**1.Câu hỏi cơ bản về Java:**

**1.1 Java ra đời vào năm nào?**

Java ban đầu được James Gosling tại Sun Microsystems (sau đó đã được Oracle mua lại) phát triển và được phát hành vào năm 1995 như một thành phần cốt lõi của nền tảng Java của Sun Microsystems.

**1.2 Giới thiệu đơn giản về JVM, JRE, JDK.**

1. **JVM (Java Virtual Machine - Máy ảo Java)**:
   * JVM là một phần mềm có khả năng thực thi các chương trình Java.
   * Nó tạo môi trường ảo để chạy mã nguồn Java (file .java) đã được biên dịch thành mã bytecode (file .class).
   * JVM cung cấp khả năng quản lý bộ nhớ, thực thi mã, và nhiều chức năng khác giúp ứng dụng Java chạy trên nhiều nền tảng khác nhau mà không cần biên dịch lại.
2. **JRE (Java Runtime Environment - Môi trường chạy Java)**:
   * JRE là một tập hợp các thư viện, mã và các thành phần cần thiết để chạy các ứng dụng Java đã biên dịch.
   * Nó bao gồm JVM cùng với các thư viện cần thiết để thực thi chương trình.
   * JRE không bao gồm các công cụ phát triển, chỉ dùng để chạy các ứng dụng Java đã tạo ra.
3. **JDK (Java Development Kit - Bộ công cụ phát triển Java)**:
   * JDK là bộ công cụ cần thiết để phát triển ứng dụng Java.
   * Nó bao gồm JRE để thực thi ứng dụng Java và các công cụ phát triển như trình biên dịch (javac), trình gỡ lỗi (debugger), trình tạo tài liệu (javadoc) và nhiều công cụ khác.
   * JDK cho phép bạn viết mã nguồn Java, biên dịch thành bytecode và chạy các ứng dụng trên nền tảng Java.

**2. Biến, mảng trong Java**

**2.1 Biến trong Java là gì?**

Trong ngôn ngữ lập trình Java, biến là một vùng bộ nhớ được sử dụng để lưu trữ dữ liệu. Biến giúp bạn đặt tên cho các giá trị dữ liệu cụ thể và thực hiện các thao tác và tính toán trên chúng. Mỗi biến phải được khai báo với một kiểu dữ liệu cụ thể, và sau khi khai báo, bạn có thể gán giá trị cho biến và sử dụng giá trị đó trong các phần khác của chương trình.

Dưới đây là cú pháp cơ bản để khai báo và sử dụng biến trong Java:

**Khai báo biến:**

Cú pháp:

type variableName;

Trong đó:

* type là kiểu dữ liệu của biến (ví dụ: int, double, String, ...)
* variableName là tên của biến

**2.2 Mảng trong Java là gì? Cách khai báo một mảng trong Java?**

**Mảng (Array) trong Java:**

Mảng là một cấu trúc dữ liệu cho phép lưu trữ nhiều giá trị cùng kiểu dữ liệu trong một biến duy nhất. Mỗi phần tử trong mảng được truy cập thông qua chỉ số (index) duy nhất. Chỉ số của mảng bắt đầu từ 0 và kết thúc tại độ dài mảng trừ đi 1.

**Khai báo mảng**:

Cú pháp khai báo một mảng trong Java như sau:

type[] arrayName = new type[length];

Trong đó:

* type là kiểu dữ liệu của mảng (ví dụ: int, double, String, ...)
* arrayName là tên của mảng
* length là số lượng phần tử trong mảng

**3. Các kiểu dữ liệu trong Java**

**3.1 Liệt kê các kiểu dữ liệu nguyên thủy (primitive) trong Java**

Trong ngôn ngữ lập trình Java, có một số kiểu dữ liệu nguyên thủy (primitive data types) được sử dụng để đại diện cho các giá trị cơ bản như số nguyên, số thực, ký tự, và giá trị logic. Dưới đây là danh sách các kiểu dữ liệu nguyên thủy trong Java:

1. **Kiểu số nguyên (Integer Types)**:
   * **byte**: 8-bit số nguyên, giới hạn trong khoảng từ -128 đến 127.
   * **short**: 16-bit số nguyên, giới hạn trong khoảng từ -32,768 đến 32,767.
   * **int**: 32-bit số nguyên, giới hạn trong khoảng từ -2^31 đến 2^31 - 1.
   * **long**: 64-bit số nguyên, giới hạn trong khoảng từ -2^63 đến 2^63 - 1.
2. **Kiểu số thực (Floating-Point Types)**:
   * **float**: 32-bit số thực động, có độ chính xác xấp xỉ 7 chữ số thập phân.
   * **double**: 64-bit số thực động, có độ chính xác cao hơn, xấp xỉ 15 chữ số thập phân.
3. **Kiểu ký tự (Character Type)**:
   * **char**: 16-bit ký tự Unicode.
4. **Kiểu logic (Boolean Type)**:
   * **boolean**: Biểu thị giá trị đúng (true) hoặc sai (false).

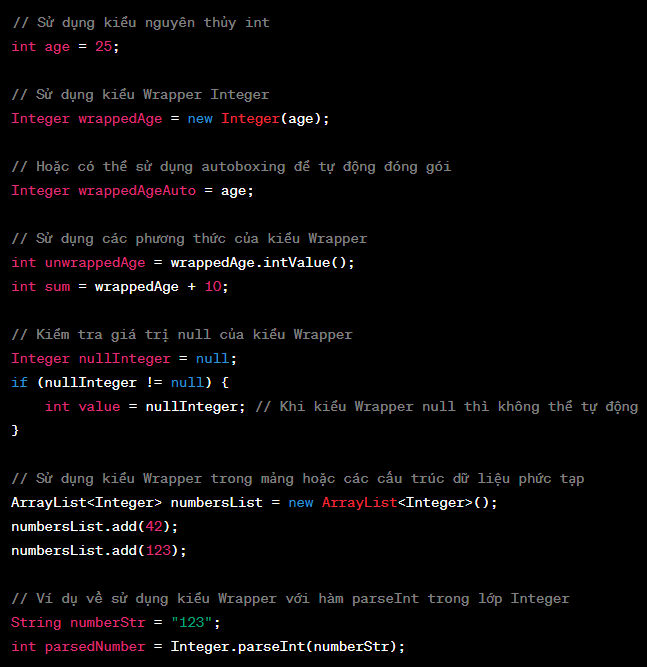
**3.2 Giải thích khái niệm kiểu Wrapper trong Java và đưa ra ví dụ.**

Kiểu Wrapper trong Java là một cách để đóng gói (wrap) các kiểu dữ liệu nguyên thủy vào các đối tượng. Mỗi kiểu dữ liệu nguyên thủy có một lớp tương ứng là kiểu Wrapper, cho phép bạn làm việc với kiểu dữ liệu nguyên thủy như là các đối tượng. Việc sử dụng kiểu Wrapper có thể hữu ích trong nhiều tình huống, chẳng hạn như khi bạn cần làm việc với các thuộc tính hoặc phương thức của một kiểu dữ liệu nguyên thủy.

Dưới đây là danh sách các kiểu dữ liệu nguyên thủy và kiểu Wrapper tương ứng trong Java:

1. **byte** - **Byte**
2. **short** - **Short**
3. **int** - **Integer**
4. **long** - **Long**
5. **float** - **Float**
6. **double** - **Double**
7. **char** - **Character**
8. **boolean** - **Boolean**

Ví dụ về việc sử dụng kiểu Wrapper:



Kiểu Wrapper giúp bạn làm việc linh hoạt hơn với các kiểu dữ liệu nguyên thủy và đối tượng, đặc biệt trong các tình huống yêu cầu làm việc với các cấu trúc dữ liệu phức tạp như Collection (ArrayList, LinkedList, ...) hoặc trong việc truyền tham số cho các phương thức yêu cầu kiểu đối tượng.

**3.3 So sánh primitive với wrapper.**

Dưới đây là một số điểm so sánh giữa kiểu dữ liệu nguyên thủy (primitive) và kiểu Wrapper trong Java:

