

LẬP TRÌNH JAVA

NHẬP MÔN LẬP TRÌNH JAVA

ThS. Dương Hữu Thành
Khoa CNTT, Đại học Mở Tp.HCM
thanh.dh@ou.edu.vn



1



Nội dung chính

- 1. Giới thiệu sơ lược Java**
- 2. Máy ảo Java**
- 3. Quản lý bộ nhớ trong Java**
- 4. Collection Framework**
- 5. Biểu thức lambda**
- 6. Stream**
- 7. Đa luồng**

2

2



Giới thiệu sơ lược Java



James Gosling – cha đẻ
ngôn ngữ Java



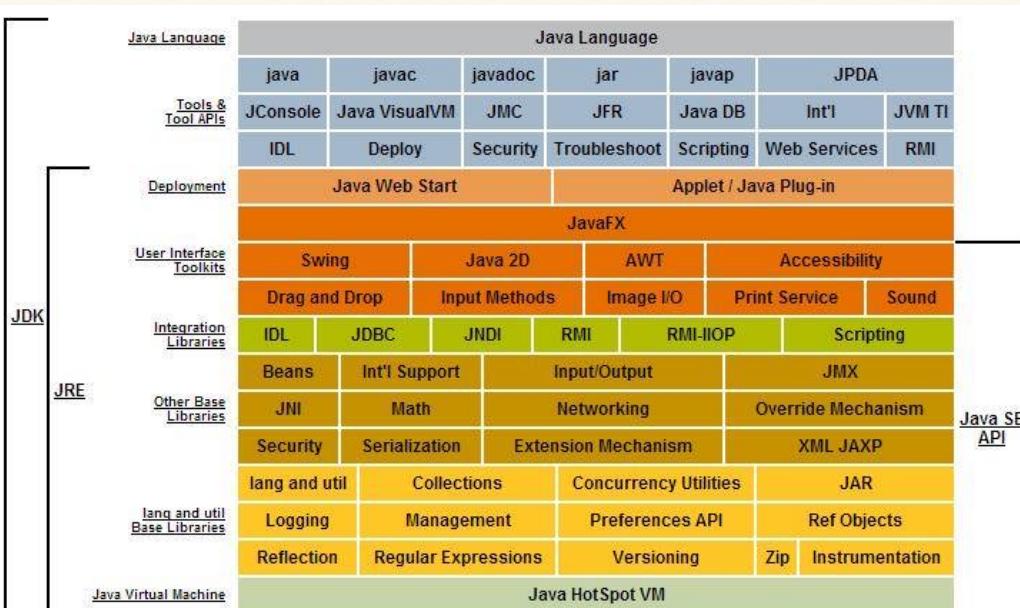
Write Once, Run Anywhere

Dương Hữu Thành, Khoa CNTT, Đại học Mở Tp.HCM

3



Cài đặt môi trường



Dương Hữu Thành, Khoa CNTT, Đại học Mở Tp.HCM

4



Máy ảo Java

- Máy ảo Java (JVM) dùng thực thi chương trình Java,
- Mỗi nền tảng sẽ có một JVM tương ứng có cơ chế hoạt động giống nhau, nên các chương trình chạy giống nhau trên bất kỳ nền tảng nào có hỗ trợ máy ảo Java.
- JVM có 3 thành phần chính: Class Loader, Runtime Data Area, Execution Engine

