PHƯƠNG THỨC VÀ XỬ LÝ NGOẠI LỆ

NỘI DUNG TRONG CHƯƠNG

- *Khái niệm về phương thức, phương thức tĩnh, biến tĩnh*
- *Tham số và các loại tham số*
- *Chồng phương thức*
- *Ngoại lệ và xử lý ngoại lệ*
- *Tạo tài liệu bằng Javadocs*

Chương 4 tập trung vào các kiến thức liên quan đến phương thức, đặc biệt là phương thức tĩnh. Các phương thức tĩnh cho phép Java viết mã tương tự như lập trình thủ tục. Các kiến thức cần biết trong phần này bao gồm việc phân biệt được phương thức thông thường và phương thức tĩnh, biến tĩnh, các loại tham số khai báo cũng như tham số truyền vào, tạo và sử dụng phương thức. Ngoài ra người đọc cũng được tiếp cận đến các kiến thức quan trọng về công dụng và các quy tắc khi dùng tham số linh động, tham số mảng và chồng phương thức. Xử lý ngoại lệ cũng là một nội dung quan trọng trong chương. Mục đích của phần này là hướng dẫn người học về các loại ngoại lệ, cách tạo ra, "ném" ngoại lệ. Cuối chương là phần tạo tài liệu bằng Javadocs, công cụ này giúp cho các nhà lập trình dễ dàng tạo ra tài liệu cho dự án một cách nhanh chóng và đúng cú pháp như các hướng dẫn về viết tài liệu của Java.

KHÁI NIỆM VỀ PHƯƠNG THỨC

Phương thức là gì?

Một **phương thức** (method) là một tập các câu lệnh để thực hiện một số tác vụ được định nghĩa trong một lớp (class), được ghép lại với nhau và được đặt tên. Một phương thức có thể hiểu là một **"hộp đen"**:

Hình 4-1 Các phương thức có thể được hiểu là các hộp đen

Phương thức cho phép giải quyết một nhiệm vụ có khả năng rất phức tạp như một **khái niệm duy nhất**. Thay vì quan tâm đến từng bước mà máy tính có thể phải thực hiện để thực hiện tác vụ đó, chỉ cần nhớ tên của phương thức. Bất cứ khi nào muốn chương trình thực hiện tác vụ, chỉ cần gọi phương thức. Các phương thức thực chất là một khối các câu lệnh để thực hiện một công việc nào đó. Một phương thức thường được so sánh với **"hộp đen"** bởi vì không thể nhìn thấy "bên trong" nó có gì (hoặc nói chính xác hơn là không muốn xem bên trong nó, bởi vì khi đó sẽ phải đối mặt với sự phức tạp của các câu lệnh bên trong mỗi phương thức). Một hộp đen cần có thông tin cho biết môi trường bên ngoài có thể tương tác như thế nào với hộp đen. Một chiếc tivi có thể coi như một hộp đen, các nút bấm điều khiển trên tivi chính là cách thức mà môi trường bên ngoài tương tác với hộp đen tivi này.

Quy tắc của hộp đen

Quy tắc về hộp đen cũng là những quy tắc sẽ áp dụng cho phương thức:

- **Giao diện nên đơn giản, rõ ràng và dễ hiểu**: Trong thế giới thực, con người sử dụng hộp đen rất nhiều. Tivi, xe hơi, điện thoại di động, tủ lạnh đều có thể được hiểu là những hộp đen... Đối với tivi, có thể bật và tắt, thay đổi kênh và tăng giảm âm lượng bằng cách sử dụng các thành phần của giao diện tivi như công tắc bật/ tắt, điều khiển từ xa. Điều tương tự cũng xảy ra với điện thoại di động, mặc dù giao diện trong trường hợp đó phức tạp hơn rất nhiều. Quy tắc này cho thấy cần thiết kế một phương thức dễ dùng, dễ gọi bởi các phương thức khác.

- **Để sử dụng, không cần hiểu về cách làm việc của nó; chỉ cần biết giao diện sử dụng:** Trên thực tế, có thể thay đổi cách triển khai bên trong hộp đen nhưng nhìn từ bên ngoài vẫn không thay đổi. Ví dụ: khi bên trong TV chuyển từ công nghệ này sang công nghệ khác, người sử dụng TV không cần biết về điều đó - hoặc thậm chí biết ý nghĩa của nó. Tương tự, có thể viết lại bên trong của một phương thức, ví dụ như sử dụng mã hiệu quả hơn mà không ảnh hưởng đến các chương trình sử dụng phương thức đó.

- **Khi tạo ra hộp đen không cần biết quá nhiều về các hệ thống lớn hơn mà hộp sẽ được sử dụng:** Tất nhiên, để có một hộp đen, phải có ai đó thiết kế và xây dựng việc triển khai ngay từ đầu. Sau đó, hộp đen có thể được sử dụng trong vô số các trường hợp khác nhau. Người triển khai những gì hoạt động bên trong hộp đen không cần biết quá nhiều về hệ thống lớn hơn mà hộp đen sẽ được sử dụng. Người triển khai đơn giản chỉ cần đảm bảo rằng hộp đen thực hiện nhiệm vụ được giao và giao diện chính xác với phần còn lại của thế giới.

ĐỊNH NGHĨA PHƯƠNG THỨC

Một phương thức **phải** được định nghĩa trong một lớp (class) theo cú pháp như sau:

```
1 modifiers return-type function-name(parameter-list) {
2     statements;
3     return;
4 }
```

Định nghĩa một phương thức phải bao gồm:

- **function-name (tên của phương thức)**, đủ thông tin để có thể gọi phương thức và mã sẽ được thực thi mỗi khi phương thức được gọi.

- **statements (các câu lệnh)** giữa cặp dấu ngoặc nhọn { và }, trong định nghĩa phương thức tạo nên phần thân của phương thức. Các câu lệnh này là phần bên trong hoặc phần triển khai của "hộp đen", như đã thảo luận trong phần trước. Đây là các câu lệnh mà máy tính thực thi khi phương thức được gọi.

- **modifier(bổ từ)** có thể xuất hiện ở đầu định nghĩa phương thức là các từ thiết lập các đặc điểm nhất định của phương thức

- **return-type** (**kiểu trả về**) được sử dụng để chỉ định kiểu giá trị được trả về nếu phương thức có công việc là tính toán một số giá trị. Phương thức thực hiện tính toán bằng các câu lệnh và trả về bằng cách sử dụng từ khóa return. Giá trị trả về có thể là một String hoặc int, hoặc thậm chí là một

kiểu mảng như double[]. Nếu phương thức không trả về giá trị, thì kiểu trả về được thay thế bằng giá trị đặc biệt **void**, cho biết rằng không có giá trị nào được trả về. Thuật ngữ "**void**" để chỉ ra rằng giá trị trả về là trống hoặc không tồn tại.

