Tiếng nói của chuyên gia về Công nghệ Java

Bí mật Java 8

Tiết lộ những chức năng mới trong Java 8

Phan Thành Nhân

Nội dung

CHAPTER 0: GIỚI THIỆU

Ebook này sẽ giới thiệu ngắn gọn về Java 8. Sau khi đọc xong, bạn sẽ hiểu cơ bản về những chức năng mới và có thể sử dụng khá là ok.

Nếu đã đọc quyển sách này thì tui mặc định bạn đã kha khá Java và JVM, nếu bạn hong hiểu gì thì có lẽ hơi khó để theo. Tui suggest bạn đọc quyển trước về Java 6 và 7 hihi.

Một số điểm mới trong Java 8 như sau:

- Lambda expressions
- Method references
- Default Methods (Defender methods)
- A new Stream API.
- Optional
- A new Date/Time API.
- Nashorn, the new JavaScript engine
- Removal of the Permanent Generation
- …

CHAPTER 1: LAMBDA EXPRESSIONS

Điểm mới bự nhất của Java 8 chính là việc language level đã support lambda expressions. Lambda expression kiểu như là một anonymous function (function không có tên), nó có thể dùng để truyền như một argument hoặc lưu vào biến. Lambda expression có thể dùng để thay thế anonymous class trong một số trường hợp nhưng nó không phải là anonymous class. Với lamda expession, code sẽ ngắn gọn hơn, dễ đọc hơn, và nhiều hơn thế nữa.

I. Cú pháp

Lambda expression gồm 3 phần: list parameter, arrow, body.

Tùy vào ngữ cảnh, compiler sẽ tự động xác định functional interface để dùng cũng như type của parameter. Có 4 quy tắc quan trọng cho cú pháp:

- Khai báo type của parameter là không bắt buộc
- Không nhất thiết phải sử dụng cặp dấu "()" nếu chỉ có 1 parameter
- Nếu body chỉ có 1 statement thì không cần cặp dấu ngoặc nhọn, lúc này statement sẽ được thay thế với expression

Một số ví dụ

```
() -> System.out.println(this)

(String str) -> System.out.println(str)

str -> System.out.println(str)

(String s1, String s2) -> { return s2.length() - s1.length(); }

(s1, s2) -> s2.length() - s1.length()
```

Ví dụ cuối có thể sử dụng để sort list, trường hợp này lambda expression sẽ implement interface Comparator để sort list bằng độ dài string.

```
Arrays.sort(strArray,

(String s1, String s2) -> s2.length() - s1.length());
```

II. Scope

Sau đây là ví dụ về sử dụng lambda expression với Runnable

```
import static java.lang.System.out;
public class Hello {
  Runnable r1 = () -> out.println(this);
  Runnable r2 = () -> out.println(toString());

  public String toString() { return "Hello, world!"; }

  public static void main(String... args) {
        new Hello().r1.run(); //Hello, world!
        new Hello().r2.run(); //Hello, world!
  }
}
```

Trường hợp trên, cả r1 và r2 đều gọi method toString() của class Hello

Lambda expression không define scope cho riêng nó, nếu trong body gọi “this” thì đó chính là “this” của block bao quanh.

III. Method reference

Với Java 8, bạn có thể sử dụng lamda expression để tạo anonymous method, nhưng đôi khi lambda chỉ làm duy nhất một việc là gọi method có sẵn, method reference sẽ giúp ích trong trường hợp này. Như cái tên của nó, method reference giúp tham chiếu đến một method có sẵn sử dụng tên method. Xét ví dụ sau:

Giả sự bạn muốn filter một list gồm các file dựa trên type của file

```
public class FileFilters {
      public static boolean fileIsPdf(File file) {/*code*/}
      public static boolean fileIsTxt(File file) {/*code*/}
      public static boolean fileIsRtf(File file) {/*code*/}
}
```

Khi bạn muốn filter, bạn có thể sử dụng method reference, giả sử đã có method getFiles() trả về Stream

```
Stream<File> pdfs = getFiles().filter(FileFilters::fileIsPdf);
Stream<File> txts = getFiles().filter(FileFilters::fileIsTxt);
Stream<File> rtfs = getFiles().filter(FileFilters::fileIsRtf);
```

Phần in đậm được gọi là method reference. Hàm getFiles() trả về Stream<File>, khi gọi filter nó sẽ truyền File object vào các method reference.

