

Lời nói đầu

Giáo trình này được biên soạn dùng để hỗ trợ giảng dạy cho môn Lập trình Hướng đối tượng với ngôn ngữ Java.

- Chúng tôi giả định bạn đọc đã biết môn Ngôn ngữ lập trình C, nên không thảo luận chi tiết cú pháp của ngôn ngữ Java, mà chỉ tập trung vào những điểm khác biệt giữa Java và C/C++ giúp bạn đọc tiếp cận Java nhanh chóng. Nếu bạn đọc chưa biết ngôn ngữ lập trình C, xin xem trước tài liệu Ngôn ngữ lập trình C, cùng người viết. Cũng vì lý do trên, cách tiếp cận Java chúng tôi chọn là sớm làm việc với đối tượng (early object).

- Chúng tôi cố gắng sắp xếp trình bày lập trình Hướng đối tượng theo cách dẫn dắt từ khái niệm đến code. Một cách trình bày khác về lập trình Hướng đối tượng, mang tính kinh điển hơn, xin bạn đọc tham khảo tài liệu Lập trình Hướng đối tượng với ngôn ngữ C++, cùng tác giả.

- Tài liệu cũng đề cập một số vấn đề nâng cao như truy cập dữ liệu bằng JDBC, lập trình mạng bằng Java, ... Chúng tôi cũng chú ý cập nhật những khái niệm của các phiên bản Java gần nhất, đến Java SE 8.

Tôi xin tri ân đến các bà đã nuôi dạy tôi, các thầy cô đã tận tâm chỉ dạy tôi. Chỉ có cống hiến hết mình cho tri thức tôi mới thấy mình đền đáp được công ơn đó.

Tôi xin cảm ơn gia đình đã hy sinh rất nhiều để tôi có được khoảng thời gian cần thiết thực hiện được giáo trình này.

Mặc dù đã dành rất nhiều thời gian và công sức, hiệu chỉnh chi tiết và nhiều lần, nhưng giáo trình không thể nào tránh được những sai sót và hạn chế. Tôi thật sự mong nhận được các ý kiến góp ý từ bạn đọc để giáo trình có thể hoàn thiện hơn. Nội dung giáo trình sẽ được cập nhật định kỳ, mở rộng và viết chi tiết hơn; tùy thuộc những phản hồi, những đề nghị, những ý kiến đóng góp từ bạn đọc.

Các bạn đồng nghiệp nếu có sử dụng giáo trình này, xin gửi cho tôi ý kiến đóng góp phản hồi, giúp giáo trình được hoàn thiện thêm, phục vụ cho công tác giảng dạy chung.

Phiên bản

Cập nhật ngày: **03/07/2017**

Thông tin liên lạc

Mọi ý kiến và câu hỏi có liên quan xin vui lòng gửi về:

Dương Thiên Tứ

91/29 Trần Tấn, P. Tân Sơn Nhì, Q. Tân Phú, Thành phố Hồ Chí Minh

Facebook: <https://www.facebook.com/tu.duongthien>

E-mail: thientu2000@yahoo.com

Trung tâm: CODESCHOOL – <http://www.codeschool.vn>

Fanpage: <https://www.facebook.com/codeschool.vn>

Java – Cài đặt

Chú ý khi xem hướng dẫn:

- Phân biệt chữ hoa chữ thường.
 - Đường dẫn cài đặt nên tránh có ký tự space. Ví dụ bạn nên tránh cài đặt theo thói quen vào thư mục **Program Files**. Nói chung nên cài đặt tương tự đề nghị trong hướng dẫn này.
 - Các project của NetBeans do bạn thực hiện, nên lưu trên đường dẫn cho phép truy cập dễ dàng, vì vậy không nên dài quá. Ví dụ: **D:\training\JavaSE\DatabaseProject**.
- Không được lưu các project vào thư mục mà đường dẫn có ký tự Unicode, ví dụ: thư mục **D:\thực hành Java**.

JDK

Bạn cài đặt JDK (<http://www.oracle.com/technetwork/java/javase/downloads>) vào thư mục đề nghị là **C:\java\jdk**.

Tạo mới biến môi trường **JAVA_HOME** có trị **C:\java\jdk**. Hầu hết các IDE cho Java sẽ tham chiếu biến môi trường **JAVA_HOME** để IDE điều hướng đến trình biên dịch Java.

Bạn có thể cài đặt cùng lúc nhiều phiên bản JDK, điều chỉnh biến môi trường **JAVA_HOME** hoặc điều chỉnh cấu hình JDK của IDE mỗi khi muốn sử dụng phiên bản JDK thích hợp.

Thêm vào cuối biến môi trường **PATH** có sẵn, đường dẫn mới **%JAVA_HOME%\bin**

Để tạo biến môi trường:

- Click phải vào **My Computer** hoặc **This PC**, chọn **Properties**. Chọn **Advanced system settings** ở cột bên trái.
- Chọn tab **Advanced**, nhấn nút **Environment Variables...**
- + Nhấn nút **New...** trong phần **User variables for...** để tạo mới biến môi trường **JAVA_HOME**.
- + Chọn biến môi trường **Path** trong phần **System variables**. Nhập đường dẫn mới vào. Chú ý, các đường dẫn trong biến môi trường **PATH** cách nhau bởi dấu (;).

Kiểm tra bằng các lệnh (mở console bằng **Windows+R** rồi nhập **cmd**):

```
echo %JAVA_HOME% (phải thấy trị của biến môi trường)
java -version (phải thấy kết quả chạy lệnh)
```

Tài liệu Java API không được kèm theo gói JDK tải về mà phải lấy từ một link khác:

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

Sau khi tải về gói tài liệu tương ứng với phiên bản JDK sử dụng, giải nén thư mục **docs** trong gói nén vào thư mục cài đặt, ví dụ: **C:\java\jdk\docs**. Đây là nguồn tài liệu tham chiếu về API quan trọng cho người phát triển Java.

NetBeans

Tải về NetBeans bản đầy đủ, mới nhất.

Khi cài đặt, trong bước **Customize Installation**, chỉ cần chọn:

Nếu chỉ dùng JavaSE: Base IDE, JavaSE

Nếu dùng cả JavaEE: JavaEE, HTML5, Apache Tomcat (chắc chắn nhớ chọn Tomcat, chú ý là không cần chọn Glassfish).

Khi wizard yêu cầu chỉ định các thư mục cài đặt, chọn:

JDK đã cài đặt trong bước trên (chọn một phiên bản, nếu cài nhiều phiên bản JDK).

NetBeans: **C:\java\NetBeans**

Tomcat: **C:\java\Tomcat**

Xem và thực hành "Phụ lục 1 – NetBeans" để nâng cao khả năng sử dụng NetBeans. Các bạn chọn IDE Eclipse, xem và thực hành "Phụ lục 2 – Eclipse".

Nếu bạn phát triển trên JavaEE:

- Đặt driver JDBC type 4 của Microsoft SQL Server (sqljdbc4.jar) vào thư mục **C:\java\Tomcat\lib**. Nếu không, thêm gói jar này vào các project Web có truy cập cơ sở dữ liệu từ Microsoft SQL Server.

- Tạo một project Web (chỉ cần vài bước theo wizard rồi thoát) để kích hoạt Java Web và Java EE.

Kiểm tra: Chọn **Tools > Servers**, phải thấy Tomcat trong danh sách server.

Để thấy menu **Tools > Servers**, phải kích hoạt Java EE trong NetBeans bằng cách tạo mới một project Java EE, sau đó thoát.

Java – Ngôn ngữ

Kiểu dữ liệu

Trị được định nghĩa như một thứ có thể đặt vào nó một biến hoặc có thể truyền nó đến một phương thức. Kiểu dữ liệu của trị chia thành hai nhóm: kiểu tham chiếu (reference) và kiểu cơ bản (primitive). Kiểu cơ bản là các trị đơn giản, không phải là đối tượng. Kiểu tham chiếu được dùng cho các kiểu phức tạp, bao gồm các kiểu do bạn định nghĩa. Kiểu tham chiếu được dùng để tạo các đối tượng.

1. Kiểu cơ bản

Java là ngôn ngữ định kiểu mạnh, Java định nghĩa 8 kiểu dữ liệu cơ bản, gồm 4 loại:

- Logic – boolean

Các trị logic được thể hiện bằng cách dùng kiểu boolean, nhận một trong hai trị: true hoặc false. Các trị này được dùng thể hiện hai trạng thái của một biến, như on/off hoặc yes/no.

Khác với C/C++, trong Java, bạn không thể ép kiểu int thành kiểu boolean.

- Văn bản – char (2)

Các ký tự đơn được thể hiện bằng cách dùng kiểu char. Một char thể hiện một ký tự Unicode không dấu 16-bit, bao bởi cặp nháy đơn (' '). Ví dụ: 'a', '\t', '\u03A6' (u là Unicode, ký tự Ø)

Bạn dùng kiểu String, không phải kiểu cơ bản mà là một lớp, để thể hiện chuỗi ký tự. Các ký tự trong chuỗi có thể là Unicode hoặc chuỗi ký tự escape với ký tự \ ngay trước. Một chuỗi được bao bởi cặp nháy kép (" ").

Khác với C/C++, chuỗi trong Java không kết thúc với ký tự \0.

- Nguyên – byte (1), short (2), int (4) và long (8)

Ngoài dạng thập phân, bạn có thể thể hiện kiểu nguyên dưới dạng nhị phân, bát phân hoặc thập lục phân như sau:

0b1010 Trị nhị phân của 10, bắt đầu bằng 0b.

077 Trị bát phân của 63, bắt đầu bằng 0.

0xBAAC Trị thập lục phân của 47788, bắt đầu bằng 0x.

Mọi kiểu số trong Java đều thể hiện số có dấu. Chuỗi số nguyên ngầm định là kiểu int, ngoại trừ có ký tự L (hoặc l) theo sau, chỉ định kiểu long. Ví dụ: 2L, 077L, 0xBAACL

Từ Java 7, cho phép dùng ký tự _ để phân cách cụm hàng ngàn trong chuỗi số nguyên. Ví dụ: int intValue = 3_000_000;

Điều này cũng áp dụng để phân cách cụm nibble cho byte. Ví dụ, char charInByte = 0B0010_0001; // ký tự !

- Dấu chấm động – float (4) và double (8).

Một chuỗi số có dấu chấm động nếu nó chứa: hoặc một dấu chấm thập phân, hoặc một phần lũy thừa (exponent part, ký tự E hoặc e), hoặc theo sau bởi ký tự F hoặc f (với float), D hoặc d (với double). Chuỗi số có dấu chấm động có cú pháp:

[whole-number].[fractional_part][e|E exp][f|F|d|D]

Ví dụ: .49F, 3.14, 6.02E23, 2.718F, 123.4E+306D. Nếu sau E không có dấu, mặc định là có dấu +.

Chuỗi số có dấu chấm động ngầm định là kiểu double.

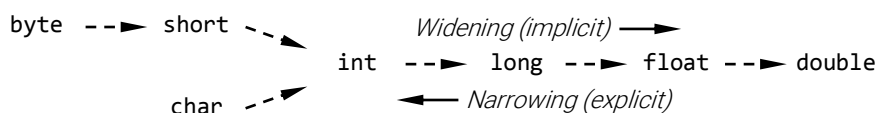
Một số trị đặc biệt được dùng như các thực thể dấu chấm động (floating-point entity):

Entity	Mô tả	Ví dụ
Infinity	Thể hiện khái niệm vô cực dương.	1.0 / 0.0, 1e300 / 1e-300, Math.abs(-1.0 / 0.0)
-Infinity	Thể hiện khái niệm vô cực âm.	-1.0 / 0.0, 1.0 / (-0.0), 1e300 / -1e-300
-0.0	Thể hiện một số âm gần 0.	-1.0 / (1.0 / 0.0), -1e-300 / 1e300
NaN	Thể hiện kết quả không xác định.	0.0 / 0.0, 1e300 * Float.NaN, Math.sqrt(-9.0)

Ngoại trừ -0.0, các thực thể trên cũng có các hằng tương đương trong lớp Float và Double.

Khi tính toán chính xác với số thực, Oracle khuyên nên dùng lớp BigDecimal thay thế cho float và double. Bạn không thể dùng các toán tử số học với BigDecimal, thay vào đó, gọi các phương thức tương ứng như add, subtract, multiply.

Với kiểu dữ liệu cơ bản, trị của kiểu dữ liệu "hẹp" có thể chuyển đổi thành trị của kiểu dữ liệu "rộng" hơn. Việc chuyển kiểu thường ngầm định, thực hiện tự động, gọi là chuyển kiểu mở rộng (widening) hay nâng cấp (promotion) kiểu. Khả năng chuyển kiểu mô tả trong sơ đồ sau:



Chuyển đổi từ một kiểu "rộng" thành một kiểu "hẹp", không theo sơ đồ trên, gọi là chuyển kiểu thu hẹp (narrowing). Kết quả có thể dẫn đến mất thông tin do kiểu gốc bị xén bớt. Việc chuyển kiểu thu hẹp phải dùng ép kiểu tường minh (cast).

Chuyển kiểu giữa char với byte và short cũng thuộc loại này: trước tiên, kiểu nguồn chuyển thành int; sau đó, kiểu int tiếp tục chuyển thành kiểu đích.

2. Kiểu tham chiếu

Một *trị tham chiếu* được trả về khi một đối tượng được tạo, một trị tham chiếu chỉ đến một đối tượng cụ thể lưu trữ trong heap. Biến lưu trị tham chiếu gọi là tham chiếu (reference). Một tham chiếu cung cấp một mục quản (handle), cho phép thông qua nó có thể truy cập *gián tiếp* đến đối tượng. Chú ý rằng trong Java, *nội dung* của một đối tượng không thể đặt vào một biến, chỉ có *trị tham chiếu* của đối tượng mới được đặt vào biến.

Java có các kiểu tham chiếu sau: annotation, array, class, enumeration và interface.

Ví dụ sau cho thấy tham chiếu chỉ đến một đối tượng cấp phát trong heap.

```
class MyDate {
    private int day = 1;
    private int month = 1;
    private int year = 2000;
```

```

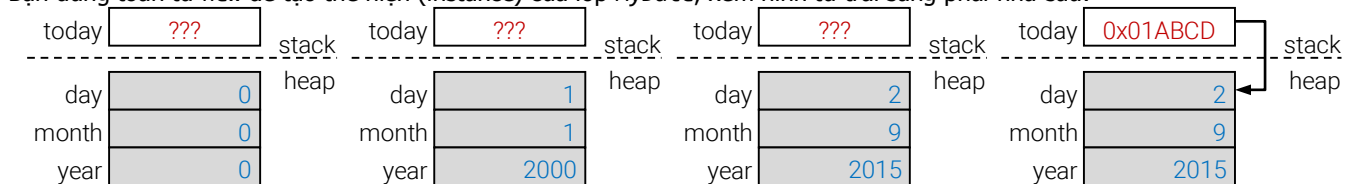
public MyDate(int day, int month, int year) {
    this.day = day;
    this.month = month;
    this.year = year;
}

@Override public String toString() {
    return String.format("%02d-%02d-%04d", day, month, year);
}
}

public class Test {
    public static void main(String[] args) {
        MyDate today = new MyDate(2, 9, 2015);
    }
}

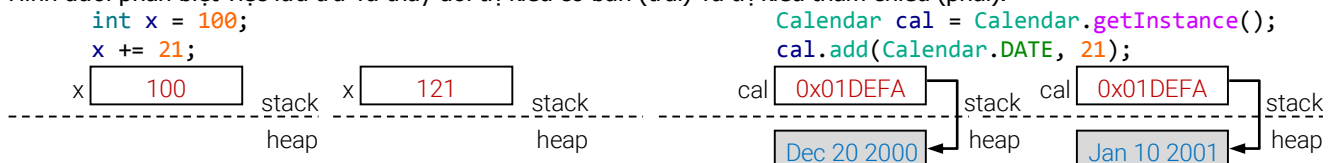
```

Bạn dùng toán tử new để tạo thể hiện (instance) của lớp MyDate, xem hình từ trái sang phải như sau:



- (1) Biến tham chiếu today được cấp phát trong stack. Vùng nhớ cho đối tượng mới được cấp phát trong heap, khởi tạo các thuộc tính là 0 hoặc null.
- (2) Khởi tạo tường minh các thuộc tính của today nếu chúng được khai báo với trị mặc định.
- (3) Constructor được gọi sẽ thực thi, các thuộc tính sẽ được khởi tạo từ tham số truyền đến constructor.
- (4) Trị trả về từ toán tử new tham chiếu đến đối tượng mới tạo trong heap, tham chiếu được lưu trong biến tham chiếu today.

Hình dưới phân biệt việc lưu trữ và thay đổi trị kiểu cơ bản (trái) và trị kiểu tham chiếu (phải).



Đối tượng trong heap không chứa các đối tượng khác, chúng chỉ chứa tham chiếu đến các đối tượng khác.

Một đối tượng có thể được chỉ đến bởi nhiều tham chiếu. Nói cách khác, các tham chiếu này lưu trữ trị tham chiếu của cùng một đối tượng. Các tham chiếu này được xem như là các alias (bí danh) của cùng một đối tượng, một đối tượng có thể được truy cập gián tiếp bởi một trong các alias của nó.

Chuyển kiểu mở rộng trên cây thừa kế gọi là *upcasting* (ép kiểu lên), thường thực hiện ngầm định, sẽ nâng cấp kiểu con (subtype) thành kiểu cha (supertype) của nó. Chuyển kiểu thu hẹp trên cây thừa kế gọi là *downcasting* (ép kiểu xuống), thường phải dùng toán tử ép kiểu, tuy nhiên vẫn có khả năng ném ra runtime exception, nên dùng toán tử instanceof để kiểm tra tương thích kiểu trước khi ép kiểu.

```
Object strObj = "upcast me"; // Widening hay upcasting: Object <---- String
```

```
Object intObj = new Integer(10);
```

```
String s1 = (String) strObj; // Narrowing hay downcasting: String <---- Object
```

```
String s2 = (String) intObj; // ném java.lang.ClassCastException, int không ép kiểu thành String được
```

Java tự động quản lý bộ nhớ bằng Garbage Collection (GC), GC có các tác vụ chính: cấp phát vùng nhớ, quản lý các đối tượng được tham chiếu, thu hồi vùng nhớ được cấp phát cho đối tượng khi không còn tham chiếu chỉ đến đối tượng.

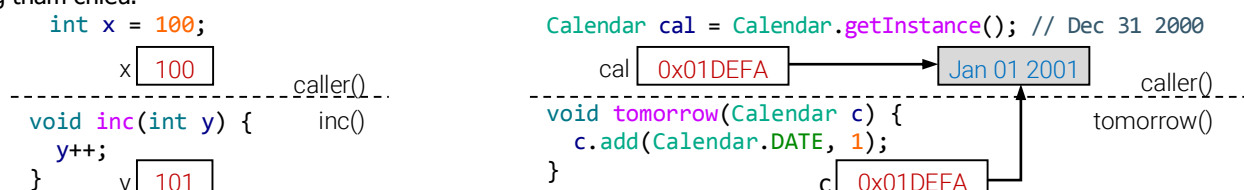
3. Truyền bằng tham chiếu

Khi nói Java xử lý các đối tượng *thông qua tham chiếu* (by reference), bạn cần tránh nhầm lẫn với khái niệm *truyền bằng tham chiếu* (pass by reference). Trong các ngôn ngữ truyền bằng tham chiếu, một trị không được truyền trực tiếp đến phương thức mà *tham chiếu đến trị* đó được truyền. Do đó, nếu phương thức thay đổi các tham số truyền cho nó, thông qua tham chiếu nó có thể thay đổi gián tiếp trị đó; những thay đổi này sẽ thể hiện sau khi gọi phương thức.

Java là ngôn ngữ *truyền bằng trị* (pass by value), khi một tham chiếu được truyền, *trị của tham chiếu* được truyền (pass by value), không phải *tham chiếu của tham chiếu* được truyền (pass by reference).

Khi truyền một đối tượng như tham số đến một phương thức, trị được truyền không phải là đối tượng mà là tham chiếu đến đối tượng. Bạn có thể thay đổi *nội dung* của đối tượng trong phương thức gọi (caller) thông qua tham chiếu, nhưng không thay đổi được tham chiếu đến đối tượng đó (do tham chiếu truyền bằng trị).

Hình dưới phân biệt truyền bằng trị một trị cơ bản (trái) và truyền bằng trị một trị tham chiếu (phải), bạn phân biệt với truyền bằng tham chiếu.



