


IS216 - LẬP TRÌNH JAVA

Chương 3 NGOẠI LỆ, GENERICS VÀ COLLECTIONS

Giảng viên: Tạ Việt Phương
E-mail: phuongtv@uit.edu.vn



1

Nội dung

1. Giới thiệu ngoại lệ
2. Các cách thức xử lý ngoại lệ
3. Generics
4. Collections

2

1. GIỚI THIỆU NGOẠI LỆ



3

Ngoại lệ

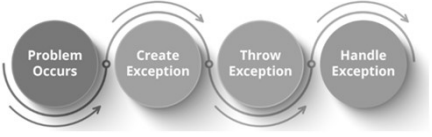
LÀ GÌ ?

4

Ngoại lệ

➤ **Định nghĩa:**

➤ Exception (ngoại lệ): là một sự kiện bất thường xảy ra trong tiến trình thực thi của một chương trình, làm gián đoạn tiến trình bình thường của chương trình.



5

Ngoại lệ

- Khi xảy ngoại lệ, nếu không xử lý chương trình sẽ kết thúc ngay.
- **Ví dụ:** Lỗi chia cho 0, vượt kích thước của mảng, lỗi mở file, Kết nối mạng bị ngắt trong quá trình thực hiện giao tác, JVM hết bộ nhớ, Truy cập vượt ngoài chỉ số của mảng ...

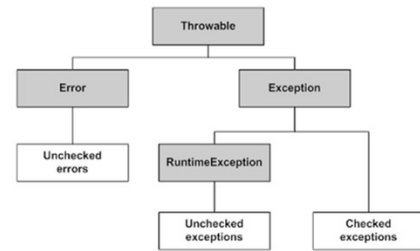
6

Ngoại lệ

- Dựa vào tính chất các vấn đề, người ta chia ngoại lệ thành hai loại:
 - Ngoại lệ được kiểm tra (Checked Exceptions).
 - Ngoại lệ không được kiểm tra (Unchecked Exceptions).
- Lỗi (Error): Error không phải là một ngoại lệ thông thường mà là lỗi nghiêm trọng của JVM

7

Ngoại lệ



8

2. CÁC CÁCH THỨC XỬ LÝ NGOẠI LỆ



9

Xử lý ngoại lệ

- Exception Handling
 - Mục đích: làm cho chương trình không bị ngắt đột ngột.
 - Có 2 cách để xử lý ngoại lệ:
 - Sử dụng các mệnh đề điều kiện kết hợp với các giá trị cờ.
 - Sử dụng cơ chế bắt và xử lý ngoại lệ.

10

Xử lý ngoại lệ

- Sử dụng các mệnh đề điều kiện kết hợp với các giá trị cờ.
 - **Mục đích:** thông qua tham số, giá trị trả lại hoặc giá trị cờ để viết mã xử lý tại nơi phát sinh lỗi.
 - **Hạn chế:**
 - Làm chương trình thêm rối, gây khó hiểu.
 - Dễ nhầm lẫn

11

Xử lý ngoại lệ

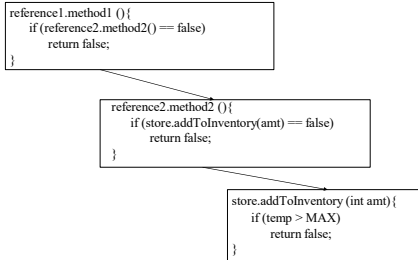
• Lớp Inventory

```
public class Inventory
{
    public final int MIN = 0;
    public final int MAX = 100;
    public final int CRITICAL = 10;
    public boolean addToInventory (int amount)
    {
        int temp;
        temp = stockLevel + amount;
        if (temp > MAX)
        {
            System.out.println("Adding " + amount + " item will cause stock ");
            System.out.println("to become greater than " + MAX + " units (overstock)");
            return false;
        }
        else
        {
            stockLevel = stockLevel + amount;
            return true;
        }
    } // End of method addToInventory
}
```

12

Xử lý ngoại lệ

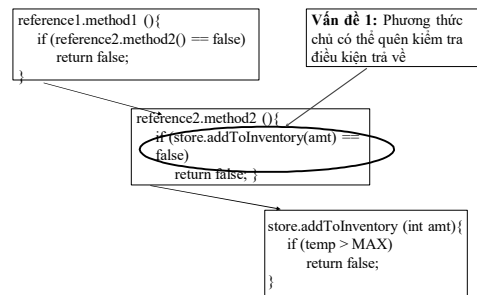
➤ Các vấn đề đối với cách tiếp cận điều kiện/cờ



13

13

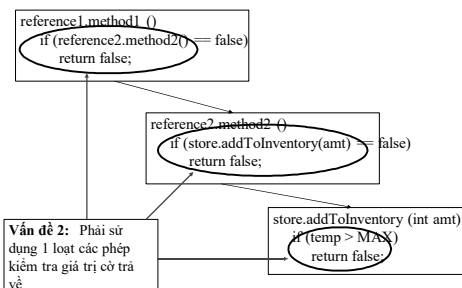
Xử lý ngoại lệ



14

14

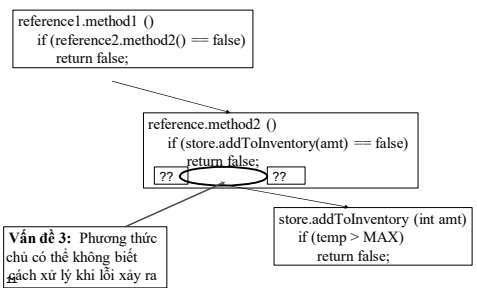
Xử lý ngoại lệ



15

15

Xử lý ngoại lệ



16

16

Xử lý ngoại lệ

- Nhược điểm của xử lý lỗi bằng giá trị cờ (boolean):
 - Khó kiểm soát được hết các trường hợp
 - Lỗi số học, lỗi bộ nhớ...
 - Lập trình viên thường quên xử lý lỗi

17

17

Xử lý ngoại lệ

➤ Sử dụng cơ chế bắt và xử lý ngoại lệ.

➤ Mục đích:

- Giúp chương trình đáng tin cậy hơn, tránh kết thúc bất thường
- Tách biệt khối lệnh có thể gây ngoại lệ và khối lệnh xử lý ngoại lệ.

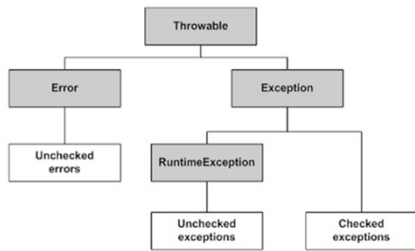
➤ Phân loại ngoại lệ:

- Ngoại lệ không cần kiểm tra (unchecked)
- Ngoại lệ phải kiểm tra (checked)

18

18

Checked vs Unchecked



19

19

Checked vs Unchecked

- Ngoại lệ unchecked
 - Là các ngoại lệ không bắt buộc phải được kiểm tra.
 - Gồm RuntimeException và các lớp con của chúng.
- Ngoại lệ checked
 - Là các ngoại lệ bắt buộc phải được kiểm tra.
 - Gồm các ngoại lệ còn lại.
- Error: Lỗi nghiêm trọng liên quan đến JVM, phần lớn Error không thể khắc phục bằng mã nguồn.

20

20

Error

➤ Lớp Error:

- Chỉ những lỗi nghiêm trọng và không dự đoán trước được:
 - OutOfMemoryError: Khi tạo quá nhiều object mà không có GC
 - StackOverflowError: Gọi đệ quy vô hạn
 - VirtualMachineError: JVM bị lỗi nội bộ
 - OutOfMemoryException: hết bộ nhớ
 - ThreadDead, LinkageError ...
- Kiểu Error ít được xử lý.

21

21

Unchecked

➤ Ngoại lệ không cần kiểm tra

- Trình biên dịch không yêu cầu phải bắt các ngoại lệ khi nó xảy ra.
 - Không cần khối try-catch
 - Các ngoại lệ này có thể xảy ra bất cứ thời điểm nào khi thi hành chương trình.
 - Thông thường là những lỗi nghiêm trọng mà chương trình không thể kiểm soát
 - Sử dụng các mệnh đề điều kiện để xử lý sẽ tốt hơn.
- Gồm các lớp **RuntimeException** và các **lớp con** của chúng

22

22

Unchecked

➤ RuntimeException: các ngoại lệ xảy ra trong quá trình thực thi chương trình do lỗi logic của lập trình viên, không bắt buộc phải xử lý bằng try-catch

- NullPointerException: con trỏ null
- ArrayIndexOutOfBoundsException: vượt quá chỉ số mảng
- ArithmeticException: lỗi toán học
- ClassCastException: lỗi ép kiểu
- NumberFormatException: lỗi khi chuyển đổi chuỗi thành số nhưng không hợp lệ. Ví dụ: Integer.parseInt("abc").
- IndexOutOfBoundsException: Lỗi truy cập danh sách ngoài phạm vi

23

23

Unchecked

➤ Ngoại lệ không cần kiểm tra: NullPointerException

```
int [] arr = null;
arr[0] = 1;
```

NullPointerException

24

24

Unchecked

- Ngoại lệ không cần kiểm tra: `ArrayIndexOutOfBoundsException`

```
arr = new int [4]; int
i;
for (i = 0; i <= 4; i++)
    arr[i] = i;

arr[i-1] = arr[i-1] / 0;
```

ArrayIndexOutOfBoundsException
(khi i = 4)

25

25

Unchecked

- Ngoại lệ không cần kiểm tra: `ArithmeticExceptions`

```
arr[i-1] = arr[i-1] / 0;
```

ArithmeticException
(Division by zero)

26

26

Checked

- Là ngoại lệ bắt buộc kiểm tra.
- Phải xử lý khi ngoại lệ có khả năng xảy ra:
 - Sử dụng khối try-catch
 - Sử dụng throw, throws
- Ví dụ:
 - `IOException`, `SQLException`

27

27

Checked

- Khối try...catch:

- try{...}: khối lệnh có khả năng gây ra ngoại lệ.
- catch{...}: nơi bắt và xử lý ngoại lệ.