1. **Đóng gói và truy cập**:
   * Primitive: Là các kiểu dữ liệu cơ bản như int, double, char, không phải là đối tượng, truy cập và thực hiện các phép toán trực tiếp.
   * Wrapper: Là các lớp đối tượng bao quanh các kiểu dữ liệu nguyên thủy, cho phép truy cập các phương thức và tính năng phức tạp hơn.
2. **Giá trị null**:
   * Primitive: Không thể chứa giá trị null.
   * Wrapper: Có thể chứa giá trị null, cho phép biểu diễn giá trị thiếu.
3. **Tính tự động đóng gói và mở gói (autoboxing/unboxing)**:
   * Primitive: Không có tính năng tự động đóng gói và mở gói.
   * Wrapper: Có tính năng tự động đóng gói (autoboxing) khi gán giá trị nguyên thủy vào đối tượng Wrapper và tự động mở gói (unboxing) khi sử dụng giá trị từ đối tượng Wrapper.
4. **Hiệu suất**:
   * Primitive: Thường nhanh hơn trong việc thực thi vì không có overhead của việc tạo và quản lý đối tượng.
   * Wrapper: Tốn một chút hiệu suất hơn do tính phức tạp của việc tạo và quản lý đối tượng.
5. **Sử dụng trong các cấu trúc dữ liệu phức tạp**:
   * Primitive: Không thể được sử dụng trực tiếp trong các cấu trúc dữ liệu phức tạp như Collection.
   * Wrapper: Có thể sử dụng trong các cấu trúc dữ liệu phức tạp như Collection.
6. **Thao tác và tính năng phức tạp**:
   * Primitive: Có hạn chế trong việc thực hiện các thao tác và tính năng phức tạp trực tiếp.
   * Wrapper: Cung cấp nhiều tính năng phức tạp hơn như phép toán so sánh, chuyển đổi, và xử lý ngoại lệ dễ dàng hơn.

Tùy thuộc vào tình huống sử dụng và yêu cầu của bạn, bạn có thể quyết định sử dụng kiểu nguyên thủy hoặc kiểu Wrapper. Kiểu nguyên thủy thường được ưu tiên trong các tình huống cần hiệu suất cao hoặc không cần tính năng phức tạp. Kiểu Wrapper thường được sử dụng trong các tình huống cần xử lý dữ liệu phức tạp, làm việc với các cấu trúc dữ liệu phức tạp, hoặc cần sử dụng tính năng mở rộng của đối tượng.

**4. String, StringBuilder, StringBuffer**

**4.1 Tại sao String là bất biến trong Java?**

Trong Java, một **String** là bất biến (immutable) vì khi một đối tượng **String** được tạo ra, nó không thể thay đổi giá trị của nó sau đó. Mọi thay đổi trên một đối tượng **String** mới tạo ra một đối tượng **String** khác mà giữ nguyên giá trị của đối tượng gốc. Điều này có nhiều lý do quan trọng:

1. **An toàn đồng thời (Thread Safety)**: Vì **String** là bất biến, nhiều luồng (threads) có thể truy cập cùng một đối tượng **String** mà không cần lo lắng về việc thay đổi dữ liệu.
2. **Bảo mật**: Trong môi trường nhiều người dùng hoặc môi trường có thể thay đổi dữ liệu, việc làm cho **String** bất biến giúp đảm bảo rằng dữ liệu của một đối tượng **String** không thể bị thay đổi một cách ngẫu nhiên.
3. **Optimization**: Do **String** là bất biến, các biểu thức nối chuỗi (concatenation) hoặc các thay đổi dữ liệu trong **String** sẽ tạo ra các đối tượng mới thay vì thay đổi trực tiếp trong đối tượng ban đầu. Điều này giúp tối ưu hóa việc quản lý bộ nhớ và xử lý dữ liệu.
4. **Tính ổn định**: Bất biến làm cho các đối tượng **String** trở nên ổn định hơn trong các tình huống khác nhau, ví dụ như khi sử dụng làm khóa cho các cấu trúc dữ liệu như HashMap hoặc khi sử dụng trong việc so sánh đối tượng.

Mặc dù **String** là bất biến, việc sử dụng các phép toán trên chuỗi như cắt, ghép và biến đổi vẫn có thể được thực hiện bằng cách tạo ra các đối tượng **String** mới. Điều này giúp đảm bảo tính ổn định và an toàn cho dữ liệu trong quá trình xử lý.

