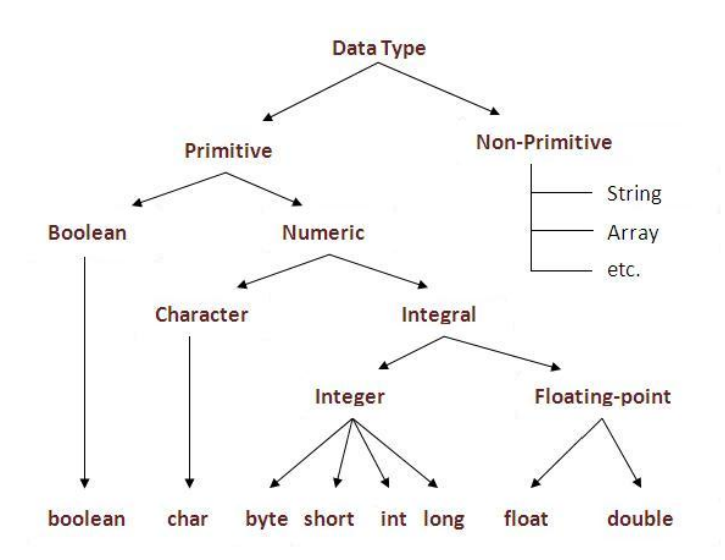
**Bài tập chương 1**

**Bài 1: Mô tả các kiểu dữ liệu cơ sở trên Java**

Trong Java có hai kiểu dữ liệu chính gồm **Kiểu dữ liệu nguyên thủy** và **Kiểu dữ liệu không nguyên thủy.**

****

**Bài 2: Mô tả cấu trúc lệnh trên Java**

Trong lập trình Java thì gồm có 3 cấu trúc lệnh:

1. Cấu trúc lệnh tuần tự: Chương trình sẽ thực thi theo hướng từ trên xuống và thực hiện dòng lệnh theo hướng từ trái sang phải.
2. Cấu trúc lệnh rẽ nhánh: Chương trình sẽ thực hiện kiểm tra một điều kiện nếu đúng chương trình sẽ thực hiện đoạn mã A, nếu sai chương trình sẽ thực hiện đoạn mã B.
3. Cấu trúc lệnh lặp: Chương trình sẽ thực hiện xét kiểm tra điều kiện nếu đúng sẽ thực hiện đoạn mã A đến khi điều kiện sai.

**Bài 3: Mô tả các môi trường lập trình Java**

* JDK: Bộ công cụ phát triển Java, cung cấp mọi thứ cần thiết để viết, biên dịch và chạy chương trình Java.
* JRE: Chỉ chứa các thành phần cần thiết để chạy ứng dụng Java, không bao gồm trình biên dịch.
* JVM: Máy ảo Java giúp thực thi mã bytecode trên nhiều hệ điều hành khác nhau.
* …

**Bài 4: Liệt kê các đặc trưng của nền tảng lập trình Java**

|  |  |
| --- | --- |
| **Nền tảng** | **Ứng dụng chính** |
| Java SE | Ứng dụng desktop, console, công cụ hệ thống |
| Java EE (Jakarta EE) | Ứng dụng web, doanh nghiệp, dịch vụ web |
| Java ME | Ứng dụng nhúng, IoT, thiết bị di động cũ |
| Android | Ứng dụng Android, thiết bị di động |
| Spring Framework | Microservices, ứng dụng web, doanh nghiệp |
| GraalIVM | Java hiệu suất cao, hỗ trợ đa ngôn ngữ |

**Bài 5: Trình bày function trên Java**

|  |  |
| --- | --- |
| **Loại phương thức** | **Đặc điểm** |
| Không có tham số, không trả về | Chỉ thực hiện hành động, không trả về dữ liệu |
| Có tham số, không trả về | Nhận dữ liệu đầu vào, thực hiện hành động |
| Có tham số, có trả về | Xử lý dữ liệu và trả về kết quả |
| Static | Gọi trực tiếp không cần đối tượng |
| Nạp chồng phương thức (Overloading) | Cùng tên nhưng khác tham số |

