Java Features

Ngôn ngữ lập trình Java và những tính năng mới qua từng phiên bản Java

Java Features Overview

Dưới đây là lịch sử nâng cấp và các tính năng nổi bật qua từng phiên bản Java

java-features2.png

Image not found or type unknown

Java 12 Features

Release Date: 19/03/2019

Java 12 là phiên bản mới nhất của JDK. Cùng tìm hiểu những tính năng và nâng cấp mới nhất.

- Collectors.teeing() in Stream API
- String API Changes
- Files.mismatch(Path, Path)
- Compact Number Formatting
- Support for Unicode 11
- Switch Expressions (Preview)

Java 11 Features

Release Date: 09/2018

Java 11 bao gồm rất nhiều cập nhật quan trọng và hữu ích.

- HTTP Client API
- Launch Single-File Programs Without Compilation
- String API Changes
- Collection.toArray(IntFunction)
- Files.readString() and Files.writeString()
- Optional.isEmpty()

Java 10 Features

Sau khi phát hành phiên bản Java 9, Java 10 được phát hành ngay sau đó. Không giống như các lần cập nhật phiên bản trước đây, Java 10 không có nhiều thay đổi tính năng nổi bật mà nó có các cập nhật thay đổi các bạn viết code và với các phiên bản java trong tương lai.

- JEP 286: Local Variable Type Inference
- JEP 322: Time-Based Release Versioning
- JEP 304: Garbage-Collector Interface

- JEP 307: Parallel Full GC for G1
- JEP 316: Heap Allocation on Alternative Memory Devices
- JEP 296: Consolidate the JDK Forest into a Single Repository
- JEP 310: Application Class-Data Sharing
- JEP 314: Additional Unicode Language-Tag Extensions
- JEP 319: Root Certificates
- JEP 317: Experimental Java-Based JIT Compiler
- JEP 312: Thread-Local Handshakes
- JEP 313: Remove the Native-Header Generation Tool
- New Added APIs and Options
- Removed APIs and Options

Java 9 Features

Release Date: 09/2017. Sự thay đổi lớn nhất là về module hóa (Java modules).

Some important features/changes in Java 9 are:

- Java platform module system
- Interface Private Methods
- HTTP 2 Client
- JShell – REPL Tool
- Platform and JVM Logging
- Process API Updates
- Collection API Updates
- Improvements in Stream API
- Multi-Release JAR Files
- @Deprecated Tag Changes
- Stack Walking
- Java Docs Updates
- Miscellaneous Other Features

Java 8 Features

Release Date: 18/03/2014

Phiên bản này bao gồm nâng cấp và bổ sung các tính năng mới:

- Lambda expression support in APIs
- Stream API
- Functional interface
- Default method & static method
- Method reference
- Optionals

- Nashorn – JavaScript runtime which allows developers to embed JavaScript code within applications
- Annotation on Java Types
- Unsigned Integer Arithmetic
- Repeating annotations
- New Date and Time API
- Statically-linked JNI libraries
- Launch JavaFX applications from jar files
- Remove the permanent generation from GC

Java SE 7 Features

Release Date: 28/07/2011

Phiên bản này được gọi là "Dolphin". Bao gồm các tính năng:

- JVM support for dynamic languages
- Compressed 64-bit pointers
- Strings in switch
- Automatic resource management in try-statement
- The diamond operator
- Simplified varargs method declaration
- Binary integer literals
- Underscores in numeric literals
- Improved exception handling
- ForkJoin Framework
- NIO 2.0 having support for multiple file systems, file metadata and symbolic links
- WatchService
- Timsort is used to sort collections and arrays of objects instead of merge sort
- APIs for the graphics features
- Support for new network protocols, including SCTP and Sockets Direct Protocol

Java SE 6 Features

Release Date: 11/12/2006

Phiên bản này được gọi là “Mustang”. Sun đã loại bỏ ".0" trong số version và đã trở thành tên gọi Java SE 6. Bao gồm các tính năng:

- Scripting Language Support
- Performance improvements
- JAX-WS
- JDBC 4.0
- Java Compiler API
- JAXB 2.0 and StAX parser
- Pluggable annotations

- New GC algorithms

J2SE 5.0 Features

Release Date: 30/09/2004

Phiên bản này được gọi là "Tiger".