4. Annotation

Annotation cung cấp một cách để liên kết meta-data (dữ liệu về dữ liệu, chú thích) với các thành phần của chương trình Java tại thời gian dịch và thời gian chạy. Annotation có thể áp dụng cho package, class, method, constructor, tham chiếu và biến. Annotation có các loại:

- Annotation tạo sẵn

+ Annotation được dùng như meta-data (chú thích), trong gói java.lang:

@Override đánh dấu một phương thức nạp chồng.
 @Deprecated đánh dấu một phương thức đã lạc hậu. Phải có chú thích kèm theo để giải thích vì sao phương thức lạc hậu và cách thay thế có thể.

@SuppressWarnings báo trình biên dịch ngăn chặn cảnh báo, có thể chỉ rõ loại cảnh báo, deprecated hoặc unchecked.

```
public interface House {
    /**
     * @deprecated use of open is discouraged,
     * use openDoor instead.
     */
    @Deprecated
    public void open();
    public void openDoor();
}
```

```
public class MyHouse implements House {
    @SuppressWarnings("deprecation")
    @Override public void open() { }
    @Override public void openDoor() { }
}
```

@SafeVarargs khẳng định phương thức không thực hiện các hoạt động có thể gây mất an toàn trên tham số của nó.

@FunctionalInterface đánh dấu một interface dự kiến dùng cho biểu thức Lambda (interface chỉ có một phương thức).

+ Annotation được dùng như meta-annotation (chú thích dùng cho chú thích), trong gói java.lang.annotation:

@Retention chỉ định cách một annotation kiểu marker được giữ lại.
 @Documented chỉ định một annotation sẽ có mặt trong Javadoc (mặc định annotation không có trong Javadoc).
 @Target chú thích một annotation là hạn chế áp dụng với thành phần Java nào đó.
 @Inherited chú thích một annotation được thừa kế từ lớp cha nào đó.
 @Repeatable chỉ định một annotation kiểu marker được áp dụng nhiều lần trong cùng một khai báo.

- Annotation định nghĩa bởi người phát triển, gọi là annotation tùy biến, có ba kiểu: marker (không tham số), single value (một tham số) và multi value (nhiều tham số).

```
@Retention(RetentionPolicy.RUNTIM) // annotation được JVM giữ lại để dùng trong môi trường chạy
public @interface Feedback { } // marker
public @interface Feedback { // single value
    String reportName();
}
public @interface Feedback { // multi value
    String reportName();
    String comment() default "None"; // thiết lập trị mặc định
}
```

Dùng annotation tùy biến như sau:

+ Đặt trực tiếp trước mục được áp dụng annotation:

```
@Feedback(reportName="Report 1")
public void myMethod() { ... }
```

+ Chương trình có thể kiểm tra sự tồn tại của annotation:

```
Feedback fb = myMethod.getAnnotation(Feedback.class);
```

- Annotation do bên thứ ba cung cấp, thường là các annotation chú thích kiểu, ví dụ @NotNull.

5. Mảng

Một mảng (array) là một cấu trúc dữ liệu định nghĩa một collection được đánh chỉ số của một tập các phần tử có kiểu dữ liệu giống nhau. Các phần tử của mảng được truy cập bằng cách dùng một trị không âm gọi là chỉ số (index, subscript), tính từ 0. Trong Java, mảng là *đối tượng*, nó có thể chứa các phần tử có kiểu dữ liệu cơ bản hoặc kiểu dữ liệu tham chiếu. Số phần tử của mảng chứa trong một trường final có tên length.

Có hai cách khai báo mảng: <element type>[] <array name>; hoặc <element type> <array name>[];

Chú ý cách khai báo mảng thứ hai tương tự như trong C/C++ nhưng *không chỉ định số phần tử*.

Khai báo như trên *không thực sự tạo mảng* mà chỉ khai báo tham chiếu chỉ đến một đối tượng mảng. Một mảng có thể được khởi tạo với số phần tử chỉ định và kiểu dữ liệu chỉ định cho các phần tử, bằng toán tử new như sau:

```
<array name> = new <element type> [<array size>];
```

Nếu số phần tử khởi tạo là số âm, exception NegativeArraySizeException sẽ được ném ra.

Khi khởi tạo mảng, các phần tử của nó có thể được khởi gán bằng khối khởi tạo.

```
int[] array1 = new int[5]; // tạo và cấp phát mảng có 5 phần tử kiểu int
int[] array2 = { 3, 5, 2, 8, 6 }; // tạo mảng và khởi gán trị cho các phần tử, dùng khối khởi tạo
```

```
int[] array3 = new int[] { 3, 5, 2, 8, 6 }; // tạo mảng vô danh, khởi gán các phần tử, rồi gán cho array3
```

```
Game[] gameList; // khai báo mảng và khởi tạo mặc định là null
gameList = new Game[10]; // cấp phát 10 phần tử cho mảng, tuy nhiên mảng vẫn rỗng
gameList[0] = new Game(); // khởi tạo cho phần tử có chỉ số 0
```

Các phần tử của mảng có thể có kiểu tham chiếu là mảng khác, khi đó ta có mảng của các mảng, thường quen gọi là mảng nhiều chiều (multidimensional array).

```
final int MAX_ROWS = 5;
final int MAX_COLS = 3;
int[][] matrix = new int[MAX_ROWS][MAX_COLS];
```

```
for (int i = 0; i < MAX_ROWS; ++i, System.out.println())
    for (int j = 0; j < MAX_COLS; ++j)
        System.out.printf("%5d", matrix[i][j] = (int) (Math.random() * 101) - 50);
```

Java cũng cho phép tạo mảng không chữ nhật (nonrectangular array), là mảng mà các hàng có thể có số cột khác nhau, còn gọi là mảng răng cưa (jagged).

```
int[][] jaggedArray = new int[5][];
jaggedArray[0] = new int[4];
jaggedArray[1] = new int[3];
jaggedArray[2] = new int[5];
jaggedArray[3] = new int[2];
jaggedArray[4] = new int[4];
for (int i = 0; i < 5; ++i, System.out.println())
    for (int j = 0; j < jaggedArray[i].length; ++j)
        System.out.printf("%5d", jaggedArray[i][j] = (int) (Math.random() * 101) - 50);
```

hoặc khởi tạo dùng cú pháp tạo mảng (array literal):

```
int[][] jaggedArray = {
    { 5, 4 },
    { 10, 15, 12, 15, 18 },
    { 6, 9, 10 },
    { 12, 5, 8, 11 }
};
```

Để tạo bản sao của một mảng cho trước, dùng phương thức `clone()` của mảng đó.

```
float[] floatArray = {5.0f, 3.0f, 2.0f, 1.5f};
float[] floatArrayCopy = floatArray.clone();
System.out.println(Arrays.toString(floatArrayCopy));
```

Biến

1. Biến và tầm vực (scope)

Tên biến là một định danh (identifier), định danh cũng dùng cho tên lớp, tên phương thức và nhãn (label). Định danh trong Java là một chuỗi ký tự và số phân biệt chữ hoa chữ thường, *ký tự đầu không được là số*. Các ký tự như underscore (`_`) hoặc ký tự currency (các ký tự tiền tệ như \$, €, ¥, hoặc £) là hợp lệ nhưng nên tránh dùng. Thực tế, không hạn chế số ký tự của định danh.

Các biến được khai báo trong các ngữ cảnh khác nhau:

- Khai báo bên trong một phương thức, gọi là biến cục bộ (local); còn gọi là biến *automatic*, hoặc *temporary*, *stack*. Bạn phải khởi tạo biến cục bộ một cách tường minh trước khi dùng chúng.

Tham số của các phương thức (kể cả constructor) cũng là các biến cục bộ và chúng được khởi tạo khi gọi phương thức. Vì tham số cũng là biến cục bộ nên trong thân phương thức bạn không được khai báo lại tham số (khai báo biến trùng tên với tham số).

Biến cục bộ là tham số của phương thức, được tạo khi gọi phương thức và tồn tại cho đến khi phương thức kết thúc.

Biến cục bộ được tạo khi điều khiển của chương trình chuyển đến phương thức và bị hủy khi phương thức kết thúc. Các biến này là cục bộ trong hàm thành viên, nên bạn có thể dùng tên biến giống nhau cho các hàm thành viên khác nhau. Tuy nhiên, dùng tên biến cục bộ giống tên biến thành viên của lớp, thường gọi là "che" biến lớp, không được khuyến khích.

- Khai báo bên ngoài phương thức nhưng bên trong một định nghĩa lớp. Chúng được khởi tạo tự động khi bạn tạo đối tượng bằng toán tử `new`. Có hai kiểu có thể:

- + `static`: còn gọi là biến cấp lớp (class variable), được tạo khi lớp nạp vào JVM, mặc dù lớp chưa tạo đối tượng nào, và tiếp tục tồn tại khi lớp còn tồn tại trong JVM. Biến `static` được truy cập và dùng chung cho tất cả các đối tượng thuộc lớp.
- + `instance`: biến thể hiện (instance variable) tồn tại khi đối tượng còn tồn tại. Biến thể hiện còn được gọi là biến thành viên (member variable) do chúng là thành viên của đối tượng tạo ra từ lớp. Trị của các biến này tại một thời điểm bất kỳ tạo thành trạng thái (state) của đối tượng.

2. Khởi tạo biến

Với các biến trong stack, trình biên dịch sẽ kiểm tra chúng được khởi tạo trước khi sử dụng. Tham chiếu `this` và các tham số của phương thức phải được gán trị khi phương thức bắt đầu thực thi. Vì vậy, các biến cục bộ phải được khởi tạo tường minh trước khi dùng.

Với các biến trong heap, do trình biên dịch khó kiểm tra việc khởi tạo, nên chúng được gán trị mặc định: `0` cho các kiểu số, `false` cho boolean và `null` cho kiểu tham chiếu.

Với mảng không được khởi gán trước, bạn phải khởi tạo các phần tử của mảng bằng toán tử `new`. Do các phần tử của mảng được tạo trong heap nên chúng sẽ được khởi tạo với trị mặc định. Ví dụ, các phần tử của mảng `array1` bên dưới sẽ chứa trị `0`.

```
public class InitArray {
    public static void main(String[] args) {
```



```

int[] array1 = new int[10];
int[] array2 = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 3 };
System.out.printf("%s%s%n", "Index", "Value");
for (int i = 0; i < array1.length; ++i)
    System.out.printf("%5d%8d%n", i, array1[i]);
System.out.println("-----");
for (int i = 0; i < array2.length; ++i)
    System.out.printf("%5d%8d%n", i, array2[i]);
System.arraycopy(array2, 0, array1, 0, array2.length);
}
}

```

Toán tử và cấu trúc điều khiển

1. Toán tử

Đa số các toán tử của Java thừa kế từ ngôn ngữ C/C++, tuy nhiên có một số khác biệt bạn cần chú ý.

- Toán tử số học

Khi làm việc với các biểu thức số học, Java tự động chuyển kiểu byte hoặc short thành int và float thành double.

```

byte a = 8;
byte b = 8;
byte c = a * b;           // Lỗi, do a * b chuyển kiểu không tường minh thành int.
int c = a * b;           // Hợp lệ, c = 10000.
byte c = (byte) (a * b);  // Hợp lệ, c = 16.

```

- Toán tử logic và toán tử điều kiện

Toán tử logic còn gọi là toán tử boolean vì trả về trị boolean. Bao gồm các toán tử !, &, ^, và |; tương ứng với các tác vụ luận lý NOT, AND, XOR và OR. Chúng thực hiện với với toán hạng kiểu boolean hoặc Boolean.

Các toán tử điều kiện, bao gồm các toán tử && và ||, cũng ngăn mạch tương tự như toán tử & và |.

Ngắn mạch (short-circuit) một biểu thức logic là đặc điểm thừa kế từ C/C++.

+ Khi toán tử && được dùng, nếu biểu thức bên trái định trị false thì không cần định trị tiếp biểu thức.

+ Khi toán tử || được dùng, nếu biểu thức bên trái định trị true thì không cần định trị biểu thức tiếp.

Khác với C/C++, trong Java, trị 0 không được dịch tự động thành false và trị khác 0 không được dịch tự động thành true.

- Toán tử logic trên bit (bitwise)

Các tác vụ thao tác trên bit, thực hiện các tác vụ cấp thấp. Các toán tử bitwise thực hiện với toán hạng kiểu nguyên, bao gồm ~, &, ^ và |; tương đương các tác vụ bitwise NOT (bù), bitwise AND, bitwise XOR và bitwise OR.

- Toán tử dịch

Java cung cấp hai toán tử dịch phải:

+ toán tử >> dịch phải số học hay dịch phải có dấu, dịch phải k bit tương đương với phép chia toán hạng thứ nhất cho 2^k .

Khi dịch số âm, bit dấu (bit trái nhất, 1 cho số âm) của toán hạng thứ nhất sẽ được giữ lại.

+ toán tử >>> dịch phải logic hay dịch phải không dấu, luôn đặt 0 vào bit trái nhất khi dịch.

Java cung cấp một toán tử dịch trái <<, dịch k bit tương đương với phép nhân toán hạng thứ nhất cho 2^k .

Toán tử dịch chỉ làm việc trên kiểu nguyên, toán tử >>> chỉ làm việc với kiểu int và long.

- Toán tử nối chuỗi

Toán tử nối chuỗi + thực hiện nối các đối tượng String, sinh ra một đối tượng String mới. Nếu một trong các toán hạng của toán tử nối chuỗi + là một đối tượng String thì các toán hạng khác tự động chuyển thành String.

- Toán tử so sánh

Hai tham chiếu bằng nhau khi chúng cùng chỉ đến một đối tượng.

```

String s1 = "java";
String s2 = new String("java"); // (s1 == s2) định trị false
String s3 = "java";           // (s1 == s3) định trị true

```

Để tránh gây nhầm lẫn, so sánh chuỗi luôn thực hiện bằng cách dùng phương thức equals() của lớp String.

- Toán tử -> dùng trong định nghĩa của biểu thức Lambda, từ Java 8, không phải toán tử truy cập thành viên từ pointer như C/C++.

2. Cấu trúc điều khiển

Các cấu trúc điều khiển của Java cũng thừa kế từ C/C++, tuy nhiên có một số khác biệt bạn cần chú ý.

- Phát biểu switch hỗ trợ kiểu int, byte, short và char. Từ Java 7, switch còn hỗ trợ kiểu enum và String.

- Bạn dùng các phát biểu sau để điều khiển phát biểu lặp, chú ý break và continue không bắt buộc phải dùng nhãn.

+ break [<label>]; thoát nhanh khỏi phát biểu switch, phát biểu lặp hoặc khối được đặt nhãn.

```

outer:
do {
    statement1;
    do {
        statement2;
        if (condition) {
            break outer;
        }
        statement3;
    } while (test_expresion1);
    statement4;
} while (test_expresion2);

```

+ continue [<label>]; bỏ qua các phát biểu kế tiếp, nhảy đến cuối thân vòng lặp và trả lại điều khiển cho phát biểu lặp.

```

test:
do {
    statement1;
    do {
        statement2;
        if (condition) {
            continue test;
        }
        statement3;
    } while (test_expression1);
    statement4;
} while (test_expression2);

```

+ <Label> : <statement> xác định một phát biểu hợp lệ để chuyển điều khiển đến đó. Với phát biểu break có dùng nhãn, nhãn được đặt cho một phát biểu bất kỳ. Với phát biểu continue có dùng nhãn, nhãn phải được đặt cho một khối lặp.

Exception và Assertion

1. Exception

Exception (ngoại lệ) là cơ chế được dùng bởi nhiều ngôn ngữ lập trình hướng đối tượng để mô tả điều phải làm khi một sự cố không mong đợi xuất hiện. Sự cố này có thể là một lỗi, ví dụ phương thức triệu gọi với tham số không hợp lệ, kết nối mạng thất bại hoặc người dùng yêu cầu mở một tập tin không tồn tại.

Java cung cấp hai loại exception: được kiểm soát (checked) và không được kiểm soát (unchecked).

- Checked exception là những sự cố mà lập trình viên dự kiến sẽ xảy ra và sẵn sàng xử lý nó trong chương trình. Checked exception xuất hiện từ các điều kiện ngoài, dễ xảy ra trong khi chương trình chạy. Ví dụ, không tìm thấy tập tin yêu cầu hoặc kết nối thất bại.

- Unchecked exception xuất hiện từ các điều kiện tạo thành lỗi (bug) của chương trình hoặc những tình huống phức tạp mà chương trình không thể xử lý hợp lý. Khi chúng xuất hiện, bạn không cần làm bất kỳ điều gì để kiểm soát chúng.

Unchecked exception lại có hai loại:

+ Fatal error: unchecked exception do lỗi của môi trường, hiếm xảy ra và khó giải quyết, gọi là lỗi (error). Ví dụ, lỗi chạy vượt ra ngoài không gian nhớ cấp phát, OutOfMemoryError. Đây là các lỗi nghiêm trọng, không thể kiểm soát được. Lớp Error là lớp cơ sở được dùng cho các exception loại này.

+ Non-fatal error: unchecked exception do lỗi của chương trình gọi là exception thời gian chạy (runtime exception), ví dụ các exception ném ra khi thử truy cập vượt khỏi phạm vi mảng (ArrayIndexOutOfBoundsException), lỗi chia cho 0 (divide by zero) và lỗi null pointer. Lớp RuntimeException, thừa kế từ lớp Exception, là lớp cơ sở được dùng cho các exception loại này. Bạn cũng phải kết thúc chương trình khi chúng xuất hiện, tuy nhiên nên cố thử bảo lưu công việc người dùng đang làm. RuntimeException chỉ ra bạn có vấn đề (bug) trong thiết kế hoặc cài đặt. Ví dụ, chương trình ném ra NullPointerException khi bạn gọi phương thức từ một tham chiếu không liên kết với một đối tượng mới tạo nào hoặc với một đối tượng có sẵn nào trong heap.

Exception và Error thừa kế từ lớp cơ sở java.lang.Throwable, lớp cha của tất cả các đối tượng có thể được ném và được bắt bằng cách dùng cơ chế xử lý exception. Khi một exception xuất hiện trong chương trình, phương thức ném ra exception có thể tự xử lý exception hoặc ném (throw) exception trở lại phương thức gọi nó (gọi là phương thức caller) để báo rằng có vấn đề xuất hiện. Phương thức nhận được exception ném đến cũng giải quyết tương tự: tự xử lý exception hoặc ném exception trở lại phương thức gọi nó.

Ví dụ về exception:

```

public class AddArguments {
    public static void main(String[] args) {
        int sum = 0;
        for (String arg : args) {
            sum += Integer.parseInt(arg);
        }
        System.out.println("Sum = " + sum);
    }
}

```

Kết quả chạy, chú ý IDE thường hỗ trợ chỉ ra số thứ tự của dòng code ném ra exception và link đến dòng code đó.

```
java AddArguments 1 two 3.0 4
```

```

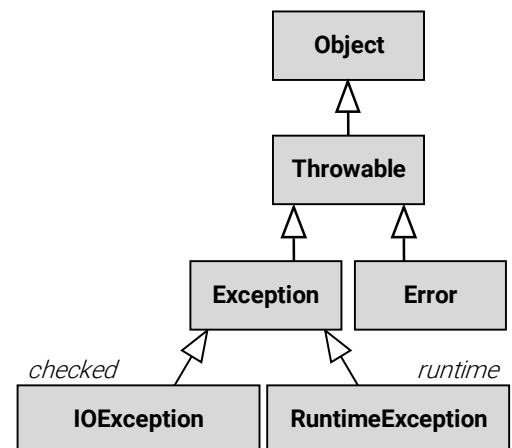
Exception in thread "main" java.lang.NumberFormatException: For input string: "two"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.lang.Integer.parseInt(Integer.java:580)
    at java.lang.Integer.parseInt(Integer.java:615)
    at TestProject.main(AddArguments.java:5)

```

Khác với C/C++, trong Java, tên của chương trình (AddArguments) không thuộc danh sách tham số dòng lệnh (args).

Một số exception thường gặp:

Exception	Mô tả
ArithmeticException	Chỉ định một điều kiện số học ngoại lệ đã diễn ra.
ArrayIndexOutOfBoundsException	Chỉ định chỉ số dùng truy cập mảng đã vượt khỏi phạm vi giới hạn.
ClassCastException	Chỉ định việc thử ép kiểu xuống lớp con đã thất bại.
DateTimeException	Chỉ định có vấn đề khi tạo, truy vấn, thao tác các đối tượng date-time.
IllegalArgumentException	Chỉ định một tham số không hợp lệ đã được truyền cho phương thức.



IllegalStateException	Chỉ định phương thức đã được gọi tại một thời điểm không thích hợp.
IndexOutOfBoundsException	Chỉ định chỉ số dùng truy cập mảng, chuỗi, vector, ... đã vượt khỏi phạm vi giới hạn.
NullPointerException	Triệu gọi phương thức từ tham chiếu chỉ đến một đối tượng null.
NumberFormatException	Chỉ định việc chuyển từ một chuỗi thành kiểu số không hợp lệ.
UncheckedIOException	Java 8, bao một IOException với một unchecked exception.

a) Phát biểu try-catch

Java cung cấp cơ chế bắt các checked exception ném ra và thử xử lý với thông tin lỗi nhận được, bạn dùng khối try bao lấy phát biểu có thể ném exception để bắt lấy exception nếu nó được ném ra và xử lý nó trong khối catch.