**Bài 6: So sánh các câu lệnh lập trên Java**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Loại vòng lặp** | **Cú pháp** | **Kiểm tra điều kiện** | **Số lần lặp tối thiểu** | **Dùng khi nào?** |
| For | for (init; condition; update){} | Trước mỗi lần lặp | 0 | Khi biết trước số lần lặp |
| While | While (condition) {} | Trước mỗi lần lặp | 0 | Khi không biết trước số lần lặp, kiểm tra trước |
| Do – While | do { } while (condition); | Sau mỗi lần lặp | 1 | Khi cần chạy ít nhất một lần |

**Bài 7: Cho ví dụ về hàm đệ quy và không đệ quy trong lập trình Java**

* **Hàm không đệ quy:**

Public class Main(){

    Public static void main(String[] args){

        Function();

    }

    Public void function(){

        System.out.println(“Hello World”);

    }

}

* **Hàm đệ quy:**

Public class Main(){

  Public static void main(String[] args){

        Function();

    }

    Public void Function(int n){

      If( n = 5)

            Return;

       If (n < 5)

            Return Funtion(n + 1);

    }

}

**Bài 8: Tóm tắt kiểu dữ liệu String**

Trong Java, String là một kiểu dữ liệu đặc biệt dùng để lưu trữ chuỗi ký tự và có tính bất biến (immutable), tức là không thể thay đổi sau khi tạo.

**1. Cách tạo chuỗi**

Chuỗi có thể được tạo bằng hai cách:

Chuỗi **literal: String s = "Hello";** (tối ưu bộ nhớ nhờ String Pool).

Từ khóa **new: String s = new String("Hello");** (luôn tạo đối tượng mới trong bộ nhớ heap).

**2. Một số phương thức phổ biến**

**length()** – Lấy độ dài chuỗi.

**charAt(index)** – Lấy ký tự tại vị trí cụ thể.

**toUpperCase(), toLowerCase()** – Chuyển đổi chữ hoa, chữ thường.

**concat(str)** – Nối chuỗi.

**contains(substring)** – Kiểm tra chuỗi con.

**equals(str), equalsIgnoreCase(str)** – So sánh chuỗi.

**3. StringBuilder và StringBuffer**

Do String không thể thay đổi, khi thao tác nhiều với chuỗi, nên dùng:

**StringBuilder** – Hiệu suất cao, không hỗ trợ đa luồng.

**StringBuffer** – Tương tự StringBuilder nhưng hỗ trợ đa luồng (thread-safe).

**Kết luận**

**String** giúp tối ưu bộ nhớ nhờ **String Pool**, nhưng nếu cần thao tác chuỗi thường xuyên, **StringBuilder hoặc StringBuffer** sẽ hiệu quả hơn.

**Bài 9: Cho ví dụ về lớp bao trên Java**

Lớp bao (**Wrapper Class**) trong Java giúp chuyển đổi các kiểu dữ liệu nguyên thủy (int, double, char, v.v.) thành đối tượng, hỗ trợ nhiều phương thức hữu ích. Dưới đây là một số ví dụ cơ bản về cách sử dụng lớp bao:

**1. Auto-boxing và Unboxing**

* **Auto-boxing** là quá trình Java tự động chuyển đổi một giá trị kiểu nguyên thủy thành đối tượng lớp bao tương ứng. Ví dụ, khi gán một giá trị int cho một biến kiểu Integer, Java tự động đóng gói nó.
* **Unboxing** là quá trình ngược lại, khi một đối tượng của lớp bao được chuyển đổi trở lại kiểu nguyên thủy. Điều này xảy ra khi sử dụng giá trị của một đối tượng Integer trong một phép toán số học.

**2. Chuyển đổi chuỗi thành số**

Lớp bao cung cấp các phương thức để chuyển đổi dữ liệu từ chuỗi (String) sang các kiểu số (int, double, v.v.). Ví dụ, Integer.parseInt("123") sẽ chuyển đổi chuỗi "123" thành số nguyên 123, giúp xử lý dữ liệu đầu vào từ người dùng một cách dễ dàng hơn.