Phiên bản này được gọi là 5.0 thay vì 1.5. Bao gồm các tính năng:

- Generics
- Annotations
- Autoboxing/unboxing
- Enumerations
- Varargs
- Enhanced for each loop
- Static imports
- New concurrency utilities in java.util.concurrent
- Scanner class for parsing data from various input streams and buffers.

J2SE 1.4 Features

Release Date: 06/02/2002

Phiên bản này được gọi là "Merlin". Bao gồm các tính năng:

- assert keyword
- Regular expressions
- Exception chaining
- Internet Protocol version 6 (IPv6) support
- New I/O; NIO
- Logging API
- Image I/O API
- Integrated XML parser and XSLT processor (JAXP)
- Integrated security and cryptography extensions (JCE, JSSE, JAAS)
- Java Web Start
- Preferences API (java.util.prefs)

J2SE 1.3 Features

Release Date: 08/05/2000

Phiên bản này được gọi là "Kestrel". Bao gồm các tính năng:

- HotSpot JVM
- Java Naming and Directory Interface (JNDI)
- Java Platform Debugger Architecture (JPDA)

- JavaSound
- Synthetic proxy classes

J2SE 1.2 Features

Release Date: 08/12/1998

Phiên bản này được gọi là "Playground". Đây là một bản phát quan trọng về số lượng các lớp được thêm vào (gần gấp ba lần kích thước). Thuật ngữ J2 J2SE được giới thiệu để phân biệt nền tảng với J2EE và J2ME. Các tính năng bao gồm:

- strictfp keyword
- Swing graphical API
- Sun’s JVM was equipped with a JIT compiler for the first time
- Java plug-in
- Collections framework

JDK 1 Features

Release Date: 23/01/1996

Đây là phiên bản phát hành đầu tiên với tên gọi ban đầu là Oak (Cây sồi). Đây là phiên bản có các API rất không ổn định và một trình duyệt web java có tên WebRunner.

Phiên bản ổn định (stable) đầu tiên là JDK 1.0.2, được gọi là Java 1.

JDK 1.1 được phát hành vào February 19, 1997 gồm các tính năng quan trọng như:

- AWT event model
- Inner classes
- JavaBeans
- JDBC
- RMI
- Reflection which supported Introspection only, no modification at runtime was possible.
- JIT (Just In Time) compiler for Windows

Java 8 - Lambda expression

1. Giới thiệu Lambda

Lamda là một hàm không có tên (unamed function) với các tham số (parameters) và nội dung thực thi (body). Nội dung thực thi của Lamda expression có thể là 1 khối lệnh hoặc 1 biểu thức. Dấu “->” tách biệt các tham số và nội dung thực thi.

Lambda là một trong những thay đổi lớn nhất và được mong đợi nhất của ngôn ngữ Java trong phiên bản phát hành này. Nó cho phép chúng ta xử lý chức năng (functionality) như một đối của phương thức (thông qua function around), hoặc xử lý code như dữ liệu (data): những khái niệm mà mỗi nhà phát triển chức năng rất quen thuộc.

Có rất nhiều ngôn ngữ trên nền tảng JVM (Groovy, Scala…) đã có lambdas từ những ngày đầu tiên, tuy nhiên Java developer thi không có sự lựa chọn cho lambdas với anonymous classes.

Việc thảo luận và thiết kế Lambdas được thực hiện với rất nhiều thời gian và nỗ lực của cộng đồng. Cuối cùng, bản thương mại đã được đề ra, một cấu trúc ngôn ngữ ngắn gọn và súc tích. Trong dạng đơn giản nhất, lambda có thể được biểu diễn như một danh sách các dấu phẩy của các thông số.

2. Cách khai báo và ví dụ

Ví dụ:

```
Arrays.asList( "a", "b", "d" ).forEach( e -> System.out.println( e ) );
```

Chú ý rằng, kiểu đối số e được suy ra từ trình biên dịch compiler. Ngoài ra, bạn cũng có thể cung cấp kiểu đối số một cách tường minh, trong dấu ngoặc đơn.

Ví dụ:

```
Arrays.asList( "a", "b", "d" ).forEach( ( String e ) -> System.out.println( e ) );
```

Trong trường hợp body của lambda phức tạp hơn, nó có thể được đặt trong dấu ngoặc nhọn – như một hàm thông thường trong Java.