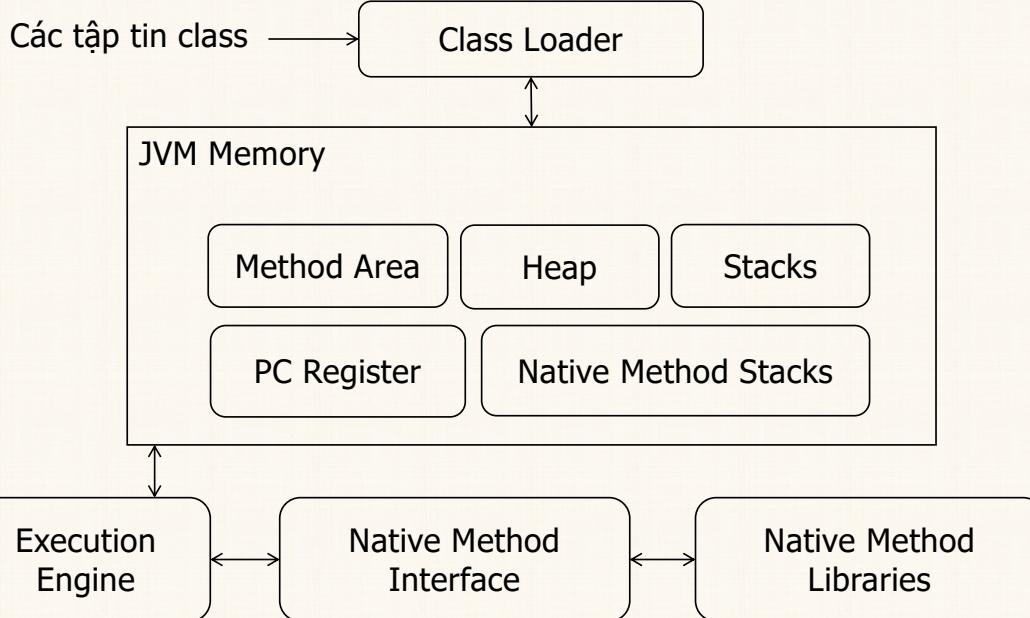
Dương Hữu Thành, Khoa CNTT, Đại học Mở Tp.HCM

5

5



Máy ảo Java



Dương Hữu Thành, Khoa CNTT, Đại học Mở Tp.HCM

6

6



Máy ảo Java

- Class Loader: dùng tìm kiếm và nạp các tập tin .class vào vùng nhớ JVM.
- Execution Engine: chuyển mã bytecode thành mã máy trên hệ điều hành tương ứng và thực thi nó.
- JVM Memory: vùng nhớ hệ điều hành cấp cho JVM.

Dương Hữu Thành, Khoa CNTT, Đại học Mở Tp.HCM

7

7



Máy ảo Java

- Method Area: lưu trữ dữ liệu lớp như tên lớp, tên lớp cha trực tiếp, các phương thức, các thuộc tính (bao gồm thuộc tính tĩnh) và mã nguồn các phương thức. Mỗi JVM chỉ có một Method Area dùng chung cho tất cả các thread.
- Heap Area: lưu trữ các đối tượng kiểu chuỗi, mảng, thể hiện của các lớp. Mỗi JVM được cấp một Heap Area, dùng chung cho tất cả các thread.

Dương Hữu Thành, Khoa CNTT, Đại học Mở Tp.HCM

8

8



Máy ảo Java

- Stack Area: lưu trữ các biến cục bộ, mỗi tiến trình thread có một vùng nhớ stack riêng. Trong stack chứa nhiều frame, mỗi frame sẽ được tạo khi hàm/phương thức được gọi để lưu trữ các biến cục bộ, và các đối số truyền vào cho nó. Frame sẽ bị hủy khi lời gọi hàm/phương thức kết thúc.
- PC Registers (Program Counter): thanh ghi lưu địa chỉ lệnh đang thực thi, mỗi thread có PC riêng.
- Native Method Stacks: chứa các hàm hệ thống dùng trong chương trình, mỗi thread có Native Method Stack riêng.



Quản lý bộ nhớ trong Java

- Khi chương trình bắt đầu, máy ảo Java (JVM) sẽ yêu cầu hệ điều hành cung cấp bộ nhớ trên RAM để chương trình hoạt động.
- JVM chia bộ nhớ được cấp phát thành hai phần: bộ nhớ Heap và bộ nhớ Stack.