**3. Sử dụng các hằng số của Wrapper Class**

Mỗi lớp bao đều có các hằng số như MAX\_VALUE và MIN\_VALUE, cho biết giới hạn lớn nhất và nhỏ nhất của kiểu dữ liệu đó. Ví dụ, Integer.MAX\_VALUE cho biết giá trị lớn nhất mà một số nguyên kiểu int có thể lưu trữ.

**4. Sử dụng Wrapper Class trong Collection**

Trong Java, các cấu trúc dữ liệu như ArrayList chỉ có thể chứa đối tượng, không thể chứa kiểu nguyên thủy. Vì vậy, nếu muốn lưu danh sách các số nguyên trong ArrayList, cần sử dụng Integer thay vì int. Điều này giúp Java tự động chuyển đổi giữa kiểu nguyên thủy và đối tượng khi thêm hoặc lấy giá trị từ danh sách.

**5. So sánh hai đối tượng Wrapper Class**

Để so sánh giá trị của hai đối tượng Wrapper Class, nên sử dụng phương thức .equals() thay vì toán tử ==. Toán tử == chỉ so sánh địa chỉ vùng nhớ, trong khi .equals() so sánh giá trị thực tế bên trong đối tượng.

**Bài 10: So sánh Java với ngôn ngữ lập trình C++**

Java và C++ đều là ngôn ngữ lập trình phổ biến, nhưng có nhiều điểm khác biệt quan trọng:

**1. Cách biên dịch và thực thi**

- Java: Chạy trên máy ảo JVM (Java Virtual Machine), giúp mã nguồn có thể chạy trên nhiều nền tảng khác nhau.

- C++: Biên dịch trực tiếp thành mã máy, nên chương trình chạy nhanh hơn nhưng phụ thuộc vào hệ điều hành.

**2. Quản lý bộ nhớ**

- Java: Tự động thu gom rác (Garbage Collection), giúp quản lý bộ nhớ dễ dàng.

- C++: Lập trình viên tự cấp phát và giải phóng bộ nhớ bằng **new** và **delete**, dễ gây lỗi tràn bộ nhớ.

**3. Lập trình hướng đối tượng**

- Java: Hoàn toàn hướng đối tượng (mọi thứ đều là đối tượng, trừ kiểu nguyên thủy).

- C++: Hỗ trợ hướng đối tượng nhưng cũng có thể lập trình theo hướng thủ tục.

**4. Tính đa kế thừa**

- Java: Không hỗ trợ đa kế thừa trực tiếp (dùng interface thay thế).

- C++: Hỗ trợ đa kế thừa, nhưng dễ gây lỗi phức tạp.

**5. Tốc độ thực thi**

- Java: Chạy chậm hơn do sử dụng JVM.

- C++: Chạy nhanh hơn vì biên dịch trực tiếp thành mã máy.

**6. Sử dụng con trỏ**

- Java: Không hỗ trợ con trỏ trực tiếp để tăng tính an toàn.

- C++: Hỗ trợ con trỏ mạnh mẽ, nhưng có thể gây lỗi truy cập bộ nhớ.

**7. Ứng dụng chính**

- Java: Phù hợp với ứng dụng web, di động (Android), phần mềm doanh nghiệp.

- C++: Dùng nhiều trong lập trình hệ thống, game, đồ họa, ứng dụng nhúng.

Kết luận

- Java dễ học hơn, an toàn hơn, phù hợp với ứng dụng đa nền tảng và doanh nghiệp.

- C++ mạnh mẽ hơn về hiệu suất, thích hợp cho lập trình hệ thống và game.

**Bài 11: Lập trình tính tổng 1 + 2 + .... + n**

**Trong file Code**

**Bài 12: Lập trình tính n!**

**Trong file code**

**Bài 13: Lập trình hàm kiểm tra số nguyên n có là số nguyên tố hay không**

**Trong file code**

**Bài 14: Lập trình hàm nhận vào họ tên, trả về Họ Tên dạng proper**

**Trong file code**

**Bài 15: Lập trình hàm xuất ra mã nhị phân của số nguyên n**

**Trong file code**

**Bài 16: Phân tích câu lệnh switch thành các lệnh if, lấy ví dụ minh họa**

* **Câu lệnh If:**

package bt16;

public class bt16 {

    public static void main(String[] args) {

        int number = 2;

        if (number == 1) {

            System.out.println("Số một");