- Cú pháp:

```
try
{
    // Code that may cause an error/exception to
    occur
}
catch (ExceptionType identifier)
{
    // Code to handle the exception
}
```

ExceptionType là một đối tượng của lớp **Throwable**

28

28

Checked

- Ngoại lệ cần phải kiểm tra: Đọc dữ liệu từ bàn phím

```
import java.io.*; class
Driver
{
    public static void main (String [] args)
    {
        BufferedReader stringInput; String
        s;
        int num;
        stringInput = new BufferedReader(new InputStreamReader(System.in));
```

29

29

Checked

- Ngoại lệ cần phải kiểm tra: Đọc dữ liệu từ bàn phím

```
try{
    System.out.print("Type an integer: ");
    s = stringInput.readLine(); System.out.println("You typed in..." + s);
    num = Integer.parseInt (s);
    System.out.println("Converted to an integer..." + num);
}
catch (IOException e){ System.out.println(e);
}
catch (NumberFormatException e{
    System.out.println(" Error format number: " + e.getMessage());
}
}
```

30

30

Checked

- Ngoại lệ cần phải kiểm tra: Ngoại lệ xảy ra khi nào

```
try
{
    System.out.print("Type an integer: ");
    s = stringInput.readLine();
    System.out.println("You typed in..." + s);
    num = Integer.parseInt(s);
    System.out.println("Converted to an integer..." + num);
}
```

31

31

Checked

- Kết quả của phương thức readLine()

```
try
{
    System.out.print("Type an integer: ");
    s = stringInput.readLine();
    System.out.println("You typed in..." + s);
    num = Integer.parseInt(s);
    System.out.println("Converted to an integer..." + num);
}
```

Ngoại lệ có thể xảy ra ở đây

32

32

Checked

- Lớp BufferedReader

<https://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html>

```
public class BufferedReader
{
    public BufferedReader (Reader in);
    public BufferedReader (Reader in, int sz);
    public String readLine () throws IOException;
    :
}
```

33

33

Checked

- Kết quả của phương thức parseInt ()

```
try
{
    System.out.print("Type an integer: ");
    s = stringInput.readLine();
    System.out.println("You typed in..." + s);
    num = Integer.parseInt(s);
    System.out.println("Converted to an integer..." + num);
}
```

Ngoại lệ có thể xảy ra ở đây

34

34

Checked

- Lớp Integer

<http://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html>

```
public class Integer
{
    public Integer (int value);
    public Integer (String s) throws NumberFormatException;
    :
    :
    public static int parseInt (String s) throws NumberFormatException;
    :
    :
}
```

35

35

Checked – Cơ chế xử lý

```
try
{
    System.out.print("Type an integer: ");
    s = stringInput.readLine();
    System.out.println("You typed in..." + s);
    num = Integer.parseInt(s);
    System.out.println("Converted to an integer..." + num);
}
catch (IOException e)
{
    System.out.println(e);
}
catch (NumberFormatException e)
{
    :
    :
    :
}
```

36

36

Checked – Cơ chế xử lý

```
Driver.main ()
{
    try
    {
        num = Integer.parseInt (s);
    }
    :
    catch (NumberFormatException e)
    {
        :
    }
}
```

```
Integer.parseInt (String s)
{
    :
    :
}
```

37

37

Checked – Cơ chế xử lý

```
Driver.main ()
{
    try
    {
        num = Integer.parseInt (s);
    }
    :
    catch (NumberFormatException e)
    {
        :
    }
}
```

```
Integer.parseInt (String s)
{
    Người sử dụng không
nhập chuỗi số
}
```

38

38

Checked – Cơ chế xử lý

```
Driver.main ()
{
    try
    {
        num = Integer.parseInt (s);
    }
    :
    catch (NumberFormatException e)
    {
        :
    }
}
```

```
Integer.parseInt (String s)
{
    NumberFormatException e =
    new
    NumberFormatException ();
}
```

39

39

Checked – Cơ chế xử lý

```
Driver.main ()
{
    try
    {
        num = Integer.parseInt (s);
    }
    :
    catch (NumberFormatException e)
    {
        :
    }
}
```

```
Integer.parseInt (String s)
{
    NumberFormatException e =
    new NumberFormatException ();
}
```

40

40

Checked – Cơ chế xử lý

```
Driver.main ()
{
    try
    {
        num = Integer.parseInt (s);
    }
    :
    catch (NumberFormatException e)
    {
        Ngoại lệ sẽ được xử lý ở đây
    }
}
```

```
Integer.parseInt (String s)
{
    :
    :
}
```

41

41

Bắt ngoại lệ

```
catch (NumberFormatException e)
{
    :
    :
    :
}

catch (NumberFormatException e)
{
    System.out.println(e.getMessage());
    System.out.println(e);
    e.printStackTrace();
}
```

42

42

Bắt ngoại lệ

```
catch (NumberFormatException e)
{
    System.out.println(e.getMessage());
    System.out.println(e);
    e.printStackTrace();
}

java.lang.NumberFormatException: For input string: "exception"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:426)
    at java.lang.Integer.parseInt(Integer.java:476)
    at Driver.main(Driver.java:39)
```

For input string: "exception"

java.lang.NumberFormatException: For input string: "exception"

43

43

Bắt ngoại lệ

- Tránh bỏ qua việc xử lý ngoại lệ

```
try
{
    s = stringInput.readLine();
    num = Integer.parseInt(s);
}
catch (IOException e)
{
    //System.out.println(e);
}
```

44

44

Bắt ngoại lệ

```
try
{
    s = stringInput.readLine();
    num = Integer.parseInt(s);
}
catch (IOException e)
{
    System.out.println(e);
}
catch (NumberFormatException e)
{
    // Do nothing here but set up the try-catch block to bypass the
    // annoying compiler error
}
```

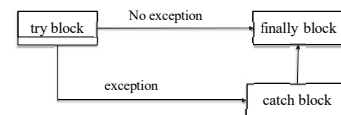
NO!

45

45

Khối finally

- Là 1 khối không bắt buộc trong khối try-catch-finally.
- Dùng để đảm bảo khối lệnh sẽ được thi hành bất kể ngoại lệ có xảy ra hay không. VD:
 - Đóng file, đóng socket, connection
 - Giải phóng tài nguyên (nếu cần)...



46

46

Khối finally

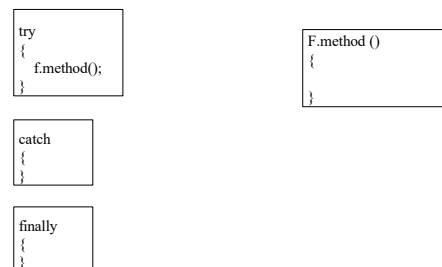
➢ Cú pháp:

```
try
{
    // Code that may cause an error/exception to occur
}
catch (ExceptionType identifier)
{
    // Code to handle the exception
}
finally
{
    // Implement code
}
```

47

47

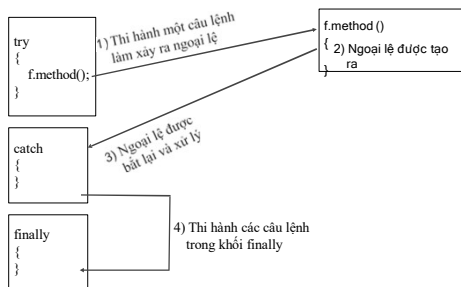
Khối finally: có ngoại lệ



48

48

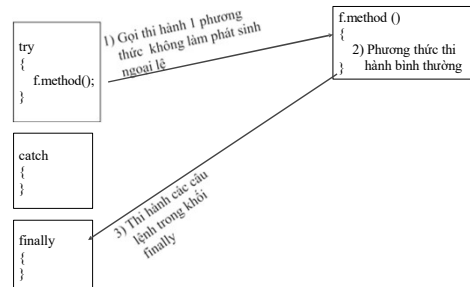
Khối finally: có ngoại lệ



49

49

Khối finally: không có ngoại lệ



50

50

Try-Catch-Finally: Ví dụ

```
class Driver  
{  
    public static void main (String [] args)  
    {  
        TCExample eg = new TCExample ();  
        eg.method();  
    }  
}
```

51

51

Try-Catch-Finally: Ví dụ

```
public class TCExample  
{  
    public void method ()  
    {  
        BufferedReader br;  
        String s;  
        int num;  
        try  
        {  
            System.out.println("Type in an integer: ");  
            br = new BufferedReader(new InputStreamReader(System.in));  
            s = br.readLine();  
            num = Integer.parseInt(s);  
            return;  
        }  
    }  
}
```

52

52

try-with-resources

- Khi dùng try-catch-finally để đọc file, cần finally để đóng tài nguyên, để quên hoặc gây rò rỉ tài nguyên
- try-with-resources được giới thiệu từ Java 7, giúp tự động đóng tài nguyên sau khi sử dụng mà không cần gọi close() trong finally.
- Áp dụng cho các lớp triển khai AutoCloseable (như BufferedReader, FileInputStream, Scanner).
- Sử dụng try-with-resources giúp tự động đóng tài nguyên, tránh lỗiOutOfMemoryError khi mở quá nhiều file hoặc kết nối mà quên đóng.

53

53

Throws

- Từ khóa throws được sử dụng để **khai báo một ngoại lệ**. Nó thể hiện thông tin cho lập trình viên rằng **có thể xảy ra một ngoại lệ**
- Giả sử có method1 và method2. Method1 gọi method2 và method2 là phương thức có khả năng xảy ra ngoại lệ:
 - Hoặc **method2** phải nằm trong khối **try/catch**.
 - Hoặc phải khai báo **method1** có khả năng ném (**throws**) ngoại lệ.

54

54

Ví dụ ngoại lệ IOException

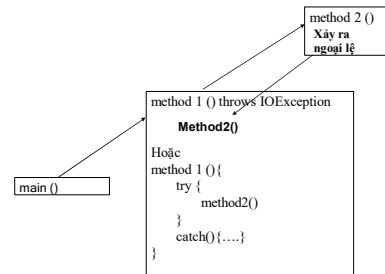
- Cách 1: try/catch

```
public static void main(String[] args)
{
    try {
        String s = buff.readLine();
    } catch (IOException e) {
        ...
    }
}
```
- Cách 2: Khai báo throws

```
public static void main(String[] args) throws IOException
{
    String s = buff.readLine();
}
```

55

Throws



56

Throws

```
import java.io.*;
public class TCExample
{
    public void method () throws IOException, NumberFormatException
    {
        BufferedReader br;
        String s; int
        num;

        System.out.print("Type in an integer: ");
        br = new BufferedReader(new InputStreamReader(System.in));
        s = br.readLine();
        num = Integer.parseInt(s);
    }
}
```

57

Hàm main xử lý ngoại lệ

```
class Driver
{
    public static void main (String [] args) {
        TCExample eg = new TCExample ();
        boolean inputOkay = true;
        do {
            try
            {
                eg.method();
                inputOkay = true;
            }
            catch (IOException e){
                e.printStackTrace();
            }
            catch (NumberFormatException e){
                inputOkay = false;
                System.out.println("Please enter a whole number.");
            }
        } while (inputOkay == false);
    } // End of main
} // End of Driver class
```

58

58

Hàm main không xử lý ngoại lệ

```
class Driver
{
    public static void main (String [] args) throws IOException, NumberFormatException {
        TCExample eg = new TCExample ();
        eg.method();
    }
}
```

throws IOException, NumberFormatException: Phương thức main() có thể phát sinh hai ngoại lệ IOException và NumberFormatException, nhưng không bắt (try-catch), mà ném lại cho trình gọi nó (JVM).

59

Throw

- Từ khoá throw được sử dụng để **ném ra một ngoại lệ cụ thể**, chủ yếu được sử dụng để ném ngoại lệ tùy chỉnh (ngoại lệ do người dùng tự định nghĩa).