Chú ý, phải đặt checked exception vào khối try, không kiểm soát runtime exception và error.

```
public class AddArguments {
    public static void main(String[] args) {
        int sum = 0;
        for (String arg : args) {
            try {
                sum += Integer.parseInt(arg);
            } catch (NumberFormatException e) {
                System.err.printf("[%s] is not an integer (not be included in the sum).", arg);
            }
        }
        System.out.println("Sum = " + sum);
    }
}
```

Kết quả chạy:

```
java AddArguments 1 two 3.0 4
[two] is not an integer (not be included in the sum).
[3.0] is not an integer (not be included in the sum).
Sum = 5
```

Một phương thức có thể ném ra nhiều loại exception, liệt kê thành danh sách trong phát biểu throws. Khi đó, khối try-catch có lỗi gọi phương thức sẽ ném ra nhiều exception, bạn dùng *nhiều* khối catch để bắt và xử lý từng loại exception riêng.

```
public void multitrouble() throws MyException, MyOtherException {
    // ném ra MyException và MyOtherException
}
```

```
public void test() {
    try {
        multitrouble();
    } catch (MyException e1) {
        // code thực hiện nếu MyException được ném
    } catch (MyOtherException e2) {
        // code thực hiện nếu MyOtherException được ném
    } catch (Exception e) {
        // code thực hiện nếu có bất kỳ exception nào khác được ném
    }
}
```

Do các lớp exception được tổ chức thành cây phân nhánh, bạn có thể bắt một *nhóm các exception* bằng cách đón bắt *lớp cơ sở* của chúng. Ví dụ, bằng cách đón bắt IOException bạn có thể bắt các exception thuộc lớp con của nó (EOFException, FileNotFoundException). Chính vì thế, bạn phải quan tâm đến *thứ tự* các khối catch, trong ví dụ trên nếu đặt khối catch cho Exception ở vị trí đầu tiên, do Exception là lớp cơ sở của các checked exception nên khối này đã xử lý tất cả các exception bắt được, các khối catch tiếp sau không bao giờ được xử lý đến.

Java 7 cho phép kết hợp các exception trong một khối catch, gọi là multiple catching exception, dĩ nhiên chúng phải không có quan hệ thừa kế với nhau.

```
try {
    // code ném ra một hoặc nhiều exception
} catch (MyException | MyOtherException e) {
    // code thực hiện nếu các exception (không quan hệ thừa kế) được ném
}
```

b) Khối finally

Khối finally định nghĩa một khối code luôn luôn được thực hiện, cho dù có một exception đã ném ra. Trong trường hợp khối catch và khối finally đều chứa phát biểu return, phát biểu return trong khối finally sẽ được thực hiện.

```
try {
    startFaucet();
    waterLawn();
} catch (BrokenPipeException e) {
    logProblem(e);
} finally {
    stopFaucet();
}
```

Chú ý là khối `catch` và `finally` là tùy chọn, nhưng bắt buộc phải có một trong hai khối. Ví dụ, bạn có thể viết khối `try-finally` mà không cần `catch`. Bạn có thể có không hoặc nhiều khối `catch`, nhưng chỉ có một khối `finally`.

c) Khai báo exception

Java yêu cầu nếu có bất kỳ checked exception (lớp con của `Exception` nhưng không phải lớp con của `RuntimeException`) nào được ném tại một điểm bất kỳ nào trong phương thức, thì phương thức phải được định nghĩa tường minh hành động nào cần thực hiện nếu có exception ném ra:

- Xử lý exception bằng cách dùng khối `try-catch-finally`.

Bao lấy phát biểu ném exception bằng khối `try-catch-finally`. Khối `catch` phải bắt exception có tên chỉ định hoặc exception cha của nó.

- Signature của phương thức phải khai báo các exception mà phương thức ném ra bằng từ khóa `throws`.

Phương thức của bạn gọi một phát biểu có ném exception. Nó có thể không xử lý exception đó mà ném tiếp exception cho phương thức gọi nó. Bạn chỉ cần khai báo có ném exception và chuyển trách nhiệm xử lý exception cho phương thức gọi nó.

```
class MyException extends Exception {
    public MyException(String message) {
        super(message);
    }
}

public class Test {
    // trouble() ném ra exception
    public void trouble() throws MyException {
        int x = -1;
        if (x < 0) {
            throw new MyException("Bug!");
        }
    }

    // delegate1() xử lý một phần và ném lại
    public void delegate1() throws MyException {
        try {
            trouble();
        } catch (MyException e) {
            System.err.println("Error caught and rethrown");
            MyException oe = new MyException("More bug!");
            oe.initCause(e);
            throw oe;
        }
    }

    // delegate2() không xử lý exception, ném tiếp
    public void delegate2() throws MyException {
        delegate1();
    }

    // resolve() xử lý exception trong khối catch
    public void resolve() {
        try {
            delegate2();
        } catch (MyException e) {
            System.err.println(e.getMessage());
            System.err.println(e.getCause().getMessage());
        }
    }

    public static void main(String s[]) {
        new Test().resolve();
    }
}
```

Lưu ý phương thức `delegate1()`, exception cho `catch` và `throw` là giống nhau. Như vậy khối `catch` chỉ xử lý một phần exception rồi ném lại exception (re-throwing). Thêm vào đó, khối `catch` tạo một `MyException` và đặt thông điệp "More bug!" vào, exception sau đó được nhúng vào đối tượng `MyException` vừa tạo bằng cách gọi phương thức `initCause()`. Cuối cùng, đối tượng mới được ném lại cho phương thức gọi. Đến phương thức `resolve()`, thông điệp trong exception "nguyên nhân" mới được truy xuất bằng phương thức `getCause()`.

Xem xét tình huống mà phương thức ném hai loại exception khác nhau:

```
public void compute() throws IOException, ParseException {
    try {
        // gọi phương thức ném ra IOException, ParseException
    } catch (Exception e) {
        throw e;
    }
}
```

}
 }
 bạn muốn ném lại hai exception bằng cách dùng một exception cha của chúng, trình biên dịch không cho phép điều này. Java 7 giải quyết bằng cách thêm từ khóa `final` trong khối `catch`, gọi là `final re-throw`:

```
catch (final Exception e) {}
```

Sử dụng `final` trong khối `catch` cho phép ném ra loại exception chính xác nếu nó xảy ra. Nghĩa là nếu `IOException` xảy ra, sau đó `IOException` sẽ được ném ra.

Khi thừa kế và viết lại một phương thức có ném exception:

- Cho phép phương thức viết lại ném ra *ít exception* hơn (kể cả không ném exception) so với phương thức mà nó thừa kế.
- Cho phép ném ra các *exception là lớp con* của exception do phương thức mà nó thừa kế ném ra. Ví dụ, phương thức lớp cha ném ra `IOException`, phương thức viết lại có thể ném `EOFException` (lớp con của `IOException`), nhưng không được ném ra `Exception` (lớp cha của `IOException`).

d) Xử lý stack-trace

Do cơ chế gọi phương thức để thực thi dựa trên stack, nên nếu một phương thức ném ra exception và exception này không được xử lý trong phương thức, thì exception được ném đến phương thức gọi. Nếu exception không được xử lý trong phương thức gọi, nó sẽ được ném đến phương thức gọi phương thức đó, và cứ tiếp tục như thế cho đến khi gặp phương thức `main()`, nếu `main()` không xử lý exception, exception in ra ngõ xuất chuẩn và chương trình dừng thực thi.

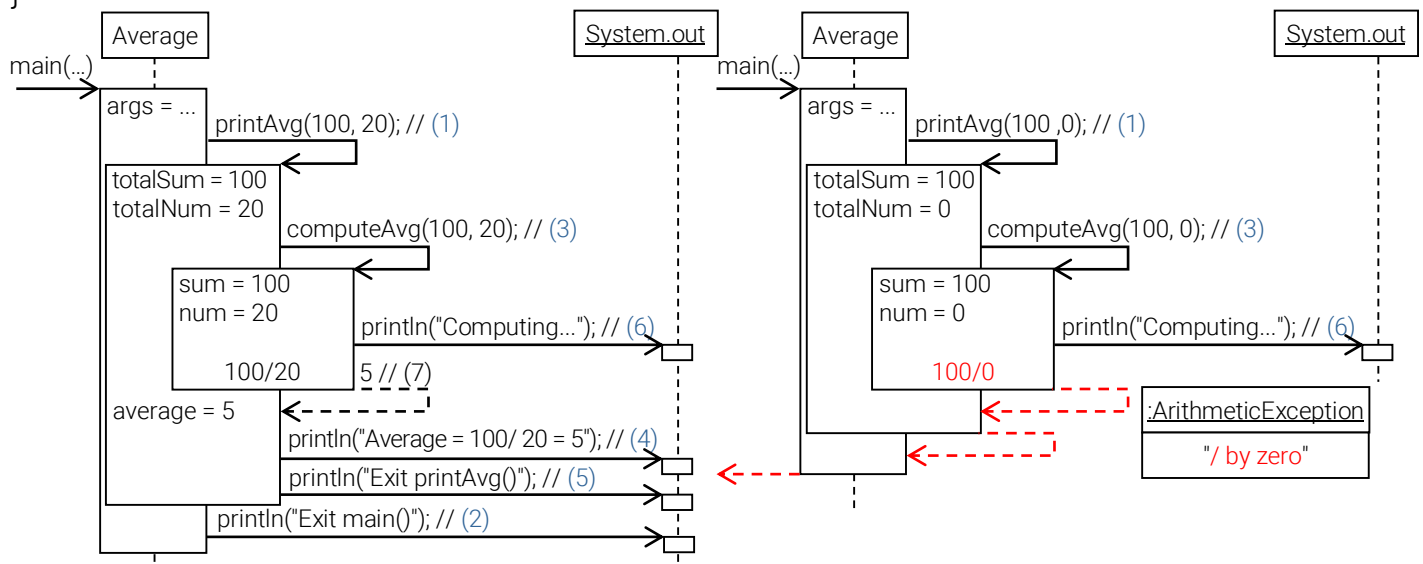
Xem ví dụ chạy thành công (trái) và ném ra exception (phải), chú ý stack-trace của chuỗi exception ném ra.

```
public class Average {
```

```
    public static void main(String[] args) {
        printAvg(100, 0); // (1)
        System.out.println("Exit main()"); // (2)
    }
```

```
    public static void printAvg(int totalSum, int totalNum) {
        int average = computeAvg(totalSum, totalNum); // (3)
        System.out.println("Average = " + // (4)
            totalSum + " / " + totalNum + " = " + average);
        System.out.println("Exit printAverage()") // (5)
    }
```

```
    public static int computeAvg(int sum, int number) {
        System.out.println("Computing..."); // (6)
        return sum / number; // (7)
    }
}
```



Computing...

Exception in thread "main" java.lang.ArithmeticException: / by zero

at Average.computeAvg(Average.java:17)

at Average.printAvg(Average.java:9)

at Average.main(Average.java:4)

Java Result: 1

Ví dụ sau minh họa cách lấy thông tin chi tiết từ stack-trace:

```
public class UsingException {
    public static void main(String[] args) {
        try {
            trouble2();
        } catch (Exception e) {
```

```

        System.err.println(e.getMessage());
        e.printStackTrace(System.err);
        // lấy thông tin stack-trace
        StackTraceElement[] traces = e.getStackTrace();
        System.out.println("Class\t\tFile\t\tLine\tMethod");
        for (StackTraceElement element : traces) {
            System.out.printf("%s\t", element.getClassName());
            System.out.printf("%s\t", element.getFileName());
            System.out.printf("%s\t", element.getLineNumber());
            System.out.printf("%s\n", element.getMethodName());
        }
    }
}

public static void trouble2() throws Exception {
    trouble1();
}

public static void trouble1() throws Exception {
    throw new Exception("Origin exception");
}
}

```

e) Tạo lớp exception riêng

Khi tạo exception định nghĩa bởi người dùng, bạn cần chú ý:

- Phải thừa kế từ lớp exception đã có sẵn, thử thừa kế từ một lớp exception có liên quan hoặc thừa kế trực tiếp lớp Exception. Cũng có thể quyết định thừa kế dựa trên loại exception. Nếu client yêu cầu xử lý exception, lớp exception mới thừa kế từ checked exception. Nếu client bỏ qua exception, lớp exception mới thừa kế từ unchecked exception.
- Trong lớp exception mới, thường chỉ khai báo các constructor sau: không tham số, có String là tham số, có Throwable là tham số, có cả String và Throwable là tham số.

```

public class ServerTimeoutException extends Exception {
    private int port;

    public ServerTimeoutException(String message, int port) {
        super(message);
        this.port = port;
    }

    public int getPort() {
        return port;
    }
}

```

Giả sử bạn viết một chương trình client-server. Trong code phía client, bạn thử kết nối đến server và dự kiến server sẽ trả lời trong 5 giây. Nếu server không trả lời, code của bạn sẽ ném ra exception có kiểu ServerTimeoutException định nghĩa ở trên.

```

public void connect(String serverName) throws ServerTimeoutException {
    boolean successful;
    int port = 80;
    successful = open(serverName, port);
    if (!successful) {
        throw new ServerTimeoutException("Could not connect", port);
    }
}

```

Đối tượng exception tạo bằng toán tử new luôn *cùng dòng* với throw, vì thông tin về số thứ tự dòng cũng chứa trong exception vừa tạo.

f) Phát biểu try-with-resources

Nhiều tác vụ liên quan đến một số tài nguyên mà bạn cần phải đóng lại sau khi sử dụng, cho dù có exception ném ra. Trước Java 7 bạn đóng các tài nguyên trong khối finally:

```

public String readFirstLineFromFile(String path) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        if (br != null) br.close();
    }
}

```

Nếu phát biểu đóng tài nguyên lại ném exception hoặc cần kiểm tra tài nguyên khác null trước khi đóng, bạn lại phải giải quyết tiếp trong khối finally. Java 7 giới thiệu phát biểu *try-with-resources*, cho phép *đóng tài nguyên tự động*.

```

public String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {

```

```

    return br.readLine();
}
}

```

Tài nguyên khai báo trong cặp () của try là tài nguyên tự động đóng, chúng phải cài đặt interface `java.lang.AutoCloseable` hoặc interface `java.io.Closeable` (thừa kế `AutoCloseable`). Bạn có thể cài đặt tài nguyên tự động đóng của riêng bạn bằng cách cài đặt interface `AutoCloseable` cho lớp tài nguyên, trong đó hiện thực phương thức `close()` của interface.

Có thể có nhiều dòng lệnh trong cặp () của try, mỗi dòng cho một loại tài nguyên `AutoCloseable`, riêng dòng lệnh cuối không cần dấu ; cuối dòng lệnh.

2. Assertion

Assertion (xác nhận) là cách để kiểm tra một giả định nào đó về logic của chương trình. Nói cách khác, assertion giúp kiểm tra tính chính xác của chương trình. Ví dụ, nếu bạn tin rằng tại một điểm cụ thể trong chương trình, trị của biến đang quan tâm luôn dương, một assertion có thể kiểm tra điều này. Assertion thường dùng để kiểm tra logic chương trình tại một điểm bên trong phương thức. Nếu giả định là sai, JVM ném một lỗi thuộc lớp `AssertionError` và chương trình ngưng hoạt động.

Một ưu điểm của assertion là chúng có thể loại bỏ hoàn toàn khỏi code khi cần. Bạn có thể kích hoạt assertion trong lúc phát triển chương trình và bất hoạt toàn bộ chúng khi biên dịch ra sản phẩm cuối.

Hai dạng cú pháp cho phép của phát biểu assertion là:

```

assert <boolean_expression> ;
assert <boolean_expression> : <detail_expression> ;

```

Nếu biểu thức boolean định trị `false`, `AssertionError` sẽ được ném, lỗi này không được bắt và chương trình kết thúc.

Nếu dùng dạng thứ hai, biểu thức thứ hai có thể có kiểu bất kỳ, sẽ chuyển thành kiểu `String` và được dùng để hỗ trợ in ra thông điệp báo lỗi.

Assertion thường dùng xác định các lỗi tiềm ẩn hoặc các lỗi logic có thể, bằng cách kiểm tra các bất biến (invariant):

- Bất biến nội (internal invariant), là tình huống luôn luôn xảy ra hoặc không bao giờ xảy ra trong code của bạn. Bạn dùng assertion để kiểm tra tính chính xác của giả định này.

- Bất biến dòng điều khiển (control flow invariant), giống bất biến nội, nhưng liên quan đến dòng điều khiển. Ví dụ, bạn tin rằng dòng điều khiển không bao giờ rơi vào trường hợp default trong phát biểu switch, thử kiểm tra điều này:

```

public static void main(String[] args) {
    int top = 0;
    int month = 0;
    int year = 2016;
    switch (month) {
        case 4: case 6: case 9: case 11: top = 30; break;
        case 1: case 3: case 5: case 7: case 8: case 10: case 12: top = 31; break;
        case 2: top = (year % 400 == 0 || year % 4 == 0 && year % 100 != 0) ? 29 : 28; break;
        default: assert false : "Unknown month";
    }
    System.out.println(top);
}

```

- Hậu điều kiện (postcondition) và bất biến lớp (class invariant).

Hậu điều kiện là giả định về trị hoặc mối quan hệ của các biến lúc kết thúc phương thức. Ví dụ, bạn kiểm tra sau khi thực hiện phương thức `pop()` trong lớp `Stack`, hậu điều kiện là `Stack` phải chứa ít hơn 1 phần tử (ngoại trừ stack rỗng).

Bất biến lớp được kiểm tra tại cuối mỗi phương thức của lớp. Ví dụ, một bất biến lớp của lớp `Stack` là số phần tử của nó luôn không âm.

Mặc định assertion là bất hoạt, đây là một ưu điểm vì ta có thể đặt `assert` khắp nơi ta muốn mà không ảnh hưởng chương trình. Khi phát triển code, ta bật (cho phép) assertion để kiểm tra tính chính xác của các bất biến, bằng một trong hai cách sau:

```

java -enableassertions MyProgram
java -ea MyProgram

```

Trong project NetBeans, cấu hình tùy chọn này bằng cách vào `Properties > Run`, thêm tùy chọn `-ea` vào `VM Options`.

3. Kiểu Optional

Một `NullPointerException` xuất hiện khi bạn thử gọi một phương thức hoặc truy cập một thuộc tính trên một tham chiếu `null`.

Cần bảo đảm rằng các tham chiếu không `null` trước khi bạn dùng chúng để gọi phương thức.

Java 8 cung cấp lớp `Optional` như một giải pháp ngăn ngừa `NullPointerException`.

- Đối tượng `Optional`

Đối tượng thuộc lớp `Optional` xem như một đối tượng bao lấy đối tượng cần kiểm soát khả năng bị `null`, gọi là *đối tượng được bao* (wrapped object).

Tạo đối tượng `Optional` rỗng:

```
Optional<Job> empty = Optional.empty();
```

Tạo đối tượng `Optional` từ đối tượng mặc định:

```
Optional<Job> oJob = Optional.of(new Job(0, "N/A"));
```

Tuy nhiên, nếu đối tượng truyền đến phương thức `of()` là `null`, bạn vẫn nhận được `NullPointerException`. Hãy dùng phương thức `ofNullable()`, khi đối tượng được bọc là `null`, phương thức sẽ trả về một đối tượng `Optional` rỗng.

```
Optional<Job> oJob = Optional.ofNullable(null);
```

```
oJob.ifPresent(j -> System.out.println(j.title));
```

- Kiểm tra đối tượng được bao

Khi bạn có một đối tượng `Optional`, bạn có thể kiểm tra đối tượng được bao của nó có `null` hay không bằng phương thức `isPresent()`. Phương thức này trả về `true` nếu và chỉ nếu đối tượng được bao không `null`. Như vậy, nếu đối tượng được bao không `null`, đối tượng `Optional` bao nó là "có mặt" (`isPresent`).

