STATIC PROPETY**
2. *Access Modifier*: Quyết định phạm vi truy cập của biến, class, method,…
3. *Static*: quyết định thành phần được khai báo thuộc về class chứ không thuộc về đối tượng

* Mục đích: quản lý bộ nhớ tốt hơn, tạo các method/class dùng chung
* Các ràng buộc khi sử dụng static:
* Các phương thức static chỉ có thể gọi đến phương thức static & biến static. Không thể gọi nonstatic
* Các nonstatic có thể gọi đến static
* Các phương thức static không thể chứa this(),super()
* Sử dụng khối static để khởi tạo giá trị cho biến static (khi gọi biến static thì cần biến static tồn tại trong bộ nhớ)

1. *Các loại biến*:

|  |  |  |
| --- | --- | --- |
| **Local** | **Instance** | **Static** |
| * Vị trí khai báo: trong phương thức, constructor,… * Phạm vi hoạt động: ở trong block * Không có AM * Vòng đời: kết thúc phương thức thì mất * Phải khai báo giá trị trước khi sử dụng * Local: stack | * Ngoài phương thức, constructor,nhưng ở trong class * Tùy AM * Có AM * Vòng đời kết thúc khi hủy đối tượng (set object=null) * Value của biến lấy theo value mặc định * heap | * Tương tự instance nhưng đi kèm từ khóa static * Tùy AM * Có AM * Chạy lại chương trình * Tương tự instance * Static area |

1. **ENCAPSULATION, INHERITANCE, POLIMORPHISM**
2. *Encapsulation*: là tính cho phép bảo vệ sự toàn vẹn dữ liệu của đối tượng. Che giấu đi các thông tin quan trọng, ngăn chặn sự truy cập bất hợp pháp
3. *Inheritance*: là cho phép lớp con sử dụng lại các đặc điểm và hành vi của lớp cha

* *Đặc điểm*:
* Kế thừa tất cả, trừ các thành phần private
* Constructor không được kế thừa
* Lớp con có thể gọi constructor của lớp cha thông qua super
* Lớp con có thể định nghĩa thêm các thuộc tính và method mới
* Java không cho phép đa kế thừa
* *Mục đích*:
* Tái sử dụng mã nguồn
* Rút ngắn code => giảm thiểu dụng lượng bộ nhớ
* Tạo ra các chương trình, ứng dụng có tính đơn giản không phức tạp
* *Tại sao java không cho phép đa kế thừa*: giảm độ phức tạp của chương trình. Tránh trường hợp khi 1 lớp con kế thừa 2 lớp cha và trong 2 lớp cha có phương thức cùng tên, lớp con sẽ không biết gọi phương thức nào
* Super:
* Super(): gọi lại constructor của lớp cha
* Super. : gọi thuộc tính or method của lớp cha
* Final:
* Đối với class thì để tạo lớp vô sinh và tạo hằng
* Đối với method thì để tạo các method chỉ có thể kế thừa mà không thể override lại
* Lớp Object: là lớp cha của mọi lớp trong java. Khi lập trình viên tạo ra 1 class mà chưa kế thừa bất kì class nào thì chương trình ngầm định hiểu lớp đó đang kế thừa lớp Object. Lớp object cho các lớp kế thừa:
* Boolean equals: kiểm tra xem 2 đối tượng hiện hành có “bằng nhau” hay không
* Class getClass() trả về lớp mà đối tượng hiện hành đã tạo từ đó
* Int hasCode() trả về mã băm của đối tượng hiện hành (một định danh của đối tượng)
* String toString() trả về biểu diễn dạng string của đối tượng ta thường cài đè phương thức này.