Bộ nhớ Heap

- Bộ nhớ này được sử dụng để lưu các đối tượng được tạo ra khi chương trình thực thi (runtime).
- Dung lượng của bộ nhớ này phụ thuộc kích thước và số lượng các đối tượng được tạo trong quá trình thực thi chương trình.
- Khi bộ nhớ trên Heap không đủ để cấp phát cho đối tượng ngoại lệ java.lang.OutOfMemoryError sẽ được ném ra.



Bộ nhớ Heap

- Thời gian sống của đối tượng trên Heap phụ thuộc hoạt động bộ thu gom rác (Garbage Collector).
- Bộ thu gom rác này chạy trên bộ nhớ Heap và sẽ tự động xóa các đối tượng khi trong chương trình không còn biến nào tham chiếu đến nó nữa.



Bộ nhớ Stack

- Bộ nhớ này thường có dung lượng nhỏ dùng lưu trữ các biến cục bộ bao gồm các biến có kiểu dữ liệu cơ bản, các biến tham chiếu đến đối tượng trên bộ nhớ Heap, các đối số được truyền vào hàm.
- Bộ nhớ này hoạt động theo cơ chế LIFO (Last-In First-Out) có nghĩa là biến được tạo sau sẽ bị hủy trước.



Bộ nhớ Stack

- Nếu bộ nhớ Stack không đủ để cấp phát cho biến cục bộ thì ngoại lệ `java.lang.StackOverflowError` sẽ được ném ra.
- Khi một phương thức được gọi, một khôi bộ nhớ trên Stack sẽ được cấp phát cho nó để lưu các biến cục bộ và khôi bộ nhớ này sẽ được giải phóng khi phương thức đã thực hiện xong.



Garbage Collector

- Garbage Collector là bộ thu gom rác trong JVM, nó tự động hủy các đối tượng không còn sử dụng (không có tham chiếu nào đến nó) trong bộ nhớ Heap và không gian này có thể dùng cấp phát cho những đối tượng khác.
- Tham chiếu đến một đối tượng có thể bị hủy trong các trường hợp sau:
 - Biến tham chiếu khai báo cục bộ hết phạm vi hoạt động.
 - Biến tham chiếu đến đối tượng khác.
 - Biến được gán giá trị null.



Garbage Collector

- Biến tham chiếu được khai báo cục bộ trong một phương thức, khi phương thức đó kết thúc thì tham chiếu đến đối tượng cũng bị hủy.
- Biến tham chiếu đến một đối tượng khác.

```
SinhVien sv = new SinhVien();  
sv = new SinhVien("111111", "Dương Lê");
```

- Biến được gán giá trị null.

```
SinhVien sv = new SinhVien();  
sv = null; // sv gán bằng null
```



Generic

- Generic là cách **tham số hóa** kiểu dữ liệu.
- Kiểu Generic **bắt buộc** là kiểu tham chiếu.
- Quy ước dùng các ký hiệu E và T để khai báo generic.
- Lớp hay giao diện đều có thể định nghĩa generic.