Ví dụ:

```
Arrays.asList( "a", "b", "d" ).forEach( e -> {
    System.out.print( e );
    System.out.print( e );
```

```
} );
```

Lambda có thể reference tới thành viên của lớp và các biến cục bộ (chắc chắn rằng các biến này là kiểu final)

Ví dụ: Hai đoạn mã sau là tương đương

```
String separator = ",";
Arrays.asList( "a", "b", "d" ).forEach(
( String e ) -> System.out.print( e + separator )
);
```

và:

```
final String separator = ",";
Arrays.asList( "a", "b", "d" ).forEach(
( String e ) -> System.out.print( e + separator )
);
```

Lambda có thể trả về kết quả giá trị. Kiểu của return value sẽ được suy ra từ trình biên dịch compiler. Câu lệnh return không yêu cầu bắt buộc nếu lambda body chỉ chứa 1 dòng code.

Ví dụ: Hai đoạn mã sau là tương đương

```
Arrays.asList( "a", "b", "d" ).sort( ( e1, e2 ) -> e1.compareTo( e2 ) );
```

và

```
Arrays.asList( "a", "b", "d" ).sort( ( e1, e2 ) -> {
int result = e1.compareTo( e2 );
return result;
} );
```

3. Biểu thức lambda minh bạch và không minh bạch (Explicit and implicit lambda expression)

Mô tả

- Một biểu thức lambda không khai báo kiểu (type) của các tham số (parameter) của nó được gọi là biểu thức lambda không minh bạch (implicit)
- Một biểu thức lambda minh bạch (explicit) là biểu thức có khai báo kiểu của các tham số của nó một cách rõ ràng.
  Trình biên dịch sẽ suy ra các kiểu tham số đối với biểu thức lambda không minh bạch.

Ví dụ 1:

Đoạn code sau tạo một interface với một phương thức duy nhất và sử dụng nó như một loại biểu thức lambda. Khi tạo ra biểu thức lambda chúng ta khai báo tham số s1 có kiểu là Integer

```
public class Main {
    public static void main(String[] args) {
        MyIntegerCalculator myIntegerCalculator = (Integer s1) -> s1 * 2;
        System.out.println("Result: " + myIntegerCalculator.calcIt(5));
    }
}
interface MyIntegerCalculator {
    public Integer calcIt(Integer s1);
}
```

Đoạn code trên sẽ tạo ra kết quả: Result: 10

Ví dụ 2:

Dưới đây là bản demo mà không sử dụng khai báo kiểu cho các tham số. Khi bỏ qua các kiểu, trình biên dịch sẽ tự nhận ra nó.

```
public class Main {
    public static void main(String[] args) {
        MyIntegerCalculator myIntegerCalculator = (s1) -> s1 * 2;
        System.out.println("Result: " + myIntegerCalculator.calcIt(5));
    }
}
interface MyIntegerCalculator {
    public Integer calcIt(Integer s1);
}
```

Đoạn code trên sẽ tạo ra kết quả: Result: 10

4. Tại sao nên sử dụng Lambda

- Giảm số dòng code

Một trong những tiện ích dễ thấy nhất khi sử dụng Lambda Expression là số lượng dòng code được giảm. Chúng ta có thể dễ dàng tạo ra một thể hiện của một functional interface bằng cách sử dụng Lambda Expression hơn là sử dụng một anonymous class.

- Hỗ trợ thực hiện tuần tự (sequential) và song song (parallel)

Một tiện ích khác của Lambda là việc hưởng lợi từ sự hỗ trợ của Stream API cho các tiến trình tuần tự và song song.

Để cho dễ hiểu, chúng ta cùng làm 1 ví dụ đơn giản, chúng ta sẽ viết một phương thức để kiểm tra xem số nhập vào có phải là số nguyên tố hay không.

Theo cách thông thường, ta sẽ viết code như sau, có thể đoạn code chưa tối ưu nhưng thể hiện được điều mình muốn làm:

```
private static boolean isPrime(int number) {
    if(number < 2) return false;
    for(int i=2; i<number; i++){
        if(number % i == 0) return false;
    }
    return true;
}
```

Vấn đề ở đoạn code trên là nó sẽ chạy tuần tự một cách tự nhiên và nếu chúng ta có 1 số cực lớn thì chúng ta sẽ mất nhiều thời gian. Ngoài ra trong đoạn code này có nhiều điểm trả về mà nó sẽ không chạy qua.