Phương thức `ifPresent()` cho phép bạn gọi phương thức (hay truy cập thuộc tính) trên đối tượng được bao nếu nó được kiểm tra là không `null`.

```
// cách tiếp cận cũ
if (job != null) System.out.println(job.title);
// cách tiếp cận dùng Optional
Optional<Job> oJob = Optional.ofNullable(job);
oJob.ifPresent(j -> System.out.println(j.title));
```

- Trả về có điều kiện với `filter()`

Tổng quát hơn phương thức `ifPresent()`, phương thức `filter()` được dùng để kiểm tra đối tượng được bao, nó nhận tham số là một predicate và trả về đối tượng `Optional`. Nếu predicate trả về `true`, đối tượng `Optional` được trả về, nếu predicate trả về `false`, một đối tượng `Optional` rỗng được trả về.

```
private Optional<Job> find(List<Job> jobs, int id) {
    return jobs.stream()
        .filter(j -> j.id == id)
        .findFirst();
}
```

- Lấy đối tượng được bao

Khi bạn nhận được đối tượng `Optional`, chẳng hạn được trả về từ phương thức trên, bạn có thể lấy đối tượng được bao trong nó, bằng các cách sau:

```
orElse dùng để lấy đối tượng được bao trong đối tượng Optional, nếu đối tượng này là null, tham số của orElse sẽ
        được trả về như một đối tượng mặc định.
        Job job = null;
        Job nJob = Optional.ofNullable(job).orElse(new Job());
        System.out.println(nJob.title);
orElseGet tương tự orElse(). Tuy nhiên, thay vì nó trả về một đối tượng mặc định nếu đối tượng được bao là null, nó nhận
        một Supplier, gọi get() của Supplier và trả về đối tượng của lời gọi.
        Job job = null;
        Job nJob = Optional.ofNullable(job).orElseGet(Job::new);
        System.out.println(nJob.title);
        Nếu đối tượng được bao là null, orElse() và orElseGet() hoạt động như nhau. Nhưng nếu đối tượng được bao
        không null, orElse() vẫn tạo ra đối tượng mặc định dù không dùng đến.
        orElseThrow thay vì trả về đối tượng mặc định sẽ ném ra exception chỉ định, thay vì ném NullPointerException:
        Job job = null;
        Job nJob = Optional.ofNullable(job).orElseThrow(IllegalArgumentException::new);
        System.out.println(nJob.title);
get chỉ trả về nếu đối tượng được bao không null, nếu không nó ném ra NoSuchElementException. Vì vậy phương
        thức này còn hạn chế và không nên dùng.
        Job job = new Job(10, "Thinking");
        Optional<Job> oJob = Optional.ofNullable(job);
        System.out.println(oJob.get().title);
```

Nếu bạn chỉ cần lấy thuộc tính nào đó của đối tượng được bao, không nhất thiết phải lấy được đối tượng được bao:

+ Nếu getter của đối tượng được bao không trả về `Optional`, sử dụng phương thức `map()`:

```
// getter của lớp Job
public String getTitle() { return title; }
// dùng map()
String title = find(jobs, 5)
    .map(Job::getTitle)
    .orElse("N/A");
```

+ Nếu getter của đối tượng được bao trả về `Optional`, sử dụng phương thức `flatMap()`:

```
// getter của lớp Job, trả về Optional
public Optional<String> getTitle() { return Optional.ofNullable(title); }
// dùng flatMap()
String title = find(jobs, 5)
    .flatMap(Job::getTitle)
    .orElse("N/A");
```

- Tiếp cận theo cách cũ, không dùng `Optional`

Trước đây, các lớp bạn viết tiềm ẩn khả năng ném ra `NullPointerException`. Bạn phải ngăn chặn điều này bằng viết code kiểm tra từng tham chiếu một có khác `null` không trước khi sử dụng chúng.

```
class Job {
    int id = 0;
    String title = "N/A";
    User assignedTo;

    Job() { }
```



```

    Job(int id, String title) {
        this.id = id;
        this.title = title;
    }

    public int getId() { return id; }
    public String getTitle() { return title; }
    public User getAssignedTo() { return assignedTo; }
    @Override public String toString() { return String.format("[%d] %s", id, title); }
}

class User {
    String username;

    User() {}
    User(String username) { this.username = username; }
    public String getUsername() { return username; }
}

// tránh NullPointerException trong các lớp không dùng Optional
private Job find(List<Job> jobs, int id) {
    for (Job job : jobs)
        if (job.id == id) return job;
    return null;
}

// nếu tìm job với id 4, NullPointerException vì không có trong danh sách
// nếu tìm job với id 1, có job nhưng NullPointerException vì job không có assignedTo
List<Job> jobs = Arrays.asList(new Job(1, "Thinking"), new Job(2, "Coding"), new Job(3, "Debugging"));
Job job = find(jobs, 1);
String username;
if (job != null) {
    User user = job.getAssignedTo();
    username = (user != null) ? user.getUsername() : "N/A";
} else throw new NoSuchElementException();

```

Một cách tiếp cận là dùng design pattern Null Object. Ý tưởng là thay vì trả về null, trả về một *đối tượng null*, thừa kế lớp cần kiểm soát. Ví dụ, để tránh getAssignedTo() tạo ra null.

```

class NullUser extends User {
    NullUser() { super("N/A"); }
    static NullUser getInstance() { return new NullUser(); }
}

```

Khi không có User gán cho assignedTo, trả về NullUser.

```

public User getAssignedTo() {
    return assignedTo == null ? NullUser.getInstance() : assignedTo;
}

```

- Tiếp cận theo cách dùng Optional

```

class Job {
    int id = 0;
    String title = "N/A";
    User assignedTo;

    Job() { }

    Job(int id, String title) {
        this.id = id;
        this.title = title;
    }

    public Optional<Integer> getId() { return Optional.ofNullable(id); }
    public Optional<String> getTitle() { return Optional.ofNullable(title); }
    public Optional<User> getAssignedTo() { return Optional.ofNullable(assignedTo); }
    @Override public String toString() { return String.format("[%d] %s", id, title); }
}

class User {
    String username;

    User() {}
    User(String username) { this.username = username; }
    public Optional<String> getUsername() { return Optional.ofNullable(username); }
}

```

// tránh NullPointerException trong các lớp có dùng Optional

```
private Optional<Job> find(List<Job> jobs, int id) {  
    return jobs.stream()  
        .filter(j -> j.id == id)  
        .findFirst();  
}  
  
List<Job> jobs = Arrays.asList(new Job(1, "Thinking"), new Job(2, "Coding"), new Job(3, "Debugging"));  
String username = find(jobs, 1)  
    .orElseThrow(NoSuchElementException::new)  
    .getAssignedTo()  
    .orElseGet(User::new)  
    .getUsername()  
    .orElse("N/A");  
System.out.println(username);
```

OOP**Encapsulation****1. Encapsulation**

Một trong những phương pháp cơ bản mà chúng ta dùng để xử lý một vấn đề phức tạp là trừu tượng hóa (abstraction). Trừu tượng hóa mô tả những thuộc tính (property) và hành vi (behavior) của một đối tượng sao cho nó phân biệt được với các đối tượng khác. Bản chất của OOP là mô hình hóa vấn đề đã trừu tượng bằng cách dùng lớp.

Một lớp (class) mô tả một nhóm các đối tượng (object) trong thực tế. Lớp giống như một bản vẽ (blueprint), một khuôn mẫu (template) chung cho nhóm đối tượng đó. Một lớp mô hình hóa một nhóm đối tượng được trừu tượng bằng cách đóng gói (encapsulation) thuộc tính và hành vi của nhóm đối tượng đó.

Tên lớp là danh từ, theo cú pháp camel case (viết hoa các ký tự đầu các từ). Ví dụ, lớp SwordFish.

Thuộc tính (attribute) của đối tượng được định nghĩa thành các trường (field) trong lớp. Một trường của lớp là một biến dùng lưu trữ thể hiện một thuộc tính riêng của đối tượng, còn gọi là dữ liệu thành viên (data member) của lớp. Tên thuộc tính bắt đầu bằng danh từ (viết thường) và các từ tiếp theo dùng camel case. Ví dụ, thuộc tính wayPoint.

Hành vi (behavior) của đối tượng được định nghĩa thành các tác vụ (operation) của lớp, còn gọi là phương thức (method) hoặc hàm thành viên (function member) trong lớp. Tên phương thức bắt đầu bằng động từ (viết thường) và các từ tiếp theo dùng camel case. Ví dụ, phương thức getWayPoint().

Để quan sát lớp một cách trực quan, bạn nên tìm hiểu sử dụng công cụ tạo sơ đồ UML: UMLet tại <http://www.umlet.com/> hoặc cài đặt plugin easyUML cho NetBeans.

Sau khi tạo, lớp được dùng làm khuôn mẫu để tạo các thể hiện (instance) thuộc lớp, quá trình này gọi là *instantiation*. Trong tài liệu này, thể hiện thuộc lớp (instance) đôi khi còn được gọi là thực thể hoặc đối tượng thuộc lớp.

Giao kết (contract) định nghĩa các dịch vụ do lớp cung cấp, nghĩa là cho biết các phương thức bạn có thể triệu gọi từ thể hiện của lớp. Cài đặt (implementation) là định nghĩa cách thực hiện các dịch vụ đó. Các đối tượng khác khi giao tiếp với đối tượng của một lớp, chỉ cần biết giao kết của lớp đó để triệu gọi, mà không cần biết dịch vụ của giao kết được cài đặt như thế nào.

Như vậy, phương thức cài đặt trong lớp cho thấy hành vi của các thể hiện thuộc lớp trong thời gian chạy. Hành vi này được triệu gọi bằng cách *truyền thông điệp* đến thể hiện đó, thường được biết như là *gọi phương thức thành viên* của thể hiện đó.

```
public class CharStack {
    private char[] stackArray;           // cài đặt stack bằng mảng
    private int top = -1;                // chỉ số của đỉnh stack.
    // constructor
    public CharStack(int capacity) {
        stackArray = new char[capacity];
    }
    // phương thức nghiệp vụ (business methods)
    public void push(char element) { stackArray[++top] = element; }
    public char pop() { return stackArray[top--]; }
    public char peek() { return stackArray[top]; } // xem trị tại đỉnh stack
    public boolean isEmpty() { return top < 0; }
    public boolean isFull() { return top == stackArray.length - 1; }

    public static void main(String[] args) {
        CharStack stack1, stack2;       // khai báo hai biến tham chiếu
        stack1 = new CharStack(10);     // tạo instance (thể hiện) của lớp
        stack2 = stack1;                // stack2 là một alias, của CharStack do stack1 tham chiếu
        stack1.push('J');
        char c = stack2.pop();
    }
}
```

CharStack
- stackArray: char[]
- top: int
+ push(char)
+ pop(): char
+ peek(): char
+ isEmpty(): boolean
+ isFull(): boolean
+ main(String[])

Các phương thức thành viên có thể truy cập tất cả thành viên non-static của lớp, do phương thức thành viên luôn được truyền một tham chiếu ngầm định *this*, là *tham chiếu chỉ đến đối tượng hiện hành*.

Tham chiếu *this* là một tham chiếu final, không thể sửa đổi. Tham chiếu *this* thường được dùng:

- Phân biệt giữa biến thành viên của lớp và tham số được truyền cho phương thức. Trong một lớp, thành viên được truy cập tường minh thông qua tham chiếu *this*, hoặc không tường minh bằng cách dùng trực tiếp, không thông qua tham chiếu *this*.

Trong trường hợp tham số truyền cho phương thức cùng tên với biến thành viên, gọi là ẩn (hides) hay che (shadows) biến thành viên, bạn phải truy cập biến thành viên thông qua tham chiếu *this* để phân biệt giữa chúng.

```
public class Point {
    int x, y;
    public Point(int x, int y) {
        x = x; // lỗi, gán tham số x cho chính nó, dẫn đến không khởi tạo cho this.x và this.y
    }

    @Override public String toString() {
        return String.format("(%d, %d)", x, y);
    }

    public static void main(String[] args) {
        System.out.println(new Point(10, 11)); // kết quả (0, 0)
    }
}
```

- Phương thức cần truyền đối tượng hiện hành như tham số đến một phương thức khác, nó dùng tham chiếu `this`.

```
class Student {
    int code;
    String name;
    public Student(int code, String name, StudentList list) {
        this.code = code;
        this.name = name;
        list.add(Student.this);
    }
}
```

```
class StudentList extends ArrayList<Student> {
}
```

Các lớp trong Java được đặt vào package (gói), package có cấu trúc phân cấp. Trình biên dịch sẽ tạo ra cấu trúc thư mục phân cấp tương ứng với khai báo package.

```
package reports.accounts.salary;
class Employee { }
reports
├── accounts
│   └── salary
│       └── Employee.class
```

2. Điều khiển truy cập

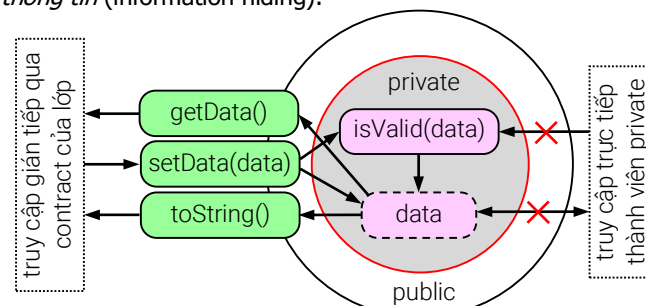
Biến và các phương thức của lớp có thể có một trong bốn cấp độ truy cập: `public`, `protected`, `default` và `private`. Các lớp cũng có cấp độ truy cập `public` hoặc `default` (còn gọi là truy cập cấp gói, package accessibility). Cấp độ truy cập được chỉ định bằng cách dùng các *bổ từ truy cập* (access modifier) hoặc *không* dùng bổ từ truy cập trong trường hợp `default`.

Bổ từ truy cập giúp một lớp định nghĩa contract (giao kết) của nó, giúp client biết chính xác các dịch vụ do lớp cung cấp.

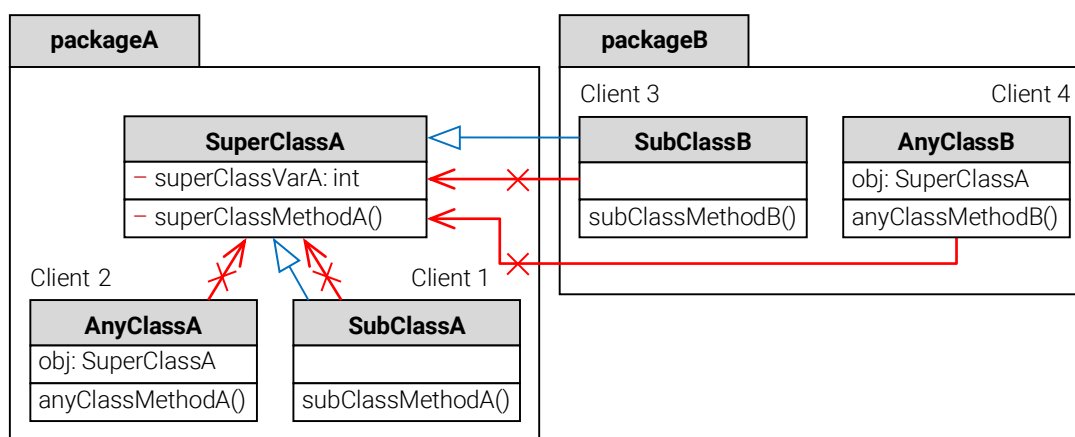
Bảng sau trình bày các cấp độ truy cập:

Access modifier	Trong cùng lớp	Trong cùng package		Khác package	
		Lớp con	Lớp khác	Lớp con	Lớp khác
Private	Yes				
Default	Yes	Yes	Yes		
protected	Yes	Yes	Yes	Yes	
Public	Yes	Yes	Yes	Yes	Yes

Một biến hoặc phương thức được đánh dấu `private` chỉ được truy cập trực tiếp bởi các phương thức và thành viên của *cùng lớp* với biến hoặc phương thức `private` đó, hoặc chỉ có thể truy cập gián tiếp qua các getter/setter (còn gọi là accessor hay mutator) của lớp. Điều này gọi là *ẩn giấu thông tin* (information hiding).

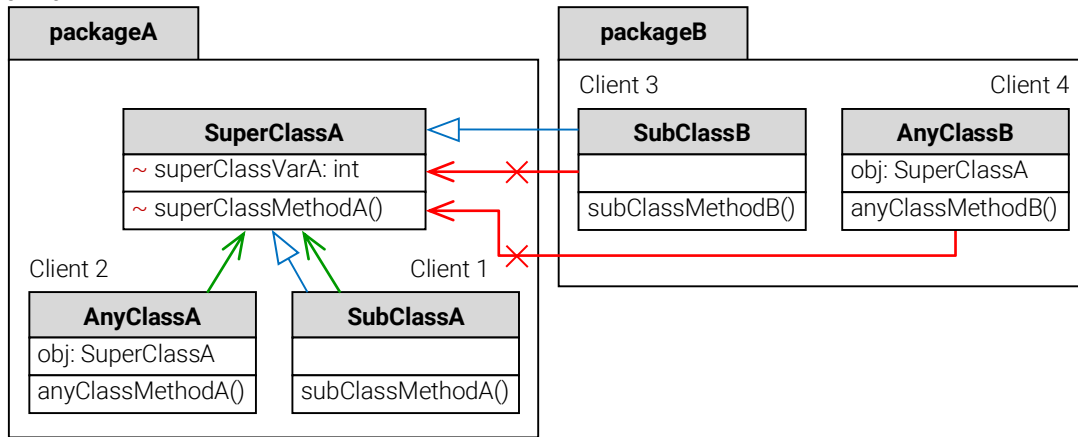


Truy cập thành viên private phải thông qua contract của lớp



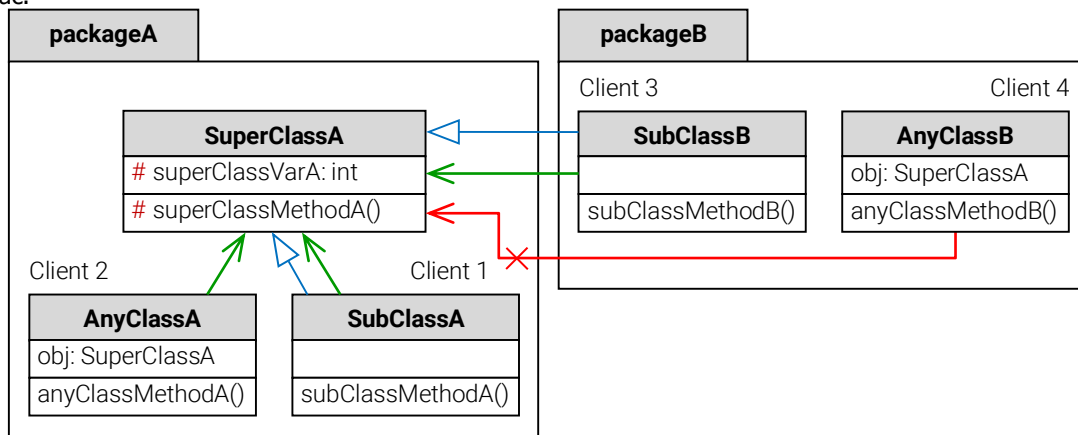
Truy cập thành viên private

Một biến, phương thức hoặc lớp có cấp độ truy cập `default`, nếu nó không có một access modifier trong khai báo của nó. Cho phép truy cập từ bất kỳ phương thức nào trong các lớp *cùng gói*. Thường gọi là *package-friendly* hoặc *package-private*.



Truy cập thành viên default

Một biến hoặc phương thức được đánh dấu `protected` có cấp độ truy cập rộng hơn `default`. Chúng được truy cập từ các phương thức trong các lớp cùng gói và từ các phương thức của lớp con của chính lớp được truy cập, cho dù lớp con này nằm trong gói khác.



Truy cập thành viên protected

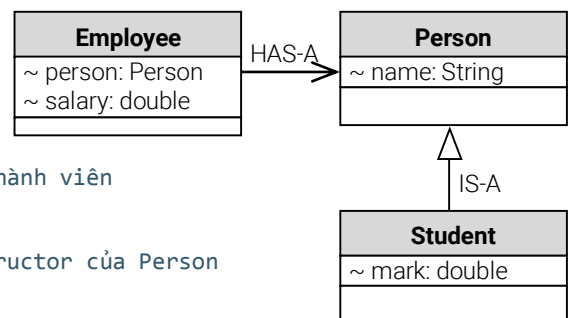
Một biến hoặc phương thức được đánh dấu `public` được truy cập từ mọi nơi. Một lớp được đánh dấu `public` phải có tên trùng với tập tin chứa nó. Lớp và interface không được đánh dấu `private` hoặc `protected`.