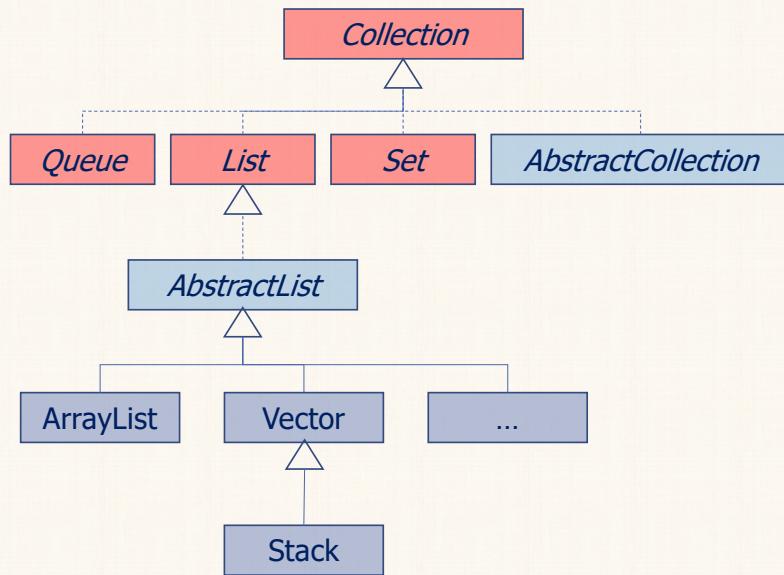


Java Collection Framework

- Java cung cấp nhiều cấu trúc dữ liệu để tổ chức và thao tác với dữ liệu hiệu quả gọi là Java Collection Framework (thuộc gói java.util).
- Nó hỗ trợ hai loại container:
 - Collection: lưu trữ tập các phần tử.
 - Map: lưu trữ các phần tử dạng cặp key/value.



Collection



Dương Hữu Thành, Khoa CNTT, Đại học Mở Tp.HCM

19

19



Collection

java.util.List<E>

- + add(index: int, e: Object): boolean
- + addAll(idx: int, c: Collection<? extends E>): boolean
- + get(index: int): E
- + indexOf(e: Object): int
- + lastIndexOf(e: Object): int
- + listIterator(): ListIterator<E>
- + listIterator(startIndex: int): ListIterator<E>
- + remove(index: int): E
- + set(index: int, e: Object): Object
- + subList(fromIndex: int, toIndex: int): List<E>

java.util.ArrayList<E>

- + ArrayList()
- + ArrayList(c: Collection<? extends E>)
- + ArrayList(initialCapacity: init)
- + trimToSize(): void

java.util.LinkedList<E>

- + LinkedList()
- + LinkedList(c: Collection<? extends E>)
- + addFirst(E: e): void
- + addLast(E: e): void
- + getFirst(): E
- + getLast(): E
- + removeFirst(): E
- + removeLast(): E

Dương Hữu Thành, Khoa CNTT, Đại học Mở Tp.HCM

20

20



Collection

- Các phương thức **tĩnh** của Collection
 - `sort()`: sắp xếp danh sách.
 - `reverse()`: đảo ngược danh sách.
 - `shuffle()`: trộn các phần tử trong danh sách với trật tự ngẫu nhiên.
 - `max()`: tìm phần tử lớn nhất trong danh sách.
 - `min()`: tìm phần tử nhỏ nhất trong danh sách.
 - `frequency()`: đếm số lần xuất hiện của một phần tử trong danh sách.



List

- Giao diện List kế thừa giao diện Collection dùng định nghĩa collection lưu trữ các phần tử liên tiếp nhau.
- Để tạo collection của giao diện List sử dụng hai lớp con cụ thể là `ArrayList` và `LinkedList`, trong đó:
 - `ArrayList` lưu trữ các phần tử thành mảng.
 - `LinkedList` lưu trữ các phần tử thành danh sách liên kết.



Lớp ArrayList

- Lớp này lưu trữ các phần tử thành mảng.
- Khi thêm một phần tử vào ArrayList vượt quá kích thước mảng đã cấp phát thì một mảng mới có kích thước lớn hơn sẽ được tạo ra và sao chép tất cả các phần tử hiện tại vào mảng mới.
- Tuy nhiên, kích thước ArrayList sẽ không tự giảm khi số phần tử nhỏ hơn kích thước mảng đã cấp phát, mà phải sử dụng phương thức `trimToSize()` để làm điều đó.