Bây giờ chúng ta sẽ xem Lambda giải quyết vấn đề này như thế nào.

```
private static boolean isPrime(int number) {
    return number > 1
        && IntStream.range(2, number).noneMatch(
            index -> number % index == 0);
}
```

IntStream là một chuỗi các phần tử có giá trị kiểu int hỗ trợ các tiến trình tổng hợp tuần tự và song song.

Để dễ đọc hơn, chúng ta cũng có thể viết theo cách sau:

```
private static boolean isPrime(int number) {
    IntPredicate isDivisible = index -> number % index == 0;
    return number > 1
            && IntStream.range(2, number).noneMatch(
                    isDivisible);
}
```

- Truyền hành động (behavior) vào phương thức

Bây giờ chúng ta sẽ truyền hành động vào một phương thức thông qua một ví dụ đơn giản.

```
public static int sumWithCondition(List numbers, Predicate predicate) {
    return numbers.parallelStream()
        .filter(predicate)
        .mapToInt(i -> i)
        .sum();
}
```

Cách sử dụng:

```
//Tổng các số
sumWithCondition(numbers, n -> true)
//Tổng các số chẵn
sumWithCondition(numbers, i -> i%2==0)
//Tổng các số lớn hơn 5
sumWithCondition(numbers, i -> i>5)
```

- Lười biếng nhưng hiệu quả

Một lợi thế của việc sử dụng Lambda là sự lười biếng. Ví dụ, ta sẽ viết một phương thức tìm ra số lẻ lớn nhất trong dãy từ 3 đến 11 và trả về bình phương của nó.

Thông thường, đoạn code sẽ được viết như sau:

```
private static int findSquareOfMaxOdd(List numbers) {
    int max = 0;
    for (int i : numbers) {
        if (i % 2 != 0 && i > 3 && i < 11 && i > max) {
            max = i;
        }
    }
    return max * max;
}
```

Chương trình trên sẽ luôn chạy tuần tự nhưng chúng ta có thể sử dụng Stream API để đạt được điều này với cách lười biếng nhất. Hãy xem cách chúng ta viết lại khi sử dụng Stream API và Lambda Expression.

```
public static int findSquareOfMaxOdd(List numbers) {
    return numbers.stream()
            .filter(NumberTest::isOdd)
```

```
            .filter(NumberTest::isGreaterThan3)
            .filter(NumberTest::isLessThan11)
            .max(Comparator.naturalOrder())
            .map(i -> i * i)
            .get();
}
public static boolean isOdd(int i) {
    return i % 2 != 0;
}
public static boolean isGreaterThan3(int i){
    return i > 3;
}
public static boolean isLessThan11(int i){
    return i < 11;
}
```

Java 8 - Functional Interface

Functional Interface là một interface có một phương thức abstract, nó cũng có thể được gọi là Single Abstract Interface (SAM) một cụm từ đôi khi chúng ta bắt gặp.

`@FunctionalInterface` annotation được thêm vào để chúng ta đánh dấu interface đó là functional interface, điều này không bắt buộc nhưng có thể là cách tốt nhất trong việc sử dụng functional interface để tránh vô tình thêm các phương thức khác. Nếu một interface đã có annotation `@FunctionalInterface` và chúng ta cố gắng thêm vào các phương thức khác thì trình biên dịch sẽ ném ra lỗi.

Lợi ích chính của functional interface là chúng ta có thể sử dụng Lambda Expression để tạo ra thể hiện (instance) cho interface đó.

Ví dụ:

```
@FunctionalInterface
public interface Functional {
    void method();
}
```

Một điều cần lưu ý: phương thức default and static không phá vỡ quy tắc của functional interface.