Method reference có 4 loại:

- Tham chiếu tới static method
- Tham chiếu tới instance method của 1 object tùy ý của 1 type cụ thể
- Tham chiếu tới instance method của 1 object cụ thể
- Tham chiếu tới constructor

1. Tham chiếu tới static method

Ví dụ chúng ta có lambda expression như sau:

`(args) -> Class.staticMethod(args)`

Sẽ được chuyển thành method reference như sau:

`Class::staticMethod`

Không cần phải truyền parameter, parameter sẽ tự động được thêm vào.

```
class Numbers {
  public static boolean isMoreThanFifty(int n1, int n2) {
    return (n1 + n2) > 50;
  }
  public static List<Integer> findNumbers(
    List<Integer> l, BiPredicate<Integer, Integer> p) {
      List<Integer> newList = new ArrayList<>();
      for(Integer i : l) {
        if(p.test(i, i + 10)) {
          newList.add(i);
        }
      }
      return newList;
  }
}

// Somewhere else in your code
List<Integer> list = Arrays.asList(12,5,45,18,33,24,40);
findNumbers(list, Numbers::isMoreThanFifty);
```

2. Tham chiếu tới instance method của 1 object tùy ý của 1 type cụ thể

Lambda expression như sau

`(obj, args) -> obj.instanceMethod(args)`

Sẽ được gọi bằng cách

`ObjectType::instanceMethod`

```
class Shipment {
  public double calculateWeight() {
    double weight = 0;
    // Calculate weight
    return weight;
  }
}
// Somewhere else in your code
public List<Double> calculateOnShipments(
  List<Shipment> l, Function<Shipment, Double> f) {
    List<Double> results = new ArrayList<>();
    for(Shipment s : l) {
      results.add(f.apply(s));
    }
    return results;
}
// Use of method reference
calculateOnShipments(l, Shipment::calculateWeight);
```

3. Tham chiếu tới instance method của 1 object cụ thể

Từ thế này

`(args) -> obj.instanceMethod(args)`

Sẽ chuyển thành thế này

`obj::instanceMethod`

```
class Car {
  private int id;
  private String color;
  // More properties
  // And getter and setters
}
class Mechanic {
  public void fix(Car c) {
    System.out.println("Fixing car " + c.getId());
  }
}

public void execute(Car car, Consumer<Car> c) {
  c.accept(car);
}

final Mechanic mechanic = new Mechanic();
Car car = new Car();
execute(car, mechanic::fix);
```

4. Tham chiếu tới constructor

Chúng ta có lambda expression như thế này

```
(args) -> new ClassName(args)
```

Sẽ được chuyển thành như thế này

```
ClassName::new
```

```
public class Resource {
      public Resource(Integer i) {
            System.out.print(i);
      }
}
// Somewhere else in your code
List<Integer> values = Arrays.asList(1,2,3,4,5);
Values.forEach(Resource::new); // It will print 12345
```

IV. Functional interface

Trong Java 8, functional interface là interface chỉ có 1 abstract method, không tính đến các default method. Miễn sao có 1 abstract method là được.

Java 8 thêm vào một số function interface mới như sau:

- Function<T,R> - takes an object of type T and returns R.
- Supplier<T> - just returns an object of type T.
- Predicate<T> - returns a boolean value based on input of type T
- Consumer<T> - performs an action with given object of type T.
- BiFunction - like Function but with two parameters.
- BiConsumer - like Consumer but with two parameters

Và một số interface tương ứng cho primitive type:

- IntConsumer
- IntFunction<R>

- IntPredicate
- IntSupplier

Cái hay của functional interface là có thể assign cho nó bất cứ cái gì đáp ứng được contract của nó

Function<String, String> atr = (name) -> {return "@" + name;};

Function<String, Integer> leng = (name) -> name.length();

Function<String, Integer> leng2 = String::length;

Mớ code này chạy ngon lành trong Java 8, cái đầu tiên nhận String và trả về String sau khi thêm dấu @ ở trước, 2 cái sau giống nhau, nhận string và return length của nó.