**4.2 Làm thế nào để lưu trữ một đối tượng String trong bộ nhớ?**

Trong Java, một đối tượng **String** được tự động lưu trữ trong bộ nhớ khi bạn tạo nó bằng cách sử dụng từ khóa **new** hoặc bằng cách gán một chuỗi giá trị cho một biến kiểu **String**. Dưới đây là một số cách để lưu trữ một đối tượng **String** trong bộ nhớ:

1. **Sử dụng toán tử gán (=)**:

String str1 = "Hello"; // Lưu trữ trong bộ nhớ

1. **Sử dụng từ khóa new để tạo đối tượng mới**:

String str2 = new String("World"); // Tạo đối tượng mới và lưu trữ trong bộ nhớ

Việc sử dụng toán tử gán (**=**) để gán giá trị cho một biến kiểu **String** sẽ thực hiện một số tối ưu hóa để sử dụng lại các đối tượng **String** đã có trong bộ nhớ (sử dụng cơ chế gọi là "string pool" hoặc "string intern pool"). Điều này giúp giảm bớt sự lãng phí về tài nguyên và tăng hiệu suất.

Ví dụ:

String str1 = "Hello"; String str2 = "Hello"; // Sử dụng lại đối tượng đã có trong bộ nhớ String str3 = new String("Hello"); // Tạo đối tượng mới trong bộ nhớ, không sử dụng lại đối tượng trong string pool

Trong ví dụ trên, **str1** và **str2** đều sử dụng lại cùng một đối tượng **String** đã có trong bộ nhớ, trong khi **str3** tạo ra một đối tượng mới trong bộ nhớ vì sử dụng từ khóa **new**.

**4.3 Sự khác nhau giữa String s = "abc"; và String s = new String("abc");?**

Trong Java, có sự khác nhau giữa việc khởi tạo một đối tượng **String** bằng cách sử dụng toán tử gán (**=**) và bằng cách sử dụng từ khóa **new**. Dưới đây là sự khác nhau giữa hai cách này:

1. **Sử dụng toán tử gán (=)**:

String s1 = "abc";

Trong trường hợp này, Java sẽ kiểm tra xem chuỗi **"abc"** đã tồn tại trong vùng nhớ đặc biệt gọi là "string pool" (hoặc "string intern pool") chưa. Nếu chuỗi đã tồn tại trong pool, biến **s1** sẽ được gán để trỏ tới đối tượng chuỗi đã tồn tại. Nếu chuỗi chưa tồn tại trong pool, nó sẽ được thêm vào pool và biến **s1** sẽ trỏ tới nó.

1. **Sử dụng từ khóa new**:

String s2 = new String("abc");

Trong trường hợp này, đối tượng chuỗi mới sẽ luôn được tạo ra, bất kể chuỗi **"abc"** có tồn tại trong string pool hay không. Điều này đảm bảo rằng một đối tượng mới sẽ luôn được tạo ra trong bộ nhớ, thậm chí khi chuỗi tương tự đã tồn tại trong string pool.

Tóm lại, sự khác nhau quan trọng nhất giữa hai cách này là cách quản lý bộ nhớ và việc tạo ra đối tượng mới. Sử dụng toán tử gán (**=**) thường được ưu tiên hơn vì nó giúp tối ưu hóa việc sử dụng lại các đối tượng chuỗi có sẵn trong string pool, trong khi sử dụng từ khóa **new** luôn tạo ra đối tượng mới trong bộ nhớ.

**4.4 So sánh sử dụng String, StringBuilder và StringBuffer trong Java**.

Trong Java, có ba lớp được sử dụng để làm việc với chuỗi: **String**, **StringBuilder**, và **StringBuffer**. Mỗi lớp có ứng dụng và ưu điểm riêng tùy theo tình huống sử dụng. Dưới đây là sự so sánh giữa ba lớp này:

1. **String**:
   * **Bất biến (Immutable)**: Một đối tượng **String** không thể thay đổi giá trị của nó sau khi đã được tạo ra. Mọi thay đổi trên chuỗi đều tạo ra một đối tượng mới.
   * **An toàn đồng thời (Thread-safe)**: Do tính bất biến, các đối tượng **String** là an toàn đồng thời mặc dù nhiều luồng có thể truy cập cùng một đối tượng.
   * **Hiệu suất yếu**: Khi thay đổi hoặc nối chuỗi, một đối tượng mới phải được tạo ra, dẫn đến hiệu suất kém trong một số tình huống.
2. **StringBuilder**:
   * **Không bất biến (Mutable)**: Các đối tượng **StringBuilder** có thể thay đổi giá trị của chuỗi mà không cần tạo ra đối tượng mới.
   * **Không an toàn đồng thời (Not thread-safe)**: **StringBuilder** không đảm bảo an toàn đồng thời, nên nếu sử dụng trong môi trường có nhiều luồng, bạn cần tự đảm bảo đồng bộ hóa.
   * **Hiệu suất cao**: Do không tạo ra các đối tượng mới khi thay đổi chuỗi, **StringBuilder** thường nhanh hơn trong việc thao tác nối chuỗi hoặc biến đổi chuỗi.
3. **StringBuffer**:
   * **Không bất biến (Mutable)**: Giống như **StringBuilder**, các đối tượng **StringBuffer** có thể thay đổi giá trị của chuỗi.
   * **An toàn đồng thời (Thread-safe)**: **StringBuffer** có tính an toàn đồng thời, tức là nhiều luồng có thể truy cập một đối tượng **StringBuffer** mà không cần đồng bộ hóa.
   * **Hiệu suất thấp hơn StringBuilder**: **StringBuffer** cung cấp tính năng an toàn đồng thời nhưng có thể chậm hơn **StringBuilder** trong việc thao tác nối chuỗi hoặc biến đổi chuỗi.