60

Throw

- Sử dụng throw **anExceptionObject** trong thân phương thức để tung ra ngoại lệ khi cần

- Ví dụ:

```
public static void method (int so) throws Exception {  
    if (so < 5) {  
        throw new Exception("Qua ít");  
    }  
}
```

61

61

Throw

- Đối với RuntimeException phương thức không cần phải khai báo throws RuntimeException vì ngoại lệ này mặc định được ủy nhiệm cho JVM

```
public static void method (int so) {  
    if (so < 5) {  
        throw new RuntimeException("Qua ít");  
    }  
}
```

62

62

Throw

- Ví dụ 1:

```
static int cal(int no, int no1){ if (no1 == 0){  
    throw new ArithmeticException ("Không thể chia cho 0");  
}  
    int num = no/no1; return num;  
}  
public static void main(String[] args) {  
    int num = cal(6,0);  
}
```

Lỗi ngoại lệ:
Exception in thread "main" java.lang.ArithmeticException: Không thể chia cho 0

63

63

Throw

- Ví dụ 2:

```
static int cal(int no, int no1) throws Exception{  
    if (no1 == 0){  
        throw new ArithmeticException ("Không thể chia cho 0");  
    }  
    int num = no/no1;  
    return num;  
}  
public static void main(String[] args) { int num =  
    cal(6,0);  
}
```

Lỗi biên dịch:
Exception in thread "main" java.lang.RuntimeException:
Uncompilable source code - unreported exception java.lang.Exception;
must be caught or declared to be thrown
at exceptionex.ExceptionEx.main(ExceptionEx.java:58)

64

64

Throw

- Sửa ví dụ 2:

```
static int cal(int no, int no1) throws Exception{  
    if (no1 == 0){  
        throw new ArithmeticException ("Không thể chia cho 0");  
    }  
    int num = no/no1;  
    return num;  
}  
public static void main(String[] args) throws Exception{ int  
    num = cal(6,0);  
}
```

Thêm: throws Exception

65

65

Throw và Throws

throw

Từ khóa throw được sử dụng để ném ra một ngoại lệ rõ ràng.

Ngoại lệ checked không được truyền ra nếu chỉ sử dụng từ khóa throw.

Sau throw là một **instance**.

Throw được sử dụng trong phương thức có thể quăng ra Exception ở bất kỳ dòng nào trong phương thức (sau đó dùng try-catch để bắt hoặc throws cho phương thức khác xử lý)

Không thể throw nhiều exceptions.

throws

Từ khóa throws được sử dụng để khai báo một ngoại lệ.

Ngoại lệ checked không được truyền ra nếu chỉ sử dụng từ khóa throws.

Sau throws là một hoặc nhiều **class**.

Throws được khai báo ngay sau dấu đóng ngoặc đơn của phương thức. Khi một phương thức có throw bên trong mà không bắt lại (try - catch) thì phải ném đi (throws) cho phương thức khác xử lý.

Có thể khai báo nhiều exceptions, Ví dụ:
public void method() throws IOException,
SQLException { }

66

66

Throw và Throws

- Nên dùng throw khi muốn ném ngoại lệ ngay lập tức. Dùng throws khi khai báo một phương thức có thể phát sinh ngoại lệ, giúp các lớp khác biết cần xử lý

```
// Sử dụng throw để ném ngoại lệ ngay lập tức
public void validateAge(int age) {
    if (age < 18) {
        throw new IllegalArgumentException("Tuổi phải >= 18");
    }
}

// Sử dụng throws để khai báo ngoại lệ có thể xảy ra
public void readFile() throws IOException {
    BufferedReader br = new BufferedReader(new FileReader("test.txt"));
    br.readLine();
}
```

67

67

Ngoại lệ cần phải kiểm tra

> Một phương thức có thể throw nhiều hơn 1 ngoại lệ:

```
public void method(int tuoi, String ten) throws ArithmeticException,
    NullPointerException {
    if (tuoi < 18) {
        throw new ArithmeticException("Chưa đủ tuổi!");
    }
    if (ten == null) {
        throw new NullPointerException("Thiếu tên!");
    }
}
```

> Lan truyền ngoại lệ:

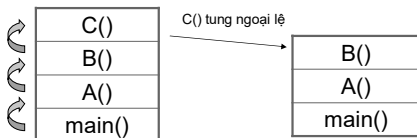
- Trong main() gọi phương thức A(), trong A() gọi B(), trong B() gọi C().
- Giả sử trong C() xảy ra ngoại lệ.

68

68

Ngoại lệ cần phải kiểm tra

- Nếu C() gặp lỗi và throw ra ngoại lệ nhưng trong C() lại không xử lý ngoại lệ này, thì nơi gọi C() là phương thức B() là nơi có thể xử lý ngoại lệ.
- Nếu trong B() cũng không xử lý thì phải xử lý ngoại lệ này trong A()... Quá trình này gọi là lan truyền ngoại lệ.
- Nếu đến main() cũng không xử lý ngoại lệ được throw từ C() thì chương trình sẽ phải dừng lại.



69

69

Ngoại lệ cần phải kiểm tra

- Ném lại ngoại lệ**
- Trong khối catch, ta có thể không xử lý trực tiếp ngoại lệ mà lại ném lại ngoại lệ đó cho nơi khác xử lý.

```
catch (IOException e) {
    throw e;
}
```

- Chú ý: Trong trường hợp trên, phương thức chứa catch phải bắt ngoại lệ hoặc khai báo throws cho ngoại lệ

70

70

Ngoại lệ cần phải kiểm tra

> Kế thừa ngoại lệ:

- Khi override một phương thức của lớp cha, phương thức ở lớp con không được phép tung ra các ngoại lệ mới.
- Phương thức ghi đè trong lớp con chỉ được phép tung ra các ngoại lệ **giống** hoặc là **lớp con** hoặc là **tập con** của các ngoại lệ được tung ra ở lớp cha.

71

71

Ví dụ kế thừa ngoại lệ

```
class Disk {
    public void readFile() throws EOFException {}
}

class FloppyDisk extends Disk {
    public void readFile() throws IOException {} // ERROR!
}

↓

class Disk {
    public void readFile() throws IOException {}
}

class FloppyDisk extends Disk {
    public void readFile() throws EOFException {} // OK
}
```

72

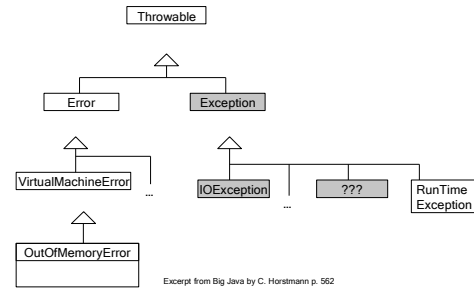
72

Ưu điểm của throws/throw

- Dễ sử dụng
 - Dễ dàng chuyển điều khiển đến nơi có khả năng xử lý ngoại lệ
 - Có thể throw nhiều loại ngoại lệ
- Tách xử lý ngoại lệ khỏi đoạn mã thông thường
- Không bỏ sót ngoại lệ (throws)
- Gom nhóm và phân loại các ngoại lệ

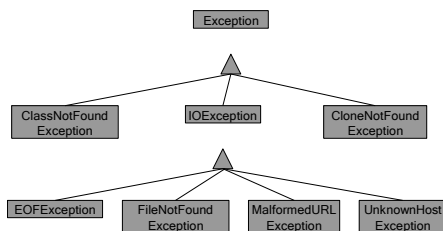
73

Tạo ra kiểu ngoại lệ mới



74

Lớp Exception



75

Tạo ngoại lệ mới

- Mục đích: tạo ra ngoại lệ do người dùng định nghĩa để kiểm soát các lỗi
 - Kế thừa lớp **Exception** hoặc lớp con của nó
 - Có tất cả phương thức của lớp **Throwable**

76

Tạo ngoại lệ tự định nghĩa

- Định nghĩa lớp ngoại lệ

```
public class MyException extends Exception { public
    MyException(String msg) {
        super(msg);
    }
    public MyException(String msg, Throwable cause){ super(msg,
        cause);
    }
}
```

77

Sử dụng ngoại lệ tự định nghĩa

- Sử dụng ngoại lệ

```
public class Example {
    public void kiểmTra(String fName1,String fName2) throws
        MyException {
        if (fName1.equals(fName2))
            throw new MyException("Trung nhau");
        System.out.println("Khong trung");
    }
}
```
- Khai báo khả năng tung ngoại lệ
- Tung ngoại lệ

78

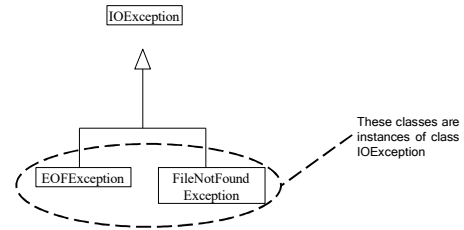
Sử dụng ngoại lệ tự định nghĩa

```
public class Test {  
    public static void main(String[] args) {  
        Example ex= new Example();  
        try {  
            String a = "Test";  
            String b = "Test";  
            ex.kiemTra(a,b);  
        } catch (MyException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

79

79

Cây thừa kế của lớp IOExceptions



80

80

Vấn đề bắt ngoại lệ

- Khi xử lý một chuỗi các ngoại lệ cần phải đảm bảo rằng các ngoại lệ lớp con được xử lý trước các ngoại lệ của lớp cha.
- Xử lý các trường hợp cụ thể trước khi xử lý các trường hợp tổng quát

81

81

Vấn đề bắt ngoại lệ

Đúng

```
try  
{  
  
}  
catch (EOFException e)  
{  
  
}  
catch (IOException e)  
{  
  
}
```

Sai

```
try  
{  
  
}  
catch (IOException e)  
{  
  
}  
catch (EOFException e)  
{  
  
}
```

82

82

Sử dụng record

- record (Java 14+) là một kiểu dữ liệu đặc biệt giúp định nghĩa class bất biến (immutable) nhanh chóng.
- Giảm boilerplate code (giảm viết thủ công getter, equals(), hashCode(), toString()) vì record tự động tạo sẵn.

83

83

Sử dụng record

```
class InvalidDataException extends Exception {  
    private final String field;  
    private final String reason;  
  
    public InvalidDataException(String field, String reason) {  
        super("Lỗi dữ liệu: " + field + " - " + reason);  
        this.field = field;  
        this.reason = reason;  
    }  
  
    public String getField() { return field; }  
    public String getReason() { return reason; }  
}
```

BEFORE

84

84

Sử dụng record

```
record InvalidData(String field, String reason) {}

class InvalidDataException extends Exception {
    public InvalidDataException(InvalidData data) {
        super("Lỗi dữ liệu: " + data.field() + " - " + data.reason());
    }
}
```

AFTER

85

85

3. GENERICS



86

86

Generics

- Xét phương thức cộng hai số nguyên kiểu int

```
public static int Cong(int a, int b)
{
    return a + b;
}
```

- Nhận xét:
 - Không thể dùng phương thức Cong trên để thực hiện cộng hai số kiểu long, float hoặc double.
 - Để cộng được các số kiểu long, float hoặc double cần viết code riêng cho từng kiểu dữ liệu.
 - Để sử dụng chung code cho nhiều kiểu dữ liệu, khi khai báo phương thức hoặc class có thể khai báo một kiểu dữ liệu chung, được gọi là kiểu Generic.

87

87

Generics

- Generics trong Java (Java Generics): là dạng tham số hóa kiểu dữ liệu. Là tham số kiểu hoặc tham số biến hoặc kiểu dữ liệu tổng quát.
- Cho phép tạo và sử dụng lớp, interface hoặc phương thức với nhiều kiểu dữ liệu khác nhau theo từng ngữ cảnh khác nhau.
- Xuất hiện từ Java 5.
- Tham số biến có thể là các kiểu dữ liệu (trừ các kiểu dữ liệu cơ sở Primary type: int, float, char...)
- Khi sử dụng, thay thế tham số biến bằng các kiểu dữ liệu cụ thể.
- Có 2 loại generic: lớp Generic và phương thức Generic.

88

88

Generics

- Có thể sử dụng bất kỳ ký tự, viết hoa hoặc thường cho các tham số generic. Tuy nhiên, có một số quy ước đặt tên:
 - E – Element (phần tử, sử dụng trong Collection Framework)
 - K – Key (khóa)
 - V – Value (giá trị)
 - N – Number (kiểu số: Integer, Long, Float, Double...)
 - T – Type (Kiểu dữ liệu bất kỳ, thuộc kiểu lớp bao: String, Integer, Long, Float...)
 - S, U, V... được sử dụng cho các kiểu loại T thứ 2, 3, 4....

89

89

Generics

- Ký tự Diamond <>: từ Java 7, có thể thay thế các đối số kiểu dữ liệu để gọi hàm khởi tạo của một lớp Generic bằng cặp dấu <>.