Ví dụ:

```
@FunctionalInterface
public interface FunctionalDefaultMethods {
    void method();

    default void defaultMethod() {
    }
}
```

Ví dụ: Trong ví dụ sau, chúng ta sẽ gán biểu thức lambda cho functional interface. Sau đó chúng ta có thể chạy biểu thức lambda bằng cách gọi phương thức được định nghĩa bên trong functional interface và truyền tham số vào.

```
public class Main {
    public static void main(String[] argv) {
        Processor stringProcessor = (String str) -> str.length();
```

```
    String name = "Java Lambda";
    int length = stringProcessor.getStringLength(name);
    System.out.println(length);
  }
}

@FunctionalInterface
interface Processor {
  int getStringLength(String str);
}
```

Đoạn code trên trả về kết quả như sau: 11

Java 8 - Default method & Static method

1. Giới thiệu

Một trong những thay đổi lớn nhất trong Java 8 là khái niệm về interface. Như chúng ta đã biết từ Java 7 trở về trước, interface chỉ cho phép chúng ta khai báo các phương thức bên trong nó. Nhưng trong Java 8 chúng ta sẽ có thêm 2 khái niệm mới đối với interface là phương thức default (default methods) và phương thức static (static methods).

Thiết kế interface luôn là một công việc rất khó khăn, bởi vì khi chúng ta thay đổi các phương thức bên trong interface nó đòi hỏi phải thay đổi tất cả các class được implements từ nó. Một khi số lượng các class được implements từ interface phát triển nhiều lên thì đến mức độ nào đó interface có thể không mở rộng được nữa. Đây là lý do tại sao khi thiết kế một ứng dụng, hầu hết các framework cung cấp một class cơ sở (base class), sau đó chúng ta sẽ mở rộng (extends) và ghi đè (override) lên các phương thức phù hợp với ứng dụng đang thực hiện.

2. Phương thức default

Để tạo một phương thức default trong interface, chúng ta sẽ sử dụng từ khóa "default".

Ví dụ: Interface1.java

```
private interface Defaulable {
    void method1(String str);
    // Interfaces now allow default methods, the implementer may or may not implement (override)
    them.
    default String notRequired() {
        return "Default implementation";
    }
}

private static class DefaultableImpl implements Defaulable {
    @Override
    public void method1(String str) {
    }
}
```

```
private static class OverridableImpl implements Defaulable {
    @Override
    public void method1(String str) {
    }

    @Override
    public String notRequired() {
        return "Overridden implementation";
    }
}
```

Giải thích: Chúng ta xây dựng một interface là Defaulable với hai phương thức:

- void method1
- default String notRequired
  Khi xây dựng các lớp implements tới lớp **Defaulable**, chúng ta có thể không cần thực hiện Override phương thức **notRequired** (lớp **DefaultableImpl**) hoặc Override phương thức **notRequired** (lớp **OverridableImpl**)

Những đặc điểm quan trọng về phương thức default trong interface:

1. Phương thức default giúp chúng ta mở rộng interface mà không phải lo ngại phá vỡ các class được implements từ nó.
2. Phương thức default giúp chúng ta tránh dùng các class tiện ích, ví dụ như tất cả phương thức của class Collections có thể được cung cấp ngay bên trong interface của nó
3. Phương thức default giúp chúng ta tháo gỡ các class cơ sở (base class), chúng ta có thể tạo phương thức default và trong class được implement có thể chọn phương thức để override
4. Một trong những lý do xuất hiện của phương thức default là để nâng cấp Collection API trong Java 8 hỗ trợ cho Lambda Expression.
5. Nếu bất kỳ class nào kế thừa những phương thức default giống nhau, thì nó sẽ không còn hiệu lực. Một điều tương tự, một phương thức default sẽ không thể override một phương thức từ java.lang.Object. Lý do rất đơn giản là bởi vì Object là base class của tất cả các class trong Java. Vì vậy nếu chúng ta có các phương thức của class Object được định nghĩa là phương thức default trong interface, nó sẽ không dùng được bởi vì các phương thức của Object luôn luôn được sử dụng. Đây lý do tại sao chúng ta sẽ không có bất cứ phương thức default nào override các phương thức của class Object.
6. Phương thức default cũng có thể được gọi là phương thức Defender (Defender Methods) hay là phương thức Virtual mở rộng (Virtual extension methods)

Chú ý:

- Phương thức default được thực hiện trên JVM mang lại hiệu quả và được hỗ trợ các chỉ dẫn byte code cho phương pháp gọi. Phương thức default cho phép các Java interface đã tồn tại phát triển thêm mà không gây lỗi trong quá trình biên dịch. Ví dụ như bổ sung các

phương thức vào interface **java.util.Collection: stream(), parallelStream(), forEach(), removeIf()**...