Lựa chọn giữa **String**, **StringBuilder** và **StringBuffer** phụ thuộc vào tình huống sử dụng của bạn. Nếu bạn chỉ cần làm việc với chuỗi không thay đổi (ví dụ: làm việc với hằng số chuỗi), sử dụng **String** là hợp lý. Nếu bạn cần thay đổi chuỗi và hiệu suất là yếu tố quan trọng, hãy sử dụng **StringBuilder**. Nếu bạn cần thay đổi chuỗi trong môi trường đa luồng, hãy sử dụng **StringBuffer**.

**5. Ép kiểu**

**5.1 Khái niệm ép kiểu dữ liệu trong Java là gì? Đưa ra ví dụ.**

Ép kiểu dữ liệu (type casting) trong Java là quá trình chuyển đổi một kiểu dữ liệu sang một kiểu dữ liệu khác. Ép kiểu có thể được thực hiện khi bạn muốn gán một biến của một kiểu dữ liệu vào một biến của kiểu dữ liệu khác, hoặc khi bạn cần thực hiện các phép toán trên các kiểu dữ liệu khác nhau.

Có hai loại ép kiểu chính trong Java: ép kiểu ngầm định (implicit casting, còn gọi là widening casting) và ép kiểu tường minh (explicit casting, còn gọi là narrowing casting).

**1. Ép kiểu ngầm định (Implicit Casting):**

Ép kiểu ngầm định xảy ra khi một giá trị của kiểu dữ liệu nhỏ hơn được gán cho một biến của kiểu dữ liệu lớn hơn. Java thực hiện ép kiểu ngầm định một cách tự động và an toàn.

Ví dụ:

int myInt = 10;

double myDouble = myInt; // Ép kiểu ngầm định từ int sang double

**2. Ép kiểu tường minh (Explicit Casting):**

Ép kiểu tường minh xảy ra khi bạn cần chuyển đổi một biến của kiểu dữ liệu lớn hơn sang kiểu dữ liệu nhỏ hơn. Bạn cần phải sử dụng từ khóa () để thực hiện ép kiểu tường minh.

Ví dụ:

double myDouble = 10.5;

int myInt = (int) myDouble; // Ép kiểu tường minh từ double sang int

Lưu ý rằng ép kiểu tường minh có thể gây ra mất mát dữ liệu nếu giá trị của biến không thể được chứa trong kiểu dữ liệu đích. Ví dụ, ép kiểu một giá trị double lớn sang int có thể làm mất dữ liệu thập phân.

double myDouble = 10.8;

int myInt = (int) myDouble; // Giá trị thập phân sẽ bị mất

Sự cẩn trọng cần được thực hiện khi sử dụng ép kiểu, đặc biệt là ép kiểu tường minh, để tránh mất mát dữ liệu không mong muốn hoặc lỗi trong chương trình.

**5.2 Khái niệm ép kiểu đối tượng trong Java là gì? Đưa ra ví dụ.**

Ép kiểu đối tượng (Object Casting) trong Java là quá trình chuyển đổi giữa các kiểu đối tượng khác nhau. Khi bạn có một biến đối tượng của một lớp cha và muốn sử dụng nó như một biến của một lớp con hoặc ngược lại, bạn có thể thực hiện ép kiểu đối tượng.

Có hai loại ép kiểu đối tượng chính trong Java: ép kiểu ngầm định (implicit casting) và ép kiểu tường minh (explicit casting).