Tổng hợp và thừa kế

Có hai cách tiếp cận khi xây dựng một lớp mới:

- **Tổng hợp** (composition) kết hợp một hay nhiều lớp có sẵn để tạo lớp mới, đối tượng của các lớp có sẵn trở thành *thành viên* của lớp mới. Ta nhận được lớp bao hay lớp phức hợp (composed class). Quan hệ giữa hai lớp là quan hệ HAS-A (có một...), nghĩa là lớp mới có tham chiếu đến lớp có sẵn như là thành viên của nó khi xây dựng lớp.
- **Thừa kế** (inheritance) thừa kế các lớp đã có để tạo lớp mới, nghĩa là lớp mới được tạo ra *trên cơ sở* lớp đã có. Ta nhận được lớp dẫn xuất (derived class). Quan hệ giữa hai lớp là quan hệ IS-A (là một..., là một loại của...), nghĩa là lớp mới thừa kế lớp có sẵn khi xây dựng lớp.

```
class Person {
    String name;
    public Person(String name) { this.name = name; }
}
// composition
class Employee {
    Person person;
    double salary;
    public Employee(String name, double salary) {
        this.person = new Person(name); // triệu gọi constructor của Person
        this.salary = salary;
    }
}
// inheritance
class Student extends Person {
    double mark;
    public Student(String name, double mark) {
        super(name); // triệu gọi constructor của Person
        this.mark = mark;
    }
}
```



Thực tế, một lớp được tạo ra bằng cách phối hợp cả hai cách trên.

1. Composition

Một lớp gọi là lớp phức hợp (composed class) nếu nó có dữ liệu thành viên là các đối tượng thuộc các lớp khác. Một lớp mà các đối tượng của nó là dữ liệu thành viên của lớp phức hợp gọi là lớp thành phần.

Quan hệ giữa hai lớp trên là quan hệ kết hợp (association). Quan hệ kết hợp có thể hai chiều, một chiều hoặc đệ quy.

- Quan hệ phụ thuộc (dependency)

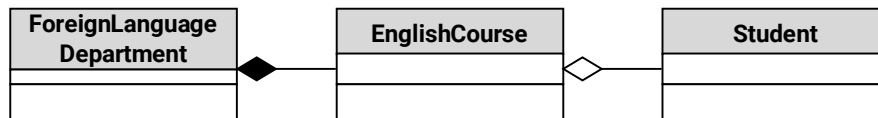
Quan hệ phụ thuộc cũng là quan hệ kết hợp giữa hai lớp nhưng chỉ là quan hệ một chiều, chỉ ra một lớp phụ thuộc vào lớp khác.

Quan hệ này cho thấy một lớp tham chiếu đến một lớp khác. Do vậy, nếu lớp được tham chiếu thay đổi sẽ ảnh hưởng đến lớp sử dụng nó. Đối tượng thuộc lớp thành phần dùng trong trường hợp này thường là biến lớp, biến cục bộ của hàm thành viên hoặc tham số của hàm thành viên thuộc lớp phức hợp.

- Quan hệ tụ hợp (aggregation)

Quan hệ tụ hợp là hình thức mạnh của quan hệ kết hợp. Quan hệ kết hợp giữa hai lớp cho thấy vai trò của chúng cùng mức, không có lớp nào quan trọng hơn. Trong lúc quan hệ tụ hợp là quan hệ giữa toàn thể và bộ phận trong đó một lớp giữ vai trò cái lớn hơn (tổng thể, lớp bao), còn lớp kia giữ vai trò cái nhỏ hơn (bộ phận, lớp thành phần).

Nếu tổng thể và thành phần được hình thành và hủy bỏ vào các *thời điểm khác nhau*, ta gọi là quan hệ tụ hợp bởi tham chiếu hoặc quan hệ kết tập (aggregation). Nếu tổng thể và thành phần được hình thành và hủy bỏ *cùng thời điểm*, ta gọi là quan hệ gộp hoặc quan hệ hợp thành (composition), đây là hình thức mạnh hơn của quan hệ tụ hợp, còn gọi là quan hệ tụ hợp bởi trị.



Quan hệ tụ hợp: quan hệ hợp thành (composition, trái) và quan hệ kết tập (aggregation, phải)

2. Inheritance

Trong lập trình thường gặp tình huống: bạn đang có một lớp và bạn cần một lớp mới chi tiết hơn của lớp gốc. Ví dụ, bạn đang có lớp Employee, bây giờ bạn muốn tạo lớp Manager. Một Manager *là một* Employee, nhưng có thêm một số đặc tính.

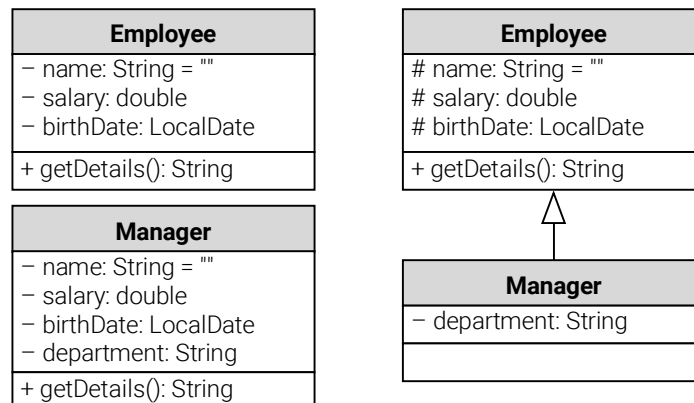
Để tránh phải khai báo, cài đặt lại dữ liệu thành viên và các phương thức của lớp gốc, bạn tạo lớp mới từ lớp đã tồn tại, điều này gọi là dẫn xuất ra lớp con (subclassing).

```

public class Employee {
    protected String name = "";
    protected double salary;
    protected LocalDate birthDate;

    public String getDetails() { ... }
}

public class Manager extends Employee {
    private String department;
}
  
```



Thay vì viết lại lớp (trái), việc dẫn xuất ra lớp con từ một lớp có sẵn (phải) cho phép dùng lại lớp đã viết.

Trong OOP, các cơ chế đặc biệt được cung cấp để cho phép bạn định nghĩa một lớp từ lớp có sẵn.

Ngôn ngữ Java cho phép một lớp thừa kế *một và chỉ một* lớp khác bằng từ khóa **extends**, gọi là đơn thừa kế (single inheritance).

Java dùng **interface** để cung cấp những lợi ích do đa thừa kế (multiple inheritance) mang lại.

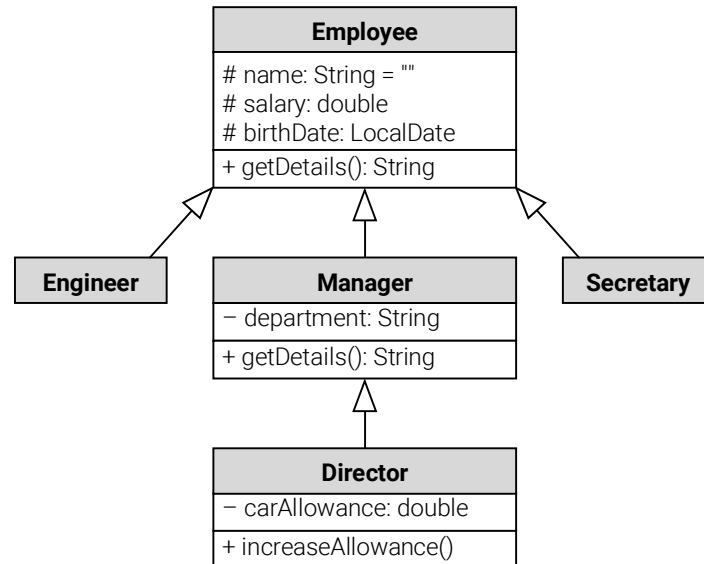
Thừa kế là dẫn xuất ra các lớp mới (thêm các thuộc tính và phương thức từ lớp cơ sở) và thay đổi các thành viên thừa kế được. Hình dưới trình bày lớp cơ sở (base class, superclass, parent class) Employee và ba lớp con (derived class, subclass, child class) thừa kế nó: Engineer, Manager và Secretary.

- Lớp Manager dẫn xuất tiếp ra lớp con Director. Lớp Manager thừa kế tất cả thuộc tính và phương thức của lớp Employee, thêm thuộc tính mới department và viết lại phương thức getDetails() của lớp cha.

- Lớp Director thừa kế tất cả thành viên của lớp Employee và Manager, thêm thuộc tính mới carAllowance và phương thức mới increaseAllowance().

Các quan hệ thừa kế trên có thể mô tả như một cây thừa kế phân cấp (inheritance hierarchy). Lớp nằm ở vị trí càng cao thì càng tổng quát (generalized), các hành vi có thể trừu tượng, mang tính khái niệm. Lớp càng thấp càng chi tiết, chuyên biệt (specialized) do bạn có thể tùy biến các hành vi thừa kế được và bổ sung các thuộc tính, hành vi mới.

Lớp con có thể ẩn (hiding) thuộc tính của lớp cha bằng cách định nghĩa thuộc tính *cùng tên* với lớp cha. Khi đó, lớp con không truy cập được trực tiếp các thuộc tính bị ẩn mà chỉ có thể truy cập thông qua từ khóa super.



3. Viết lại (overriding) phương thức

Ngoài việc tạo ra lớp mới dựa trên lớp cũ bằng cách thêm vào các đặc tính mới, bạn cũng có thể *thay đổi những hành vi có sẵn* của lớp cha.

Nếu một phương thức được định nghĩa trong lớp con cùng signature với phương thức của lớp cha, thì phương thức mới được gọi là viết lại (override) phương thức cũ. Nhắc lại, signature của một phương thức bao gồm tên phương thức, kiểu và số lượng các tham số, bao gồm cả thứ tự của chúng.

Từ Java 5, kiểu trả về của phương thức viết lại có thể là lớp con của kiểu trả về của phương thức được thừa kế, đặc tính này gọi là *covariant return* (trả về một hiệp biến). Chú ý rằng covariant return chỉ áp dụng cho kiểu tham chiếu, vì giữa các kiểu cơ bản không có quan hệ thừa kế.

Lớp Manager có phương thức getDetails() do nó thừa kế từ lớp Employee. Tuy nhiên, phương thức gốc thuộc lớp Employee đã được thay thế, hay viết lại, trong phiên bản lớp con.

```

class Info {
    private String info;
    public Info(Employee employee) {
        this.info = String.format("%s (%d)", employee.name, employee.byear);
    }
    @Override public String toString() { return info; }
}

class ExInfo extends Info {
    private String title;
    public ExInfo(Manager manager) {
        super(manager);
        this.title = manager.title;
    }
    @Override public String toString() { return title + " " + super.toString(); }
}

class Employee {
    String name;
    int byear;
    public Employee(String name, int byear) {
        this.name = name;
        this.byear = byear;
    }
    public Info getDetails() { return new Info(this); }
}

class Manager extends Employee {
    String title;
    public Manager(String name, int byear, String title) {
        super(name, byear);
        this.title = title;
    }
    @Override public ExInfo getDetails() { return new ExInfo(this); }
}

```

Phương thức của lớp con có thể triệu gọi phương thức của lớp cha bằng cách dùng từ khóa super. Từ khóa super tham chiếu đến lớp cha của lớp mà từ khóa super đang được sử dụng. Nó dùng để tham chiếu các biến thành viên hoặc các phương thức

của lớp cha. Nếu phương thức do super triệu gọi không được định nghĩa trong lớp cha, nó có thể được gọi từ một lớp tổ tiên nào đó nằm cao hơn trong cây thừa kế.

Phương thức viết lại không được có *cấp độ truy cập thu hẹp* (narrow) so với phương thức gốc mà phải mở rộng (widen) hơn. Ví dụ, nếu bạn viết lại một phương thức public và đánh dấu chúng thành private là vi phạm.

```
interface Relocatable {
    void move(); // mặc định có truy cập public
}

class CEO implements Relocatable {
    // lỗi, gán truy cập default thu hẹp so với truy cập public của interface
    @Override void move() { }
```

Phương thức viết lại có thể: ném (throws) tất cả hoặc không, hoặc tập con của các checked exception (kể cả lớp con của chúng) được chỉ định trong mệnh đề throws của phương thức được thừa kế.

Bạn nên dùng annotation @Override để giúp dễ nhận ra các phương thức được viết lại.

Đa hình (polymorphism)

Trong ví dụ trên, Manager là một Employee. Manager có tất cả thành viên, thuộc tính và phương thức, của lớp cha Employee. Điều này nghĩa là bất kỳ tác vụ nào hợp lệ trên một Employee thì cũng hợp lệ trên một Manager.

Có vẻ không thực tế khi tạo một Manager và cố gán nó đến một biến có kiểu cha Employee. Tuy nhiên, có lý do để ta muốn thực hiện điều này. Một *đối tượng* chỉ có một hình dạng (form) do ta gán cho nó khi tạo. Tuy nhiên, một *biến* là đa hình do nó có thể tham chiếu đến một nhóm các đối tượng có hình dạng khác nhau. Java, giống như các ngôn ngữ hướng đối tượng khác, cho phép bạn tham chiếu đến một đối tượng bằng cách dùng một biến có kiểu là một trong các lớp cha:

```
Employee e = new Manager();
// Dùng biến e, bạn có thể truy cập một phần của đối tượng, là phần thừa kế từ lớp cha; phần đặc tả cho lớp con là ẩn. Điều này
// do trình biên dịch xem e là một Employee, không phải là Manager. Do vậy, code sau là không hợp lệ:
// không hợp lệ khi thử gán thuộc tính của Manager, do biến được khai báo như một Employee
e.department = "Sales";
```

1. Triệu gọi phương thức ảo

Giả sử bạn có:

```
Employee e = new Employee();
Manager m = new Manager();
// Nếu triệu gọi e.getDetails() và m.getDetails(), bạn đã triệu gọi hai hành vi khác nhau. Đối tượng Employee thực hiện phiên
// bản getDetails() liên kết với lớp Employee, và đối tượng Manager thực hiện phiên bản getDetails() liên kết với lớp Manager.
// Điều này xác định trong thời gian biên dịch. Còn nếu bạn viết:
Employee e = new Manager();
e.getDetails();
```

Thực tế, bạn sẽ nhận được hành vi liên kết với đối tượng được tham chiếu *trong thời gian chạy*. Kiểu của đối tượng thực hiện hành vi không được xác định trong thời gian biên dịch. *Hành vi đa hình* này là một đặc điểm quan trọng của các ngôn ngữ hướng đối tượng, thường được hiểu như triệu gọi đến phương thức ảo (virtual method invocation).

Kiểu của đối tượng được tham chiếu trong thời gian biên dịch là Employee, gọi là kiểu khai báo (declare type), kiểu của đối tượng được tham chiếu trong thời gian chạy là Manager, gọi là *kiểu động* (dynamic type).

Đa hình cho phép bạn làm việc với kiểu trừu tượng trong code để chương trình có mức trừu tượng (tổng quát hóa) cao. Nhưng khi chương trình chạy, nó lại hoạt động chính xác với các kiểu con cụ thể.

Xem ví dụ sau, ta dùng hai mảng kiểu trừu tượng Shape và kiểu interface IDrawable để quản lý các đối tượng thừa kế hoặc cài đặt từ chúng. Trong thời gian biên dịch, các đối tượng trong mảng shapes có kiểu Shape và các đối tượng trong mảng drawables có kiểu IDrawable. Trong thời gian chạy, khi duyệt các mảng và gọi phương thức ảo, kiểu thật của đối tượng trong mảng mới được xác định (dynamic binding) và phương thức đa hình được cài đặt cho lớp cụ thể của nó mới được triệu gọi.

```
interface IDrawable {
    void draw();
}

abstract class Shape implements IDrawable {
    public abstract String area();
    @Override public abstract void draw();
}

class Circle extends Shape {
    @Override public String area() {
        return "R^2 * PI";
    }

    @Override public void draw() {
        System.out.println("[Circle]");
    }
}
```

```

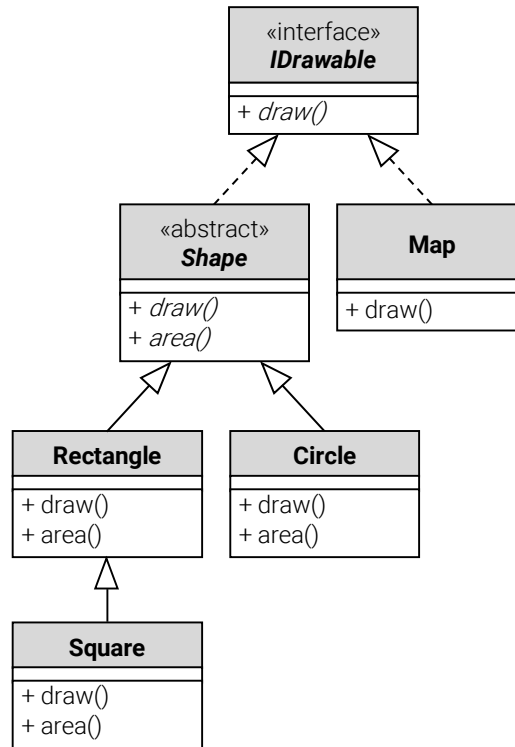
class Rectangle extends Shape {
    @Override public String area() {
        return "Width * Height";
    }
    @Override public void draw() {
        System.out.println("[Rectangle]");
    }
}

class Square extends Rectangle {
    @Override public String area() {
        return "Side^2";
    }
    @Override public void draw() {
        System.out.println("[Square]");
    }
}

class Map implements IDrawable {
    @Override public void draw() {
        System.out.println("[Map]");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape[] shapes = { new Circle(), new Rectangle(), new Square() };
        IDrawable[] drawables = { new Circle(), new Rectangle(), new Map() };
        System.out.println("Shape's area:");
        for (Shape shape : shapes)
            System.out.println(shape.area()); // R^2 * PI, Width * Height, Side^2
        System.out.println("Draw drawables:");
        for (IDrawable drawable : drawables)
            drawable.draw(); // [Circle], [Rectangle], [Map]
    }
}

```



2. Collection không đồng nhất

Bạn thường tạo một collection chứa các đối tượng thuộc một lớp chung. Collection như vậy gọi là collection đồng nhất (homogeneous collection). Ví dụ:

```

MyDate[] dates = new MyDate[2];
dates[0] = new MyDate(2, 9, 1945);
dates[1] = new MyDate(30, 4, 1975);

```

Do tất cả các lớp đều thừa kế lớp Object, bạn có thể tạo một collection chứa thành viên thuộc các lớp khác nhau. Collection như vậy gọi là collection không đồng nhất (heterogeneous collection). Với các thành viên có kiểu cơ bản, bạn tạo đối tượng từ các kiểu cơ bản bằng cách dùng lớp bao (wrapper class).

Thực tế, ta thường xuyên dùng các collection không đồng nhất chứa các đối tượng được dẫn xuất từ một lớp cha chung gần gũi hơn là lớp Object:

```

Shape[] shapes = { new Circle(), new Rectangle(), new Square() };

```

3. Các tham số đa hình

Bạn có thể viết phương thức nhận một đối tượng tổng quát (trong trường hợp này là lớp Employee) và nó làm việc đúng trên đối tượng thuộc lớp con bất kỳ của đối tượng này (ví dụ Manager).

Điều này hợp lệ do Manager là một Employee. Dĩ nhiên, phương thức findTaxRate() chỉ truy cập đến các thành viên được định nghĩa trong lớp Employee.

```

public class TaxService {
    public TaxRate findTaxRate(Employee e) {
        // tính toán và trả về một TaxRate cho e
    }
}

```

```

TaxService taxSvc = new TaxService();
TaxRate t = taxSvc.findTaxRate(new Manager());

```

4. Toán tử instanceof

Bạn có thể truyền các đối tượng khác nhau cho một phương thức, bằng cách dùng tham chiếu đến lớp cha của chúng. Tuy nhiên, đôi khi bạn cần biết lớp cụ thể của tham số đa hình này. Toán tử instanceof sẽ giúp bạn.

Giả sử bạn có cây phân cấp lớp như sau:

```

public class Employee extends Object // không cần thiết, chỉ dùng để nhấn mạnh

```

```
public class Manager extends Employee
public class Engineer extends Employee
```

Nếu phương thức của bạn nhận đối tượng dùng tham chiếu kiểu Employee, và bạn cần xác định đối tượng được truyền đến là Manager hoặc Engineer. Bạn có thể kiểm tra bằng cách dùng toán tử instanceof như sau:

```
public void doSomething(Employee e) {
    if (e instanceof Manager) {
        Manager m = (Manager) e;
        // xử lý Manager
    } else if (e instanceof Engineer) {
        Engineer r = (Engineer) e;
        // xử lý Engineer
    } else {
        // xử lý kiểu khác của Employee
    }
}
```

Khi bạn đã xác định bằng toán tử instanceof, tham chiếu nhận được thuộc lớp con cụ thể nào, bạn có thể truy cập đầy đủ chức năng của đối tượng đó bằng cách ép kiểu tham chiếu (casting). Nếu bạn không kiểm tra trước bằng toán tử instanceof, bạn có nguy cơ gặp lỗi khi ép kiểu:

- Ép kiểu lên (upcasting) trong cây phân cấp là luôn cho phép; thực tế, không cần toán tử ép kiểu, bạn chỉ đơn giản gán.
- Ép kiểu xuống (downcasting) phải bảo đảm ép đúng kiểu, kiểu của đối tượng sẽ được kiểm tra trong thời gian chạy, nếu ép kiểu không phù hợp sẽ ném ra exception. Ví dụ, thử ép kiểu Manager thành Engineer là không cho phép do Engineer không là Manager. Kiểu được ép phải là một lớp con của kiểu hiện có.