```
// Trước Java 7
List<Integer> listInt = new ArrayList<Integer>();
// Sử dụng cặp dấu <> từ phiên bản Java 7
List<Integer> listInt = new ArrayList<>();
```

- Khi dùng <>, không thể tạo mảng generic trực tiếp vì Java **không** hỗ trợ

```
T[] arr = new T[10];
```

90

90

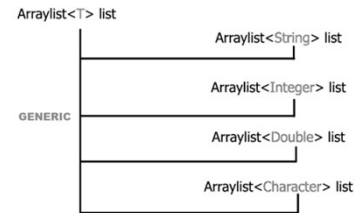
Generics

- Java sử dụng Type Erasure khi biên dịch
 - Khi khai báo một class hoặc method generic, kiểu tham số (T) chỉ tồn tại trong mã nguồn và sẽ bị xóa bỏ (erased) khi biên dịch. Giúp đảm bảo tương thích ngược với các phiên bản Java trước đó.
 - Khi chạy chương trình, các kiểu Generic (T, E, K, V) không còn tồn tại trong bytecode.
 - Java thay thế T bằng một kiểu dữ liệu cụ thể (thường là Object hoặc một kiểu giới hạn cụ thể như Number nếu có T extends Number)
 - Nhưng mảng trong Java cần biết kiểu thực sự tại runtime để kiểm soát kiểu dữ liệu được lưu trữ, điều này xung đột với type erasure, nên không tạo mảng generic được.

91

91

Generics



92

92

Generics

- Ưu điểm của Generics:
 - Kiểm tra kiểu dữ liệu trong thời điểm biên dịch để đảm bảo tính chặt chẽ của kiểu dữ liệu.

```

class A {}
class B extends A {
    public void methodB() {}
}
class C {}
    
```

```

public class GenericEx {
    public static void main(String[] args) {
        List list = new ArrayList();
        list.add(new A());
        list.add(new B());
        list.add(new C());

        ((B)list.get(0)).methodB(); //compile OK
        ((B)list.get(1)).methodB(); //compile OK
        ((B)list.get(2)).methodB(); //compile OK
    }
}
    
```

Exception in thread "main" java.lang.ClassCastException: class GenericEx cannot be cast to class GenericA (GenericEx and GenericA are in unnamed module of loader "app")

at GenericEx.main(GenericEx.java:10)

21/08/2021 10:00:00 AM [GenericEx] [main] [INFO] The following error occurred while executing this line:

21/08/2021 10:00:00 AM [GenericEx] [main] [INFO] java.lang.RuntimeException: java.lang.ClassCastException: class GenericEx cannot be cast to class GenericA (GenericEx and GenericA are in unnamed module of loader "app")

93

93

Generics

```

public class GenericEx {
    public static void main(String[] args) {
        List<A> list = new ArrayList<A>();
        list.add(new A()); //OK Compile
        list.add(new B()); //OK Compile

        list.add(new C());

        for (A item : list){
            if (item instanceof B) ((B)item).methodB();
        }
    }
}
    
```

incompatible types: C cannot be converted to A

.....

(Alt-Enter shows hints)

94

94

Generics

- Loại bỏ việc ép kiểu dữ liệu.

```

//Phải ép kiểu
List list = new ArrayList();
list.add("abc");
String s = (String) list.get(0);
    
```

```

//Không phải ép kiểu
List<String> list = new ArrayList<>();
list.add("abc");
String s = list.get(0);
    
```

95

95

Generics

- Cho phép thực hiện các xử lý tổng quát: thực hiện các thuật toán tổng quát với các kiểu dữ liệu tùy chọn khác nhau.

```

public static <T extends Number> double add(T a, T b){
    return a.doubleValue() + b.doubleValue();
}
    
```

```

public static void main(String[] args) {
    System.out.println(add(1,2));
    System.out.println(add(7.5,15.0));
}
    
```

96

96

Generics

- Generic Collections nhanh hơn Raw Type vì tránh được ClassCastException khi truy xuất phần tử.

97

97

Generic Class

- Generic Class là dạng class có một hoặc nhiều tham số biến sử dụng trong class.
- Một class có thể tham chiếu bất kỳ kiểu đối tượng nào.
- Cú pháp Generic class:
 - class Tên_Class<T1,T2,...,Tn> {}
- Khi sử dụng, khai báo <T> với kiểu dữ liệu cụ thể nào thì trong generic class sẽ chỉ xử lý kiểu dữ liệu đó.
- Phạm vi và ý nghĩa của kiểu T sẽ là trong toàn class
- Sử dụng generic class khi:
 - Khi xây dựng class, chưa xác định được kiểu dữ liệu của biến thành viên, thuộc tính hoặc biến cục bộ của phương thức.
 - Khi nhiều class có cùng chung về mặt logic (các biến, các phương thức) chỉ khác biệt về kiểu dữ liệu.

98

98

Generics class

```
public class KeyValue<K, V> {
    private K key;
    private V value;

    public KeyValue(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public void setKey(K key) {
        this.key = key;
    }

    public V getValue() {
        return value;
    }

    public void setValue(V value) {
        this.value = value;
    }
}
```

99

99

Generics class

```
public static void main(String[] args) {
    // TODO code application logic here
    //KeyValue với kiểu Integer và String
    KeyValue<Integer, String> keyValue = new KeyValue<Integer, String>(1234, "Test");
    Integer id = keyValue.getKey();
    String name = keyValue.getValue();
    System.out.println("ID: " + id + "Name = " + name);

    //KeyValue với kiểu String và String
    KeyValue<String, String> keyString = new KeyValue<>("ABC", "XYZ");
    String key = keyString.getKey();
    String value = keyString.getValue();
    System.out.println("Key: " + key + "Value = " + value);
}
```

100

100

Generic Method

- Generic method (**generic phương thức**) là dạng phương thức có một hoặc nhiều tham số biến.
- Phương thức có thể được gọi với nhiều kiểu dữ liệu khác nhau.
- Cú pháp generic phương thức:
Tiền tố <T1,T2,...,Tn> Kiểu trả về Tên_Method([tham số]){}
- Khi sử dụng, khai báo <T> với kiểu dữ liệu cụ thể nào thì trong generic method sẽ chỉ xử lý kiểu dữ liệu đó.
- Phạm vi và ý nghĩa của kiểu T sẽ là toàn bộ trong method. Hai tham biến cùng kiểu dữ liệu chỉ cần khai báo một kiểu dữ liệu giá T.
- Thao tác trên kiểu dữ liệu <T> giống như là một kiểu dữ liệu bình thường.

101

101

Generic Method

```
public static <T> T max(T a, T b) { // Chỉ cần một T cho cả hai tham số
    return (a.toString().compareTo(b.toString()) > 0) ? a : b;
}
```

- Nhưng nếu hai tham số có kiểu dữ liệu khác nhau, cần nhiều tham số kiểu

```
public static <T, U> void print(T a, U b) { // Cần 2 kiểu khác nhau
    System.out.println(a + " - " + b);
}
```

102

102

Generic Method

- Kiểu <T> có thể được sử dụng làm kiểu trả về của phương thức.
- Sử dụng generic method khi logic của phương thức giống nhau và chỉ khác biệt nhau về kiểu dữ liệu thì có thể cài đặt phương thức theo generic.

```
public static <T> T getMiddle(T... a) {
    return a[a.length/2];
}

public static void main(String[] args) {
    String middle = getMiddle("ABC", "DEF", "IJK");
    System.out.println(middle);
    int midInt = getMiddle(5, 8, 1, 17, 19, 20);
    System.out.println(midInt);
}
```

103

103

Generic Method

```
public static <T extends Comparable> T timMax(T a, T b, T c) {
    T max = a;
    if (max.compareTo(b) < 0)
        max = b;
    if (max.compareTo(c) < 0)
        max = c;
    return max;
}

public static void main(String[] args) {
    int maxInt = timMax(8, 35, 20);
    System.out.println(maxInt);
    double max = timMax(5.7, 8.4, 30.0);
    System.out.println(max);
}
```

104

104

Ký tự đại diện Generic

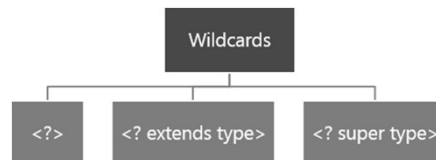
- Ký tự đại diện <?> (wildcard): đại diện cho một kiểu không xác định.
- Có thể được sử dụng trong nhiều tình huống: tham số, biến cục bộ, thuộc tính hoặc có thể là một kiểu trả về.
- Không sử dụng như là một đối số cho lời gọi một phương thức generic, khởi tạo đối tượng class generic, hoặc kiểu cha.
- VD: `Collection<?> coll = new ArrayList<String>();`
`Pair<String, ?> pair = new Pair<String, Integer>();`
- Tham số ký tự đại diện không thể tham gia trong toán tử new

```
List<? extends Object> list = new ArrayList<? extends Object>();
//Lỗi
```

105

105

Ký tự đại diện Generic



106

106

Ký tự đại diện Generic

- Có thể dùng để hạn chế kiểu dữ liệu của các tham số:
 - <? extends type>: kiểu dữ liệu kế thừa từ type hoặc đối tượng của type.
 - <? super type>: kiểu dữ liệu là kiểu cha type hoặc đối tượng của type.