for (String s : args) out.println(leng2.apply(s));

V. Comparisons to Java 7

Để hiểu hơn về lợi ích của việc sử dụng lambda expression, chúng ta sẽ cùng xem một số ví dụ cho thấy code trong Java 8 sẽ ngắn hơn trong Java 7

Tạo ActionListener

```
// Java 7
ActionListener al = new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
         System.out.println(e.getActionCommand());
    }
};
// Java 8
ActionListener al8 = e ->
System.out.println(e.getActionCommand());
```

In ra các phần tử của list string

```
// Java 7
for (String s : list) {
  System.out.println(s);
}
//Java 8
list.forEach(System.out::println);
```

Sort list các string

```
// Java 7
Collections.sort(list, new Comparator<String>() {
  @Override
  public int compare(String s1, String s2) {
        return s1.length() - s2.length();
  }
});
//Java 8
Collections.sort(list, (s1, s2) -> s1.length() - s2.length());
// or
list.sort(Comparator.comparingInt(String::length))
```

Sort: Giả sử bạn có class Person

```
public static class Person {
  String firstName;
  String lastName;
  public String getFirstName() {
        return firstName;
  }
  public String getLastName() {
        return lastName;
  }
}
```

Đây là cách bạn sort list<Person> bằng last name và first name

```
// Java 7
Collections.sort(list, new Comparator<Person>() {
  @Override
  public int compare(Person p1, Person p2) {
        int n = p1.getLastName().compareTo(p2.getLastName());
        if (n == 0) {
              return p1.getFirstName()
                    .compareTo(p2.getFirstName());
        }
        return n;
  }
});
```

```
// Java 8
list.sort(Comparator.comparing(Person::getLastName)
    .thenComparing(Person::getFirstName));
```

CHAPTER 2: DEFAULT METHODS

Để thêm method stream() vào Collection API, Java cần thêm một feature là Default methods, cách này giúp add method vào interface List mà không ảnh hưởng đến code cũ.

Default method là method của interface mà có thể implement được body như trong class bình thường.

Default method có thể dùng ở bất kỳ interface nào. Class nào implements interface có default method thì không nhất thiết phải override default method.

I. Default and Functional

Interface có thể có 1 hoặc nhiều default method mà vẫn là functional interface, miễn sao nó có 1 abstract method.

```
@FunctionalInterface
public interface Iterable {
	Iterator iterator();
	default void forEach(Consumer<? super T> action) {
		Objects.requireNonNull(action);
		for (T t : this) {
			action.accept(t);
		}
	}
}
```

Interface Iterable vừa có iterator() vừa có forEach() nhưng nó vẫn là functional

II. Multiple Defaults

Trường hợp 1 class implements 2 hay nhiều interface mà chúng có cùng default method, Java sẽ báo compilation error. Bạn phải override method đó hoặc chọn 1 trong 2 sử dụng từ khóa super.

```
interface Foo {
	default void talk() {
		out.println("Foo!");
	}
}
interface Bar {
	default void talk() {
		out.println("Bar!");
	}
}
class FooBar implements Foo, Bar {
	@Override
	void talk() { Foo.super.talk(); }
}
```

III. Static Methods on Interface

Cho dù cái này chả liên quan gì tới default method nhưng nó cũng là một thay đổi tương tự đối với interface, method trong interface được gắn từ khóa static sẽ phải implement body và có thể gọi như static method của class.

CHAPTER 3: STREAMS

Interface Stream là một phần chính của Java 8 nên nó xứng đáng là một chapter :D

I. Stream là gì?

Interface stream nằm trong package java.util.stream, stream đại diện cho sequence of objects, nó hỗ trợ nhiều method và parallel execution.

Stream cung cấp map/filter/reduce pattern và execute lazily

Một số stream cho primitive type như IntStream, DoubleStream, and LongStream để hỗ trợ performance.