Constructor

1. Các phương thức nạp chồng (overloading methods)

Trong một vài tình huống, bạn muốn viết một số phương thức trong cùng lớp, có cùng tác vụ cơ bản nhưng nhận tham số khác nhau. Ví dụ, tác vụ cơ bản là xuất dưới dạng văn bản tham số nhận được, ta gọi là println().

Giả sử bạn cần các phương thức in khác nhau cho các kiểu int, float và String; do các kiểu này cần định dạng và xử lý khác nhau. Như vậy bạn phải tạo ba phương thức printInt(), printFloat() và printString().

Java, cũng như các ngôn ngữ hướng đối tượng khác, cho phép bạn dùng lại một tên phương thức cho một hay nhiều phương thức, phân biệt lời gọi dựa trên số lượng và kiểu các tham số.

```
public void println(int i);
public void println(float f);
public void println(String s);
```

Khi bạn viết code gọi một trong các phương thức trên, phương thức được chọn tùy theo kiểu của tham số hoặc số các tham số được cung cấp trong lời gọi.

Hai luật được áp dụng cho các phương thức nạp chồng:

- Danh sách tham số bắt buộc phải khác nhau.

Danh sách tham số của phát biểu gọi phải đủ khác để phân biệt rõ phương thức được gọi. Chú ý kiểu tham số mở rộng có thể gây nhầm lẫn khi gọi hàm. Ví dụ, không phân biệt giữa phương thức có tham số float và phương thức có tham số double.

- Kiểu trả về có thể khác nhau.

Kiểu trả về của các phương thức nạp chồng *có thể khác nhau*, nhưng không được dùng để phân biệt các phương thức nạp chồng. Một biến thể của nạp chồng hàm là phương thức có *số tham số bất kỳ*. Số tham số của một phương thức, gọi là *arity* của phương thức đó. Ví dụ phương thức tính trung bình cộng các tham số int truyền cho nó:

```
public class Statistics {
    public float average(int x1, int x2) { }
    public float average(int x1, int x2, int x3) { }
    public float average(int x1, int x2, int x3, int x4) { }
}
```

```
float gradePointAverage = new Statistics().average(4, 3, 4);
```

```
float averageAge = new Statistics().average(24, 32, 27, 18);
```

Java 5 cung cấp phương thức có arity thay đổi, gọi là *variable arity* hoặc *varargs*:

```
public class Statistics {
    public float average(int... nums) {
        int sum = 0;
        for (int x : nums) sum += x;
        return nums.length == 0 ? 0 : (float) sum / nums.length;
    }
}
```

nums trong ví dụ trên là một mảng int[] có kích thước tùy theo số tham số được truyền cho phương thức.

2. Constructor nạp chồng (overloading constructor)

Mục đích chính của constructor là khởi gán tập thuộc tính của một đối tượng, khi đối tượng đó được tạo ra bằng cách dùng toán tử new:

- Constructor có cùng tên với tên lớp.

- Constructor không có kiểu trả về, tuy nhiên cho phép dùng phát biểu return; (không trả về trị) trong constructor. Bạn cũng có thể đặt tên phương thức thành viên khác có tên giống với constructor nhưng có kiểu trả về, tuy nhiên không khuyến khích.

Vì vậy, nếu bạn viết constructor có kiểu trả về, constructor đó được xem như phương thức thành viên bình thường.

Khi khởi gán một đối tượng, chương trình có thể cung cấp nhiều constructor dựa trên dữ liệu dùng khởi tạo đối tượng, cho phép khởi gán đối tượng bằng nhiều cách khác nhau. Ví dụ, hệ thống payroll khởi gán cho đối tượng Employee với dữ liệu cá nhân cơ bản như name, salary và birthDate. Đôi khi hệ thống khởi gán cho đối tượng mà không có thông tin về salary hoặc birthDate:

```
public class Employee {
    private static final double BASE_SALARY = 15000.00;
    private String name;
    private double salary;
    private Date birthDate;

    public Employee(String name, double salary, Date dOB) {
        // constructor mặc định được gọi ngầm định tại đây
        this.name = name;
        this.salary = salary;
        this.birthDate = dOB;
    }
    // copy constructor
    public Employee(Employee other) {
        this(other.name, other.salary, other.dOB);
    }

    public Employee(String name, double salary) {
        this(name, salary, null);
    }

    public Employee(String name, Date dOB) {
        this(name, BASE_SALARY, dOB);
    }

    public Employee(String name) {
        this(name, BASE_SALARY);
    }
}
```

Lớp trên có năm constructor, constructor thứ nhất khởi gán cho tất cả biến thực thể. Trong các constructor sau, từ khóa this dùng tham chiếu đến các constructor khác cùng lớp. Constructor thứ hai cho phép tạo đối tượng sao chép từ một đối tượng khác tương tự. Constructor thứ ba và thứ tư gọi constructor thứ nhất, constructor thứ năm gọi constructor thứ ba. Từ khóa this trong constructor phải nằm trong *dòng đầu tiên của khối code* trong constructor, những khởi gán khác nằm sau lời gọi this này.

Constructor mặc định là constructor không có tham số. Một lớp trong Java có ít nhất một constructor, nếu lớp không chỉ định bất kỳ constructor nào, constructor mặc định sẽ được trình biên dịch sinh ra một cách ngầm định.

Trong constructor, không nên gọi các phương thức có thể được viết lại (overridable). Bạn có thể gọi các phương thức static và final trong constructor. Nếu vậy, hoặc đánh dấu phương thức được gọi là static hay final, hoặc chỉ định rõ lớp của tham chiếu this dùng gọi phương thức.

```
public class Student {
    private String code;
    private String name;

    public Student(String name) {
        this.code = genCode(name);
        Student.this.setName(name);
        updateDB(Student.this);
    }
    // private ngầm định là final
    private String genCode(String name) {
        return String.format("STU%05d", System.currentTimeMillis() % 100_000);
    }
    // overridable method
    public void setName(String name) {
        this.name = (name == null || name.isEmpty()) ? "noname" : name;
    }
    // final method, không thừa kế
    public final void updateDB(Student student) {
        // ...
    }
}
```

3. Triệu gọi constructor lớp cha

Mặc dù lớp con thừa kế tất cả phương thức và biến thành viên từ lớp cha, nhưng lớp con *không thừa kế các constructor* của lớp cha. Khi viết một lớp, bạn viết một constructor hoặc nếu không viết một constructor nào, lớp sẽ có một constructor mặc định là constructor không có tham số. Constructor của lớp cha luôn được triệu gọi trong constructor của lớp con.

Giống các phương thức, các constructor có thể gọi các constructor (non-private) từ lớp cha trực tiếp của nó.

Một phần tham số bạn cung cấp cho constructor lớp con được dùng để truyền cho constructor của lớp cha. Bạn có thể triệu gọi constructor cụ thể của lớp cha như một phần công việc khởi tạo lớp con, bằng cách dùng từ khóa `super` ngay *dòng đầu tiên* của constructor lớp con. Để triệu gọi constructor chỉ định, bạn phải cung cấp các đối số tương ứng cho `super()`. Khi không gọi `super`, constructor không tham số của lớp cha sẽ được gọi không tường minh. Khi đó, nếu lớp cha không có constructor không tham số, sẽ xuất hiện lỗi biên dịch.

```
public class Manager extends Employee {
    private String department;

    public Manager(String name, double salary, String dept) {
        super(name, salary);
        this.department = dept;
    }

    public Manager(String name, String dept) {
        super(name);
        this.department = dept;
    }

    public Manager(String dept) {        // có lỗi, không có super()
        this.department = dept;
    }
}
```

Lỗi trong code trên do trình biên dịch chen vào không tường minh lời gọi `super()` không tham số, nhưng lớp `Employee` lại không cung cấp một constructor không tham số.

Constructor được gọi cũng có thể gọi `super()` hoặc `this()`, triệu gọi một chuỗi các constructor.

4. Khởi gán cho đối tượng

Khởi gán đối tượng là một quá trình khá phức tạp do nó gây ra một chuỗi lời gọi constructor, dùng `this()` hoặc `super()`.

Đầu tiên, vùng nhớ cho đối tượng được cấp phát trong heap và trị mặc định cho các biến thực thể được gán. Sau đó, constructor mức cao nhất được gọi và đệ quy trong cây thừa kế:

1. Kết nối (bind) các tham số cho constructor.
2. Nếu có lời gọi `this()` tường minh, gọi đệ quy và nhảy đến bước 5.
3. Gọi đệ quy đến `super()` tường minh hoặc không tường minh, ngoại trừ `Object`, do `Object` không có lớp cha.
4. Thực hiện khởi gán tường minh các biến thực thể.
5. Thực hiện thân của constructor hiện hành.

Ví dụ:

```
public class Employee {
    private String name;
    private double salary = 15000.00;
    private Date birthDate;

    public Employee(String name, Date dOB) {
        // gọi super() không tường minh
        this.name = name;
        this.birthDate = dOB;
    }

    public Employee(String name) {
        this(name, null);
    }
}

public class Manager extends Employee {
    private String department;

    public Manager(String name, String dept) {
        super(name);
        this.department = dept;
    }
}
```

Khi gọi `new Manager("Joe Smith", "Sales")`:

0 Khởi gán cơ bản

- 0.1 Cấp phát bộ nhớ trong heap cho đối tượng `Manager`
- 0.2 Khởi gán tất cả các biến thực thể với trị mặc định của chúng

1 Gọi constructor: `Manager("Joe Smith", "Sales")`

- 1.1 Kết nối các tham số cho constructor: `name = "Joe Smith", dept = "Sales"`
- 1.2 Không có lời gọi `this()` tường minh
- 1.3 Gọi `super(name)`, tức `Employee(String)`
 - 1.3.1 Kết nối các tham số cho constructor: `name = "Joe Smith"`

- 1.3.2 Gọi `this(name, null)`, tức `Employee(String, Date)`
 - 1.3.2.1 Kết nối các tham số cho constructor: `name = "Joe Smith", doB = null`
 - 1.3.2.2 Không có lời gọi `this()` tường minh
 - 1.3.2.3 Gọi `super()`, tức `Object()`
 - 1.3.2.3.1 Không có kết nối tham số cho constructor
 - 1.3.2.3.2 Không có lời gọi `this()` tường minh
 - 1.3.2.3.3 Không có lời gọi `super()`, do `Object` là gốc cây thừa kế
 - 1.3.2.3.4 Không có khởi gán tường minh các biến thực thể cho `Object`
 - 1.3.2.3.5 Không có lời gọi thân constructor `Object()`
 - 1.3.2.4 Khởi gán tường minh các biến thực thể cho `Employee: salary = 15000.00`
 - 1.3.2.5 Thực hiện thân của constructor `Employee(String, Date): name = "Joe Smith", birthDate = null`
- 1.3.2 - 1.3.4 Bỏ qua các bước này
- 1.3.5 Thực hiện thân của constructor `Employee(String)`, không có
- 1.4 Không có khởi gán tường minh các biến thực thể cho `Manager`
- 1.5 Thực hiện thân của constructor `Manager(String, String): department = "Sales"`

Lớp Object

Lớp `Object` là gốc của cây thừa kế (inheritance hierarchy) cho tất cả các lớp trong Java. Nếu trong khai báo của một lớp không có mệnh đề `extends` thì trình biên dịch thêm ngầm định `extends java.lang.Object` vào khai báo của nó. Tất cả các mảng có kiểu tham chiếu đều là kiểu con của `Object[]`, `Object[]` cũng là kiểu con của `Object`. Phần lớn các phương thức non-final của lớp `Object` sẽ được viết lại trong các lớp dẫn xuất từ nó. Các phương thức này tạo thành contract chung cho một đối tượng trong Java.

1. Phương thức `equals()`

Toán tử `==` thực hiện một so sánh tương đương. Như vậy, với hai tham chiếu `x` và `y`, `x == y` trả về `true` nếu và chỉ nếu `x` và `y` tham chiếu đến cùng một đối tượng.

Lớp `java.lang.Object` có phương thức `public boolean equals(Object object)` dùng so sánh hai đối tượng có bằng nhau hay không. Khi không viết lại, phương thức `equals()` trả về `true` nếu hai tham chiếu được so sánh *cùng tham chiếu đến một đối tượng*. Tuy nhiên, mục đích cụ thể của phương thức `equals()` là so sánh nội dung của hai đối tượng. Ví dụ, phương thức `equals()` của lớp `String` trả về `true` nếu và chỉ nếu tham số không `null` và là một đối tượng `String` thể hiện cùng chuỗi ký tự với đối tượng `String` đang triệu gọi phương thức `equals()`.

Một cài đặt của phương thức `equals()` phải thỏa mãn các tính chất của một quan hệ tương đương:

- Reflexive: tự phản chiếu. Nghĩa là `self.equals(self)` luôn trả về `true`.
- Symmetric: đối xứng. Với hai tham chiếu `x` và `y` bất kỳ, `x.equals(y)` là `true` nếu và chỉ nếu `y.equals(x)` cũng là `true`.
- Transitive: bắc cầu. Với ba tham chiếu `x`, `y` và `z` bất kỳ, `x.equals(y)` và `y.equals(z)` là `true` thì `x.equals(z)` cũng là `true`.
- Consistency: nhất quán. Với hai tham chiếu `x` và `y` bất kỳ, nhiều lần triệu gọi `x.equals(y)` đều cho về cùng kết quả.
- null comparison: với tham chiếu `object` không `null`, lời gọi `object.equals(null)` luôn trả về `false`.

Bạn nên viết lại phương thức `hashCode()` khi bạn viết lại phương thức `equals()`. Phương thức `hashCode()` cài đặt bằng cách dùng toán tử XOR trên các thuộc tính của `MyDate` hoặc theo cách sinh mã băm nào đó hợp lý. Phải đảm bảo rằng `hashCode()` của hai đối tượng bằng nhau có cùng trị và `hashCode()` của hai đối tượng không bằng nhau có trị khác nhau.

Các đối tượng của một lớp viết lại phương thức `equals()` có thể được dùng như các phần tử trong một collection. Nếu bạn cũng viết lại phương thức `hashCode()` của chúng, chúng có thể được dùng như các phần tử trong một `HashSet` và như các khóa trong một `HashMap`. Ngoài ra, nếu chúng cài đặt interface `Comparable`, chúng có thể được dùng như các phần tử trong một collection được sắp xếp và như các khóa trong một map được sắp xếp.

```
public class MyDate implements Comparable<MyDate> {
    private int day;
    private int month;
    private int year;

    public MyDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    @Override public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof MyDate)) return false;    // làm việc cả khi o là null.
        MyDate d = (MyDate) o;
        return day == d.day && month == d.month && year == d.year;
    }

    @Override public int hashCode() {
        return 31 * (31 * (31 * 11 + day) + month) + year;
    }

    @Override public int compareTo(MyDate o) {
        Calendar left = new GregorianCalendar(year, month + 1, day);
```

```

    Calendar right = new GregorianCalendar(o.year, o.month + 1, o.day);
    return left.compareTo(right);
}
}

```

2. Phương thức toString()

Phương thức `toString()` chuyển đổi một đối tượng thành thể hiện của đối tượng đó dưới dạng một `String`. Nó được tham chiếu bởi trình biên dịch tại những nơi có tự động chuyển kiểu chuỗi. Ví dụ, gọi `System.out.println(object)`.

Lớp `Object` định nghĩa một phương thức `toString()` mặc định, trả về tên lớp và địa chỉ tham chiếu của nó, nhưng không hữu dụng. Nhiều lớp viết lại phương thức `toString()` để cung cấp thêm thông tin hữu ích. Ví dụ, tất cả các lớp bao (wrapper class) đều viết lại phương thức `toString()` để cung cấp trị mà chúng thể hiện dưới dạng chuỗi.

Lớp bao

1. Lớp bao (wrapper class)

Java không xem các kiểu cơ bản (primitive) là đối tượng. Để thao tác trên các kiểu cơ bản như là đối tượng, Java cung cấp các lớp bao (wrapper), trong đó dữ liệu của kiểu cơ bản được bao trong một đối tượng. Mỗi kiểu dữ liệu cơ bản đều có lớp bao tương ứng trong gói `java.lang`.

Chú ý, lớp bao cài đặt các đối tượng không thể thay đổi (immutable), nghĩa là sau khi trị có kiểu cơ bản được khởi gán trong đối tượng bao, không thể thay đổi trị này. Thay đổi một đối tượng lớp bao nghĩa là tạo một đối tượng mới.

Kiểu dữ liệu cơ bản	Lớp bao
<code>boolean</code>	<code>Boolean</code>
<code>byte</code>	<code>Byte</code>
<code>char</code>	<code>Character</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>
<code>void</code>	<code>Void</code>

Từ Java 5, bạn có thể khởi tạo một đối tượng lớp bao, bằng cách gọi constructor tương ứng với trị có kiểu cơ bản cần bao. Ta gọi là boxing (đóng gói) một trị có kiểu cơ bản:

```

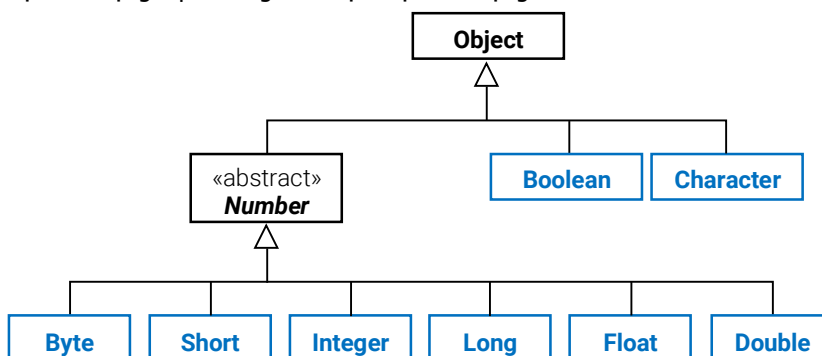
int p1 = 420;
Integer p1W = new Integer(p1); // boxing: Integer <---- int
int p2 = p1W.intValue();      // unboxing

```

Java 5 cũng cung cấp thêm nhiều phương thức hữu dụng cho lớp bao. Ví dụ lớp `Integer` có thêm các phương thức thao tác trên bit: `rotateRight`, `rotateLeft`, `bitCount`, `numberOfLeadingZeros`, `numberOfTrailingZeros`, ... Lớp `Double` bọc kiểu dữ liệu `double` chứa ba trường: `NaN` (Not-a-Number), `POSITIVE_INFINITY`, `NEGATIVE_INFINITY` và các phương thức `isNaN`, `isInfinite` để kiểm tra các điều kiện `NaN` và `Infinity`.

Lớp bao cần đến khi bạn đưa trị vào collection, các element của collection phải là đối tượng. Ngoài ra, khi bạn cần lấy trị có kiểu cơ bản từ một chuỗi, bạn cần phải "parse" chuỗi đó bằng phương thức static của lớp bao thích hợp.

```
int x = Integer.parseInt(strX);
```



2. Autoboxing

Nếu bạn thay đổi kiểu dữ liệu cơ bản thành đối tượng tương đương (gọi là boxing), bạn cần một lớp bao. Nếu bạn lấy dữ liệu có kiểu cơ bản từ đối tượng (gọi là unboxing), bạn cần gọi các phương thức của lớp bao. Từ Java 5, Java cung cấp autoboxing cho phép bạn gán và lấy dữ liệu kiểu cơ bản mà không cần lớp bao.

So sánh code bên dưới với phần 1 ở trên:

```

int p1 = 420;
Integer p1W = p1; // autoboxing
int p2 = p1W;     // autounboxing

```

Trình biên dịch sẽ tạo đối tượng bao tự động khi gán một kiểu cơ bản đến một biến kiểu lớp bao. Trình biên dịch cũng sẽ rút trích dữ liệu kiểu cơ bản khi gán một đối tượng lớp bao đến biến có kiểu cơ bản.

Điều này cũng được thực hiện khi truyền tham số đến phương thức hoặc định trị các biểu thức.

```

// auto-boxing
List list = new ArrayList();
list.add(100); // list.add(new Integer(100));
Byte b = 100; // chuyển int thành byte, rồi boxing byte thành Byte

// auto-unboxing
while (Boolean.TRUE) { ... }
if (new Integer(2) > 1) { ... }
int i = 10 + new Integer(2) * 3;

```

Từ khóa static

Từ khóa `static` khai báo các thành viên (thuộc tính, phương thức, lớp lồng) liên kết với lớp thay vì liên kết với các thể hiện (instances) của lớp. Nghĩa là chúng thuộc về lớp mà chúng được khai báo nhưng không phải là một phần của bất kỳ thể hiện nào của lớp.