1. *Polimorphism*: khi với cùng 1 method nhưng có thể có cách ứng xử khác nhau ở những lớp cình giao diện

|  |  |
| --- | --- |
| **Overload** | **Override** |
| * Cùng tên nhưng khác tham số (khác số lượng hoặc kiểu dữ liệu) * Xảy ra trong cùng 1 class * Đa hình tại compile time * Không quan trọng AM của các method | * Cùng tên, cùng danh sách tham số, cùng kiểu dữ liệu trả về * Xảy ra trong 2 class có quan hệ kế thừa * Đa hình tại runtime * AM method ở lớp con phải lớn hơn hoặc bằng lớp cha |

* *Ép kiểu*: là chuyển biến kiểu dữ liệu này thành kiểu dữ liệu khác
* Kiểu nguyên thủy:

+ Ngầm định: ép kiểu từ kiểu dữ liệu nhỏ lên lớn (dễ gây ra dư thừa dữ liệu)

+ Tường minh: từ lớn xuống nhỏ (Dễ gây ra mất mát or sai trái về dữ liệu)

* Kiểu đối tượng:

+ Upcasting: ép từ cha lên con

+ Downcasting: ép từ con lên cha

1. **ABSTRACTION & INTERFACE**
2. *Interface* là cấu trúc tương tự như lớp, nhưng chỉ chứa các hằng số và abstract method

* Interface quy định các hành vi chung cho các lớp triển khai nó
* Sử dụng từ khóa interface để định nghĩa interface

1. *Abstraction:*

* Là tính chất cho phép loại bỏ đi những thuộc tính không liên quan đến chương trình, ẩn đi các chi tiết triển khai, chỉ hiển thị kết quả cho người dùng
* Thể hiện qua abstract class và interface

1. *Abstract class:* là lớp trừu tượng đến mức không thể tạo được đối tượng (khi tạo đối tượng thì đối tượng vô nghĩa)

* *Đặc điểm:*
* Abstract class vẫn có constructor nhưng không thể tạo đối tượng (constructor sinh ra để cho lớp con sử dụng)
* 1 class có abstract class method thì bắt buộc đó phải là abstract class, ngược lại thì không
* 1 abstract class khi extend 1 abstract class khác thì không cần phải override lại các abstract method
* 1 class thường khi extend1 abstract class thì bắt buộc phải override lại các abstract method
* Abstract không đi kèm với final, private
* *Abstract class & Interface :*

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| * *Có constructor* * *Có thể chứa biến, method thông thường* * *1 class thường thì extend abstract class* * *1 class thường chỉ extend được 1 abstract class* * *1 abstract class chỉ có thể extend 1 abstract class* * *Xuất hiện ở mối quan hệ is-a* | * *Không có constructor* * *Trước java 8 chỉ có hằng siis và abstract method (sau java 8 có thêm default, static)* * *Class thường implement interface* * *1 class thường có thể implement nhiều interface* * *1 interface có thể extend nhiều interface* * *Xuất hiện ở mối quan hệ has-a, can-do* |

1. **CLEAN CODE**

Clean code là thuật ngữ để chỉ đến những mà nguồn tốt

* *Đặc điểm:*
* Đơn giản, trực tiếp, dễ đọc
* Dễ cải tiến, mở rộng
* Có unit test
* Các định danh đều thể hiện rõ nghĩa
* Có ít sự phụ thuộc
* Không có mã bị trùng lặp
* Thể hiện được ý tưởng của thiết kế

1. **DSA (Data structures algrithms)**
2. *Collection*

* Collection: cung cấp các interface/ class để lưu trữ dữ liệu

+ Interface: List, Set, Queue

+ Class: ArrayList, LinkedList, Stack, ArrayDeque, Priority, HashSet, LinkedHashSet, TreeSet,..

* Collections: cung cấp các method để thao tác với cấu trúc dữ liệu

1. *ArrayList:* cung cấp cấu trúc lưu trữ dữ liệu dưới dạng mảng động

* *Vì sao vẫn giữ Array mà không xóa bỏ?*
* Array là gốc của ArrayList nên nếu xóa bỏ Array thì không còn gốc để ArrayList thao tác
* ArrayList hao tốn bộ nhớ khi capacity tự động kéo dài length
* *…*
* *Vì sao ArrayList thao tác add, delete chậm hơn Array?*
* Vì ArrayList khi thực hiện một thao tác thì cần phải thực hiện thêm nhiều thao tác khác (copy array, dịch chuyển phần tử khi add,move,...)
* Array là mảng cố định nên khi add or move thì nó chỉ cần đóng phần tử ở index đó chứ không thực hiện thao tác nào