Lớp ArrayList

```
List<String> lstFruit = new ArrayList<>();  
  
// Thêm phần tử vào ArrayList  
lstFruit.add("Apple");  
lstFruit.add("Banana");  
// Thêm một collection vào ArrayList  
String[] fruits = {"Lemon", "Orange"};  
lstFruit.addAll(Arrays.asList(fruits));  
// Duyệt các phần tử dùng for each  
for (String f: lstFruit)  
    System.out.print(f.toUpperCase() + "\t");  
// Duyệt các phần tử dùng Iterator  
Iterator<String> iter = lstFruit.iterator();  
while (iter.hasNext()) {  
    System.out.print(iter.next().toUpperCase() + "\t");  
}
```



Lớp LinkedList

- Lớp này lưu trữ các phần tử thành danh sách liên kết.

```
LinkedList<String> lstColor = new LinkedList<>();  
// Thêm phần tử đầu danh sách  
lstColor.addFirst("red");  
// Thêm phần tử cuối danh sách  
lstColor.addLast("blue");  
// Thêm phần tử vị trí 1 trong danh sách  
lstColor.add(1, "green");  
// Duyệt các phần tử dùng foreach  
for (String c: lstColor)  
    System.out.print(c.toUpperCase() + "\t");
```



Set

- Tập hợp (Set) là một cấu trúc dữ liệu không cho phép có hai phần tử trùng nhau, giao diện Set kế thừa từ giao diện Collection cho phép tạo ra cấu trúc dữ liệu như vậy.
- Để tạo được các thể hiện của Set sử dụng một trong các lớp con cụ thể sau: HashSet, LinkedHashSet, TreeSet.
- Điểm khác nhau các phần tử HashSet không đúng thứ tự khi thêm vào thì các phần tử trong LinkedHashSet sẽ theo đúng thứ tự khi thêm vào.

```
// Tạo set rỗng, rồi thêm các phần tử vào
Set<String> colors = new HashSet<>();
colors.add("Blue");
colors.add("Green");
colors.add("Blue"); // đã có phần tử này
colors.add("Brown");
System.out.println(colors.size()); // 3
System.out.println(colors); // [Brown, Blue, Green]
// Tạo set từ một danh sách cho trước
Set<String> cols = new HashSet<>(
    Arrays.asList("Blue", "Red", "Blue")
);
System.out.println(cols); // [Red, Blue]
// Hủy phần tử Browns từ tập colors
colors.remove("Brown");
System.out.println(colors); // [Blue, Green]
```

```
// Lấy những phần tử chung các tập colors và cols
colors.retainAll(cols);
System.out.println(colors); // [Blue]
```

- Map dùng lưu trữ các collection, trong đó mỗi phần tử là cặp key/value, với key là một object và không trùng với các phần tử khác trong map.
- Map cho phép việc tìm kiếm, xóa, cập nhật phần tử trong collection nhanh chóng thông qua key.
- Để tạo các thể hiện của Map sử dụng các lớp con: HashMap, LinkedHashMap và TreeMap.

```
Map<String, String> countries = new HashMap<>();  
countries.put("US", "America");  
countries.put("VN", "Vietnam");  
countries.put("AU", "Australia");  
countries.put("JP", "Japan");  
System.out.println(countries);  
// Lấy kích thước map  
System.out.println(countries.size());  
// Lấy danh sách key  
System.out.println(countries.keySet());  
// Lấy danh sách value  
System.out.println(countries.values());  
// Tìm quốc gia có mã là VN  
System.out.println(countries.get("VN"));  
// Xóa quốc gia có mã AU  
countries.remove("AU");  
System.out.println(countries);
```



Biểu thức lambda

- Biểu thức lambda là một đặc trưng mới, nổi bật được giới thiệu từ Java 8.
- Với biểu thức lambda giúp cho việc lập trình trở nên **ngắn gọn, đơn giản và dễ hiểu hơn**.
- **Lập trình hàm** trong Java thể hiện thông qua biểu thức lambda.



Biểu thức lambda

- Cú pháp biểu thức lambda

```
(<kiểu1> p1, <kiểu2> p2, ...) -> <câu-lệnh>  
(<kiểu1> p1, <kiểu2> p2, ...) -> {  
    <các-câu-lệnh>;  
}
```

- Biểu thức lambda sử dụng chính để hiện thực các interface có **duy nhất một** phương thức trừu tượng (gọi là các functional interface hay SAM interface (Single Abstract Method interface)).