**1. Ép kiểu ngầm định (Implicit Casting)**:

Ép kiểu ngầm định xảy ra khi bạn chuyển đối từ một lớp con sang một lớp cha mà các lớp này có mối quan hệ kế thừa. Java thực hiện ép kiểu ngầm định một cách tự động và an toàn.

Ví dụ:

class Animal { }

class Dog extends Animal { }

Dog myDog = new Dog();

Animal myAnimal = myDog; // Ép kiểu ngầm định từ Dog sang Animal

**2. Ép kiểu tường minh (Explicit Casting)**:

Ép kiểu tường minh xảy ra khi bạn chuyển đối từ một lớp cha sang một lớp con hoặc giữa các lớp không có mối quan hệ kế thừa. Bạn cần sử dụng từ khóa **()** và chỉ định kiểu cần ép kiểu.

Ví dụ:

class Animal { }

class Dog extends Animal { }

Animal myAnimal = new Dog();

Dog myDog = (Dog) myAnimal; // Ép kiểu tường minh từ Animal sang Dog

**6. Constructor:**

**6.1 Constructor là gì trong Java?**

Constructor trong Java là một kiểu phương thức đặc biệt mà được sử dụng để khởi tạo đối tượng.

**6.2 Có mấy loại constructor?**

-Có hai loại constructor, hàm khởi tạo mặc định(default constructor or no-arg constructor)

-Hàm khởi tạo có tham số (parameterized constructor)

**6.3 Cách nào để tạo constructor nhanh trong IDE Eclipse hoặc Intellij?**

  - Nhấn tổ hợp phím alt + shift + s (generate setter and getter)

**7.  Access modifier:**

**7.1**  **Liệt kê và giải thích các phạm vi truy cập (access modifier) trong Java:** private, default, protected, public.

* Private: chỉ được truy cập trong phạm vi lớp
* Default: chỉ được phép truy cập trong cùng package.
* protected: được truy cập bên trong package và bên ngoài package nhưng phải kế thừa.
* public: được truy cập ở mọi nơi.

**7.2** **Khi nào chúng ta sử dụng private, default, protected và public trong việc khai báo các thành viên lớp?**

* Private: sử dụng trong việc khai báo các thành viên lớp truyền dữ liệu trong cùng một lớp (class).
* Default: sử dụng trong việc khai báo các thành viên lớp truyền dữ liệu trong cùng một lớp (class) và một gói (package).
* Protected: sử dụng khi trong việc khai báo thành viên truyền dữ liệu trong cùng một lớp, package, lớp con (subclass).
* Public: sử dụng khi trong việc khai báo thành viên truyền dữ liệu đến bất cứ khu vực nào.

**8. Non-access modifier:**

**8.1** Liệt kê và giải thích các non-access modifier: abstract, static, final, synchronized, transient, volatile (khi kết hợp với field, method hoặc class).

* abstract: Một lớp abstract có thể không bao giờ được khởi tạo. Một lớp được khai báo là abstract thì mục đích duy nhất cho lớp này là để được kế thừa. Một lớp không thể vừa abstract và final (một lớp final không thể được kế thừa).
* static: Sử dụng để tạo các biến mà sẽ tồn tại một cách độc lập trong bất kỳ instance được tạo cho lớp đó. Chỉ có một bản sao biến static tồn tại cho dù có nhiều instance của lớp. Biến static cũng được biết đến như là các biến class. Biến local không thể được khai báo là static.
* final: biến final có thể được khởi tạo một cách rõ ràng chỉ một lần. Một biến tham chiếu khai báo final có thể không bao giờ được gán để tham chiếu tới đối tượng khác. Final thường được sử dụng với static để tạo một hằng số cho một biến class.
* synchronized: sử dụng để chỉ rằng một phương thức có thể được truy cập bởi chỉ một thread tại một thời điểm. Synchronized có thể được áp dụng với bất kỳ một trong 4 mức độ access modifier.
* transient: Một biến instance được đánh dấu là transient để chỉ rằng JVM bỏ qua biến cụ thể khi xếp thứ tự đối tượng đang chứa nó.
* volatile: sử dụng để chỉ cho JVM biết rằng một thread đang truy cập biến đó phải luôn luôn sáp nhập bản sao biến private của riêng nó với bản sao master trong bộ nhớ. Một tham chiếu đối tượng Volatile có thể là null.