```
public static <T extends Comparable> T timMax(T a, T b, T c) {
    T max = a;
    if (max.compareTo(b) < 0)
        max = b;
    if (max.compareTo(c) < 0)
        max = c;
    return max;
}
```

```
class A {}
class B extends A {}
class C {}
```

```
public static void main(String[] args) {
    List<B> listB = new ArrayList<B>();
    methodB(listB);
    List<A> listA = new ArrayList<A>();
    methodB(listA);
    List<Object> listO = new ArrayList<Object>();
    methodB(listO);
    List<C> listC = new ArrayList<C>();
    methodB(listC);
}
```

107

107

Hạn chế của Generic

- Không thể khởi tạo generic với kiểu dữ liệu cơ sở
- Không tạo được đối tượng của kiểu T

```
KeyValue<int, double> keyValue = new KeyValue<>(1234, 30.3); // Lỗi
```

```
private T obj;
public GenExample() {
    obj = new T();
}
```

- Không là kiểu static trong class

```
class GenExample<T> {
    //Lỗi
    static T obj;
    //Lỗi
    static T getObj() {
        return obj;
    }
}
```

108

108

Hạn chế của Generic

- Có thể khai báo một mảng generic nhưng không thể khởi tạo mảng Generic

109

Hạn chế của Generic

```
public T[] arrayT; //OK
public T[] array = new T[10]; //Lỗi

//Lỗi
GenExample<String> gens[] = new GenExample<>[10];

//OK
GenExample<?> gens[] = new GenExample<?>[10];
gens[0] = new GenExample<Integer>(13);
gens[1] = new GenExample<String>("Gen");
```

Không thể tạo class ngoại lệ là generic

```
a generic class may not extend java.lang.Throwable
----
(Alt-Enter shows hints)

public class GenException<T> extends Exception {}
```

110

109

110

Ví dụ mảng Generic

```
public class GenericArray<T> {
    public T[] array;

    public GenericArray(T[] arr){
        this.array = arr;
    }

    //Phương thức lấy phần tử tại vị trí index
    public T Get(int index){
        if (index >= 0 && index < array.length) return array[index];
        return null;
    }
}

public static void main(String[] args) {
    //Tạo một mảng String
    String[] country = new String[]{"US","UK","AU"};
    GenericArray<String> gArr = new GenericArray<>(country);
    String indCoun = gArr.Get(1);
    System.out.println(indCoun);
}
```

111

111

Bounded Type Parameters

- Dùng **T extends Number** để giới hạn kiểu dữ liệu generic.
- Sử dụng **T extends Number** để đảm bảo T chỉ nhận các kiểu dữ liệu số (Integer, Double, Float...), giúp hạn chế lỗi không mong muốn khi thực hiện các phép toán số học.

```
public class MathUtils<T extends Number> {
    private T number;

    public MathUtils(T number) {
        this.number = number;
    }

    public double square() {
        return number.doubleValue() * number.doubleValue();
    }
}
```

112

112

Bounded Type Parameters

```
class Utils {
    public static <T extends Comparable<T>> T findMax(T a, T b) {
        return (a.compareTo(b) > 0) ? a : b;
    }
}
```

113

113

4. COLLECTIONS



114

114

Collections

- Collections là đối tượng có khả năng chứa các đối tượng khác. Là tập hợp các đối tượng riêng lẻ được biểu diễn như một đơn vị duy nhất.
- Các thao tác thông thường trên collections:
 - Thêm/Xoá đối tượng vào/khỏi collections
 - Kiểm tra một đối tượng có ở trong collections không
 - Lấy một đối tượng từ collections
 - Duyệt các đối tượng trong collections
 - Xoá toàn bộ collections

115

115

Collections Framework

- Là tập hợp các interface và các lớp hỗ trợ thao tác trên tập hợp các đối tượng.
- Hỗ trợ thực hiện các thao tác trên dữ liệu: tìm kiếm, sắp xếp, phân loại, thêm, sửa, xóa...
- Cung cấp các thành phần sau:
 - Interface: là kiểu dữ liệu abstract biểu diễn collection; cho phép các collection thao tác độc lập.
 - Implementations (triển khai): là việc triển khai cụ thể của collection interface.
 - Algorithms (thuật toán): là các phương thức để thực thi các phép toán (tìm kiếm, sắp xếp...) trên các đối tượng đã triển khai các interface collection.

116

116

Collections Framework

- Các collection đầu tiên của Java:
 - Mảng
 - Vector: Mảng động
 - Hashtable: Bảng băm
- Collections Framework (từ Java 1.2)
 - Là một kiến trúc hợp nhất để biểu diễn và thao tác trên các collection.
 - Giúp cho việc xử lý các collection độc lập với biểu diễn chi tiết bên trong của chúng.

117

117

Collections Framework

- Một số lợi ích của Collections Framework
 - Giảm thời gian lập trình
 - Tăng cường hiệu năng chương trình
 - Dễ mở rộng các collection mới
 - Khuyến khích việc sử dụng lại mã chương trình

118

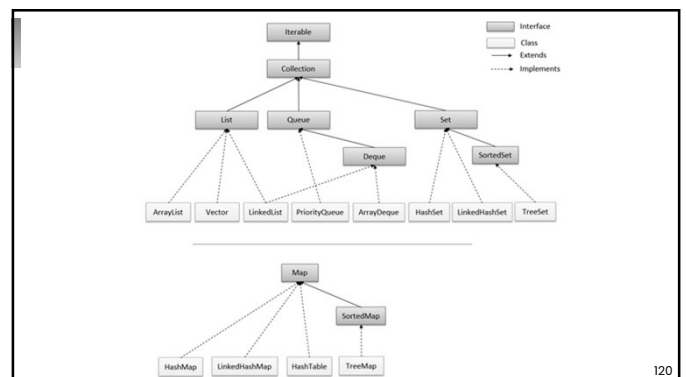
118

Collection Framework

- Thuộc package java.util.
`import java.util.*;`
- Gồm 2 loại chính: Interface Collections, Class Collections.
- Ngoài ra, còn có Map Interface và các class của Map lưu trữ theo cặp key/value

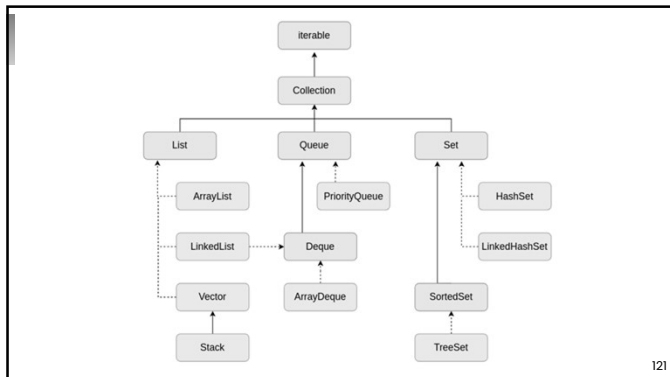
119

119

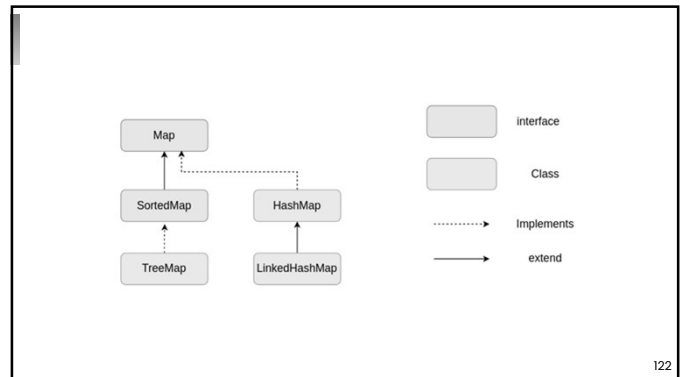


120

120



121



122

Collection Framework

- Iterable
 - Iterable<T> là interface cốt lõi của Java Collection Framework, được tất cả các collection kế thừa.
 - Cung cấp phương thức iterator() để tạo một đối tượng Iterator<T> giúp duyệt qua các phần tử trong collection.
 - Có trong java.lang nên không cần import.
 - Dùng khi muốn tạo một tập hợp có thể duyệt bằng for-each

123

123

Collection Framework

- Ví dụ Iterable

```

import java.util.*;

public class IterableExample {
    public static void main(String[] args) {
        Iterable<String> iterable = Arrays.asList("A", "B", "C"); // List implement Iterable
        for (String s : iterable) {
            System.out.println(s);
        }
    }
}
  
```

124

124

Collection Framework

- Iterable interface: chứa phương thức tạo ra một Iterator.
- Iterator interface:
 - Là đối tượng có trạng thái lặp.
 - Truy xuất các phần tử từ đầu đến cuối của một collection.
 - Xóa phần tử khi lặp một collection.
 - Có 3 phương thức trong Iterator:

Phương thức	Mô tả
public boolean hasNext()	True nếu iterator còn phần tử kế tiếp phần tử đang duyệt.
public Object next()	Trả về phần tử hiện tại và di chuyển con trỏ tới phần tử tiếp theo.
public void remove()	Loại bỏ phần tử cuối được trả về bởi Iterator.

125

125

Collection Framework

- Ví dụ sử dụng Iterator

```

import java.util.*;

public class IteratorExample {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("A", "B", "C");

        Iterator<String> iterator = list.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
  
```

126

126

Collection Framework

- Khi nào dùng Iterator thay vì for-each?
 - Khi cần thêm và xóa phần tử trong khi duyệt (for-each không cho phép).
 - Khi muốn duyệt Set hoặc Map không hỗ trợ truy xuất chỉ mục.

127

127

Collection Framework

- Ví dụ xóa phần tử khi duyệt

```
import java.util.*;

public class RemoveElementIterator {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C", "D"));

        Iterator<String> iterator = list.iterator();
        while (iterator.hasNext()) {
            String s = iterator.next();
            if (s.equals("B")) {
                iterator.remove(); // Xóa phần tử "B"
            }
        }

        System.out.println("After removal: " + list);
    }
}
```

128

128

Collection Framework

- Interface Collections:
 - Là interface chính của Java Collection Framework (thuộc java.util).
 - Là tập hợp đại diện cho nhóm các đối tượng. Trong collection interface có các interface chính như: List interface, Set, SortedSet, Map và SortedMap
 - Một số interface cho phép lưu trữ các phần tử giống nhau hoặc không giống nhau.
 - Tùy từng loại collection, các phần tử có thể có thứ tự hoặc không.
 - Bao gồm các phương thức: thêm (add), xóa (clear), so sánh (compare), duy trì (retaining) đối tượng...
 - Kế thừa lớp Iterable interface, có thể sử dụng Iterator để duyệt từng phần tử.

129

129

Collection Framework

- Ví dụ: Collection sử dụng List

```
import java.util.*;

public class CollectionExample {
    public static void main(String[] args) {
        Collection<String> collection = new ArrayList<>();
        collection.add("One");
        collection.add("Two");
        collection.add("Three");

        for (String item : collection) { // Duyệt for-each do Collection kế thừa Iterable
            System.out.println(item);
        }
    }
}
```

130

130

Collection Framework

- Class Collections: là các lớp tiêu chuẩn dùng để thực thi các Interface Collections.
- Chứa các phương thức tĩnh (static) để thao tác trên Collection, như sắp xếp, tìm kiếm, đảo ngược, tạo danh sách bất biến, v.v.
- Không kế thừa Collection, chỉ hỗ trợ các collection.
 - Trước JDK 1.5 là dạng non-generic; về sau là dạng generic.
 - Non-generic: `ArrayList arr = new ArrayList();`
 - Generic: `ArrayList<String> arr = new ArrayList<String>();`

131

131

Collection Framework