Biểu thức lambda

```
public class Demo {  
    public static void main(String[] args) {  
        SayHello s = (name) -> {  
            System.out.println("Hello " + name);  
        };  
        s.say("A");  
        s.say("B");  
    }  
}  
  
interface SayHello {  
    public void say(String name);  
}
```



Stream

- Khái niệm stream được giới thiệu từ Java 8, tương tự như iterator, cho phép thực hiện nhiều thao tác trên các collection.
- Mỗi stream các đối tượng của các lớp hiện thực **giao diện Stream** hoặc các giao diện của các kiểu dữ liệu cơ bản như IntStream, LongStream, DoubleStream.
- Stream có thể **gọi liên tiếp** nhiều phương thức thực hiện một yêu cầu nào đó (stream pipeline).



Stream

Java

forEach()	duyệt các phần tử trong stream.
average()	tính trung bình các phần tử trong stream.
count()	đếm số phần tử trong stream.
min()	tìm phần tử nhỏ nhất trong stream.
max()	tìm phần tử lớn nhất trong stream.
reduce()	trả về giá trị duy nhất của các phần tử bằng một hàm tính toán chỉ định.
filter()	lọc các phần tử trong stream.
sorted()	sắp xếp các phần tử trong stream.
map()	thay đổi giá trị các phần tử trong stream theo một quy tắc chỉ định.
distinct()	trả về các phần tử khác nhau từng đôi một.
limit()	chỉ định số phần tử muốn thao tác.

Dương Hữu Thành, Khoa CNTT, Đại học Mở Tp.HCM

35

35



Stream

Java

```
int[] values = {5, 6, 4, 9, 8};
// duyệt các phần tử
IntStream.of(values).forEach(v -> System.out.println(v));
// tính tổng các phần tử
int sum = IntStream.of(values).sum();
System.out.println("sum = " + sum); // 32
// tính trung bình các phần tử
double avg = IntStream.of(values).average().getAsDouble();
System.out.println("Avg = " + avg); // 6.4
// tìm phần tử nhỏ nhất
int min = IntStream.of(values).min().getAsInt();
System.out.println("min = " + min); // 4
// tìm phần tử lớn nhất
int max = IntStream.of(values).max().getAsInt();
System.out.println("max = " + max); // 9
```

Dương Hữu Thành, Khoa CNTT, Đại học Mở Tp.HCM

36

36



Stream

```
// tính tổng của nhân đôi từng phần tử.  
int kq = IntStream.of(values)  
                    .reduce(0, (x, y) -> x + 2 * y);  
System.out.println(kq);  
  
// nhân đôi từng phần tử trong stream, rồi xuất kết quả.  
IntStream.of(values).map(v -> v * 2)  
                    .forEach(v->System.out.printf("%d\t", v));  
  
// lọc các số chẵn trong stream và sắp xếp tăng  
IntStream.of(values).filter(v -> v % 2 == 0).sorted()  
                    .forEach(v -> System.out.printf("%d\t", v));
```



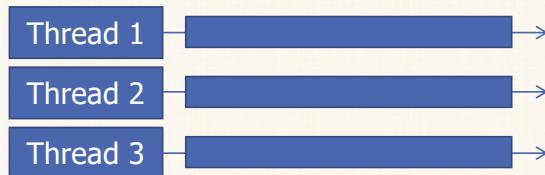
Đa luồng

- Đa luồng (multithreading) cho phép nhiều nhiệm vụ (task) trong chương trình thực thi đồng thời.
- Một thread là một luồng thực thi từ bắt đầu đến kết thúc của một nhiệm vụ (task).
- Một trong những đặc trưng mạnh mẽ của Java là hỗ trợ sẵn tạo và thực thi đa luồng, cũng như khoá các tài nguyên liên quan để ngăn xung đột.
- Đa luồng giúp tăng hiệu năng và tăng tính tương tác của chương trình.