**9. Exception và xử lý exception:**

**9.1 Khái niệm về exception trong Java là gì?**

* Exceptions là một sự kiện xảy ra khi một chương trình đang chạy (thực thi), sự kiện đó làm cho luồng xử lý thông thường của chương trình không thể thực hiện một cách bình thường, thậm chí chết chương trình.
* Có 3 loại exception là Error, checked exception, và unchecked exception (runtime exception).
* Error: là những lỗi nghiêm trọng xảy ra lỗi khi chương trình hoạt động mà lập trình viên không thể kiểm soát. Ví dụ như lỗi phần cứng, hay lỗi của JVM.
* Checked exceptions: là những exception ta phải xử lý ngay khi viết code, vì nó được kiểm tra bởi trình biên dịch Javac. Ví dụ: ClassNotFoundException, NoSuchFieldException…
* Unchecked exceptions: là những exception chỉ xảy ra khi chương trình chạy, nghĩa là trình biên dịch Javac không "phát hiện" ra khi biên dịch, do vậy programmer không thể xử lý khi viết code. Ví dụ: NullPointerException, ArrayIndexOutOfBoundsException, DivideByZeroException...

**9.2 Làm thế nào để xử lý exception trong Java?**

* Sử dụng try catch để khắc phục exception

**10. Abstract và Interface:**

**10.1 Khái niệm abstract class và abstract method trong Java là gì?**

* abstract clasṣ (lớp trừu tượng):
* Lớp trừu tượng có thể có các phương thức abstract hoặc non-abstract.
* Lớp trừu tượng có thể khai báo 0, 1 hoặc nhiều method trừu tượng bên trong.
* Không thể khởi tạo 1 đối tượng trực tiếp từ một lớp trừu tượng.
* abstract method (phương thức trừu tượng):
* Một phương thức được khai báo là abstract và không có trình triển khai thì đó là phương thức trừu tượng.
* Phương thức abstract sẽ không có định nghĩa, được theo sau bởi dấu chấm phẩy, không có dấu ngoặc nhọn theo sau.

**10.2 Khái niệm interface trong Java là gì?**

* interface là một cách chỉ định hành vi của một lớp bằng cách cung cấp một tập hợp phương thức trừu tượng
* Interface được sử dụng để đạt được tính trừu tượng mà còn tính đa hình và đa kế thừa.

**10.3 Sự khác nhau giữa abstract class và interface trong Java**.

Abstract class và interface là hai khái niệm quan trọng trong Java để thực hiện tính trừu tượng và đa kế thừa. Dưới đây là sự khác nhau giữa chúng:

1. **Abstract Class (Lớp trừu tượng):**
   * Lớp trừu tượng có thể chứa các phương thức trừu tượng (phương thức mà chỉ có khai báo mà không có cài đặt) và cũng có thể chứa phương thức có cài đặt.
   * Một lớp con kế thừa từ lớp trừu tượng phải triển khai tất cả các phương thức trừu tượng của lớp cha, trừ khi lớp con cũng là một lớp trừu tượng.
   * Một lớp trừu tượng không thể được khởi tạo trực tiếp (không thể tạo đối tượng từ lớp trừu tượng).
   * Lớp trừu tượng có thể có các biến thành viên (fields) và cài đặt các phương thức không trừu tượng.
   * Một lớp con chỉ có thể kế thừa từ một lớp trừu tượng duy nhất.
2. **Interface (Giao diện):**
   * Giao diện chỉ chứa các phương thức trừu tượng và không thể cài đặt các phương thức có sẵn.
   * Một lớp có thể triển khai nhiều giao diện, cho phép thực hiện đa kế thừa của hành vi.
   * Các lớp triển khai giao diện phải triển khai tất cả các phương thức trừu tượng của giao diện đó.
   * Giao diện không thể chứa biến thành viên, trừ những biến được khai báo là hằng số (final).
   * Giao diện là một cách để đảm bảo rằng các lớp khác nhau có thể thực hiện cùng một hành vi mà không cần phải kế thừa cấu trúc lớp.

Tóm lại, một abstract class thường được sử dụng khi bạn có một số phương thức đã được cài đặt và bạn muốn chia sẻ chúng giữa các lớp con. Một interface được sử dụng để định nghĩa hành vi chung mà các lớp khác nhau có thể triển khai mà không cần quan tâm đến cấu trúc của lớp.