Stream chỉ được sử dụng 1 lần, không thể dùng lại. Các method của stream được chia làm 2 loại:

- Intermediate operations: return stream và không làm gì cho đến khi terminal operations được gọi (laziness), có các method như map, filter, sorted, limit, distinct, skip, …
- Terminal operations: Tạo kết quả từ stream pineline. Một số method: collect, count, forEach, reduce …

II. Tạo Streams

Streaming Collections

Cách tạo stream đơn giản nhất là từ Collections. Interface Collection có 2 default method để tạo stream:

- stream(): trả về sequential stream từ collection, thứ tự được bảo toàn
- parallelStream(): trả về parallel Stream (nếu có thể) từ collection

Streaming Files

```
BufferedReader.lines() trả về Stream
try (FileReader fr = new FileReader("file");
      BufferedReader br = new BufferedReader(fr)) {
```

```
        br.lines().forEach(System.out::println);
}
```

Streaming File Trees

Class Files có một số method để navigate file tree sử dụng Stream

- list(Path dir) – Stream of files in the given directory.
- walk(Path dir)[2] – Stream that traverses the file tree depth-first starting at the given directory.
- walk(Path dir, int maxDepth) – Same as walk(dir) but with a maximum depth.

Streaming Text Patterns

Class Pattern có method splitAsStream(CharSequence) để tạo Stream

```
import java.util.regex.Pattern;
Pattern patt = Pattern.compile(",");
patt.splitAsStream("a,b,c").forEach(System.out::println);
```

Ví dụ trên tách chuỗi "a,b,c" theo dấu "," rồi trả về stream

Infinite Streams

Sử dụng method generate() hoặc iterate() của Stream, bạn có thể tạo stream vô tận

```
Stream.generate(() -> new Dragon());
Stream.generate(() -> Math.random());
```

Method iterate() cũng tương tự, nhưng nó cần 1 giá trị khởi đầu và 1 nguyên tắc để generate giá trị sau

```
Stream.iterate(1, i -> i+1).forEach(System.out::print);
```

Chương trình sẽ in ra 123456… đến khi dừng

Ranges

Tạo range cho Stream

```
IntStream.range(1, 11).forEach(System.out::println);
```

In ra 1 đến 10

Streaming Anything

Bạn có thể tạo Stream cho bất cứ type nào

```
Stream<Integer> s = Stream.of(1, 2, 3);
```

```
Stream<Object> s2 = Arrays.stream(array);
```

III. For Each

Cái này không cần phải nói nhiều, như foreach bên loop thôi

```
Files.list(Paths.get(".")).forEach(System.out::println);
```

IV. Map/Filter/Reduce

Giả sử bạn có list string chứa player name, bạn muốn tìm người có điểm số cao nhất

```
public static class PlayerPoints {
	public final String name;
	public final long points;
	public PlayerPoints(String name, long points) {
		this.name = name;
		this.points = points;
	}
	public String toString() {
		return name + ":" + points;
	}
}

public static long getPoints(final String name) {
	// gets the Points for the Player
}
```

Để tìm người có điểm số cao nhất

```
PlayerPoints highestPlayer =
names.stream().map(name -> new PlayerPoints(name,
                              getPoints(name)))
             .reduce(new PlayerPoints("", 0.0),
                (s1, s2) -> (s1.points > s2.points) ? s1 : s2);
```

Hàm map có nghĩa là: với mỗi name từ list names, tôi sẽ tạo mới PlayerPoints và đưa nó vào stream. Sau tất cả, sẽ có 1 stream của PlayerPoints

Hàm reduce bắt đầu từ 1 temp PlayerPoints với điểm số 0.0, với phần tử đầu tiên của stream đã lấy từ hàm map, nếu nó có point lớn hơn point của temp PlayerPoints thì sẽ lấy

nó, không thì sẽ lấy temp, giá trị lấy này được dùng để chạy với phần tử tiếp theo của stream. Lặp lại đến khi hết stream thì chúng ta có người chơi có số điểm cao nhất

V. Parallel

Thay vì gọi method stream(), gọi parallelStream() sẽ tạo parallel stream để hỗ trợ việc chạy stream song song trên nhiều thread

```
double bestGpa = students
      .parallelStream() // Tạo stream song song
      .filter(s -> (s.graduationYear == THIS_YEAR))
            // Lấy sinh viên tốt nghiệp năm nay
      .mapToDouble(s -> s.gpa) // Với mỗi sinh viên, lấy GPA
      .max().getAsDouble(); // Lấy điểm cao nhất, return Double
```