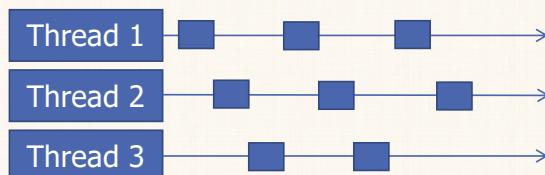


Đa luồng

- Các thread có thể thực thi đồng thời trong các hệ thống multi-processor.
- Nhiều thread chạy trên nhiều cpu



- Nhiều thread chia sẻ cùng một CPU



Sử dụng giao diện Runnable

- Các task là các đối tượng được tạo ra từ lớp hiện thực giao diện Runnable.
- Giao diện này có một phương thức trừu tượng run() cần được hiện thực chỉ định cách thức thread làm việc.



Sử dụng giao diện Runnable

```
public class MyTask implements Runnable {  
    private String name;  
    private int times;  
  
    public MyTask(String name, int times) {  
        this.name = name;  
        this.times = times;  
    }  
  
    @Override  
    public void run() {  
        for (int i = 0; i < this.times; i++)  
            System.out.print(this.name);  
    }  
}
```



Sử dụng giao diện Runnable

▪ Tạo các thread

```
public class MyClient {  
    public static void main(String[] args) {  
        Runnable task1 = new MyTask("a", 50);  
        Thread thread1 = new Thread(task1);  
        Runnable task2 = new MyTask("b", 50);  
        Thread thread2 = new Thread(task2);  
  
        thread1.start();  
        thread2.start();  
    }  
}
```



Sử dụng giao diện Runnable

- Phương thức start() để cho JVM biết thread đã sẵn sàng, khi đó phương thức run() sẽ thực thi.
- Khi chương trình thực thi, hai thread sẽ chia sẻ CPU sẽ in kết quả trên console giống như sau:

```
aaaabbbbbbbbbbbaaaaaaaabbbbbbbbbb  
bbbaaaaabbbbbbbbbbbaaaaaaabaaaaaaaaaa  
aaaaaaaaaaaaaa
```



Sử dụng lớp Thread

- Lớp Thread hiện thực Runnable nên ta có thể tạo lớp task kế thừa Thread.

```
public class MyTask extends Thread {  
    private String name;  
    private int times;  
    public MyTask(String name, int times) {  
        this.name = name;  
        this.times = times;  
    }  
    @Override  
    public void run() {  
        for (int i = 0; i < this.times; i++)  
            System.out.print(this.name);  
    }  
}
```



Sử dụng lớp Thread

- Tạo các thread thực thi

```
public class MyClient {  
    public static void main(String[] args) {  
        Thread thread1 = new MyTask("a", 50);  
        Thread thread2 = new MyTask("b", 50);  
  
        thread1.start();  
        thread2.start();  
    }  
}
```



Sử dụng lớp Thread

- Một số phương thức thông dụng lớp Thread
 - Thread.yield(): tạm giải phóng cho thread khác.
 - Thread.sleep(): cho thread tạm dừng (sleep) trong khoảng thời gian chỉ định để thread khác thực thi.
 - join(): bắt một thread phải chờ một thread khác hoàn tất.
 - Ngoài ra, ta cũng có thể thiết lập và lấy giá trị độ ưu tiên thread thông qua các phương thức getter và setter của thuộc tính priority.



Sử dụng lớp Thread

- Ví dụ sử dụng join()

```
public void run() {  
    Thread t = new AnotherTask("@", 10) ;  
    t.start();  
    try {  
        for (int i = 0; i < this.times; i++) {  
            System.out.print(this.name);  
            if (i == 10)  
                t.join();  
        }  
    } catch (InterruptedException ex) {  
        System.err.println(ex.getMessage());  
    }  
}
```



Thread Pool

- Đối với các chương trình quá số lượng task lớn, việc tạo mỗi thread cho từng task sẽ làm hiệu năng của chương trình không hiệu quả. Thread pool là một giải pháp cho vấn đề này.
- Java cung cấp
 - giao diện Executor để thực thi các task trong thread pool
 - giao diện ExecutorService để quản lý và điều khiển các task.