- **parameter-list (danh sách tham số)** là một phần của giao diện của phương thức. Chúng đại diện cho thông tin được truyền vào phương thức từ bên ngoài. Các thông tin này sẽ được sử dụng bởi các tính toán bên trong của phương thức. Danh sách tham số trong phương thức có thể trống hoặc trường hợp có nhiều tham số thì sẽ được phân cách bởi dấu phẩy. Dưới đây là một vài ví dụ về định nghĩa phương thức:

```
 1  public static void playGame () {
 2      // "public" và "static" là các bổ ngữ xác định đây là một phương thức tĩnh côn
g khai;
 3      // "void" là kiểu trả về;
 4      // "playGame" là tên phương thức;
 5      // danh sách tham số trống.
 6      . . . // Các câu lệnh xác định những gì playGame thực hiện ở đây.
 7  }
 8  int getNextN (int N) {
 9      // Không có bổ từ nào;
10      // "int" là kiểu trả về;
11      // "getNextN" là tên phương thức;
12     // Danh sách tham số bao gồm một tham số có tên là "N" và có kiểu là "int".
13      . . . // Các câu lệnh xác định getNextN thực hiện ở đây.
14  }
15  static boolean lessThan (double x, double y) {
16      // "static" là một bổ ngữ xác định đây là một phương thức tĩnh;
17      // "boolean" là kiểu trả về;
18      // "lessThan" là tên phương thức;
19      // danh sách tham số bao gồm hai tham số có tên là "x" và "y" và kiểu của từng
tham số này là "double".
20      . . . // Các câu lệnh xác định những gì lessThan thực hiện ở đây.
21  }
```

Trong ví dụ thứ hai, **getNextN** là một phương thức **non-static** (không tĩnh), vì định nghĩa của nó không bao gồm bộ điều chỉnh "**static**". Bổ từ khác được hiển thị trong các ví dụ là "**public**". Bổ từ này chỉ ra rằng phương thức có thể được gọi từ bất kỳ đâu trong chương trình, ngay cả từ bên ngoài lớp nơi phương thức được định nghĩa. Có một bổ từ khác, "**private**", chỉ ra rằng phương thức chỉ có thể được gọi từ các phương thức trong cùng lớp. Nếu không có chỉ định truy cập nào được cung cấp cho một phương thức, thì phạm vi truy cập của nó được gọi là mặc định "default", khi đó phương thức đó có thể được gọi từ bất kỳ đâu trong package chứa lớp đó, nhưng không được gọi từ bên ngoài package.

Hãy lưu ý rằng **main()** cũng là một ví dụ tiêu biểu của phương thức:

```
1  public static void main (String [] args) {
2  }
```

Trong phương thức trên, các bổ từ là public và static, kiểu trả về là void, tên phương thức là main và danh sách tham số là "String[] args". Trong trường hợp này, kiểu cho tham số là kiểu mảng String[].

PHƯƠNG THỨC TĨNH

Phương thức tĩnh là gì

Một phương thức tĩnh (static) là một phương thức được định nghĩa với từ khóa **static**. Một phương thức tĩnh thuộc lớp và được gọi thông qua lớp chứ không phải thông qua đối tượng của lớp. Điều đó có nghĩa là một phương thức static gọi mà không cần tạo một thể hiện (instance) của một lớp. Phương thức static có thể truy cập biến static và có thể thay đổi giá trị của nó.

Gọi phương thức tĩnh

Để **gọi** phương thức tĩnh từ một phương thức tĩnh khác trong lớp, viết tên phương thức và truyền các tham số đầu vào. Ví dụ:

```
playGame();
```

Để gọi phương thức từ một lớp khác, cần ghi rõ tên lớp:

```
Poker.playGame();
```

Việc gọi phương thức có thể xảy ra ở bất kỳ đâu trong cùng một lớp nơi định định nghĩa phương thức. Vì playGame() là một phương thức public, nó cũng có thể được gọi từ các lớp khác, nhưng trong trường hợp đó, phải cho máy tính biết nó đến từ lớp nào. Vì playGame() là một phương thức static (tĩnh), tên đầy đủ của nó bao gồm tên của lớp mà nó được định nghĩa. Ví dụ, giả sử playGame() được định nghĩa trong một lớp có tên Poker. Việc sử dụng tên lớp ở đây cho máy tính biết lớp nào cần tìm để tìm phương thức. Nó cũng cho phép phân biệt giữa các phương thức Poker.playGame() và các phương thức playGame() khác được xác định trong các lớp khác, chẳng hạn như Roulette.playGame() hoặc Blackjack.playGame(). Lưu ý rằng danh sách tham số có thể trống, như trong ví dụ playGame(), nhưng phải luôn có các dấu ngoặc đơn. Số lượng và kiểu dữ liệu các tham số cung cấp khi gọi một phương thức phải khớp với số lượng và kiểu dữ liệu được chỉ định trong danh sách tham số trong định nghĩa phương thức.

Tổng quát hơn, câu lệnh gọi phương thức cho phương thức tĩnh có dạng

```
tên_lớp.tên_phương thức(các tham số);
```

Xét ví dụ sau:

```
1  public class Main {
2    // Phương thức tĩnh (static)
3    static void myStaticMethod() {
4      System.out.println("Phương thức tĩnh có thể gọi không qua đối tượng");
5    }
6    // Phương thức không tĩnh (non-static)
7    void myPublicMethod() {
8      System.out.println("Phương thức này cần gọi qua đối tượng");
9    }
10   // Main method
11   public static void main(String[] args) {
```

```
12      myStaticMethod(); // Gọi phương thức tĩnh
13      Main myObj = new Main(); // Tạo đối tượng
14      myObj.myPublicMethod();  // Gọi phương thức từ đối tượng
15    }
16  }
```

Kết quả hiển thị:

```
1  Phương thức tĩnh có thể gọi không qua đối tượng
2  Phương thức này cần gọi qua đối tượng
```

BIẾN TĨNH

Các biến thành viên hay còn gọi là biến thuộc tính là các biến được khai báo trong lớp tương tự phương thức. Khi đó biến đó sẽ được gọi là thuộc tính của lớp. Ví dụ:

```
1  class Person{
2     String usersName;
3  }
```

Một biến thành viên có thể được khai báo là các **biến tĩnh** khi sử dụng thêm từ khóa **static**:

```
1  class Player{
2     public static int gamesPlayed;
3     public static int gamesWon;
4  }
```

Các biến tĩnh này thuộc về lớp và nó luôn tồn tại kể cả khi không có đối tượng nào được tạo ra từ biến. Bộ nhớ được cấp phát cho biến khi lớp được tải lần đầu tiên bởi trình thông dịch Java. Bất kỳ câu lệnh gán nào gán giá trị cho biến sẽ thay đổi nội dung của vùng nhớ đó, bất kể câu lệnh gán đó nằm ở đâu trong chương trình. Bất kỳ khi nào biến được sử dụng trong một biểu thức, giá trị sẽ được lấy từ cùng bộ nhớ đó, bất kể biểu thức đó nằm ở đâu trong chương trình. Điều này có nghĩa là giá trị của một biến tĩnh có thể được sử dụng dung bởi nhiều phương thức.

Xét ví dụ sau:

```
 1  static int gamesPlayed;
 2  static int gamesWon;
 3  public static void main(String[] args) {
 4     System.out.println("Chọn 1 số từ 1 đến 100");
 5     boolean playAgain;
 6     do {
 7         playGame();
 8         System.out.print("Bạn có muốn chơi lại? ");
 9         playAgain = new Scanner(System.in).nextBoolean();
10     } while (playAgain);
11     System.out.println("Bạn đã thắng " + gamesWon + "/" + gamesPlayed + " games,");
12     System.out.println("Tạm biệt");
13  }
14  static void playGame() {
15     int computersNumber;
```

```
16      int usersGuess;
17      int guessCount;
18      computersNumber = (int) (100 * Math.random()) + 1;
19      guessCount = 0;
20      gamesPlayed++;
21      System.out.println();
22      System.out.print("Bạn đoán số nào? ");
23      while (true) {
24          usersGuess = new Scanner(System.in).nextInt();
25          guessCount++;
26          if (usersGuess == computersNumber) {
27              System.out.println("Bạn đã đoán đúng, số đó là: " + computersNumber);
28              gamesWon++;
29              break;
30          }
31          if (guessCount == 6) {
32              System.out.println("Bạn đã thua, số đó là " + computersNumber);
33              break;
34          }
35          if (usersGuess < computersNumber)
36              System.out.print("Số quá thấp, hãy đoán lại: ");
37          else if (usersGuess > computersNumber)
38              System.out.print("Số quá cao, hãy đoán lại: ");
39      }
40  }
```

Trong ví dụ trên, chương trình sẽ thêm 2 vào biến tĩnh là gamesPlayed, và gamesWon để xác định số lượt chơi và số lượt chơi chiến thắng. Hai biến này được sử dụng trong các phương thức như playGame() và phương thức main() sẽ in ra giá trị của cả hai biến khi trò chơi kết thúc.