VI. Peek

Peek() giúp chạy một action nào đó khi đang stream mà không làm gián đoạn stream, ví dụ như bạn muốn print ra để debug. Đừng thay đổi giá trị element của stream trong peek(), nếu muốn hãy sử dụng hàm map()

```
Files.list(Paths.get("."))
      .map(Path::getFileName)
      .peek(System.out::println)
      .forEach(p -> doSomething(p));
```

VII. Limit

Method limit(int n) sử dụng để giới hạn số phần tử của stream. Ví dụ

```
Random rnd = new Random();
rnd.ints().limit(10)
      .forEach(System.out::println);
```

Sẽ in ra 10 giá trị integer random.

VIII. Sort

Method sorted() sử dụng để sort stream, giống như những intermediate methods khác, sorted() execite lazily, sẽ không có gì được sort nếu không gọi terminal methods.

Lưu ý đối với stream vô tận thì bạn phải gọi sort() sau limit(), không thì error liền

```
    Files.list(Paths.get("."))
          .map(Path::getFileName) // still a path
          .map(Path::toString) // convert to Strings
          .filter(name -> name.endsWith(".java"))
          .sorted() // sort them alphabetically
          .limit(5) // first 5
          .forEach(System.out::println);
```

Đoạn code trên sẽ:

- Lists the files in the current directory.
- Maps those files to file names.
- Finds names that end with ".java".
- Takes only the first five (sorted alphabetically).
- Prints them out.

Trường hợp trên, method sorted() sử dụng default behavior, nhưng nếu bạn muốn sort theo cách khác thì hãy truyền vào interface Comparator như lúc gọi Collections.sort(), interface này là functional nên có thể dùng lambda expression

```
sorted((e1, e2) -> …)
```

IX. Collectors and Statistics

Stream làm lazy, nó cũng có thể execute tuần tự hay song song, do đó cần phải có một cách đặc biệt để lấy kết quả, là Collector.

Collector là một cách để combine các element trong stream thành 1 kết quả

Java 8 đã build sẵn Collector, chỉ việc import và dùng

import static java.util.stream.Collectors.*;

Simple Collectors

Collector đơn giản nhất là toList() và toCollection()

```
// Accumulate names into a List
List<String> list = dragons.stream()
      .map(Dragon::getName)
      .collect(toList());

// Accumulate names into a TreeSet
Set<String> set = dragons.stream()
      .map(Dragon::getName)
      .collect(toCollection(TreeSet::new));
```

Joining

Nếu bạn đã quen với Apache Commons' StringUtil.join thì joining collector y chang như vậy luôn, combine stream sử dụng ký tự phân cách.

```
String names = dragons.stream()
      .map(Dragon::getName)
      .collect(joining(","));
```

Statistics

Sử dụng averaging collector để lấy trung bình

```
System.out.println("\n----->Average line length:");
System.out.println(
      Files.lines(Paths.get("Nio.java"))
      .map(String::trim)
      .filter(s -> !s.isEmpty())
      .collect(averagingInt(String::length))
);
```

Đoạn code phía trên tính độ dài trung bình của các line không empty trong file Nio.java

Nếu muốn collect nhiều thông tin hơn thì hãy dùng SummaryStatistics

```
import java.util.IntSummaryStatistics;
IntSummaryStatistics stats = Files.lines(Paths.get("Nio.java"))
      .map(String::trim)
      .filter(s -> !s.isEmpty())
      .collect(summarizingInt(String::length));

System.out.println(stats.getAverage());
System.out.println("count=" + stats.getCount());
System.out.println("max=" + stats.getMax());
System.out.println("min=" + stats.getMin());
```

Mớ code trên cũng tính giá trị trung bình như ví dụ trước, nhưng nó cũng tính thêm maximum, minimum cũng như số phần tử của stream. Bạn cũng có thể map stream sang sang primitive type rồi gọi summaryStatistics() để có kết quả tương tự

```
IntSummaryStatistics stats = Files.lines(Paths.get("Nio.java"))
            .map(String::trim)
            .filter(s -> !s.isEmpty())
            .mapToInt(String::length)
            .summaryStatistics();
```