1. Biến cấp lớp

Đôi khi bạn cần có biến dùng chung cho tất cả các thực thể của lớp. Ví dụ, bạn có thể dùng biến này như một cơ sở giao tiếp giữa các thực thể hoặc dùng để đếm các thực thể được tạo ra.

```
public class Count {
    private int serialNumber;
    public static int counter = 0;

    public Count() {
        counter++;
        serialNumber = counter;
    }
}
```

Biến dùng chung này có từ khóa `static`, được gọi là biến cấp lớp (class variable) hoặc thuộc tính cấp lớp (class attribute). Bạn phân biệt với biến thực thể (hay thuộc tính thành viên), riêng biệt cho từng thực thể mà không dùng chung.

Biến `static` tên `counter` được tạo và gán trị 0 trước khi các thực thể của lớp `Count` được tạo. Các thực thể của lớp `Count` được tạo và gán một trị `serialNumber` duy nhất từ `counter`, constructor của nó sẽ tăng `counter`.

Nếu biến `static` không đánh dấu `private`, bạn có thể truy cập đến nó từ bên ngoài lớp, *thông qua tên lớp*, không cần phải thông qua thực thể của lớp:

```
public class OtherClass {
    public void incrementNumber() {
        Count.counter++;
    }
}
```

2. Phương thức cấp lớp

Đôi khi bạn cần gọi các phương thức của một lớp mà không thông qua một thực thể nào của lớp. Phương thức đánh dấu `static` có thể thực hiện được điều này, chúng gọi là phương thức cấp lớp (class method).

```
public class Count2 {
    private int serialNumber;
    private static int counter = 0;

    public static int getTotalCount() {
        return counter;
    }

    public Count2() {
        counter++;
        serialNumber = counter;
    }
}
```

Như vậy, phương thức cấp lớp chính là phương thức `static`. Bạn triệu gọi chúng *thông qua tên lớp*, thay vì triệu gọi thông qua thực thể của lớp như các phương thức `non-static`:

```
public class TestCounter {
    public static void main(String[] args) {
        System.out.println("Number of counter is " + Count2.getTotalCount()); // 0
        Count2 count = new Count2();
        System.out.println("Number of counter is " + Count2.getTotalCount()); // 1
    }
}
```

Do bạn triệu gọi phương thức `static` mà không kèm theo bất kỳ thực thể nào của lớp, nên trong phương thức `static` không có tham chiếu `this`. Kết quả là phương thức `static` không thể truy cập các biến nào khác ngoại trừ: các thuộc tính `static`, biến cục bộ và tham số của phương thức `static`. Thử truy cập các thuộc tính `non-static` sẽ gây lỗi biên dịch.

Các thuộc tính `non-static` luôn kết nối đến một thực thể và chỉ có thể được truy cập thông qua tham chiếu đến thực thể đó.

Khi dùng phương thức `static`, bạn cần nhận thức các điều sau:

- Bạn không thể viết lại (overridden) một phương thức `static`, nhưng bạn có thể ẩn (hiding) chúng.

Phương thức viết lại phải `non-static`. Hai phương thức `static` có cùng signature trong cùng cây phân cấp có nghĩa là hai phương thức cấp lớp độc lập. Nếu gọi phương thức cấp lớp thông qua tham chiếu đến đối tượng của một lớp, phương thức được triệu gọi sẽ thuộc lớp của đối tượng được dùng.

```
public class Count3 extends Count2 {
    // ...
    // ẩn phương thức static getTotalCount() của Count2
    public static int getTotalCount() {
        return 0;
    }
}
```

```
Count3 c3 = new Count3();
Count2 c2 = c3;
c3.getTotalCount(); // triệu gọi getTotalCount() của Count3
c2.getTotalCount(); // triệu gọi getTotalCount() của Count2
```

- Phương thức `main()` buộc phải là `static` do JVM không thể tạo một thể hiện của lớp để gọi phương thức `main()`. JVM buộc phải gọi phương thức `main()` thông qua tên lớp.

3. Khởi tạo static

Một lớp có thể chứa một hoặc nhiều khối static (static block), bên ngoài các phương thức. Khối static chỉ *thực thi một lần* khi lớp được nạp. Nếu lớp chứa hơn một khối static, chúng được thực thi theo thứ tự xuất hiện trong lớp. Khối khởi tạo static (static initializers) này không được thừa kế trong lớp con.

```
public class FruitJar {
    public static ArrayList<Fruit> fruits;
    static {
        fruits = new ArrayList<Fruit>();
        fruits.add(new Fruit("Apple"));
        fruits.add(new Fruit("Guava"));
    }
}

public class TestStaticInit {
    public static void main(String[] args) {
        ArrayList<Fruit> fruits = FruitJar.fruits;
        for (Fruit fruit : fruits)
            System.out.println(fruit);
    }
}
```

Java cũng cho phép khối khởi tạo non-static, thực thi một lần trước tất cả các constructor.

```
public class FruitJar {
    public ArrayList<Fruit> fruits;
    {
        fruits = new ArrayList<Fruit>();
        fruits.add(new Fruit("Apple"));
        fruits.add(new Fruit("Guava"));
    }
}

public class TestStaticInit {
    public static void main(String[] args) {
        ArrayList<Fruit> fruits = new FruitJar().fruits;
        for (Fruit fruit : fruits)
            System.out.println(fruit);
    }
}
```

Từ khóa final

1. Lớp final

Java cho phép bạn áp dụng từ khóa `final` cho lớp. Khi đó, lớp sẽ *không thể thừa kế*. Tất cả phương thức trong lớp `final` ngầm định là phương thức `final`.

Ví dụ, lớp `java.lang.String` là một lớp `final`. Điều này là do an toàn, để bảo đảm nếu một phương thức tham chiếu đến một chuỗi, thì chuỗi đó có kiểu `String`, không phải thuộc về một kiểu con của lớp `String` *đã bị thay đổi* (ngăn lập trình viên thừa kế lớp `String`). Các lớp bao của các kiểu dữ liệu cơ bản cũng là `final`.

Lớp `final` nằm thấp nhất trong cây thừa kế phân cấp của nó. Vì vậy, chỉ có lớp được *định nghĩa đầy đủ*, nghĩa là các phương thức của nó đều được cài đặt, lớp mới có thể đánh dấu `final`. Lớp `abstract` và `interface`, do cần thừa kế hoặc cài đặt nên không thể `final`. Kiểu `enum` ngầm định là `final`, nên cũng không thể khai báo tường minh là `final`.

2. Phương thức final

Bạn cũng có thể đánh dấu một phương thức là `final`. Phương thức đánh dấu `final` sẽ *không thể viết lại* (overridden). Do an toàn, bạn nên tạo một phương thức `final` nếu phương thức đó có một cài đặt không nên thay đổi và quan trọng cho việc bảo đảm trạng thái nhất quán của đối tượng.

Phương thức khai báo `final` có thể được tối ưu bởi trình biên dịch. Trình biên dịch sẽ sinh mã cho một lời gọi phương thức *trực tiếp* (kết nối tĩnh), thay vì triệu gọi một phương thức ảo được xác định trong thời gian chạy. Các phương thức đánh dấu `static` hoặc `private`, ngầm định là `final`, có thể được tối ưu bởi trình biên dịch khi chúng được đánh dấu `final`, do kết nối động không thể áp dụng trong trường hợp này.

Các tham số hình thức của một phương thức có thể được khai báo với `final`. Một tham số `final` là một biến `final` trống, cho đến khi nó được gán khi gọi phương thức. Sau đó trị của nó không thể thay đổi suốt vòng đời của biến.

```
public double bake(int quantity, final double price, final Pizza pizza) {
    pizza.meat = "chicken"; // cho phép
    pizza = new Pizza("pork"); // gán lại, không cho phép
    price /= 2.0; // gán lại, không cho phép
    return quantity * price;
}
```

3. Biến final

Nếu một biến được đánh dấu là `final`, nó được *xem như một hằng số* mặc dù vẫn là biến, không thể thay đổi trị của nó một khi nó đã khởi tạo, nghĩa là chỉ gán trị cho nó một lần khi khởi tạo. Thử gán trị lại cho biến `final` sẽ gây lỗi biên dịch.

```
public class Bank {
    private static final double DEFAULT_INTEREST_RATE = 3.2;
    // các khai báo khác
}
```

Nếu đánh dấu một biến kiểu tham chiếu (kiểu lớp nào đó) là `final`, biến này không thể tham chiếu đến một đối tượng nào khác. Tuy nhiên, bạn có thể thay đổi nội dung của đối tượng, do chỉ có tham chiếu chỉ đến nó là `final`.

Một biến `final` không phải khởi tạo khi khai báo gọi là biến `final` trống (blank final variable), khởi tạo được hoãn nhưng phải thực hiện khởi tạo trước khi dùng biến. Một biến `final` trống chỉ được *khởi tạo một lần trong một constructor*.

Các biến static, biến cục bộ kể cả tham số của phương thức, cũng có thể áp dụng `final`. Các biến `final` static thường dùng để định nghĩa các hằng có tên (named constants hoặc manifest constant), ví dụ `Integer.MAX_VALUE`.

Các biến được định nghĩa trong một interface là `final` ngầm định.

```
public class Customer {
    private final long customerID;

    public Customer() {
        customerID = createID();
    }

    public long getID() {
        return customerID;
    }

    private long createID() {
        return ... // sinh ID mới
    }
    // các khai báo khác
}
```

Một tham số của phương thức có thể đánh dấu `final`, gọi là tham số `final` (final parameter). Một tham số `final` cũng được xem như một biến `final` trống, nghĩa là nó sẽ trống (không được khởi tạo) cho đến khi có trị gán cho nó, khi truyền tham số trong gọi phương thức, sau đó trị của biến không thể thay đổi suốt vòng đời của biến.

```
public double calcPrice(int quantity, final double price) {
    price = price / 2.0; // lỗi, không cho phép
    return quantity * price;
}
```

Kiểu liệt kê (Enumrated Type)

Trong lập trình, đôi khi bạn phải làm việc với một tập hữu hạn các tên thể hiện tập trị của một thuộc tính. Ví dụ, để thể hiện thuộc tính chất (suit) của một bộ bài, bạn phải tạo một tập các tên: SPADES, HEARTS, CLUBS, và DIAMONDS. Lúc này, bạn dùng kiểu liệt kê. Chú ý phân biệt kiểu liệt kê với lớp `java.util.Enumeration`, chúng không liên quan với nhau.

1. Kiểu liệt kê cũ

Kiểu liệt kê được tạo bằng cách dùng hằng nguyên cho tên của mỗi trị được liệt kê.

```
package cards.domain;
public class PlayingCard {
    public static final int SUIT_SPADES = 0;
    public static final int SUIT_HEARTS = 1;
    public static final int SUIT_CLUBS = 2;
    public static final int SUIT_DIAMONDS = 3;

    private int suit;
    private int rank;

    public PlayingCard(int suit, int rank) {
        this.suit = suit;
        this.rank = rank;
    }

    public int getSuit() {
        return suit;
    }

    public String getSuitName() {
        String name = "";
        switch (suit) {
            case SUIT_SPADES: name = "Spades"; break;
            case SUIT_HEARTS: name = "Hearts"; break;
            case SUIT_CLUBS: name = "Clubs"; break;
        }
    }
}
```

```

    case SUIT_DIAMONDS: name = "Diamonds"; break;
    default: System.err.println("Invalid suit.");
}
return name;
}
}

```

Kiểu liệt kê cũ có một số vấn đề:

- Không an toàn kiểu. Do thuộc tính suit là một int, bạn có thể truyền một trị int bất kỳ cho suit, áp dụng các toán tử số học cho suit, dẫn đến trị vô nghĩa, không an toàn.
- Không namespace. Vì vậy, khi đặt tên, bạn phải thêm một chuỗi tiền tố (SUIT_) cho mỗi trị liệt kê int để tránh đụng độ tên với kiểu int liệt kê khác.
- Hằng số thời gian biên dịch. Do kiểu int liệt kê là các hằng trong thời gian biên dịch, nếu một hằng mới được thêm vào giữa hai hằng đã tồn tại hoặc thứ tự giữa các hằng thay đổi, client phải biên dịch lại.
- Trị chứa ít thông tin. Do chúng chỉ là trị int, bạn chỉ in ra được số, không thể hiện thông tin một cách thân thiện, vì vậy cần viết thêm phương thức getSuitName().

2. Kiểu liệt kê mới

Java 5 cung cấp kiểu liệt kê an toàn kiểu, có thuộc tính và có phương thức, giống một lớp thông thường.

```

package cards.domain;
public enum Suit {
    // liệt kê các hằng enum trong danh sách tách biệt bởi dấu phẩy
    SPADES   ("Spades") { @Override int getValue() { return 1; } },
    HEARTS   ("Hearts") { @Override int getValue() { return 2; } },
    CLUBS    ("Clubs") { @Override int getValue() { return 3; } },
    DIAMONDS ("Diamonds") { @Override int getValue() { return 4; } };
    // phần tùy chọn: constructor, field và method
    private final String name;
    private Suit(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    abstract int getValue();
}

```

Các hằng của enum ngầm định là final và static.

Constructor của enum luôn luôn dùng mức truy cập private, vì vậy không thể tạo một đối tượng kiểu enum với toán tử new.

Tham số của constructor được cung cấp sau mỗi khai báo trị. Ví dụ, chuỗi "Spades" là tham số constructor của enum cho trị SPADES.

```

package cards.domain;
public class PlayingCard {
    private Suit suit;
    private int rank;

    public PlayingCard(Suit suit, int rank) {
        this.suit = suit;
        this.rank = rank;
    }

    public Suit getSuit() {
        return suit;
    }

    public int getRank() {
        return rank;
    }
}

package cards.test;
import cards.domain.*;
public class TestPlayingCard {
    public static void main(String[] args) {
        PlayingCard card = new PlayingCard(Suit.SPADES, 2);
        System.out.println("Card is the " + card.getRank() + " of " + card.getSuit().getName());
    }
}

```

Giống truy cập thành viên static, truy cập trị kiểu liệt kê cũng thông qua tên namespace. Ví dụ, truy cập trị SPADES của kiểu Suit bằng cách dùng Suit.SPADES.

Java 5 cung cấp khả năng *static import* cho phép truy cập đến kiểu liệt kê mà không cần tên namespace.


```
package cards.test;
import cards.domain.*;
import static cards.domain.Suit.*;
public class TestPlayingCard {
    public static void main(String[] args) {
        PlayingCard card = new PlayingCard(SPADES, 2);
        System.out.println("Card is the " + card.getRank() + " of " + card.getSuit().getName());
    }
}
```

Một ví dụ khác của static import, dùng khi gọi các phương thức static của một lớp:

```
import static java.lang.Math.*;
public class StaticImportTest {
    public static void main(String[] args) {
        System.out.printf("sqrt(900.0) = %.1f\n", sqrt(900.0));
        System.out.printf("ceil(-9.8) = %.1f\n", ceil(-9.8));
        System.out.printf("PI = %f\n", PI);
    }
}
```

Lớp trừu tượng

Khi làm việc với nhiều đối tượng thuộc các lớp có những đặc điểm chung, ta thường trừu tượng hóa chúng thành một lớp cha chung. Sau đó, áp dụng tính đa hình, bạn có thể đưa các đối tượng này vào một collection không đồng nhất có kiểu cha chung để dễ quản lý, ví dụ duyệt collection không đồng nhất và triệu gọi các phương thức ảo từ chúng.

Chú ý rằng lớp cha chung này chỉ *mang tính khái niệm*, vì vậy một số phương thức của nó cũng mang tính khái niệm, không nhất thiết phải được cài đặt.

Java cho phép chỉ định trong một lớp cha các phương thức chỉ khai báo mà không cần cung cấp cài đặt, gọi là các phương thức trừu tượng (abstract method). Một lớp có một hoặc nhiều phương thức trừu tượng gọi là lớp trừu tượng (abstract class). Cài đặt cho các phương thức trừu tượng chỉ được cung cấp trong lớp con cụ thể.

Ngoài các phương thức trừu tượng (không bắt buộc) phải có, lớp trừu tượng cũng có các thuộc tính, các phương thức được cài đặt và constructor.

Do constructor không được thừa kế nên không khai báo abstract được. Constructor của lớp trừu tượng thường được đánh dấu là protected vì nó chỉ có ý nghĩa khi được triệu gọi từ lớp con cụ thể.

Phương thức static (non-private) được thừa kế nhưng không thể viết lại nên cũng không khai báo abstract được.

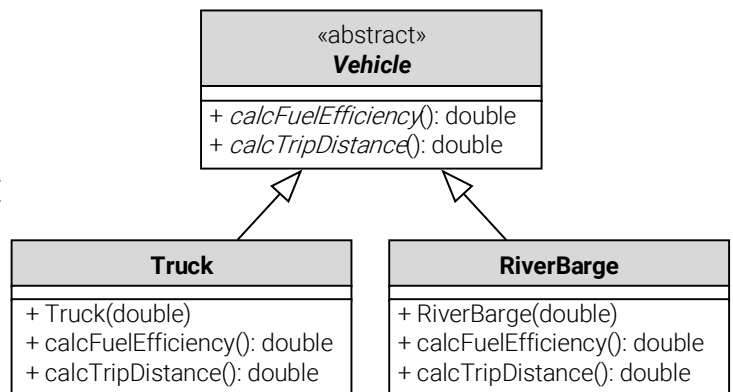
Do lớp trừu tượng chỉ mang tính khái niệm, trình biên dịch ngăn cản tạo thể hiện của lớp trừu tượng. Lớp trừu tượng có thể thừa kế từ một lớp cụ thể, nhưng không nên thực hiện điều đó.

Ví dụ, để quản lý các loại phương tiện vận chuyển (Truck, RiverBarge), ta đưa chúng vào collection không đồng nhất có kiểu Vehicle. Lớp trừu tượng Vehicle được trừu tượng hóa từ các lớp con Truck, RiverBarge. Nó chỉ mang tính khái niệm, nên các phương thức trừu tượng của nó (calcFuelEfficiency() và calcTripDistance()) không cần (và không thể) cài đặt.

```
public abstract class Vehicle {
    public abstract double calcFuelEfficiency();
    public abstract double calcTripDistance();
}

public class Truck extends Vehicle {
    public Truck(double maxLoad) { ... }
    @Override public double calcFuelEfficiency() {
        // tính lượng xăng tiêu thụ của truck
    }
    @Override public double calcTripDistance() {
        // tính khoảng cách đường bộ
    }
}

public class RiverBarge extends Vehicle {
    public RiverBarge(double maxLoad) { ... }
    @Override public double calcFuelEfficiency() {
        // tính lượng xăng tiêu thụ của river barge
    }
    @Override public double calcTripDistance() {
        // tính khoảng cách đường thủy
    }
}
```



Dùng trong hệ thống shipping, tạo báo cáo tuần về các phương tiện vận chuyển và lượng xăng dự kiến sẽ tiêu thụ:

```
public class FuelNeedsReport {
    private Company company;

    public FuelNeedsReport(Company company) {
        this.company = company;
    }
}
```

```

    }

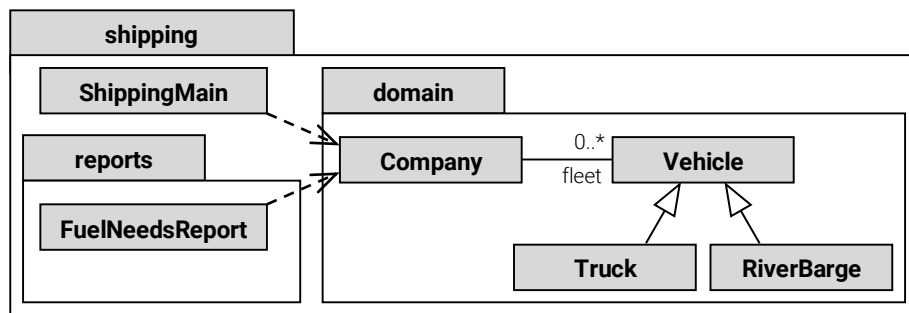
    public void generateReport(PrintStream out) {
        double fuel;
        double total_fuel = 0.0;
        final ArrayList<Vehicle> list = company.getFleet();
        for (Vehicle v : list) {
            fuel = v.calcTripDistance() / v.calcFuelEfficiency();
            out.println("Vehicle " + v.getName() + " needs " + fuel + " liters of fuel.");
            total_fuel += fuel;
        }
        out.println("Total fuel needs is " + total_fuel + " liters.");
    }
}

public class ShippingMain {
    public static void main(String[] args) {
        Company c = new Company();

        c.addVehicle(new Truck(15000.0));
        c.addVehicle(new RiverBarge(750000.0));

        FuelNeedsReport report = new FuelNeedsReport(c);
        report.generateReport(System.out);
    }
}

```



Interface

1. Interface

Một interface của một lớp được xem như một giao kết (contract) về các dịch vụ mà lớp đó sẽ cung cấp. Interface chỉ khai báo các phương thức mà không cung cấp bất kỳ cài đặt nào, mục đích là đẩy mạnh tính móc nối lỏng (loose coupling) trong thiết kế. Một lớp cụ thể cài đặt (implements) một interface bằng cách định nghĩa tất cả các phương thức được khai báo trong interface đó. Nếu một lớp cài đặt interface, nhưng không cung cấp cài đặt đầy đủ các phương thức của interface, lớp đó phải được khai báo abstract.