        } else if (number == 2) {

            System.out.println("Số hai");

        } else if (number == 3) {

            System.out.println("Số ba");

        } else {

            System.out.println("Không xác định");

        }

    }

}

* **Câu lệnh Switch**

package bt16;

public class bt16 {

    public static void main(String[] args) {

        int number = 2;

        switch (number) {

            case 1:

                System.out.println("Số một");

                break;

            case 2:

                System.out.println("Số hai");

                break;

            case 3:

                System.out.println("Số ba");

                break;

            default:

                System.out.println("Không xác định");

        }

    }

}

**Đánh Giá Và Phân Tích**

Tính chất :

**IF :**

Có thể kiểm tra nhiều điều kiện với các toán tử so sánh (==, !=, <, >, <=, >=) và các biểu thức logic (&&, ||, !).

**SWITCH :**

Switch chỉ so sánh giá trị cụ thể của một biến với các case cố định (int, char, String, enum)

|  |  |  |
| --- | --- | --- |
| **Tiêu chí** | **if-else** | **switch** |
| Điều kiện | Kiểm tra bất kỳ điều kiện nào với các toán tử (==,>,<,>=,<=,&&,…) | Chỉ kiểm tra với các giá trị cố định |
| Tính đọc hiểu | Có thể khó đọc nếu có quá nhiều điều kiện lồng nhau | Dễ đọc hơn với các giá trị cố định |
| Hiệu suất | Có thể chậm hơn do phải kiểm tra từng điều kiện | Thường nhanh hơn nhờ tối ưu hóa của JVM |
| Khả năng mở rộng | Linh hoạt hơn, có thể kiểm tra phạm vi điều kiện | Hạn chế hơn, chỉ kiểm tra giá trị cố định |
| Hỗ trợ kiểu dữ liệu | Bất kỳ kiểu dữ liệu nào | Chỉ hỗ trợ int, char,String, enum |
| Sử dụng break | Không cần | Cần break để tránh thực thi tiếp các case khác |

**Khi nào dùng?**

IF:

* Kiểm tra các điều kiện phức tạp như so sánh lớn hơn, nhỏ hơn (x > 10, y < 20).
* Kiểm tra nhiều điều kiện logic (&&, ||).
* Không thể dùng switch vì điều kiện không phải là int, char, String hoặc enum.

SWITCH:

* Cần kiểm tra một biến với nhiều giá trị cố định.
* Cần hiệu suất tốt hơn khi có nhiều giá trị cần kiểm tra.
* Cần mã nguồn dễ đọc hơn.

**Bài 17: Phân tích câu lệnh try catch, lấy ví dụ minh họa**

Try – catch :

package bt17;

public class Main {

    public static void main(String[] args) {

        try {

            int result = 10 / 0; // Lỗi chia cho 0

            System.out.println(result);

        } catch (ArithmeticException e) {

            System.out.println("Lỗi: Chia cho 0 không hợp lệ!");

        } finally {

            System.out.println("Khối finally luôn được thực thi.");

        }

    }

}

**Phân Tích Try- Catch**

**Chức năng:**

* try: Chứa đoạn code có thể gây lỗi.
* catch: Bắt và xử lý lỗi khi có ngoại lệ xảy ra.

**Ý nghĩa :**

* try: Chứa đoạn mã có thể phát sinh lỗi.
* catch: Chặn và xử lý ngoại lệ nếu xảy ra.
* finally *(tùy chọn)*: Chạy dù có lỗi hay không (thường dùng để đóng tài nguyên như file, database...).

**Bài 18: Biến đổi cấu trúc lệnh lặp for thành lệnh lặp điều kiện trước bằng ví dụ**

package bt18;

public class Main {

    public static void ForLoopExample () {

        for (int i = 1; i <= 5; i++) {

            System.out.println("Giá trị i: " + i);

        }

    }

    public static void WhileLoopExample(){

        int i = 1;

        while (i <= 5) {

            System.out.println("Giá trị i: " + i);

            i++;

        }

    }

    public static void main(String[] args ){

        System.out.println("Day la vong lap for");

        ForLoopExample();

        System.out.println("Day la vong lap while");

        WhileLoopExample();

    }

}

**Phân Tích & Tính Chất**

FOR

Dùng khi biết trước số lần lặp.