SỬ DỤNG CÁC THAM SỐ

Nếu phương thức là một hộp đen, **tham số** là danh sách các thông tin có thể đưa vào hộp đen. Ví dụ điều hòa nhiệt độ là một hộp đen, thì thông tin các thông tin như nhiệt độ mong muốn, mức gió mong muốn được gọi là các tham số. Có hai loại tham số

- **Tham số hình thức** (formal parameter hay còn gọi là **parameter**) là tham số trong định nghĩa phương thức
- **Tham số thực** (actual parameter hay còn gọi là **argument**) là giá trị thực sự sẽ truyền khi gọi phương thức

Một **tham số hình thức** là một khai báo biến đầu vào. Nó có kiểu được chỉ định như int, boolean, String hoặc mảng như double[]. **Tham số thực** là một giá trị sẽ đưa vào khi gọi/sử dụng phương thức. Kiểu của tham số thực phải là kiểu phù hợp với kiểu của tham số hình thức. Đó sẽ là kiểu mà có thể thực hiện được **lệnh gán**. Ví dụ: nếu tham số hình thức có kiểu double, thì sẽ hợp lệ nếu chuyển một số int làm tham số thực sự vì int có thể được gán hợp lệ cho double.

Ví dụ, tham số hình thức định nghĩa trong phương thức doTask():

```
1  static void doTask(int N, double x, boolean test) {
2      // statements to perform the task go here
```

```
3 }
```

Tham số thực là các giá trị đưa vào khi gọi phương thức:

```
doTask(17, Math.sqrt(z+1), z >= 10);
```

Khi máy tính thực hiện câu lệnh này, về cơ bản nó có tác dụng tương tự như khối lệnh:

```
1 {
2   int N;
3   double x;
4   boolean test;
5   N = 17;
6   x = Math.sqrt(z+1);
7   test = (z >= 10);
8   // các câu lệnh sử dụng các biến trên
9 }
```

Tham số linh động

Tham số linh động (varargs) trong Java cho phép một phương thức có thể nhận số lượng tham số vào khác nhau. Ví dụ:

```
1 public static int sum(int i, int...j){
2     int sum = i;
3     for(int x : j){
4         sum+=x;
5     }
6     return sum;
7 }
```

Chú ý:

- Một phương thức chỉ có một tham số linh động
- Chỉ tham số cuối cùng trong phương thức có thể là tham số linh động
- Hãy cẩn thận khi chồng phương thức với tham số linh động

Tham số mảng

Có thể cho kiểu của một **tham số là một kiểu mảng** (array). Ví dụ:

```
1 static void printValuesInList(int[] list) {
2     System.out.print('[');
3     int i;
4     for ( i = 0; i < list.length; i++ ) {
5         if ( i > 0 )
6             System.out.print(',');
7         System.out.print(list[i]);
8     }
```

```
 9      System.out.println(']');
10  }
```

Điều này có nghĩa là toàn bộ một mảng có thể được truyền cho phương thức dưới dạng một tham số duy nhất. Ví dụ trên, chúng ta muốn một phương thức in tất cả các giá trị trong một mảng số nguyên, được phân tách bằng dấu phẩy và được đặt trong một cặp dấu ngoặc vuông. Để sử dụng phương thức này, cần một mảng truyền vào phương thức.

```
1  int[] numbers;
2  numbers = new int[3];
3  numbers[0] = 42;
4  numbers[1] = 17;
5  numbers[2] = 256;
6  printValuesInList(numbers);
```

Kết quả được tạo ra bởi câu lệnh cuối cùng sẽ là

```
[42,17,256]
```

Tham số dòng lệnh

Tham số dòng lệnh (Command-line Arguments) là các tham số được truyền vào khi gọi chương trình, các tham số này sẽ được truyền cho phương thức main(). Phương thức main() của chương trình có tham số kiểu String[]. Khi phương thức main() được gọi, một số mảng chuỗi phải được truyền cho main() dưới dạng giá trị của tham số. Khi sử dụng giao diện dòng lệnh, người dùng gõ lệnh để yêu cầu hệ thống thực thi một chương trình. Người dùng có thể thêm đầu vào trong lệnh này, ngoài tên của chương trình. Đầu vào này trở thành đối số dòng lệnh. Hệ thống nhận các đối số dòng lệnh, đặt chúng vào một mảng chuỗi và chuyển mảng đó đến phương thức main().

Ví dụ, nếu tên của chương trình là myProg, thì người dùng có thể gõ để khởi động chương trình không có tham số:

```
java myProg
```

Nếu người dùng muốn truyền tham số vào có thể gõ:

```
java myProg one two three
```

Trong trường hợp trên các đối số dòng lệnh là các chuỗi "one", "two" và "three". Hệ thống đặt các chuỗi này vào một mảng String[] và chuyển mảng đó làm tham số cho chương trình main().

Ví dụ, đây là một chương trình in ra bất kỳ tham số dòng lệnh nào do người dùng nhập:

```
1  public class CLDemo {
2     public static void main(String[] args) {
3        System.out.println("Bạn đã nhập " + args.length + " tham số dòng lệnh");
4        if (args.length > 0) {
5           System.out.println("Đó là :");
6           int i;
7           for ( i = 0; i < args.length; i++ )
8              System.out.println("   " + args[i]);
```

```
 9         }
10     }
11 }
```

Trong thực tế, các tham số dòng lệnh thường được sử dụng để chuyển tên của tệp vào một chương trình. Ngày nay, hầu hết các chương trình được chạy trong **môi trường đồ họa (GUI)** nên các đối số dòng lệnh không còn quan trọng như trước đây. Nhưng ít nhất chúng cung cấp một ví dụ về cách các tham số mảng có thể được sử dụng.

CHỒNG PHƯƠNG THỨC

Chồng phương thức là một tính năng cho phép định nghĩa nhiều phương thức cùng tên nhưng khác nhau về các tham số đầu vào. Ví dụ:

```
1 println(int)                  println(double)
2 println(char)                 println(boolean)
3 println()
```

Để gọi một phương thức một cách hợp lệ, cần biết nó có bao nhiêu tham số hình thức và kiểu của từng tham số. Thông tin này được gọi là **chữ ký của phương thức**. Chữ ký của phương thức doTask, được sử dụng như một ví dụ ở trên, có thể được biểu diễn dưới dạng: doTask(int, double, boolean). Lưu ý rằng chữ ký không bao gồm tên của các tham số; trong thực tế, nếu bạn chỉ muốn sử dụng phương thức, bạn thậm chí không cần biết tên tham số hình thức là gì, vì vậy tên tham số không phải là một phần của giao diện. Java cho phép các phương thức khác nhau trong cùng một lớp có cùng tên, miễn là **chữ ký của chúng khác nhau.** Ví dụ đối tượng System.out bao gồm nhiều phương thức khác nhau có tên là println. Tất cả các phương thức này đều có các chữ ký khác nhau, chẳng hạn println(int) hay println(char). Chính vì vậy lời gọi System.out.println(17) sẽ gọi phương thức có println(int), trong khi lời gọi System.out.println('A') sẽ gọi phương thức println(char). Tất nhiên tất cả các phương thức khác nhau này đều có tác dụng tương tự nhau.