1. *LinkedList:* cung cấp cấu trúc lưu trữ dữ liệu dưới dạng các node liên kết

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| * Lưu trữ dưới dạng mạng động * Duyệt, tìm kiếm nhanh vì các phần tử sắp xếp theo index * Thêm, xóa chậm | * Lưu trữ dưới dạng các node liên kết * Duyệt tìm kiếm chậm * Thêm, xóa nhanh vì chỉ cần bẻ gãy liên kết rồi thực hiện thao tác |

1. *Set:* là cấu trúc dữ liệu lưu trữ các phần tử dưới dạng tập hợp không trùng lặp

|  |  |  |
| --- | --- | --- |
| **HashSet** | **LinkedHashSet** | **TreeSet** |
| Lưu không theo thứ tự truyền vào | Lưu theo thứ tự truyền vào | Lưu có sắp xếp |

1. **DSA STACK QUEUE**

* *Generic* là cơ chế cho phép truyền dũ liệu như tham số, hay còn gọi là tham số hóa kiểu dữ liệu
* *Ưu điểm:*
* Kiểu dữ liệu an toàn
* Phát hiện lỗi lúc compile
* Tái sử dụng mã nguồn
* *Nhược điểm:*
* Không dùng được với kiểu nguyên thủy (dùng thông qua auto\_boxing)
* Không tạo được đối tượng Generic
* *Stack* (là 1 class) cấu trúc dữ liệu danh sách mà các phần tử được lưu trữ theo cơ chế FILO = LIFO
* Pop: lấy phần tử và xóa (ảnh hưởng size)
* Peek: lấy ra sau đó bỏ vào lại (không ảnh hưởng size)
* *Queue* (là 1 interface) cơ chế FIFO = LILO
* Các lớp triển khai Queue:
* ArrayDequeue (theo thứ tự)
* LinkedList
* PriorityQueue(ưu tiên nhỏ nhất)

1. **MAP TREE**

*Map thuộc Java Collection framework, chứ không thuộc Collection. Vì Map lưu trữ dữ liệu dưới dạng cặp key-value, khác với cấu trúc dữ liệu còn lại chỉ lưu trữ value. Map đồng cấp với Collection*

1. *Map* là cấu trúc dữ liệu dưới dạng cặp key-value

* *Đặc điểm:*
* 1 key tương ưng 1 value
* Key không cho phép trùng lặp, value thì có thể
* Cặp key-value gọi là entry

|  |  |  |
| --- | --- | --- |
| **HashMap** | **LinkedMap** | **TreeMap** |
| * Không duy trì thứ tự truyền vào * Cho phép 1 key null (vì Map không cho phép trùng lặp nếu không khi có nhiều key null sẽ không biết gọi key nào) | * Duy trì * Cho phép 1 key null (tương tự HashMap) | * Sắp xếp tăng dần theo key * Không cho phép key null (vì sẽ không biết sắp xếp ở đâu NulllPointerException) |

1. **XỬ LÝ NGOẠI LỆ VÀ DEBUG**

* Exception là ngoại lệ sinh ra có thể dẫn đến việc sai luồng thực thi or chết chương trình

|  |  |
| --- | --- |
| Checked | Unchecked |
| * Xảy ra trong quá trình compile * Bắt buộc phải xử lí * FileNotFoundException, IO Exception, DB Exception | * Xảy ra trong quá trình runtime * Không bắt buộc phải xử lí |

* Error: lỗi gây chết chương trình & không thể hadle
* Có 2 cách xử lí ngoại lệ:
* Try – catch – finally : xử lí ngay tại vị trí có khả năng sinh ra exception
* Throw, Throws: ném cho người khác xử lí

|  |  |
| --- | --- |
| Throw | Thows |
| * Ném ngoại lệ * Chỉ ném được 1 * Vị trí ném nằm ở body method | * Khai báo ngoại lệ * Khai báo 1 or nhiều * Vị trí khai báo nằm ở phần header method phía sau dấu () |