- Mặc dù vậy, phương thức default cũng cần được sử dụng một cách cẩn thận bởi nguyên nhân: Trước khi khai báo phương thức là default, cần chắc chắn là nó là cần thiết bị nó có thể gây ra sự nhập nhằng và biên dịch lỗi trong hệ thống phân cấp phức tạp.

3. Phương thức static

Phương thức static cũng giống như phương thức default ngoại trừ việc nó không thể được override chúng trong class được implements.

Một tính năng thú vị cung cấp bởi Java 8 là interface có thể khai báo (và cung cấp implementation) các phương thức tĩnh. Dưới đây là một số ví dụ.

Ví dụ 1:

```
private interface DefaulableFactory {
    // Interfaces now allow static methods
    static Defaulable create( Supplier< Defaulable > supplier ) {
        return supplier.get();
    }
}
```

Main class:

```
public static void main( String[] args ) {
    Defaulable defaulable = DefaulableFactory.create( DefaultableImpl::new );
    System.out.println( defaulable.notRequired() );

    defaulable = DefaulableFactory.create( OverridableImpl::new );
    System.out.println( defaulable.notRequired() );
}
```

Kết quả:

```
Default implementation
Overridden implementation
```

Ví dụ 2:

`MyData.java`

```
public interface MyData {
    default void print(String str) {
        if (!isNull(str))
            System.out.println("MyData Print::" + str);
    }
    static boolean isNull(String str) {
        System.out.println("Interface Null Check");
        return str == null ? true : "".equals(str) ? true : false;
    }
}
```

Bây giờ sẽ xem class được implements có phương thức isNull()

`MyDataImpl.java`

```
public class MyDataImpl implements MyData {

    public boolean isNull(String str) {
        System.out.println("Impl Null Check");
        return str == null ? true : false;
    }

    public static void main(String args[]){
        MyDataImpl obj = new MyDataImpl();
        obj.print("");
        obj.isNull("abc");
    }
}
```

Phương thức `isNull(String str)` là một phương thức đơn giản, nó không override phương thức của interface. Ví dụ nếu chúng ta thêm annotation `@Override` cho phương thức `isNull()`, trình biên dịch sẽ báo lỗi.

Bây giờ chúng ta sẽ chạy ứng dụng và xem kết quả:

```
Interface Null Check
Impl Null Check
```

Nếu chúng ta chuyển `static` thành `default`, thì kết quả như sau:

```
Impl Null Check
MyData Print::
```

```
Impl Null Check
```

Phương thức static chỉ hiển thị trong phương thức của interface, nếu chúng ta xóa phương thức isNull() trong class `MyDataImpl`, chúng ta sẽ không thể sử dụng nó cho đối tượng (object) của `MyDataImpl`. Tuy nhiên, giống như các phương thức static khác, chúng ta có thể sử dụng phương thức static của interface thông qua tên của class. Ví dụ sau đây là cách sử dụng hợp lệ:

```
boolean result = MyData.isNull("abc");
```

Những đặc điểm quan trọng về phương thức static trong interface:

1. Phương thức static là một thành phần của interface, chúng ta có thể sử dụng nó trong class được implements từ nó.
2. Phương thức static rất hữu ích trong việc cung cấp các phương thức tiện ích, ví dụ như là kiểm tra null, sắp xếp tập hợp ...
3. Phương thức static giúp chúng ta bảo mật, không cho phép class implements từ nó có thể override
4. Chúng ta không thể định nghĩa phương thức static của các phương thức thuộc class Object, chúng ta sẽ gặp lỗi "This static method cannot hide the instance method from Object". Điều này không cho phép trong Java, khi Object là base class cho tất cả các class và chúng ta không thể có một phương thức static và một phương thức khác cùng định dạng
5. Chúng ta có thể sử dụng phương thức static để bỏ đi những những phương thức dạng tiện ích như là Collections và làm cho tất cả các phương thức có thể liên lạc với interface, chúng ta sẽ dễ dàng tìm thấy và sử dụng những phương thức đó.