While

Dùng khi không biết trước số lần lặp, chỉ cần điều kiện còn đúng thì tiếp tục chạy.

|  |  |  |
| --- | --- | --- |
| Tiêu chí | For | While |
| Khi nào dùng? | Khi biết trước số lần lặp | Khi chưa biết trước số lần lặp |
| Cấu trúc | Gồm khởi tạo, điều kiện, cập nhật trong một dòng | Chỉ có điều kiện, khởi tạo & cập nhật tách rời |
| Tính đọc hiểu | Dễ đọc nếu số lần lặp cố định | Dễ hơn nếu điều kiện thay đổi theo luồng chương trình |
| Hiệu suất | Ngang nhau, tùy vào cách sử dụng | Ngang nhau tùy vào cách sử dụng |
| Tính linh hoạt | Ít linh hoạt hơn | Linh hoạt hơn khi cần điều kiện phức tạp |
| Nguy cơ lặp vô hạn | Ít xảy ra vì có cập nhật trong dòng for | Dễ xảy ra nếu quên cập nhật điều kiện |

**Bài 19: Phân tích quá trình vận hành của chương trình Java bao gôm nhiều lời gọi hàm cụ thể**

package bt1;  
  
public class bt19 {  
 public static void main(String[] args) {  
 System.*out*.println("Chương trình bắt đầu.");  
  
 int number = 5;  
  
 // Gọi hàm tính tổng từ 1 đến n  
 int sumResult = *sumToN*(number);  
 System.*out*.println("Tổng từ 1 đến " + number + " là: " + sumResult);  
  
 // Gọi hàm tính giai thừa  
 int factorialResult = *factorial*(number);  
 System.*out*.println("Giai thừa của " + number + " là: " + factorialResult);  
  
 // Gọi hàm kiểm tra số nguyên tố  
 boolean isPrime = *checkPrime*(number);  
 if (isPrime) {  
 System.*out*.println(number + " là số nguyên tố.");  
 } else {  
 System.*out*.println(number + " không phải là số nguyên tố.");  
 }  
  
 System.*out*.println("Chương trình kết thúc.");  
 }  
  
 // Hàm tính tổng từ 1 đến n  
 public static int sumToN(int n) {  
 int sum = 0;  
 for (int i = 1; i <= n; i++) {  
 sum += i;  
 }  
 return sum;  
 }  
  
 // Hàm tính giai thừa của một số nguyên dương  
 public static int factorial(int n) {  
 if (n <= 1) { // Điều kiện đơn giản hơn  
 return 1;  
 }  
 return n \* *factorial*(n - 1);  
 }  
  
 // Hàm kiểm tra số nguyên tố  
 public static boolean checkPrime(int n) {  
 if (n < 2) return false;  
 for (int i = 2; i <= Math.*sqrt*(n); i++) {  
 if (n % i == 0) return false;  
 }  
 return true;  
 }  
}

**1. Quá Trình Biên Dịch và Chạy Chương Trình**

* Khi chạy chương trình, Java Compiler dịch file bt19.java thành bytecode .class, sau đó JVM thực thi chương trình.
* Phương thức main là điểm bắt đầu của chương trình, nơi gọi các phương thức con để thực hiện từng tác vụ.

**2. Bắt Đầu Chương Trình**



Khởi tạo biến



**2.1. Gọi Hàm sumToN(int n)**

**Quá trình thực thi:**

* sumToN(5) thực hiện vòng lặp từ 1 đến 5 và tính tổng:

sum = 1 + 2 + 3 + 4 + 5 = 15

* Trả về 15, sau đó in ra màn hình



**2.2. Gọi Hàm** factorial(int n) **(Tính Giai Thừa)**

#### **Quá trình đệ quy:**

1. factorial(5) gọi factorial(4)
2. factorial(4) gọi factorial(3)
3. factorial(3) gọi factorial(2)
4. factorial(2) gọi factorial(1)
5. factorial(1) = 1 (Điều kiện dừng)
6. Quay ngược lại nhân dần:
7. Trả về 120, sau đó in ra:



**2.3. Gọi Hàm** checkPrime(int n)

* Kiểm tra number = 5 có phải số nguyên tố không:
  + 5 > 2 → Không loại ngay.
  + Duyệt từ 2 đến √5 ≈ 2.23 (chỉ kiểm tra i = 2).
  + 5 % 2 != 0 → Không chia hết.
  + Kết luận 5 là số nguyên tố.
* In ra:



Kết thúc chương trình

**Bài 20: Phân tích quá trình vận hành của chương trình Java bao gồm nhiều lời gọi hàm đệ quy bằng ví dụ cụ thể.**

package bt1;  
  
public class bt20 {  
 public static void main(String[] args) {  
 int n = 5;  
 System.*out*.println("Tổng từ 1 đến " + n + " là: " + *sumToN*(n));  
 }  
  
 public static int sumToN(int n) {  
 if (n == 1) return 1; // Điều kiện dừng  
 return n + *sumToN*(n - 1); // Gọi lại với n - 1  
 }  
  
}

**1. Quá Trình Biên Dịch và Chạy Chương Trình**

* **Bước 1**: Chương trình được biên dịch (javac bt20.java).
* **Bước 2**: JVM thực thi chương trình (java bt20).
* **Bước 3**: Chương trình bắt đầu thực thi từ phương thức main.

**2. Chi Tiết Vận Hành**

**2.1. Gọi Hàm sumToN(5)**

* **Trong main()**, biến n = 5 được khởi tạo.
* Gọi hàm sumToN(5) để tính tổng.

**2.2. Quá Trình Đệ Quy**

Hàm sumToN(int n) hoạt động theo cơ chế đệ quy giảm dần:

* sumToN(5) → **Chưa đạt điều kiện dừng**, gọi sumToN(4)
* sumToN(4) → **Chưa đạt điều kiện dừng**, gọi sumToN(3)
* sumToN(3) → **Chưa đạt điều kiện dừng**, gọi sumToN(2)
* sumToN(2) → **Chưa đạt điều kiện dừng**, gọi sumToN(1)
* sumToN(1) → **Thỏa điều kiện dừng**, trả về 1.

**2.3. Quá Trình Quay Lại (Tính Tổng)**

Sau khi đạt điều kiện dừng (sumToN(1) = 1), các hàm đang chờ sẽ quay ngược lại và tính tổng:

sumToN(2) = 2 + sumToN(1) = 2 + 1 = 3

sumToN(3) = 3 + sumToN(2) = 3 + 3 = 6

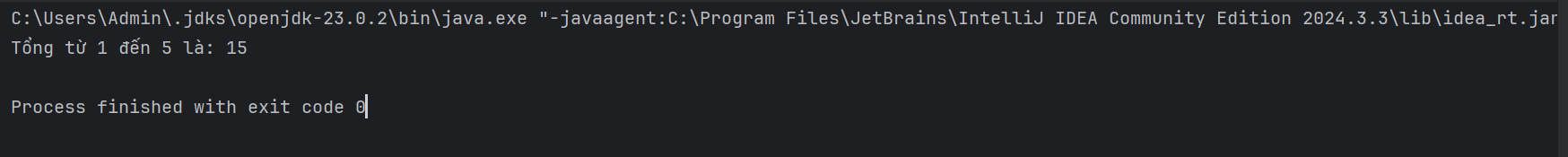
sumToN(4) = 4 + sumToN(3) = 4 + 6 = 10

sumToN(5) = 5 + sumToN(4) = 5 + 10 = 15

Kết quả trả về là 15.

**2.4. In Kết Quả**

Sau khi hàm sumToN(5) trả về kết quả 15, main() in ra màn hình:



Kết thúc chương trình

**Đánh Giá Hiệu Suất**

| **Phương pháp** | **Độ phức tạp** | **Nhược điểm** |
| --- | --- | --- |
| **Đệ quy (sumToN)** | O(n) | Tốn bộ nhớ do stack gọi hàm đệ quy |
| **Vòng lặp (for)** | O(n) | Tiết kiệm bộ nhớ hơn, dễ hiểu hơn |
| **Công thức toán học**: S = n \* (n + 1) / 2 | O(1) | Tối ưu nhất |