```
import java.util.*;

public class CollectionsExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(3, 1, 4, 2);

        Collections.sort(numbers); // Sắp xếp tăng dần
        System.out.println("Sorted List: " + numbers);

        Collections.reverse(numbers); // Đảo ngược thứ tự
        System.out.println("Reversed List: " + numbers);

        int index = Collections.binarySearch(numbers, 3); // Tìm kiếm nhị phân (phải sắp xếp trước)
        System.out.println("Found 3 at index: " + index);
    }
}
```

132

132

Interface Collections

- Một số collections có hỗ trợ đa luồng (đồng bộ - synchronized), nhưng nhiều loại mặc định là không đồng bộ (không synchronized) để tối ưu hiệu suất.
 - Không đồng bộ (Non-Synchronized): Mặc định, hầu hết các lớp trong Java Collections Framework không đồng bộ để tối ưu hiệu suất. Không an toàn trong môi trường đa luồng, vì có thể xảy ra xung đột dữ liệu nếu nhiều luồng truy cập cùng lúc.
 - Đồng bộ (Synchronized): An toàn trong môi trường đa luồng, nhưng có chi phí về hiệu suất do phải khóa (lock) khi truy cập dữ liệu. Ví dụ: Vector (hiện thay bằng Collections.synchronizedList() hoặc CopyOnWriteArrayList), Hashtable, Stack

133

133

Interface Collections

- Nếu cần sử dụng Collection không đồng bộ trong môi trường đa luồng, có thể dùng Collections.synchronizedXXX().
- Ví dụ: Collections.synchronizedList(new ArrayList<>())
Collections.synchronizedMap(new HashMap<>())
- Java đã phát triển các Collection hỗ trợ đa luồng hiệu suất cao trong java.util.concurrent
- Ví dụ: ConcurrentHashMap - Bản nâng cấp của HashMap
CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>()

134

134

Interface Collections

```
import java.util.concurrent.CopyOnWriteArrayList;

public class CopyOnWriteArrayListExample {
    public static void main(String[] args) {
        CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();
        list.add("A");
        list.add("B");
        list.add("C");

        for (String s : list) {
            System.out.println(s);
        }
    }
}
```

135

135

Interface Collections

- Các Interface và Class Collections:
 - **List**: cấu trúc dữ liệu tuyến tính, các phần tử được sắp xếp theo thứ tự xác định và giá trị có thể trùng nhau. Gồm các class:
 - **ArrayList**: kiểu danh sách sử dụng cấu trúc mảng (có kích thước thay đổi được – **mảng động**) để lưu trữ phần tử; thứ tự các phần tử dựa theo thứ tự lúc thêm vào, giá trị có thể trùng nhau và không phân biệt kiểu dữ liệu của từng phần tử.
 - **LinkedList**: danh sách liên kết đôi (double-linked list), duy trì thứ tự các phần tử được thêm vào và giá trị có thể giống nhau.
 - **Vector**: tương tự ArrayList; kích thước có thể thay đổi được; là dạng synchronized (đồng bộ)
 - **Stack**: lưu trữ trên cơ sở cấu trúc dữ liệu ngăn xếp (stack) LIFO.

136

136

Interface Collections

```
import java.util.*;

public class ListExample {
    public static void main(String[] args) {
        // 1. ArrayList
        List<String> arrayList = new ArrayList<>();
        arrayList.add("A");
        arrayList.add("B");
        System.out.println("ArrayList: " + arrayList);

        // 2. LinkedList
        List<String> linkedList = new LinkedList<>();
        linkedList.add("C");
        linkedList.add("D");
        System.out.println("LinkedList: " + linkedList);

        // 3. Vector (lưu trữ)
        List<String> vector = new Vector<>();
        vector.add("E");
        vector.add("F");
        System.out.println("Vector: " + vector);

        // 4. Stack (LIFO)
        Stack<String> stack = new Stack<>();
        stack.push("G");
        stack.push("H");
        System.out.println("Stack pop: " + stack.pop()); // Lấy phần tử cuối cùng
    }
}
```

137

137

Interface Collections

- Các Interface và Class Collections:
 - **Set**: mỗi phần tử chỉ xuất hiện một lần (giá trị các phần tử không được giống nhau). Gồm các class:
 - **HashSet**: các phần tử được lưu trữ dưới dạng mảng băm (hash table); thứ tự các phần tử không dựa theo lúc thêm vào mà được sắp xếp **ngẫu nhiên không thứ tự** và giá trị các phần tử **không trùng nhau**.
 - **LinkedHashSet**: kế thừa lớp HashSet và implement interface Set; chứa các phần tử duy nhất, **đảm bảo thứ tự phần tử được thêm vào**, cho phép chứa phần tử Null. Có hiệu suất thấp hơn HashSet do cần duy trì danh sách liên kết nội bộ.

138

138

Interface Collections

```
import java.util.*;

public class SetExample {
    public static void main(String[] args) {
        // 1. HashSet
        Set<String> hashSet = new HashSet<>();
        hashSet.add("A");
        hashSet.add("B");
        hashSet.add("A"); // Bị loại bỏ vì trùng lặp
        System.out.println("HashSet: " + hashSet);

        // 2. LinkedHashSet
        Set<String> linkedHashSet = new LinkedHashSet<>();
        linkedHashSet.add("C");
        linkedHashSet.add("D");
        linkedHashSet.add("C"); // Bị loại bỏ
        System.out.println("LinkedHashSet: " + linkedHashSet);
    }
}
```

HashSet: [A, B] hoặc HashSet: [B, A]

LinkedHashSet: [C, D]

Dùng khi chỉ cần tập hợp không trùng lặp mà không quan tâm thứ tự (tối ưu tốc độ).

Dùng khi muốn giữ thứ tự chèn ban đầu nhưng vẫn cần tập hợp không trùng lặp.

139

139

Interface Collections

- Các Interface và Class Collections:
 - **SortedSet**: dạng riêng của Set Interface; giá trị các phần tử mặc định được sắp xếp tăng dần.
 - **TreeSet**: các phần tử mặc định **sắp xếp tăng dần** và giá trị của các phần tử là duy nhất. Tốc độ chậm hơn ($O(\log n)$)
 - Các loại khác: ConcurrentSkipListSet, NavigableSet (mở rộng SortedSet),

140

140

Interface Collections

```
import java.util.*;

public class SortedSetExample {
    public static void main(String[] args) {
        SortedSet<Integer> treeSet = new TreeSet<>();
        treeSet.add(5);
        treeSet.add(3);
        treeSet.add(8);
        System.out.println("TreeSet: " + treeSet); // Tự động sắp xếp
    }
}
```

Dùng TreeSet khi cần dữ liệu sắp xếp tự nhiên.

141

141

Interface Collections

- Các Interface và Class Collections:
 - **Queue**: được thực thi theo kiểu FIFO; có các loại queue: priority queue (queue có ưu tiên), interface deque (queue 2 chiều)...
 - **LinkedList**: là LinkedList trong interface List.
 - **PriorityQueue**: các phần tử được sắp xếp theo trật tự tự nhiên (các phần tử so sánh được với nhau – thi hành Comparable) hoặc theo một bộ so sánh Comparator được cung cấp cho PriorityQueue.
 - **ArrayDeque**: là dạng queue 2 chiều, thực thi dựa trên mảng.
 - LinkedList: Hỗ trợ cả List & Queue.

142

142

Interface Collections

```
import java.util.*;

public class QueueExample {
    public static void main(String[] args) {
        // 1. LinkedList (Queue)
        Queue<String> queue = new LinkedList<>();
        queue.add("A");
        queue.add("B");
        System.out.println("Queue poll: " + queue.poll());
        // Lấy phần tử đầu

        // 2. PriorityQueue
        PriorityQueue<Integer> priorityQueue = new
        PriorityQueue<>();
        priorityQueue.add(3);
        priorityQueue.add(1);
        priorityQueue.add(2);
        System.out.println("PriorityQueue poll: " +
        priorityQueue.poll()); // Lấy phần tử nhỏ nhất
    }
}
```

```
// 3. ArrayDeque (thay thế Stack)
Deque<String> deque = new ArrayDeque<>();
deque.push("X");
deque.push("Y");
System.out.println("ArrayDeque pop: " +
deque.pop()); // Lấy phần tử cuối cùng
}
```

```
Queue poll: A
PriorityQueue poll: 1
ArrayDeque pop: Y
```

143

143

Interface Collections

- Lưu ý:
 - **Nên dùng**:
 - ArrayList (thay vì Vector).
 - HashSet (trừ khi cần giữ thứ tự).
 - TreeSet (khi cần sắp xếp).
 - ArrayDeque (thay cho Stack).
 - **Tránh dùng**:
 - Vector (lỗi thời, thay bằng ArrayList).
 - Stack (Stack vẫn có thể sử dụng trong một số trường hợp, nhưng ArrayDeque có hiệu suất tốt hơn).
 - LinkedList (nếu không cần thêm/xóa nhiều).

144

144

Interface Collections

- Các Interface và Class Collections:
 - **Map (đồ thị/ánh xạ):** dữ liệu của phần tử được quản lý theo dạng cặp key/value; key là duy nhất và ứng với mỗi key là một value. Không kế thừa từ Collection Interface.
 - **HashMap:** giá trị mỗi phần tử bao gồm 2 phần: khóa (key – là duy nhất) được lưu trữ dưới dạng bảng băm và giá trị tương ứng (value); truy xuất trực tiếp dữ liệu bằng khóa; cho phép 1 key null và nhiều giá trị null.
 - **LinkedHashMap:** có thể chứa 1 key là null và nhiều giá trị null; thứ tự các phần tử theo thứ tự thêm (Giữ nguyên thứ tự).
 - **HashTable:** không cho phép key hoặc giá trị null, giống HashMap nhưng đồng bộ hóa (synchronized), hiệu suất chậm hơn.

145

145

Interface Collections

```
import java.util.*;

public class MapExample {
    public static void main(String[] args) {
        // 1. HashMap (Không đảm bảo thứ tự)
        Map<Integer, String> hashMap = new
        HashMap<>();
        hashMap.put(3, "Ba");
        hashMap.put(1, "Một");
        hashMap.put(2, "Hai");
        System.out.println("HashMap: " + hashMap); //
        Không đảm bảo thứ tự

        // 2. LinkedHashMap (Giữ nguyên thứ tự chèn)
        Map<Integer, String> linkedHashMap = new
        LinkedHashMap<>();
        linkedHashMap.put(3, "Ba");
        linkedHashMap.put(1, "Một");

        linkedHashMap.put(2, "Hai");
        System.out.println("LinkedHashMap: " +
        linkedHashMap); // Giữ thứ tự chèn

        // 3. Hashtable (Thread-safe, không cho
        phép null)
        Map<Integer, String> hashtable = new
        Hashtable<>();
        hashtable.put(3, "Ba");
        hashtable.put(1, "Một");
        hashtable.put(2, "Hai");
        System.out.println("Hashtable: " +
        hashtable);
    }
}
```

146

146

Interface Collections