X. Grouping and Partitioning

groupingBy nhóm các phần tử theo function truyền vào

```
// Group by first letter of name
List<Dragon> dragons = getDragons();
Map<Character,List<Dragon>> map = dragons.stream()
     .collect(groupingBy(dragon -> dragon.getName().charAt(0)));
```

partitioningBy tạo list dựa vào boolean key

```
// Group by whether or not the dragon is green
Map<Boolean,List<Dragon>> map = dragons.stream()
.collect(partitioningBy(Dragon::isGreen));
```

Để có thể grouping parallel (không quan tâm đến thứ tự), bạn có thể dùng method groupingByConcurrent, stream phải là unordered để cho phép chạy song song. Ví dụ

```
dragons.parallelStream()
     .unordered()
```

```
.collect(groupingByConcurrent(Dragon::getColor));
```

XI. Comparisons to Java 7

Lại đến chuyên mục so sánh với Java 7, chúng mình cùng nhìn xem 2 thằng này nó khác nhau cái gì nhé.

Tìm giá trị lớn nhất

```
// Java 7
double max = 0;


for (Double d : list) {
      if (d > max) {
            max = d;
      }
}
// Java 8
max = list.stream().reduce(0.0, Math::max);
// or
max = list.stream()
      .mapToDouble(Number::doubleValue)
      .max()
      .getAsDouble();
```

Tính giá trị trung bình

```
double total = 0;
double ave = 0;
// Java 7
for (Double d : list) {
total += d;
}
ave = total / ((double) list.size());
//Java 8
ave = list.stream().mapToDouble(Number::doubleValue)
      .average().getAsDouble();
```

In giá trị từ 1 tới 10

```
// Java 7
for (int i = 1; i < 11; i++) {
      System.out.println(i);
}
// Java 8
 IntStream.range(1, 11)
      .forEach(System.out::println);
 //or
 Stream.iterate(1, i -> i+1).limit(10)
      .forEach(System.out::println);
```

Nối Strings

```
// Java 7 using commons-util
List<String> names = new LinkedList<>();
for (Dragon dragon : dragons)
      names.add(dragon.getName());
String names = StringUtils.join(names, ",");
// Java 8
 String names = dragons.stream()
      .map(Dragon::getName)
      .collect(Collectors.joining(","));
```

CHAPTER 4: OPTIONAL

Java 8 giới thiệu java.util.Optional để tránh null value (NullPointerException) tương tự với Optional của Google Guava, Maybe của Nat Pryce's, Option của Scala.

Optional đại diện cho một giá trị có thể có hoặc không tồn tại, nếu không tồn tại thì Optional sẽ là empty, chứ không phải là null.

Bạn có thể dùng Optional.of(x) để tạo giá trị Option với x không null, Optional.empty() để biểu diễn một giá trị không tồn tại (kiểu như null), Optional.ofNullable(x) sẽ tạo một Optional với x có thể null hoặc không null.

Sau khi tạo Optional, dùng method isPresent() để kiểm tra xem giá trị có tồn tại hay không, get() để lấy giá trị. Optional cũng cung cấp một số cách để chơi với giá trị không tồn tại.

- `orElse(T)` – Trả về T nếu Optional là empty.
- `orElseGet(Supplier<T>)` – Gọi Supplier để trả về giá trị khi Optional là empty.
- `orElseThrow(Supplier<X extends Throwable>)` – Gọi Supplier để throw exception khi Optional là empty.

Optional cũng có có những method để chơi với functional style:

- `filter(Predicate<? super T> predicate)` – Filter values và trả về Optional.
- `flatMap(Function<? super T,Optional<U>> mapper)` – Chạy tác vụ mapping để trả về Optional.
- `ifPresent(Consumer<? super T> consumer)` – Chạy Consumer trong trường hợp Optional tồn tại.
- `map(Function<? super T,? extends U> mapper)` – Sử dụng map để return Optional

Stream cũng có vài method trả về Optional:

- `reduce(BinaryOperator<T> accumulator)` – Tạo giá trị duy nhất từ stream.
- `max(Comparator<? super T> comparator)` – Tìm giá trị lớn nhất.
- `min(Comparator<? super T> comparator)` – Tìm giá trị nhỏ nhất.