Thread Pool

▪ Ví dụ

```
// Tạo pool với 2 thread cố định  
ExecutorService executor = Executors.newFixedThreadPool(2);  
  
// Đặt các task vào executor  
executor.execute(new MyTask("a", 10)); // (1)  
executor.execute(new MyTask("b", 20)); // (2)  
executor.execute(new MyTask("c", 5)); // (3)  
  
// Ngừng executor  
executor.shutdown();
```



Thread Pool

- Đoạn chương trình tạo cố định 2 thread, hai task (1) và (2) sẽ được thực thi đồng thời trên 2 thread này, task (3) phải chờ một trong 2 task này hoàn tất để thực thi trên thread đó.
- Trong trường hợp, ta muốn chương trình sẽ tạo thread mới cho các task đang chờ, ta có thể tạo pool như sau:

```
ExecutorService executor  
= Executors.newCachedThreadPool();
```



Thread Synchronization

- Trường hợp các thread có tranh chấp tài nguyên thì việc thực thi đa luồng có thể dẫn đến sai sót.
- Khi đó ta có thể sử dụng từ khoá **synchronized** để đồng bộ phương thức, một thời điểm phương thức đó chỉ được một thread truy cập.
- Phương thức synchronized sẽ **khoá** (lock) phương thức mỗi khi thực thi, lock là cơ chế để cô lập việc sử dụng tài nguyên.
- Đối với phương thức tĩnh synchronized sẽ được khoá cấp class.



Thread Synchronization

- Ví dụ ta có lớp Account như sau:

```
public class Account {  
    private String name;  
    private double amount;  
    public Account(String name, double amount) {  
        this.name = name;  
        this.amount = amount;  
    }  
    public void withdraw(double a) throws InterruptedException {  
        if (this.getAmount() >= a) {  
            Thread.sleep(1000);  
            this.setAmount(this.getAmount() - a);  
            System.out.println(this.amount);  
        } else  
            System.err.println("NOT ENOUGH MONEY");  
    }  
    // getter, setter  
}
```



Thread Synchronization

- Lớp Task rút tiền

```
class WithdrawTask implements Runnable {  
    private Account acct;  
    public WithdrawTask(Account acct) {  
        this.acct = acct;  
    }  
    @Override  
    public void run() {  
        try {  
            this.acct.withdraw(10);  
        } catch (InterruptedException ex) {  
            System.err.println(ex.getMessage());  
        }  
    }  
}
```



Thread Synchronization

- Một chương trình main thực thi đa luồng rút tiền trên cùng một tài khoản.

```
ExecutorService executor  
    = Executors.newCachedThreadPool();  
  
Account a = new Account("A", 15);  
  
executor.execute(new WithdrawTask(a)); // (1)  
executor.execute(new WithdrawTask(a)); // (2)  
  
executor.shutdown();
```



Thread Synchronization

- Lệnh rút tiền (1) thì tài khoản a có đủ tiền nên thực hiện thành công, lúc đó amount chỉ còn 5.
- Lệnh rút tiền (2) thì tài khoản a không còn đủ tiền rút nên nó phải không thực hiện rút tiền.
- Tuy nhiên, khi thực hiện (1) sau khi kiểm tra điều kiện rút tiền, ta đã dùng giả lập thực tế dùng sleep tạm dừng thread này trong 1s, trong thời gian này thread của (2) cũng thực thi, nhưng số tiền a chưa trừ từ thread (1), nên thread (2) vẫn thực thi rút tiền nên kết quả 5 và -5.



Thread Synchronization

- Để giải quyết vấn đề này, ta gắn synchronized vào phương thức withdraw như sau:

```
public synchronized void withdraw(double a) {  
    // ...  
}
```

Q&A

57

57