```
HashMap: {1=Một, 2=Hai, 3=Ba} // Có thể thay đổi thứ tự
LinkedHashMap: {3=Ba, 1=Một, 2=Hai} // Giữ đúng thứ tự chèn
Hashtable: {1=Một, 2=Hai, 3=Ba} // Không đảm bảo thứ tự nhưng đồng bộ
```

- Cho phép null
 - HashMap, LinkedHashMap: Key 1 null, Value nhiều null
 - Hashtable: Không cho phép null
- Sử dụng:
 - HashMap: Khi cần hiệu suất cao
 - LinkedHashMap: Khi cần giữ thứ tự chèn
 - Hashtable: Khi cần đồng bộ hóa

147

147

Interface Collections

- Các Interface và Class Collections:
 - **SortedMap:** là dạng riêng của Map Interface, giá trị key được sắp xếp tăng dần.
 - **TreeMap:** giá trị mỗi phần tử bao gồm 2 phần: khóa (key – là duy nhất) và giá trị tương ứng (value); key được sắp xếp tăng dần.
 - Khác: ConcurrentSkipListMap (Hỗ trợ đa luồng, thay thế TreeMap). ImmutableSortedMap

148

148

Interface Collections

```
import java.util.*;

public class SortedMapExample {
    public static void main(String[] args) {
        // 1. TreeMap (Sắp xếp theo Key)
        SortedMap<Integer, String> treeMap = new TreeMap<>();
        treeMap.put(3, "Ba");
        treeMap.put(1, "Một");
        treeMap.put(2, "Hai");
        System.out.println("TreeMap: " + treeMap); // Tự động sắp xếp theo key tăng dần
    }
}
```

TreeMap: {1=Một, 2=Hai, 3=Ba} // Sắp xếp theo key

149

149

Interface Collections

- Các phương thức trong Interface Collections:

Phương thức	Mô tả
boolean add(Object element)	Thêm một phần tử vào collection.
boolean addAll(Collection c)	Thêm các phần tử collection được chỉ định.
boolean remove(Object element)	Xóa phần tử từ collection.
boolean removeAll(Collection c)	Xóa tất cả các phần tử của collection được chỉ định.
boolean retainAll(Collection c)	Giữ lại các phần tử collection được chỉ định.
int size()	Tổng số các phần tử trong collection.
void clear()	Xóa tất cả các phần tử khỏi collection.
boolean contains(Object element)	True nếu collection chứa phần tử được chỉ định.
boolean containsAll(Collection c)	True nếu collection chứa collection con được chỉ định.
Iterator iterator()	Trả về một iterator.
Object[] toArray()	Trả về mảng chứa tất cả phần tử của collection.
boolean isEmpty()	True nếu collection rỗng.
boolean equals(Object element)	So sánh một đối tượng với collection.
int hashCode()	Trả về giá trị hashcode của collection.

150

150

A - ArrayList

- Là lớp thực thi của List Interface
- Sử dụng cấu trúc mảng để lưu trữ phần tử.
- Thứ tự các phần tử dựa theo lúc thêm vào và giá trị có thể trùng nhau.
- Chứa dữ liệu thuộc bất cứ kiểu dữ liệu nào
- Các phần tử có thể có kiểu dữ liệu khác nhau (non-generic). Thuộc java.util.ArrayList.
- Là loại không đồng bộ (non-synchronized). Cho phép truy cập ngẫu nhiên.
- Lưu trữ theo chỉ mục, tốc độ truy xuất (get) nhanh.
- Thêm (add), xóa (remove) chậm vì cần nhiều sự chuyển. Thường dùng khi cần truy xuất phần tử nhiều hơn cập nhật và xóa phần tử.

151

151

A - ArrayList

- Khai báo và khởi tạo ArrayList có 2 cách:
 - Cách 1: khởi tạo một ArrayList rỗng

```
ArrayList<String> list = new ArrayList<String>();
```
 - Cách 2: khởi tạo và cung cấp số lượng phần tử ban đầu

```
ArrayList<Integer> listInt = new ArrayList<>(10);
```
- Truy xuất phần tử dùng phương thức get(int index).

```
String s = list.get(1);  
Integer num = listInt.get(2);
```

152

152

A - ArrayList

- Duyệt ArrayList dùng vòng lặp For:

```
for (int i = 0; i < list.size(); i++){  
    System.out.println(list.get(i));  
}
```
- Hoặc

```
for (int num : listInt){  
    System.out.println(num);  
}
```
- Duyệt ArrayList sử dụng Iterator:
 - Thuộc java.util.Iterator.
 - Khai báo Iterator cùng kiểu với ArrayList muốn duyệt.

```
Iterator<String> itr = list.iterator();
```
 - Dùng hàm hasNext() và hàm next() để duyệt.

```
while (itr.hasNext()){  
    System.out.println(itr.next());  
}
```

153

153

A - ArrayList

- Duyệt ArrayList sử dụng ListIterator:
 - Thuộc java.util.ListIterator.
 - Khai báo ListIterator cùng kiểu với ArrayList muốn duyệt.

```
ListIterator<int> listItr = list.listIterator();
```
 - Dùng hàm hasNext() và next() để duyệt list từ đầu tới cuối

```
while (listItr.hasNext()){  
    System.out.println(listItr.next());  
}
```
 - Dùng hàm hasPrevious() và previous() để duyệt list từ cuối về đầu.

```
while (listItr.hasPrevious()){  
    System.out.println(listItr.previous());  
};
```

154

154

A - ArrayList

- Một số phương thức của ArrayList:
 - Thêm phần tử vào cuối danh sách: add(Object o)

```
list.add("XYZ");
```
 - Thêm collection vào cuối danh sách: addAll(Collection c)

```
list.addAll(listString);
```
 - Thêm phần tử vào vị trí bất kỳ trong danh sách:

```
add(int index, Object value)
```

 - ```
list.add(2, "XYZ");
```
  - Thêm collection vào vị trí bất kỳ trong danh sách  

```
addAll(int index, Collection c)
```

    - ```
list.addAll(3, listString);
```
 - Cập nhật giá trị phần tử: set(int index, Object o) list.set(3, "ABC");

155

155

A - ArrayList

- Xóa phần tử tại vị trí index: Remove(int index)

```
list.Remove(3);
```
- Xóa tất cả các phần tử: Clear()

```
list.Clear();
```

156

156

A - ArrayList

Phương thức	Mô tả
boolean isEmpty()	True nếu ArrayList rỗng.
int indexOf (Object o)	Trả về vị trí index trong list của phần tử o xuất hiện đầu tiên, hoặc -1 nếu List không chứa phần tử này.
int lastIndexOf(Object o)	Trả về vị trí index của phần tử o cuối cùng, hoặc -1 nếu List không chứa phần tử này.
boolean removeAll(Collection c)	Xóa tất cả các phần tử của ArrayList được chỉ định.
boolean retainAll(Collection c)	Giữ lại các phần tử ArrayList được chỉ định.
void removeRange(int fromIndex, int toIndex)	Gỡ bỏ từ list này tất cả phần tử từ vị trí fromIndex đến toIndex.
int size()	Tổng số các phần tử trong ArrayList.
boolean contains(Object element)	True nếu ArrayList chứa phần tử được chỉ định.
void trimToSize()	Cắt kích thước của ArrayList này về kích thước hiện tại.
Object[] toArray()	Trả về một mảng chứa tất cả phần tử của list.
Object clone()	Trả về một bản copy của ArrayList này ⁿⁿ .

157

157

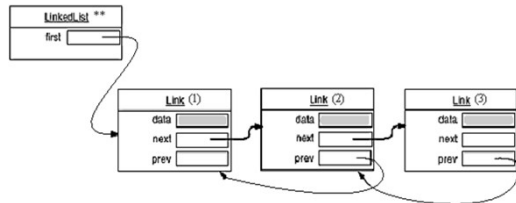
B - LinkedList

- Là lớp thực thi của List Interface và Deque Interface
- Sử dụng cấu trúc danh sách liên kết **Doubly Linked List** để lưu trữ phần tử.
- Duy trì thứ tự các phần tử thêm vào và giá trị có thể trùng nhau.
- Thuộc **java.util.LinkedList**.
- Là loại không đồng bộ (non-synchronized).
- Thêm (add), xóa (remove) nhanh vì không cần phải dịch chuyển nếu bất kỳ phần tử nào thêm/xóa khỏi danh sách.
- Có thể sử dụng như danh sách (list), ngăn xếp (stack) hoặc hàng đợi (queue).
- Thường dùng khi cần làm việc với thêm/xóa lượng lớn phần tử.

158

158

B - LinkedList



159

159

B - LinkedList

- Khai báo và khởi tạo LinkedList:
 - Khởi tạo một LinkedList rỗng
`LinkedList<String> list = new LinkedList<String>();`
 - Khởi tạo với danh sách phần tử:
`LinkedList(Collection c)`
- Truy xuất phần tử dùng phương thức `get(int index)`.
`String s = list.get(1);`
- Duyệt LinkedList dùng vòng lặp For hoặc Iterator hoặc ListIterator giống với ArrayList
`Integer num = listInt.get(2);`

160

160

B - LinkedList

Phương thức	Mô tả
boolean add(Object o)	Thêm phần tử vào cuối list
boolean add(int index, Object o)	Thêm phần tử vào vị trí index của list.
boolean addAll(Collection c)	Thêm một collection vào cuối list.
boolean addAll(int index, Collection c)	Thêm một collection vào vị trí index của list
boolean addFirst(Object o)	Thêm phần tử vào đầu list
boolean addLast(Object o)	Thêm phần tử vào cuối list
void clear()	Xóa tất cả các phần tử khỏi list.
boolean contains(Object o)	True nếu list chứa phần tử được chỉ định.
Object get(int index)	Trả về phần tử tại vị trí index của list
Object getFirst()	Trả về phần tử đầu tiên của list
Object getLast()	Trả về phần tử cuối của list
int indexOf(Object o)	Trả về vị trí index của phần tử o xuất hiện đầu tiên trong list

161

161

B - LinkedList

Phương thức	Mô tả
int lastIndexOf(Object o)	Trả về vị trí index của phần tử o cuối cùng, hoặc -1 nếu List không chứa phần tử này.
Object remove(int index)	Xóa phần tử tại vị trí index trong list.
boolean remove(Object o)	Xóa phần tử o xuất hiện đầu tiên trong list.
Object removeFirst()	Xóa phần tử đầu tiên trong list.
Object removeLast()	Xóa phần tử cuối cùng trong list.
Object set(int index, Object o)	Thay thế phần tử tại vị trí index bằng phần tử o
int size()	Trả về số phần tử trong list
Object[] toArray()	Trả về một mảng chứa tất cả phần tử của list.