Tuy nhiên, sẽ không hợp lệ khi có hai phương thức trong cùng một lớp giống nhau tham số hình thức nhưng có kiểu trả về khác nhau. Ví dụ, sẽ là một lỗi cú pháp cho một lớp chứa hai phương thức được định nghĩa:

```
1 int    getln() { ... }
2 double getln() { ... }
```

Đây là lý do tại sao các phương thức trong đối tượng Scan lại được đặt tên khác nhau là nextInt(), nextDouble()...

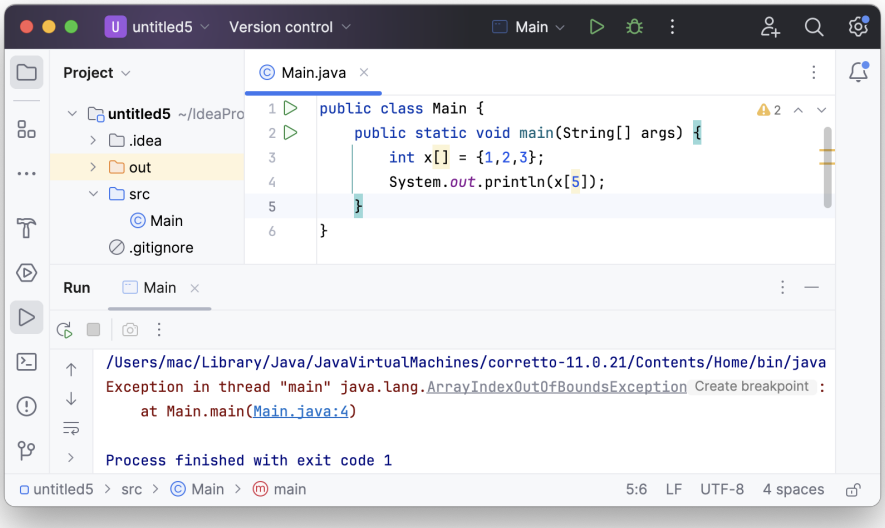
NGOẠI LỆ

Từ "ngoại lệ" (exception) được sử dụng thay cho từ "lỗi" (error). Nó bao gồm bất kỳ trường hợp nào phát sinh khi **chương trình được thực thi (runtime)**. Một ngoại lệ có thể là một lỗi hoặc nó có thể chỉ là một trường hợp đặc biệt. Đôi khi, ngoại lệ có thể không được coi là lỗi, mà chỉ là một cách khác để tổ chức một chương trình.

Nếu một ngoại lệ xảy ra trong quá trình thực thi một chương trình, có thể nói rằng ngoại lệ đó được "ném" ra. Khi điều này xảy ra, chương trình cần xử lý ngoại lệ này nếu không sẽ khiến chương bị treo hoặc bị kết thúc ngay lập tức. Xét ví dụ đoạn mã sau:

```
static void Main(string[] args){
    int[] mynumbers = new int[] {1,2,3};
    int i = mynumbers[9];     // chỗ này phát sinh lỗi
}
```

Đoạn mã trên khi chạy sẽ phát sinh vấn đề bởi vì, mynumbers là mảng có 3 phần tử, nhưng nếu truy cập truy cập đến phần tử thứ 10 bằng mynumber[9] thì sẽ phát sinh một ngoại lệ. Một ngoại lệ sẽ xuất hiện ra màn hình và chương trình sẽ bị kết thúc ngay lập tức.



Hình 4-2 Ngoại lệ phát sinh được hiển thị trên cửa sổ Terminal

Các ngoại lệ trong Java được biểu diễn dưới dạng các đối tượng kiểu Exception. Ví dụ trong trường hợp trên, ngoại lệ là một đối tượng có kiểu là **IndexOutOfRangeException**. Các ngoại lệ này để là lớp con của **Exception** (kế thừa từ Exception). Các lớp con khác nhau đại diện cho các loại ngoại lệ khác nhau. Có rất nhiều loại ngoại lệ. Ví dụ một ngoại lệ **NumberFormatException** có thể xảy ra khi cố gắng chuyển đổi một chuỗi thành một số. Ví dụ:

```
public static void main(String[] args) {
    int a = Integer.parseInt("123.33");
}
```

Các chuyển đổi như vậy được thực hiện bởi các phương thức như **Integer.parseInt** hay **Double.parseDouble**. Hãy xem xét lời gọi phương thức Integer.parseInt(str) trong đó str là một biến kiểu String. Nếu giá trị của str là chuỗi "42", thì lệnh gọi phương thức sẽ chuyển đổi chính xác chuỗi thành int 42. Tuy nhiên, nếu giá trị của str là "forty two", thì lệnh gọi phương thức sẽ không thành công vì "forty two" không phải là một biểu diễn chuỗi hợp lệ của một giá trị int. Trong trường hợp này, một ngoại lệ của kiểu **NumberFormatException** được ném ra. Nếu không có gì được thực hiện để xử lý ngoại lệ, chương trình sẽ bị đóng ngay lập tức.

Xử lý ngoại lệ

Khi một ngoại lệ xảy ra, chúng tôi nói rằng ngoại lệ được "ném". Ví dụ, chúng ta nói rằng Integer.parseInt(str) ném một ngoại lệ kiểu **NumberFormatException** khi giá trị của str là không hợp lệ.

Khi một ngoại lệ được ném ra, ta có thể "bắt" ngoại lệ và ngăn nó làm hỏng chương trình. Điều này được thực hiện với câu lệnh **try**..**catch**. Ở dạng đơn giản, cú pháp cho câu lệnh **try**..**catch** có thể là:

```
try {
   //khối lệnh-1
}
catch ( exception-class-name  variable-name ) {
   //khối lệnh-2
}
```

Trong đó **exception-class-name** là tên lớp ngoại lệ, nó có thể là NumberFormatException, hay IllegalArgumentException hoặc một số lớp ngoại lệ khác. Khi máy tính thực hiện câu lệnh try..catch này, nó thực thi **khối-lệnh-1** trong phần try. Nếu không có ngoại lệ nào xảy ra trong quá trình thực thi các **khối-lệnh-1**, thì máy tính chỉ cần bỏ qua **khối-lệnh-2** và tiếp tục với phần còn lại của chương trình. Tuy nhiên, nếu một ngoại lệ có kiểu **exception-class-name** xảy ra trong quá trình thực thi các **khối-lệnh-1**, máy tính sẽ ngay lập tức nhảy từ điểm mà ngoại lệ xảy ra đến phần bắt và thực hiện **khối-lệnh-2**, và bỏ qua bất kỳ câu lệnh nào còn lại trong **khối-lệnh-1**.

Trong quá trình thực thi **khối-lệnh-2**, có thể sử dụng **variable-name** để đại diện cho đối tượng ngoại lệ. Biến này sẽ chứa thông tin về loại và nguyên nhân của ngoại lệ và có thể được in ra màn hình nếu cần. Sau khi kết thúc phần **catch**, máy tính tiếp tục với phần còn lại của chương trình; ngoại lệ đã được bắt và xử lý và không làm hỏng chương trình.