* Lưu ý khi sử dụng Try – catch – finally:
* 1 try có thể có 1 hoặc nhiều catch
* 1 try thì chỉ có thể 1 finally
* Với mỗi khối try thì bắt buộc phải có 1 khối catch hoặc finally đi kèm
* Các khối catch phải được sắp xếp theo thứ tự từ chi tiết đến chung nhất
* Tại mỗi thời điểm chỉ có 1 khối catch được thực thi
* Finally luôn thực thi dù có hay không xảy ra ngoại lệ. Ngoại trừ : System.exit(), sinh ra error ở try or catch

1. **SERIALIZATION**

* *Serialization:* là cơ chế cho phép chuyển đổi object sang luồng byte stream. Ngược lại thì có cơ chế Deserialization
* **Lưu ý:**

+ Sử dụng transient hoặc static để ngăn chặn việc ghi thuộc tính vào file

+ Super class đã implement serializable thì supclass không cần phải implements

* IO Stream là luồng nhập xuất dữ liệu, có 2 loại: Character & Byte

1. **STRING**

|  |  |  |
| --- | --- | --- |
| STRING | STRING BULDER | STRING BUFFER |
| * Immutable * Tốc độ chậm nhất * Có 2 cách để khởi tạo giá trị * An toàn trên mọi trường hợp | * Mutable * Nhanh hơn Buffer * Chỉ có thể sử dụng new * Luồng bất đồng bộ -> không an toàn trong khi xử lý đa luồng (luồng nào thực thi xong trước cũng được ) | * Mutable * Chậm hơn Builder * Chỉ có thể sử dụng new * Luồng đồng bộ -> an toàn (luồng thực thi từ đầu đến cuối) |

* Regex là mẫu hoặc biểu mẫu được sử dụng để tìm, thay thế chuẩn hóa dữ liệu
* Pattern : không có constructor. Tạo object thông qua Pattern.compile
* Matcher: không có constructor. Tạo object thông qua method matcher của pattern

1. **SOLID**

* *SOLID* là một trong những nguyên tắc để giúp chúng ta xây dựng các ứng dụng OOP hiệu quả . Giúp cho LTV viết ra những đoạn code dễ đọc, dễ hiểu, dễ bảo trì
* *S- Single-responsibility principle: Nguyên lí trách nhiệm duy nhất*
* Mỗi lớp chỉ nên đảm nhiệm 1 nhiệm vụ duy nhất
* Lí do:
* Dễ quản lí mã nguồn
* Các lớp tập trung vào nhiệm vụ của mình
* Giảm tính phụ thuộc giữa các thành phần
* Có thể phát triển đồng thời với các lớp độc lập với nhau
* Dễ dàng mở rộng bảo trì
* *O- Open- closed principle: Nguyên lí đóng mở*
* Có thể dễ dàng mở rộng 1 lớp mà không cần thay đổi mã nguồn của lớp đó
* Lí do: dễ mở rộng dễ thay đổi
* *L-Liskov Substitution principle : Nguyên lý thay thế Liskov*
* Nếu S là một lớp con của T thì các đối tượng của lớp T có thể được thay thế bằng các đối tượng của lớp S mà không làm ảnh hưởng tới bất cứ hành vi nào của hệ thống
* Lí do : tránh các sai sót khi mở rộng thiết kế
* *I-Interface- segregation principle: Nguyên lý phân tách Interface*
* Các Interface chỉ nên chưa các method cần thiết vừa đủ với mục đích của nó. Nên tách các Interface thành nhiều interface nếu các method của chúng không liên quan chặt chẽ đến nhau
* Lí do: tránh dư thừa mã nguồn không cần thiết
* *D - Dependency Inversion principle: Nguyên lý đảo ngược phụ thuộc*
* Các module ở mức trên không nên phụ thuộc vào các module ở mức dưới, mà cả hai nên phụ thuộc vào trừu tượng