162

162

So sánh ArrayList và LinkedList

ArrayList	LinkedList
Sử dụng mảng động để lưu trữ phần tử	Sử dụng danh sách liên kết (Doubly Linked List) để lưu trữ phần tử.
Lưu trữ dữ liệu trên chỉ mục (index), mỗi phần tử (element) liên kết với một index.	Mỗi phần tử (node) lưu trữ 3 thông tin: giá trị phần tử, tham chiếu phần tử trước và tham chiếu phần tử sau.
Thao tác thêm, xóa chậm vì sử dụng nội bộ mảng; sau khi thêm, xóa cần sắp xếp lại	Thêm, xóa nhanh hơn ArrayList; không cần sắp xếp lại phần tử, chỉ cập nhật lại tham chiếu tới phần tử trước và sau.
Truy xuất nhanh (dựa trên index)	Truy xuất chậm hơn nhiều ArrayList vì phải duyệt lần lượt các phần tử từ đầu tới cuối
Truy xuất ngẫu nhiên nhiều phần tử	Không thể truy xuất ngẫu nhiên.
Chỉ hoạt động như một list vì là implements của List Interface	Hoạt động như list, stack hoặc queue, vì là implements của List và Deque Interface
Ít tốn bộ nhớ.	Cần nhiều bộ nhớ.
Tốt trong việc lưu trữ và truy xuất (get)	Tốt trong việc thực hiện thêm, xóa.

163

163

C - TreeSet

- Là lớp thực thi của SortedSet Interface Sử dụng TreeMap để lưu trữ phần tử.
- Các phần tử mặc định được sắp xếp tăng dần hoặc dựa trên bộ so sánh Comparator tùy chỉnh lúc khởi tạo.
- Giá trị phần tử là duy nhất và không null.
- Là loại không đồng bộ (non-synchronized).
- Thuộc **java.util.TreeSet**.
- Duyệt TreeSet dùng vòng lặp For hoặc Iterator.

164

164

C - TreeSet

- Khai báo và khởi tạo TreeSet:
 - Khởi tạo một TreeSet rỗng
`TreeSet<String> list = new TreeSet <>();`
 - Khởi tạo với danh sách phần tử: `TreeSet<Sorter< s> s)`
`TreeSet <Integer> listInt = new TreeSet <>((Int));`
 - Khởi tạo với với bộ so sánh Comparator tùy chỉnh.
`TreeSet<String> list = new
TreeSet<>(String.CASE_INSENSITIVE_ORDER);`
- Hoặc `TreeSet<String> list = new
TreeSet<>(Comparator.reverseOrder());`
- Hoặc `TreeSet<String> list = new TreeSet<>(new Comparator<String>() {
@Override
public int compare(String s1, String s2) { return
s2.compareTo(s1);
}
});`

165

165

C - TreeSet

- Các phương thức trong TreeSet

Phương thức	Mô tả
<code>void add(Object o)</code>	Thêm phần tử vào TreeSet
<code>boolean addAll(Collection c)</code>	Thêm một collection vào TreeSet.
<code>boolean remove(Object o)</code>	Xóa phần tử khỏi TreeSet
<code>void clear()</code>	Xóa tất cả các phần tử khỏi TreeSet
<code>boolean contains(Object o)</code>	True nếu TreeSet chứa phần tử được chỉ định.
<code>Object first()</code>	Trả về phần tử đầu tiên (nhỏ nhất) của TreeSet
<code>Object last()</code>	Trả về phần tử cuối cùng (lớn nhất) của TreeSet
<code>SortedSet subSet(Object fromElement, Object toElement)</code>	Trả về SortedSet từ fromElement đến phần tử đứng trước toElement
<code>SortedSet headSet(E toElement)</code>	Trả về SortedSet từ phần tử đầu tiên đến phần tử đứng trước toElement
<code>SortedSet tailSet(E fromElement)</code>	Trả về SortedSet từ phần tử lớn hơn hoặc bằng fromElement đến phần tử cuối cùng
<code>int size()</code>	Trả về số phần tử trong TreeSet
<code>boolean isEmpty()</code>	True nếu TreeSet rỗng.

166

166

D – TreeMap

- Là lớp thực thi của SortedMap Interface
- Lưu trữ phần tử (entry) dưới dạng key-value (khóa – giá trị) với key là duy nhất và không null.
- Có thể có nhiều giá trị null
- Các phần tử được sắp xếp theo key tăng dần.
- Thuộc **java.util.TreeMap**.
- Duyệt TreeMap dùng vòng lặp For hoặc Iterator.

167

167

D – TreeMap

- Khai báo và khởi tạo TreeMap:
 - Khởi tạo một TreeMap rỗng
`TreeMap<Integer, String> list = new TreeMap <>();`
 - Khởi tạo với danh sách phần tử:
`TreeMap <Integer , String> list = new TreeMap <>(map);`
 - Hiển thị toàn bộ TreeMap:
`for (Map.Entry<Integer, String> entry : list.entrySet()) {
System.out.println("Key: " + entry.getKey() + ", Value: " +
entry.getValue());
}`
- Hoặc:
`list.forEach((key, value) -> System.out.println("Key: " + key +
", Value: " + value));`

168

168

D – TreeMap

- Các phương thức trong TreeSet

Phương thức	Mô tả
<code>void add(Object o)</code>	Thêm phần tử vào TreeSet
<code>boolean addAll(Collection c)</code>	Thêm một collection vào TreeSet.
<code>boolean remove(Object o)</code>	Xóa phần tử khỏi TreeSet
<code>void clear()</code>	Xóa tất cả các phần tử khỏi TreeSet
<code>boolean contains(Object o)</code>	True nếu TreeSet chứa phần tử được chỉ định.
<code>Object first()</code>	Trả về phần tử đầu tiên (nhỏ nhất) của TreeSet
<code>Object last()</code>	Trả về phần tử cuối cùng (lớn nhất) của TreeSet
<code>SortedSet subSet(Object fromElement, Object toElement)</code>	Trả về SortedSet từ fromElement đến phần tử đứng trước toElement
<code>SortedSet headSet(E toElement)</code>	Trả về SortedSet từ phần tử đầu tiên đến phần tử đứng trước toElement
<code>SortedSet tailSet(E fromElement)</code>	Trả về SortedSet từ phần tử lớn hơn hoặc bằng fromElement đến phần tử cuối cùng
<code>int size()</code>	Trả về số phần tử trong TreeSet
<code>boolean isEmpty()</code>	True nếu TreeSet rỗng.

169

Lựa chọn Collection

- Dựa trên cấu trúc dữ liệu tốt nhất và thuật toán hỗ trợ. Phần tử null?
- Trùng lặp phần tử?
- Truy cập phần tử theo index hay key?
- Thao tác thêm xóa hoặc sửa như thế nào? Sắp xếp, tìm kiếm?

170

Sử dụng Java Streams API

- Sử dụng Java Streams API để xử lý Collections hiệu quả
 - Stream API (Java 8+) là một cách hiện đại để xử lý Collections theo phong cách hàm (Functional Programming).
 - Thay vì sử dụng for hoặc Iterator, Stream giúp viết code ngắn gọn hơn và tận dụng tối ưu đa luồng (parallel processing).
 - Viết code ngắn gọn, dễ hiểu hơn so với vòng lặp for.
 - Tăng hiệu suất với `parallelStream()`.
 - Chỉ đọc, không thay đổi dữ liệu gốc

171

Sử dụng Java Streams API

- Cách tạo:
 - Cách 1: Tạo Stream từ Collection

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
// Tạo Stream từ danh sách
Stream<String> stream = names.stream();
```

- Cách 2: Tạo Stream từ Mảng

```
String[] arr = {"Java", "Python", "C++"};
Stream<String> stream = Arrays.stream(arr);
```

- Cách 3: Tạo Stream trực tiếp

```
Stream<Integer> numberStream = Stream.of(1, 2, 3, 4, 5);
```

172

Sử dụng Java Streams API

- Các thao tác chính trên Stream
 - Intermediate Operation (Trả về Stream, có thể xâu chuỗi)
 - Terminal Operation (Trả về giá trị hoặc Collection, kết thúc Stream)

Loại thao tác	Phương thức phổ biến
Intermediate	<code>filter()</code> , <code>map()</code> , <code>sorted()</code> , <code>distinct()</code>
Terminal	<code>forEach()</code> , <code>collect()</code> , <code>reduce()</code> , <code>count()</code> , <code>allMatch()</code>

173

Sử dụng Java Streams API

- Ví dụ: Duyệt Collection với `forEach()`

```
import java.util.*;

public class StreamForEachExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        // Cách truyền thống
        for (String name : names) {
            System.out.println(name);
        }

        // Cách dùng Stream API
        names.stream().forEach(System.out::println);
    }
}
```

Stream viết ngắn gọn hơn!

174

Sử dụng Java Streams API

- Ví dụ: Lọc danh sách số chẵn từ List<Integer>

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamFilterExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);

        // Lọc số chẵn
        List<Integer> evenNumbers = numbers.stream()
            .filter(n -> n % 2 == 0)
            .collect(Collectors.toList());

        System.out.println("Số chẵn: " + evenNumbers);
    }
}
```

175

175

Sử dụng Java Streams API

- Ví dụ: Sắp xếp danh sách số

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamSortExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(5, 1, 4, 3, 2);

        List<Integer> sortedNumbers = numbers.stream()
            .sorted()
            .collect(Collectors.toList());

        System.out.println("Danh sách sau khi sắp xếp: " + sortedNumbers);
    }
}
```

Kết quả: [1, 2, 3, 4, 5]

176

176

Sử dụng Java Streams API

- Xử lý song song với parallelStream(): parallelStream() giúp xử lý dữ liệu nhanh hơn bằng cách chạy nhiều luồng. Nếu dữ liệu lớn, dùng parallelStream() để tăng hiệu suất. Sinh viên tự tìm hiểu thêm.
- Không nên dùng parallelStream() khi dữ liệu nhỏ hoặc khi xử lý có nhiều thao tác I/O, vì việc chia nhỏ dữ liệu sẽ gây tốn tài nguyên mà không tăng hiệu suất.

177

177

Q & A

Giảng viên: Tạ Việt Phương
E-mail:phuongtv@uit.edu.vn

178

178

Bài tập

- Viết chương trình nhập hai số nguyên a và b, sau đó thực hiện phép chia a / b. Nếu b = 0, ném ra ngoại lệ và hiển thị thông báo "Lỗi: Không thể chia cho 0!".
- Viết chương trình yêu cầu người dùng nhập một số nguyên từ bàn phím. Nếu người dùng nhập chuỗi không phải số, hiển thị "Lỗi: Vui lòng nhập số nguyên hợp lệ!"
- Viết một phương thức generic để tìm số lớn nhất trong hai giá trị bất kỳ có thể so sánh được (T extends Comparable<T>).
- Tạo một lớp Student với thuộc tính id và name. Dùng ArrayList<Student> để lưu danh sách sinh viên. Thêm 3 sinh viên vào danh sách và hiển thị danh sách sinh viên.

179

179

Bài tập

- Tạo lớp Book với các thuộc tính id, title, author.
Tạo lớp Generic Library<T> để quản lý danh sách sách (dùng ArrayList<T>).
Viết các phương thức:
 addBook(T book): Thêm sách vào thư viện.
 removeBook(int id): Xóa sách theo id, nếu không tìm thấy thì ném ngoại lệ BookNotFoundException.
 displayBooks(): Hiển thị danh sách sách.
Trong main(), thêm một số sách vào thư viện, xóa một quyển sách và xử lý ngoại lệ.
Gợi ý: Dùng Generic Library<T> để lưu trữ sách. Dùng Exception Handling để kiểm tra sách có tồn tại không.

180

180