Hãy lưu ý rằng dấu ngoặc nhọn { và } là một phần của cú pháp của câu lệnh try..catch. Chúng được yêu cầu ngay cả khi chỉ có **một câu lệnh** giữa các dấu ngoặc nhọn. Điều này khác với những câu lệnh khác trước đó, khi dấu ngoặc nhọn xung quanh một câu lệnh đơn là tùy chọn.

Xét một ví dụ khác, giả sử có một chương trình tìm giá trị trung bình của một dãy số thực do người dùng nhập vào và yêu cầu người dùng báo hiệu kết thúc dãy bằng cách nhập vào một dòng trống. Người dùng có thể nhập một giá trị không đúng yêu cầu khiến một ngoại lệ xuất hiện, vì vậy cần sử dụng try..catch để tránh sự cố chương trình:

```
public static void main(String[] args) {
   String str;     // Đầu vào của người dùng.
   double number;  // Đầu vào được chuyển đổi thành số.
   double total;   // Tổng tất cả các số đã nhập.
   double avg;     // Giá trị trung bình của các số.
   int count;      // Số lượng các số đã nhập.
   total = 0;
   count = 0;
   System.out.println("Nhập số của bạn, nhấn quay lại để kết thúc.");
   while (true) {
       str = new Scanner(System.in).next();
       if (str.equals("")) {
           break; // Thoát vòng lặp, vì dòng đầu vào trống.
       }
       try {
           number = Double.parseDouble(str); //Nếu xảy ra ngoại lệ tại đây, hai dòng
tiếp theo sẽ bị bỏ qua!
           total = total + number;
           count = count + 1;
```

```
        } catch (NumberFormatException e) {
            System.out.println("Số không hợp lệ! Hãy thử lại.");
        }
    }
    avg = total / count;
    System.out.printf("Giá trị trung bình của %d số là %1.6g%n", count, avg);
}
```

Dây chuyền xử lý ngoại lệ

Một chương trình sẽ bao gồm việc tham gia của nhiều phương thức khác nhau, các phương thức này có thể gọi lẫn nhau, và ngoại lệ có thể xảy ra tại bất kỳ vị trí nào trong các phương thức này. Ví dụ:

Hình 4-3 Minh họa việc ném và bắt ngoại lệ

Ví dụ phương thức **main(),** gọi phương thức **writeToFile(),** phương thức **writeToFile()** lại gọi phương thức khởi tạo **FileWriter(),** phương thức **write()** để đọc file. Những phương thức này thực hiện việc mở và ghi file có thể xuất hiện một ngoại lệ vì vậy **writeToFile()** bắt buộc phải có khối try...catch như sau:

```
public static void writeToFile() {
    try {
        BufferedWriter fw = new FileWriter("myFile.txt");
        fw.write("Test");
        fw.close();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
public static void main(String[] args) {
    writeToFile();
}
```

Tuy nhiên nếu phương thức **writeToFile()** không muốn xử lý ngoại lệ đó, nó có thể khai báo ném ngoại lệ đó để phương thức gọi nó xử lý bằng cách sử dụng từ khóa **throws** như sau:

```
public static void writeToFile() throws IOException {
    BufferedWriter bw = new BufferedWriter(new FileWriter("myFile.txt"));
    bw.write("Test");
    bw.close();
}
public static void main(String[] args) {
    try {
        writeToFile();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

Trong câu lệnh trên khi ngoại lệ xảy ra ta, trong khối catch gọi phương thức **printStackTrace()** của đối tượng **ex.** Phương thức này sẽ in ra **"Stack Trace"** hay còn được dịch là "Ngăn xếp lệnh", đây là một danh sách thể hiện thứ tự chi tiết các phương thức được gọi khiến ngoại lệ xảy ra. Ví dụ "Stack Trace" xuất hiện khi không thể mở được file để ghi:

```
java.io.FileNotFoundException: C:/myFile.txt (No such file or directory)
    at java.base/java.io.FileOutputStream.open0(Native Method)
    at java.base/java.io.FileOutputStream.open(FileOutputStream.java:291)
    at java.base/java.io.FileOutputStream.<init>(FileOutputStream.java:234)
    at java.base/java.io.FileOutputStream.<init>(FileOutputStream.java:123)
    at java.base/java.io.FileWriter.<init>(FileWriter.java:66)
    at Program.writeToFile(Program.java:8)
    at Program.main(Program.java:15)
```

Đọc và hiểu được "Stack Trace" là một trong những kỹ năng lập trình vô cùng quan trọng, ví dụ như trong kết quả trên thì ngoại lệ xuất phát từ việc phương thức **main()** gọi phương thức **writeToFile()** (ở dòng 15 file **Program.java**), phương thức **writeToFile()** gọi phương thức khởi tạo FileWriter() (ở dòng 8 file **Program.java**), phương thức **FileWriter()** lại tiếp túc gọi các phương thức khác và ngoại lệ xảy ra khi phương thức **open0()** đã không thực hiện được việc mở file. Tất nhiên chỉ có file Program.java là file của lập trình viên tạo ra và lập trình, vì vậy chỉ cần quan tâm đến các dòng lệnh có xuất hiện ngoại lệ trong file này. "Stack Trace" thực sự là một công cụ hữu ích để nhà phát triển biết được nên sửa lỗi tại đâu.

Sử dụng finally

Bên cạnh việc sử dụng cú pháp **try ... catch**. Lập trình viên có thể sử dụng cú pháp **try ... catch ... finally**. Cú pháp như sau:

```
try {
}
catch (Exception e ) {
}
finally {
}
```

Mệnh đề **finally** khai báo khối các câu lệnh sẽ được đảm bảo sẽ được thực hiện như là bước cuối cùng trong quá trình thực thi các câu lệnh trong khối **try**, cho dù có hay không ngoại lệ xảy ra. Các câu lệnh trong khối **finally** xảy ra ngay cả khi trong khối **catch** xuất hiện một ngoại lệ mới. Khối lệnh này thường được sử dụng để thực hiện việc dọn dẹp cần thiết mà không được bỏ qua trong bất kỳ trường hợp nào ví dụ như đóng kết nối tới các tài nguyên đã truy cập. Ví dụ:

```
try {
   //Tạo một kết nối mạng, thực hiện vào ra
}
catch (IOException e ) {

}
finally {
   //Kiểm tra nếu kết nối vẫn còn mở thì thực hiện đóng lại
}
```

Trong ví dụ trên, **finally** đảm bảo rằng kết nối mạng chắc chắn sẽ được đóng lại, cho dù có xảy ra lỗi hay không.

Phân cấp ngoại lệ

Tất cả các đối tượng ngoại lệ phải thuộc về một lớp con của **java.lang.Throwable**. Các lớp con này được sắp xếp theo một hệ thống phân cấp khá phức tạp cho thấy mối quan hệ giữa các loại ngoại lệ khác nhau. **Throwable** có hai lớp con trực tiếp, Error và Exception. Hai lớp con này lần lượt có nhiều lớp con khác. Ngoài ra, một lập trình viên có thể tạo các lớp ngoại lệ mới để đại diện cho các loại ngoại lệ mới không có trong danh sách các ngoại lệ chuẩn.

Hình 4-4 Sơ đồ phân cấp lớp ngoại lệ

Hầu hết các lớp con của lớp **Error** đại diện cho các **lỗi nghiêm trọng** trong **máy ảo Java** (**JVM**), Ví dụ như các lỗi liên quan đến việc thư viện chung bị lỗi hay có vấn đề về bộ nhớ. Nói chung, không nên cố gắng bắt và xử lý những lỗi như vậy. Ví dụ **java.lang.OutOfMemoryError** là lỗi sẽ xuất hiện khi máy ảo Java không thể cấp phát bộ nhớ cho chương trình và bộ thu gom rác không thể dọn dẹp được bộ nhớ.

Mặt khác, các lớp con của lớp **Exception** đại diện cho các ngoại lệ được dùng để bắt. Đây là những "lỗi" không được xử lý trong chương trình mà lập trình viên đã “vô tình” hoặc “cố tình” bỏ qua. Lớp ngoại lệ có lớp là **RuntimeException**, đại diện cho các ngoại lệ xuất hiện khi chạy chương trình. Các ngoại lệ như **NullPointerException** hay **ArithmeticException** là các lớp con của lớp **RuntimeException** này.

Tất cả các ngoại lệ này đều kế thừa từ **Throwable**. Các lớp này đều có phương thức quan trọng hay được sử dụng như:

- **getMessage()** là một phương thức trả về một chuỗi mô tả ngoại lệ.
- **toString ()** trả về một chuỗi có chứa tên của lớp mà ngoại lệ xảy ra.
- **printStackTrace ()** trả về **“stack trace”** chứa danh sách thể hiện thứ tự chi tiết các phương thức được gọi khiến ngoại lệ xảy ra.

Xét một ví dụ khác:

```
1  double x = arr[1]/arr[2];
2  System.out.println("x = " + x);
```

Ở đây, máy tính sẽ cố gắng thực thi khối lệnh "**try**". Nếu không có ngoại lệ nào xảy ra trong quá trình thực thi khối này, thì phần "**catch**" của câu lệnh sẽ bị bỏ qua. Tuy nhiên, sẽ có một số tình huống xảy ra như sau:

Trường hợp 1, mảng arr chỉ có 2 phần tử như vậy việc truy cập vào arr[2] là một ngoại lệ, chương trình sẽ đưa ra một ngoại lệ là **ArrayIndexOutOfBoundsException,** để đảm bảo ngoại lệ không xảy ra có thể sử dụng khối **try ... catch** như sau:

```
1  try {
2      double x = arr[1]/arr[2];
3      student.score = x;
4  }
5  catch (ArrayIndexOutOfBoundsException ex) {
6      System.out.println("Phần tử mảng không tồn tại");
7  }
```

Tuy nhiên điều này chưa đảm bảo được chương trình hết ngoại lệ.

Trường hợp 2 khi phần tử arr[2] có tồn tại trong mảng nhưng có giá trị bằng 0. Trong khi arr[1] lại là một giá trị khác 0. Khi đó phép toán arr[1]/arr[2] đã thực hiện một pháp chia cho 0. Điều này khiến xuất hiện một ngoại lệ có kiểu là **ArithmeticException** xuất hiện. Chương trình bị dừng lại vì ngoại lệ này chưa được xử lý. Để xử lý cả hai loại ngoại lệ, chương trình cần được sửa lại như sau:

```
try {
    double x = arr[1]/arr[2];
    student.score = x;
} catch (ArithmeticException ex) {
    System.out.println("Phép toán không thực hiện được ");
} catch (ArrayIndexOutOfBoundsException ex) {
    System.out.println("Phần tử mảng không tồn tại");
}
```

Tuy nhiên, điều này vẫn chưa đảm bảo rằng chương trình hết ngoại lệ. Ví dụ đối tượng student có thể chưa được khởi tạo, vì vậy khiến câu lệnh student.score = x; không thể thực hiện được, vì vậy một ngoại lệ **NullPointerException**, có thể xảy ra. Như vậy chương trình cần bắt thêm ngoại lệ **NullPointerException.** Đến bây giờ thì có thể **"khá chắc chắn"** các ngoại lệ đều được xử lý tuy nhiên cũng có thể bỏ sót một số ngoại lệ chưa tính đến. Để đảm bảo ngoại lệ đều được bắt, có thể catch loại ngoại lệ có kiểu là **RuntimeException**, **Exception** và thậm chí là **Throwable**. Ví dụ:

```
try {
    double x = arr[1] / arr[2];
    student.score = x;
} catch (ArithmeticException ex) {
    System.out.println("Phép toán không thực hiện được ");
} catch (ArrayIndexOutOfBoundsException ex) {
    System.out.println("Phần tử mảng không tồn tại");
} catch (NullPointerException ex) {
    System.out.println("Đối tượng rỗng");
} catch (RuntimeException ex) {
    System.out.println("Có lỗi xảy ra");
}
```

Với cách viết mã như trên, hệ thống sẽ bắt được hết tất cả các lỗi RuntimeException. Nếu cần thiết có thể bắt tiếp ngoại lệ có kiểu tổng quát hơn như là Exception. Tuy nhiên điều cần lưu ý ở đây là các ngoại lệ chung cần phải để xuống dưới, các ngoại lệ kế thừa từ ngoại lệ chung để lên phía trên. Điều đấy có nghĩa là việc xử lý như sau là sai cú pháp:

```
try {
    double x = arr[1] / arr[2];
    student.score = x;
} catch (Exception ex) {
    System.out.println("Đối tượng rỗng");
} catch (ArithmeticException ex) {
    System.out.println("Phép toán không thực hiện được ");
} catch (ArrayIndexOutOfBoundsException ex) {
    System.out.println("Phần tử mảng không tồn tại");
} catch (NullPointerException ex) {
```

```
11      System.out.println("Đối tượng rỗng");
12  }
```

Rõ ràng, các loại ngoại lệ như **ArithmeticException**, **ArrayIndexOutOfBoundsException**, hay **NullPointerException** đều là lớp con (cháu) của lớp Exception, vì vậy việc bắt ngoại lệ có kiểu Exception phải viết xuống bên dưới như sau:

```
1  try {
2      double x = arr[1] / arr[2];
3      student.score = x;
4  } catch (Exception ex) {
5      System.out.println("Đối tượng rỗng");
6  } catch (ArithmeticException ex) {
7      System.out.println("Phép toán không thực hiện được ");
8  } catch (ArrayIndexOutOfBoundsException ex) {
9      System.out.println("Phần tử mảng không tồn tại");
10 } catch (NullPointerException ex) {
11     System.out.println("Đối tượng rỗng");
12 }
```

Ném ngoại lệ và tạo lớp ngoại lệ

Từ khóa **throw** được sử dụng để ném một ngoại lệ từ một phương thức hoặc bất kỳ khối mã nào. Ví dụ:

```
throw new ArithmeticException("Ngoại lệ toán học");
```

Ở đây, một ngoại lệ kiểu "Toán học" được tạo ra và ném ra tại một vị trí bất kỳ trong chương trình. Các ngoại lệ được ném ra thường là những ngoại lệ do người dùng tự định nghĩa. Khi đó các ngoại lệ đó phải là các đối tượng được tạo ra bởi các lớp mà các lớp này kế thừa từ các lớp ngoại lệ đã có hoặc từ lớp Throwable. Ví dụ tạo một lớp ngoại lệ như sau:

```
1  public class MyException extends RuntimeException {
2      String message;
3      MyException(String message){
4          this.message = message;
5      }
6
7      @Override
8      public String getMessage() {
9          return message;
10     }
11 }
```

Để ném loại ngoại lệ trên, viết mã lệnh như sau:

```
throw new MyException("Ngoại lệ tự định nghĩa");
```

Việc tạo ra một lớp ngoại lệ, kế thừa và tạo đối tượng từ lớp tự định nghĩa sẽ tìm hiểu kỹ hơn trong chương lập trình hướng đối tượng.

Xử lý ngoại lệ bắt buộc

Trong ví dụ trước về đọc file:

```
public static void writeToFile() throws IOException {
   BufferedWriter bw = new BufferedWriter(new FileWriter("myFile.txt"));
   bw.write("Test");
   bw.close();
}
public static void main(String[] args) {
   try {
       writeToFile();
   } catch (IOException ex) {
       ex.printStackTrace();
   }
}
```

Có thể thấy rằng trong trường hợp này, việc xử lý ngoại lệ là bắt buộc. Nếu không làm như vậy sẽ có một lỗi cú pháp và sẽ được trình biên dịch thông báo. Các ngoại lệ bắt buộc thường là các ngoại lệ liên quan đến các hoạt động vào/ra. Chúng cũng được gọi là "**check exception**" một số các phương thức luôn khai báo chúng sẽ có thể "**throws**" ra những ngoại lệ này. Vì vậy bắt buộc những phương thức gọi phương thức này bắt buộc phải xử lý ngoại lệ. Tức là phương thức đó phải đặt câu lệnh trong khối lệnh **try** có mệnh đề bắt xử lý ngoại lệ; trong trường hợp này, ngoại lệ được xử lý trong chương trình con, do đó không người gọi chương trình con nào có thể nhìn thấy ngoại lệ. Cách khác là khai báo rằng phương thức con có thể ném ngoại lệ. Điều này được thực hiện bằng cách thêm mệnh đề "**throws**" vào tiêu đề chương trình con, điều này cảnh báo bất kỳ người gọi nào về khả năng ngoại lệ có thể được tạo ra khi chương trình con được thực thi. Như vậy người gọi sẽ bị buộc phải xử lý ngoại lệ trong câu lệnh try hoặc khai báo ngoại lệ trong mệnh đề ném trong tiêu đề của chính nó.

JAVA DOCS

Khi viết các phương thức, phần mô tả của phương thức nên được đưa vào khối chú thích

```
/**
*/
```

Ví dụ:

```
/**
* phương thức tính diện tích hình chữ nhật
* @param width Chiều rộng
* @param height Chiều cao
* @return Diện tích
* @exception IllegalArgumentException xảy ra khi kích thước vào âm
*/
public static double areaOfRectangle(double height, double width) {
   if (width < 0 || height < 0)
       throw new IllegalArgumentException("Sides must have positive length.");
   double area;
   area = width * height;
   return area;
}
```

Để sinh ra tài liệu mô tả về phương thức có thể dùng lệnh:

```
javadoc Main.java
```

Chú ý trong intelliJ IDEA hỗ trợ add JavaDoc, và Gen JavaDoc rất tiện.

THỰC HÀNH

Tìm ước của một số

Hãy viết một phương thức để tính toán và in ra tất cả các ước của một số nguyên dương cho trước. Số nguyên sẽ là một tham số của phương thức. Trường hợp này có thể khai báo phương thức là:

```
1  static void printDivisors( int N ) {
2      int D;
3      System.out.println("Ước của " + N + " là các số:");
4      for ( D = 1; D <= N; D++ ) {
5         if ( N % D == 0 )
6            System.out.println(D);
7      }
8  }
```

Trò chơi đoán số

Viết một chương trình chơi trò chơi đoán với người dùng. Máy tính sẽ chọn một số ngẫu nhiên từ 1 đến 100 và người dùng sẽ cố gắng đoán nó. Máy tính cho người dùng biết dự đoán cao hay thấp hoặc chính xác. Nếu người dùng đoán đúng sau sáu lần đoán trở xuống, người dùng sẽ thắng trò chơi. Sau mỗi trò chơi, người dùng có tùy chọn tiếp tục với trò chơi khác.

```
1  public class Main {
2     public static void main(String[] args) {
3         System.out.println("Chọn 1 số từ 1 đến 100");
4         boolean playAgain;
5         do {
6             playGame();
7             System.out.print("Bạn có muốn chơi lại? ");
8             playAgain = new Scanner(System.in).nextBoolean();
9         } while (playAgain);
10        System.out.println("Tạm biệt");
11    }
12    static void playGame() {
13        int computersNumber;
14        int usersGuess;
15        int guessCount;
16        computersNumber = (int) (100 * Math.random()) + 1;
17        guessCount = 0;
18        System.out.println();
19        System.out.print("Bạn đoán số nào? ");
20        while (true) {
21            usersGuess = new Scanner(System.in).nextInt();
22            guessCount++;
```

```
            if (usersGuess == computersNumber) {
                System.out.println("Bạn đã đoán đúng, số đó là: " + computersNumber);
                break;
            }
            if (guessCount == 6) {
                System.out.println("Bạn đã thua, số đó là " + computersNumber);
                break;
            }
            if (usersGuess < computersNumber)
                System.out.print("Số quá thấp, hãy đoán lại: ");
            else if (usersGuess > computersNumber)
                System.out.print("Số quá cao, hãy đoán lại: ");
        }
    }
}
```

Chồng phương thức

Viết một phương thức có tên printChar() in ra một ký tự.

```
private static void printChar(char ch) {
    System.out.print( ch );
}
```

Hãy viết một phương thức trùng tên với phương thức trên và in ra in ra một dòng văn bản chứa N bản sao của ký tự ch. Phương thức phải có một tham số ch kiểu char và một tham số N kiểu int.

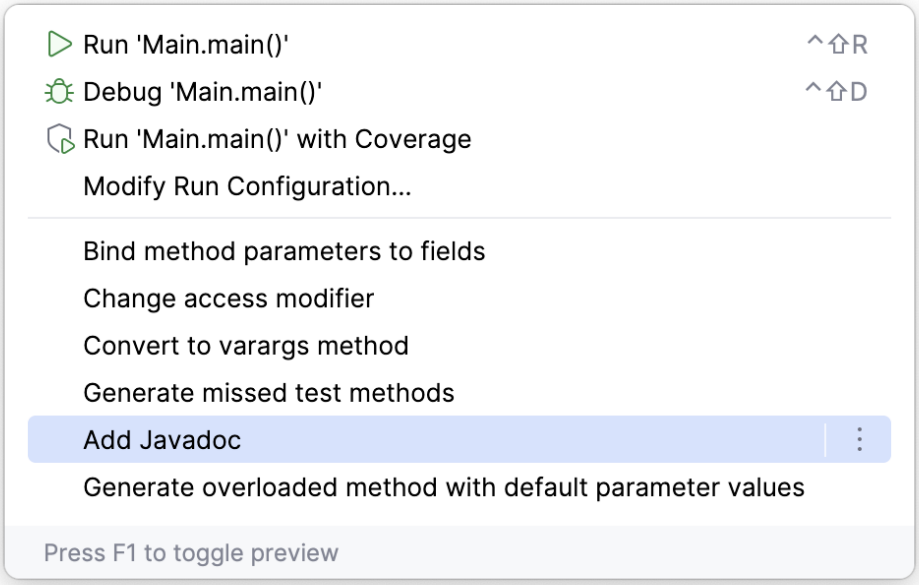
Hãy viết một phương thức nhận một chuỗi làm tham số. Đối với mỗi ký tự trong chuỗi, nó sẽ in ra một dòng hiển thị 25 bản sao của ký tự đó.