Java 8 - Method reference

Phương thức tham chiếu cung cấp các cú pháp (syntax) hữu ích để truy cập trực tiếp tới các phương thức hoặc hàm dựng đã tồn tại của các lớp hoặc đối tượng trong Java. Với sự kết hợp của Lambda expression, phương thức tham chiếu làm cho cấu trúc ngôn ngữ trông nhỏ gọn và súc tích.

Ví dụ: Car class với các định nghĩa phương thức khác nhau

```
public static class Car {

    public static Car create( final Supplier< Car > supplier ) {
        return supplier.get();
    }

    public static void collide( final Car car ) {
        System.out.println( "Collided " + car.toString() );
    }

    public void follow( final Car another ) {
        System.out.println( "Following the " + another.toString() );
    }

    public void repair() {
        System.out.println( "Repaired " + this.toString() );
    }
}
```

Giải thích:

1. Tham chiếu hàm dựng

Loại thứ nhất của phương thức tham chiếu là hàm dựng tham chiếu với cú pháp `Class::new` hoặc cách tổng quát, `Class<T>::new`

```
final Car car = Car.create(Car::new);
final List<Car> cars = Arrays.asList(car);
```

Chú ý rằng, hàm dựng này không có đối

2. Tham chiếu phương thức static

Loại thứ hai là tham chiếu tới phương thức static với cú pháp `Class::staticMethod`

```
cars.forEach(Car::collide);
```

Chú ý rằng, phương thức này chấp nhận chính xác tham số của Car

3. Tham chiếu phương thức instance của đối tượng tùy ý

Loại thức ba là tham chiếu tới phương thức instance của đối tượng tùy ý của loại cụ thể với cú pháp `Class::method`

```
cars.forEach(Car::repair);
```

Chú ý rằng, phương thức này có thể chấp nhận không có đối số

4. Tham chiếu phương thức instance của lớp instance

Loại thứ tư là tham chiếu tới phương thức instance của lớp instance cụ thể với cú pháp `instance::method`

```
final Car police = Car.create(Car::new);
cars.forEach(police::follow);
```

Chú ý rằng, phương thức này chấp nhận chính xác tham số của Car

Kết quả:

```
Collided vn.laptrinh.Car@36baf30c
Repaired vn.laptrinh.Car@36baf30c
Following the vn.laptrinh.Car@36baf30c
```

Java 8 - Repeating annotation

Annotations support từ khi được giới thiệu trong phiên bản Java 5, tính năng này đã trở thành một tính năng hữu ích và được sử dụng rộng rãi.

Tuy nhiên, hạn chế của việc sử dụng annotatioin này là các annotation có thể không được khai báo nhiều hơn một lần cùng một vị trí. Java 8 đã giới thiệu tính năng repeating annotation. Nó cho phép các annotation giống nhau có thể được khai báo nhiều lần cùng một vị trí.

Repeating annotation sử dụng chú thích `@Repeatable`.

Ví dụ:

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Repeatable;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

public class RepeatingAnnotations {
    @Target(ElementType.TYPE)
    @Retention(RetentionPolicy.RUNTIME)
    public @interface Filters {
        Filter[] value();
    }

    @Target(ElementType.TYPE)
    @Retention(RetentionPolicy.RUNTIME)
    @Repeatable(Filters.class)
    public @interface Filter {
        String value();
    };

    @Filter("filter1")
    @Filter("filter2")
    public interface Filterable {
```

```
    }

    public static void main(String[] args) {
        for (Filter filter : Filterable.class
                .getAnnotationsByType(Filter.class)) {
            System.out.println(filter.value());
        }
    }
}
```

Giải thích: Đây là lớp sử dụng Filter annotation với `@Repeatable(Filters.class)`. Lớp Filters chỉ nắm giữ Filter annotation nhưng trình biên dịch Java compiler cố gắng để ch giấu sự hiện diện của nó từ developer. Như vậy Filterable interface có chú thích Filter được định nghĩa hai lần (không đề cập tới Filters).

Ngoài ra, Reflection API cũng cung cấp phương thức mới là `getAnnotationsByType()` để trả về các repeating annotation. Chú ý rằng `Filterable.class.getAnnotation(Filters.class)` sẽ trả về instance của Filters bởi trình biên dịch.

Kết quả khi chạy chương trình:

```
filter1
filter2
```