CHAPTER 5: NASHORN

Nashorn thay thế Rhino như là Javascript engine mặc định của JVM. Nashorn nhanh hơn rất nhiều vì nó sử dụng invokedynamic của JVM. Nó cũng hỗ trợ command line tool (jjs). Để hiểu được invokedynamic thì có cả một topic dài, nhưng bạn chỉ cần hiểu rằng nó là một instruction mới của JVM được thêm vào từ Java 7 để compiler có thể generate code để gọi method.

Để dễ hiểu hơn thì nên đọc Ebook về Java 6 trước, ở đó có nói về JavaScript engine.

Tui đã thông báo rồi nha, đọc không hiểu ráng chịu.

I. jjs

JDK 8 include thêm command line tool jjs để chạy Javascript

Bạn có thể chạy JavaScript file từ command line, nếu đã cài folder bin của Java 8 vào PATH

```
$ jjs script.js
```

II. Scripting

Chạy jjs –scripting để vô shell cho phép bạn chơi với JavaScript

```
jjs> var date = new Date()
jjs> print("${date}")
```

III. ScriptEngine

Trên Java, để chạy code JavaScript thì bạn hãy dùng ScriptEngine, import như sau

```
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
```

Sau đó, sử dụng ScriptEngineManager để lấy Nashorn engine

```
ScriptEngineManager engineManager = new ScriptEngineManager();
```

```
ScriptEngine engine = engineManager.getEngineByName("nashorn");
```

Ok bây giờ có thể chơi với JavaScript trên Java

```
engine.eval("function p(s) { print(s) }");
engine.eval("p('Hello Nashorn');");
```

Engine cũng có thể lấy input từ FileReader

```
engine.eval(new FileReader('library.js'));
```

IV. Importing

Trên Javascript, để import Java class, sử dụng JavaImporter

```
var imports = new JavaImporter(java.util, java.io, java.nio.file);
with (imports) {
      var paths = new LinkedList();
      print(paths instanceof LinkedList); //true
      paths.add(Paths.get("file1"));
      paths.add(Paths.get("file2"));
      paths.add(Paths.get("file3"));
      print(paths) // [file1, file2, file3]
}
```

V. Extending

Trên JavaScript của Nashorn, bạn có thể extend Java class và interface, sử dụng method Java.type() Java.extend(). Ví dụ sau sẽ extend Callable interface và implement nó

```
var concurrent = new JavaImporter(java.util, java.util.concurrent);
var Callable = Java.type("java.util.concurrent.Callable");
with (concurrent) {
      var executor = Executors.newCachedThreadPool();
      var tasks = new LinkedHashSet();
      for (var i=0; i < 200; i++) {
            var MyTask = Java.extend(Callable, {call: function()
            {print("task " + i)}})
            var task = new MyTask();
            tasks.add(task);
            executor.submit(task);
      }
}
```

VI. Invocable

Trên Java, để gọi JavaScript function, bạn phải cast ScriptEngine sang Invocable, sau đó sử dụng method invokeFunction

```
Invocable inv = (Invocable) engine;
engine.eval("function p(s) { print(s) }");
inv.invokeFunction("p", "hello");
```

Hàng thứ 1: cast sang Invocable để sử dụng hàm invokeFunction

Hàng thứ 2: tạo một function trong JavaScript, function p(s)

Hàng thứ 3: gọi hàm p(s) của JavaScript từ Java, truyền vào parameter “hello”

Invocable cũng hỗ trợ method getInterface() lấy function của JavaScript để implement interface của Java. Function của JavaScript và method của interface phải cùng tên.

Giả sử trên Java mình có interface sau:

```
public static interface JPrinter {
 void p(String s);
}
```

Như ví dụ trước, trong JavaScript đã có sẵn function p(s). Bây giờ trên Java, bạn có thể implement Jprinter sử dụng function p(s) như sau

```
JPrinter printer = inv.getInterface(JPrinter.class);
printer.p("Hello again!");
```

OK XONG. TẠM BIỆT CÁCH TÌNH YÊU!