```
private static void printRowsFromString(String str) {
    for (int i = 0; i < str.length(); i++ ) {
        printChar(str.charAt(i), 25);
    }
}
```

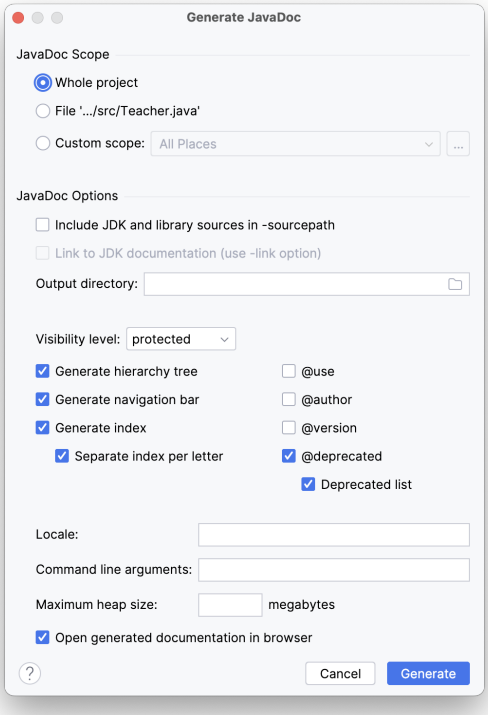
Hãy viết phương thức main() để gọi và thử nghiệm phương thức trên.

Tạo JavaDoc

Sử dụng công cụ InteliJ IDEA. tạo một Project, khai báo một số phương thức sau đó viết JavaDoc cho dự án bằng cách chọn phương thức đó sau đó chọn tùy chọn Add JavaDoc:



Để tạo JavaDoc cho một Project chọn Tool – JavaDoc sau đó khai báo thông tin để sinh ra thư mục JavaDocs.



Sau khi bấm Generate, hãy kiểm trả kết quả được tạo ra.

CÂU HỎI ÔN TẬP LÝ THUYẾT

1. Đâu không phải là quy tắc của một "hộp đen":

- A. Giao diện của hộp đen nên đơn giản, rõ ràng và dễ hiểu
- B. Hộp đen cần phải che dấu toàn bộ thông tin và dữ liệu bên trong để đảm bảo sự an toàn của chương trình

C. Để sử dụng hộp đen, không cần biết bất cứ điều gì về cách thực hiện của nó; tất cả những gì cần biết là giao diện của nó.

D. Người triển khai hộp đen không cần biết quá nhiều về các hệ thống lớn hơn mà hộp sẽ được sử dụng.

2. Từ khóa static dùng để:

A. Tạo một khối lệnh tĩnh

B. Tạo một lớp tĩnh

C. Tạo một đối tượng tĩnh

D. Tạo một phương thức tĩnh

3. Đâu là khẳng định chính xác về biến tĩnh:

A. Biến tĩnh phải được gọi thông qua đối tượng

B. Biến tĩnh có thể được truy cập bởi phương thức tĩnh

C. Biến tĩnh được tạo ra mỗi khi đối tượng ở trạng thái tĩnh

D. Biến tĩnh không thể thay đổi giá trị và luôn được khai báo kèm từ khóa final

4. Từ khóa catch được sử dụng để:

A. Chụp một ngoại lệ

B. Điều hướng khối lệnh

C. Bắn ngoại lệ

D. Bắt một địa chỉ của lỗi

5. Từ khóa finally được sử dụng để:

A. Kết thúc hàm và trả về giá trị

- B. Kết thúc một vòng lặp
- C. Tạo một khối lệnh luôn được thực hiện khi có ngoại lệ
- D. Kết thúc một chuỗi xử lý

6. Để ném ra một ngoại lệ dùng từ khóa:

- A. throws
- B. catch
- C. try
- D. throw

7. Chồng phương thức là:

- A. Việc gọi liên tiếp các phương thức nhằm thực hiện một công việc cụ thể
- B. Việc để các phương thức chồng lên nhau
- C. Việc khai báo lại một phương thức đã có ở lớp cha
- D. Việc khai báo các phương thức trùng tên nhưng khác nhau tham số đầu vào

8. Tham số linh động là tham số:

- A. Có thể khai báo nhiều trong một phương thức
- B. Linh động về kích thước
- C. Có thể linh động về kiểu
- D. Luôn được khai báo ở cuối cùng trong danh sách tham số truyền vào

9. Tham số dòng lệnh là tham số:

- A. Tham số của hàm main

- B. Tham số truyền vào kết thúc chương trình
- C. Có dạng mảng số nguyên
- D. Tất cả các phương án trên đều sai

10. Java docs được tạo ra bởi lệnh:

- A. javadoc
- B. javadocument
- C. javahelp
- D. javadocs

11. Hãy cho biết kết quả thực hiện đoạn mã sau:

```
1  public class Program {
2      public static void main(String[] args) {
3          System.out.println(stars(10));
4      }
5
6      public String stars(int count) {
7          String res = "";
8          for (int i = 0; i < count; i++) {
9              res = res + "*";
10         }
11         return res;
12     }
13 }
```

- A. Hiển thị 9 dấu *
- B. Hiển thị 10 dấu *
- C. Lỗi biên dịch
- D. Lỗi khi chạy

BÀI TẬP TỰ THỰC HÀNH

Chồng phương thức tĩnh

Viết một lớp MathUtils với các phương thức tính toán như sau:

- max: nhận vào 2 số nguyên và trả về số lớn hơn trong 2 số đó
- max: nhận vào 2 số thực và trả về số lớn hơn trong 2 số đó
- max: nhận vào một mảng số nguyên và trả về số lớn nhất trong mảng đó
- max: nhận vào một mảng số thực và trả về số lớn nhất trong mảng đó

Lớp MathUtils có chứa phương thức tĩnh main, trong phương thức này:

- Tạo một đối tượng MathUtils
- Gọi phương thức max với các tham số là 2 số nguyên
- Gọi phương thức max với các tham số là 2 số thực
- Tạo một mảng số nguyên và gọi phương thức max với mảng này
- Tạo một mảng số thực và gọi phương thức max với mảng này

Tham số linh động

Viết một phương thức sử dụng tham số linh động để có thể tính tổng của một danh sách số bất kỳ được truyền vào.

Xử lý ngoại lệ

Viết một phương thức tính toán phép chia hai số nguyên. Nếu số thứ hai bằng 0, phương thức sẽ ném ra một ngoại lệ với thông báo "Chia cho không".

Thực hành về JavaDocs

Tạo một lớp trong đó có các phương thức:

- Tính diện tích hình chữ nhật
- Tính diện tích hình tròn
- Tính diện tích hình tam giác
- Tính diện tích hình thang

Yêu cầu:

- Với mỗi phương thức viết đầy đủ các hướng dẫn
- Viết hướng dẫn chung cho lớp
- Sinh ra Javadoc cho Project.

TÀI LIỆU THAM KHẢO

[1] Core Java: Fundamentals (2021) , Cay Horstmann (Oracle Press Java)

[2] Head First Java: A Brain-Friendly Guide (2022), Kathy Sierra, O'Reilly Media

[3] Java OOP Done Right: Create object oriented code you can be proud of with modern Java Paperback (2019), Mr Alan Mellor, Mellor Books

[4] Murach's Java Programming (5th Edition) (2017), Joe Murach, Mike Murach & Associates

[5]. Java for Absolute Beginners Learn to Program the Fundamentals the Java 9+ Way

[6]. Modern Java Recipes: Simple Solutions to Difficult Problems in Java 8 and 9 (2017), by Ken Kousen, O'Reilly Media

[7] Effective Java (2018), Joshua Bloch, Addison-Wesley Professional