Tên interface là tính từ, thường kết thúc với "able", hoặc là danh từ. Tên interface dùng camel case, như tên lớp.

Interface là trừu tượng theo định nghĩa, nên không thể tạo được thể hiện từ một interface và cũng không có constructor.

Cú pháp khai báo của interface:

```

[modifier] interface <tên interface> [mệnh đề extends (nếu có)] {
    [các khai báo hằng]
    [các khai báo phương thức trừu tượng]
    [các khai báo Lớp Lồng]
    [các khai báo interface Lồng]
}

```

Các phương thức trong interface mặc định là public và abstract, các bổ từ này thường được bỏ qua, dù vẫn có thể khai báo tường minh.

```

interface IStack {
    void push(Object item);
    Object pop();
}

class StackImpl implements IStack {
    protected Object[] stackArray;
    protected int tos; // top of stack

    public StackImpl(int capacity) {
        stackArray = new Object[capacity];
        tos = -1;
    }
}

```

```

@Override public void push(Object item) {
    stackArray[++tos] = item;
}

@Override public Object pop() {
    Object objRef = stackArray[tos];
    stackArray[tos--] = null;
    return objRef;
}

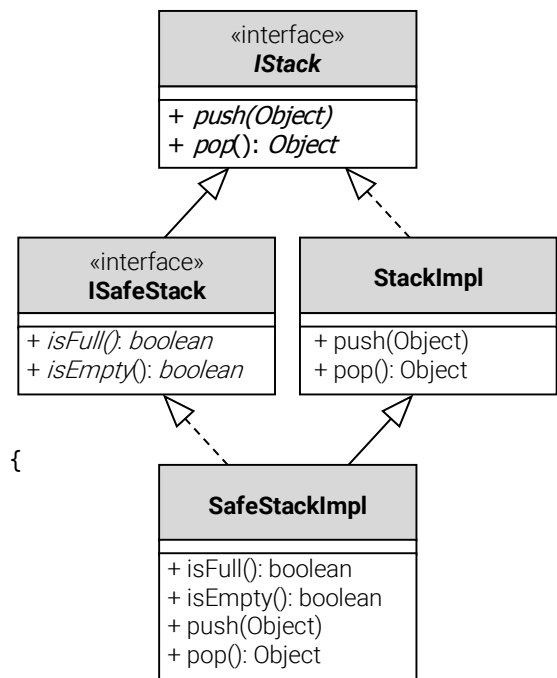
public Object peek() {
    return stackArray[tos];
}
}

interface ISafeStack extends IStack {
    boolean isEmpty();
    boolean isFull();
}

class SafeStackImpl extends StackImpl implements ISafeStack {
    public SafeStackImpl(int capacity) { super(capacity); }
    @Override public boolean isEmpty() { return tos < 0; }
    @Override public boolean isFull() {
        return tos >= stackArray.length-1;
    }
}

public class StackUser {
    public static void main(String[] args) {
        SafeStackImpl safeStackRef = new SafeStackImpl(10);
        StackImpl stackRef = safeStackRef;
        ISafeStack isafeStackRef = safeStackRef;
        IStack istackRef = safeStackRef;
        Object objRef = safeStackRef;
        safeStackRef.push("Dollars");
        stackRef.push("Kroner");
        System.out.println(isafeStackRef.pop()); // Kroner
        System.out.println(istackRef.pop()); // Dollars
        System.out.println(objRef.getClass()); // class SafeStackImpl
    }
}

```



Tất cả các thuộc tính của interface là public, static và final. Nói cách khác, chúng được xem như là hằng số. Ví dụ:

```

interface Constants {
    double PI_APPROXIMATION = 3.14;
    String AREA_UNITS = "sq.cm.";
    String LENGTH_UNITS = "cm.";
}

public class Client implements Constants {
    public static void main(String[] args) {
        double radius = 1.5;
        // (1) truy cập trực tiếp
        System.out.printf("Area of circle is %.2f %s\n", PI_APPROXIMATION * radius * radius, AREA_UNITS);
        // (2) truy cập bằng tên đầy đủ
        System.out.printf("Circumference of circle is %.2f %s\n",
            2.0 * Constants.PI_APPROXIMATION * radius, Constants.LENGTH_UNITS);
    }
}

```

Giống như lớp trừu tượng, bạn có thể áp dụng kết nối động với interface, khai báo đối tượng kiểu interface, ép kiểu từ một interface hoặc đến một interface, xác định kiểu interface được cài đặt bằng toán tử instanceof.

Có ba kiểu quan hệ thừa kế khi dùng các lớp và interface:

- Đơn thừa kế giữa hai lớp: một lớp thừa kế một lớp khác.
- Đa thừa kế giữa các interface: một interface có thể thừa kế một hoặc nhiều interface khác.

```
interface Transformable extends Scalable, Translatable, Rotatable { }
```

Trong trường hợp một interface thừa kế từ nhiều interface, mà các interface đó định nghĩa một phương thức giống nhau. Do các interface chỉ khai báo phương thức mà không định nghĩa nên interface con (và các lớp cài đặt nó) chỉ dùng một phương thức duy nhất trong các phương thức giống nhau đó:

```

interface BaseInterface1 { String getName(); }
interface BaseInterface2 { String getName(); }
interface IMyInterface extends IBase1, IBase2 { }

```

```
public class TestInf implements IMyInterface {
    @Override public String getName() {
        return "name";
    }
}
```

- Cài đặt nhiều interface: một lớp có thể cài đặt *một hoặc nhiều* interface.

Bạn dùng một interface để:

- Khai báo các phương thức mà một hay nhiều lớp dự kiến sẽ cài đặt. Điều này giúp dễ chia nhỏ, phân công hiện thực lớp.
- Phác thảo giao diện lập trình của một đối tượng mà không cần trình bày rõ thân của lớp, tương tự như tập tin header (.h) của C/C++. Điều này thường dùng trong RMI, hoặc khi chuyển một gói các lớp cho người phát triển khác.
- Nắm bắt các điểm tương đồng giữa các lớp không quan hệ với nhau mà không cần ép chúng phải có mối liên kết. Nhiều lớp *thuộc các cây thừa kế khác nhau* có thể cài đặt cùng một interface.
- Giả lập đa thừa kế bằng cách khai báo một lớp cài đặt vài interface.
- Cài đặt interface phù hợp với dịch vụ được cung cấp. Ví dụ, `Arrays.sort()` sẽ sắp xếp một mảng nếu lớp thành viên của mảng có cài đặt interface `Comparable<T>`.

2. Interface đánh dấu

Interface với thân rỗng, không khai báo phương thức nào, được sử dụng như một interface đánh dấu (tagging/marker interface, hoặc ability interface) giúp trình biên dịch nhận biết và xử lý đặc biệt các lớp cài đặt interface này. Khi cài đặt interface đánh dấu, thực thể của lớp trở thành một thực thể hợp lệ của interface đó, toán tử `instanceof` có thể xác định điều này để xử lý phù hợp.

Một số interface đánh dấu trong Java API: `java.lang.Cloneable`, `java.io.Serializable`, `java.util.EventListener`, `java.util.RandomAccess`.

```
// trước khi sắp xếp list, cần xác định nó cho phép truy cập ngẫu nhiên
// nếu không, tạo bản sao truy cập ngẫu nhiên cho nó
List list = ...; // ArrayList cài đặt RandomAccess, LinkedList thì không
if (list.size() > 2 && !(list instanceof RandomAccess))
    list = new ArrayList(list);
sortList(list);
```

3. Phương thức static trong interface

Trước Java 8, nếu cần phương thức factory để sinh đối tượng cài đặt mặc định interface đó hoặc để tạo các lớp helper hỗ trợ cài đặt một interface, Java API phải cung cấp các cặp interface/companion class (lớp đồng hành) như `Collection/Collections`, `Path/Paths`, ...

Từ Java 8, vấn đề này được giải quyết tốt hơn bằng cách cho phép phương thức static trong interface:

```
interface Calculator {
    static Calculator getInstance() {
        return new BasicCalculator();
    }
    int add(int left, int right);
    int subtract(int left, int right);
    int divide(int number, int divisor);
    int multiply(int left, int right);
}

class BasicCalculator implements Calculator {
    @Override public int add(int left, int right) {
        return left + right;
    }

    @Override public int subtract(int left, int right) {
        return left - right;
    }

    @Override public int divide(int number, int divisor) {
        if (divisor == 0)
            throw new IllegalArgumentException("divisor can't be zero.");
        return number / divisor;
    }

    @Override public int multiply(int left, int right) {
        return left * right;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calculator = Calculator.getInstance();
        System.out.println(calculator.divide(27, 4));
    }
}
```

```

    }
}

```

4. Phương thức default trong interface

Java 8 cũng cho phép mở rộng interface bằng cách đưa vào interface phương thức *cài đặt mặc định*, gọi là phương thức default. Gọi là cài đặt mặc định vì nó chỉ định cách phương thức được thực hiện khi lớp cài đặt interface *không viết lại* phương thức này. Đôi khi, phương thức default cũng được gọi là phương thức *optional* hoặc *extension*, do bạn *không bắt buộc* lớp cài đặt interface trước đây phải cài đặt thêm *phương thức mở rộng* này.

```

interface Calculator {
    static Calculator getInstance() {
        return new BasicCalculator();
    }
    int add(int left, int right);
    int subtract(int left, int right);
    int divide(int number, int divisor);
    int multiply(int left, int right);
    // phương thức mới thêm vào API, không bắt buộc các lớp cài đặt Calculator phải cài đặt thêm
    default int remainder(int number, int divisor) {
        return subtract(number, multiply(divisor, divide(number, divisor)));
    }
}

```

```

class BasicCalculator implements Calculator {
    // ...
}

public class Main {
    public static void main(String[] args) {
        Calculator calculator = Calculator.getInstance();
        System.out.println(calculator.remainder(27, 4));
    }
}

```

Ta cũng nhận thấy nhờ phương thức default, các phương thức hỗ trợ (helper method) cho API có thể được khai báo trong interface, thay vì phải khai báo như các phương thức static trong một lớp helper tách biệt như trước đây. Điều này giúp lập trình viên dễ dàng định vị và gọi các phương thức hỗ trợ.

Phương thức default cũng giúp giải quyết vấn đề interface có nhiều phương thức, nhưng khi cài đặt người dùng chỉ cài đặt cho một vài phương thức, còn lại phải cài đặt rỗng. Trước đây, vấn đề này được giải quyết bằng lớp adapter, cài đặt interface bằng cách cài đặt rỗng tất cả các phương thức, người dùng thừa kế lớp này và viết lại phương thức cần dùng. Dùng phương thức default đơn giản hơn. Ví dụ:

```

public interface MouseListener {
    default void mouseClicked(MouseEvent event) { }
    default void mousePressed(MouseEvent event) { }
    // ...
}

```

Trước đây Java thực hiện đa thừa kế bằng interface, gọi là đa thừa kế cấp độ kiểu. Từ Java 8, do cho phép phương thức default trong interface, Java có đa thừa kế cấp độ hành vi. Điều này đưa đến vấn đề *thừa kế hình thoi* (diamond inheritance) hoặc *thừa kế nhiều đường* (multi path inheritance) như trong C++. Khi đó, việc chọn phương thức nào sẽ chạy dựa trên ba quy tắc:

- Phương thức khai báo trong lớp ưu tiên hơn phương thức khai báo trong interface.

```

interface A {
    default void foo() {
        System.out.println("A::foo()");
    }
}

class App implements A {
    @Override public void foo() {
        System.out.println("App::foo()"); // được gọi
    }
}

public class Main {
    public static void main(String[] args) {
        new App().foo();
    }
}

```

- Nếu không, interface cụ thể nhất được chọn.

```

interface A {
    default void foo() {
        System.out.println("A::foo()");
    }
}

```

```

}

interface B { }

interface C extends A {
    @Override default void foo() {
        System.out.println("C::foo()"); // được gọi
    }
}

class App implements A, B, C { }

public class Main {
    public static void main(String[] args) {
        new App().foo();
    }
}

```

- Nếu không, lớp phải gọi tường minh cài đặt mà nó muốn.

```

interface A {
    default void foo() {
        System.out.println("A::foo()");
    }
}

interface B extends A {
    @Override default void foo() {
        System.out.println("B::foo()"); // được gọi
    }
}

interface C extends A {
    @Override default void foo() {
        System.out.println("C::foo()");
    }
}

class App implements A, B, C {
    @Override public void foo() {
        B.super.foo();
    }
}

public class Main {
    public static void main(String[] args) {
        new App().foo();
    }
}

```

5. Functional interface

Trong Java 8, functional interface là interface chỉ có một phương thức trừu tượng (SAM – Single Abstract Method). Bạn có thể đánh dấu chúng bằng annotation `@FunctionalInterface`. Thể hiện của functional interface sẽ được tạo với biểu thức Lambda.

`@FunctionalInterface`

```

interface InterfaceName {
    // chỉ có duy nhất một phương thức abstract
    public String doAbstractTask(String s);
    // cho phép nhiều phương thức default, chúng có thể gọi phương thức abstract
    default public String performTask1() {
        return doAbstractTask("task 1");
    }
    default public String performTask2() {
        return doAbstractTask("task 2");
    }
}

```

// tạo thể hiện của functional interface bằng biểu thức Lambda

```

InterfaceName interfaceName = s -> s.toUpperCase();
System.out.println(interfaceName.performTask1());

```

Functional interface sẽ được thảo luận chi tiết trong phần Biểu thức Lambda.

Lớp lồng

Lớp được khai báo nhưng bên trong một lớp khác gọi là lớp lồng (nested). Mục đích của lớp lồng không chỉ đơn giản là che giấu một lớp với bên ngoài mà còn được sử dụng hiệu quả trong nhiều tình huống. Lớp lồng có thể được khai báo không chỉ trong định nghĩa lớp bọc ngoài, mà còn trong phương thức của một lớp hoặc thậm chí trong tham số của một phương thức.

Lớp lồng được sử dụng để:

- Nhóm logic các lớp có liên quan với nhau.

Đôi khi một lớp chỉ được dùng cho một lớp khác, vì vậy nó không có "ý nghĩa bên ngoài". Ví dụ, các lớp listener có thể là lớp lồng của lớp mà nó lắng nghe sự kiện vì lớp listener chỉ liên quan đến lớp đó.

- Cung cấp đóng gói tăng cường.

Lớp lồng có quyền truy cập tất cả thành viên của lớp bọc ngoài (outer), nhưng nó lại vô với bên ngoài.

- Dễ đọc và dễ bảo trì code.

Lớp lồng thường có quan hệ với lớp bọc ngoài, code chung với lớp bọc ngoài nên dễ theo dõi và bảo trì.

1. Lớp lồng static

Sử dụng từ khóa `static` ở phía trước khai báo lớp. Giống như phương thức `static`, lớp lồng static không thể tham chiếu thành viên non-static của lớp bọc ngoài. Để truy cập lớp lồng static, bạn cần phải sử dụng tên đầy đủ, theo cú pháp như ví dụ sau:

```
class OuterClassName{
    static class StaticNestedClassName {
    }
}
```

```
OuterClassName.StaticNestedClassName object = new OuterClassName.StaticNestedClassName ();
```

2. Lớp lồng non-static

Còn gọi là lớp nội (inner class). Lớp nội kết hợp với một thể hiện của lớp bọc ngoài, vì vậy nó có quyền truy cập trực tiếp vào các thuộc tính và phương thức của lớp bọc ngoài. Nói cách khác, lớp nội được xem như thành viên của lớp bọc ngoài, gọi là lớp thành viên (member class).

Một lớp nội không thể định nghĩa bất kỳ thành viên static nào vì nó luôn liên kết với một thể hiện (instance). Một thể hiện của một lớp bên trong chỉ có thể tồn tại trong một thể hiện của lớp bên ngoài. Bạn có thể tạo ra nhiều thể hiện của cùng một lớp nội bên trong một thể hiện duy nhất của lớp bọc ngoài. Để tạo một thể hiện của lớp nội, bạn phải có một đối tượng của lớp bọc ngoài.

```
public class DynamicEvensGenerator {
    private final static int SIZE = 25;
    private int[] arrayOfInts = new int[SIZE];
    public DynamicEvensGenerator() {
        for (int i = 0; i < SIZE; i++) {
            arrayOfInts[i] = (int) (Math.random() * SIZE);
        }
    }

    public void printEvens() {
        InnerEvensIterator iterator = this.new InnerEvensIterator();
        while (iterator.hasNext()) {
            int returnValue = iterator.getNext();
            if (returnValue != -1) {
                System.out.print(returnValue + " ");
            }
        }
        System.out.println();
    }

    private class InnerEvensIterator {
        private int next = 0;
        public boolean hasNext() {
            return (next <= SIZE - 1);
        }

        public int getNext() {
            int retValue = arrayOfInts[next++];
            return retValue % 2 == 0 ? retValue : -1;
        }
    }

    public static void main(String s[]) {
        DynamicEvensGenerator numbers = new DynamicEvensGenerator();
        numbers.printEvens();
    }
}
```

Lưu ý trong cú pháp tạo lớp nội, bạn sử dụng lời gọi `this.new`, với `this` tham chiếu như thể hiện của lớp bọc ngoài. Nếu lớp nội được khai báo với từ khóa `public`, bạn có thể tạo thể hiện của nó bên ngoài lớp bọc nó. Ví dụ:

```

class Outer {
    private int counter = 0;
    public class Inner {
        public void incCount() {
            counter++;
        }
    }

    public int getCount() {
        return counter;
    }
}

class Other {
    public void foo() {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
        inner.incCount();
        System.out.println(outer.getCount());
    }
}

```

Khi một biến khai báo bên trong một tầm vực cụ thể (khối, phương thức, lớp nội) có cùng tên với một biến khai báo trong tầm vực của lớp bọc ngoài, ta gọi biến của lớp bọc ngoài bị che (shadowed). Điều này cho phép vì bạn có thể phân biệt chúng.

Một số điểm quan trọng khi sử dụng lớp nội:

- Tên của lớp nội phải khác với tên của lớp bọc ngoài.
 - Khi biên dịch lớp bọc ngoài, trình biên dịch tạo ra một tập tin .class riêng cho mỗi lớp nội của nó, có dạng: OuterClassName\$InnerClassName.
 - Lớp nội có thể sử dụng biến lớp và biến thực thể của các lớp bọc ngoài và biến cục bộ của khối bọc ngoài.
 - Lớp nội có thể được khai báo bằng cách sử dụng bộ từ truy cập bất kỳ. Nếu dùng private, lớp nội chỉ có thể được truy cập trong phạm vi lớp bọc ngoài.
 - Một lớp nội có thể là một interface. Một lớp nội khác có thể cài đặt interface này.
 - Một lớp nội có thể abstract.
 - Không thể khai báo một thành viên static bên trong một lớp nội trừ khi lớp nội được khai báo static.
 - Một lớp nội muốn sử dụng một biến static phải được khai báo static.
 - Một lớp static không được có lớp lồng static.
- Các loại khác của lớp lồng: lớp cục bộ và lớp đều thuộc lớp nội.

3. Lớp cục bộ (local)

Một lớp nội định nghĩa bên trong một khối được gọi là một nội bộ cục bộ (local inner) hoặc đơn giản là lớp cục bộ (local). Khối ở đây thường là phương thức, nhưng cũng có thể là khối khởi tạo static (static initializer) hoặc constructor.

```

class OuterClass {
    public OuterClass() {
        class Local {
            Local() { }
            String say() { return "Hello"; }
        }
        System.out.println(new Local().say());
    }

    public void instanceMethod() {
        new OuterClass();
    }
}

```

Một số điểm quan trọng khi sử dụng lớp cục bộ:

- Lớp cục bộ chỉ hiển thị và có thể sử dụng được trong khối mã mà nó được định nghĩa.
- Ngoài việc truy cập các thuộc tính được định nghĩa bởi lớp chứa, các lớp cục bộ có thể truy cập biến cục bộ, các tham số của phương thức, hoặc các tham số exception nằm trong tầm vực định nghĩa phương thức, miễn là chúng được khai báo với từ khóa final.
- Lớp cục bộ không thể dùng từ khóa new và super.
- Lớp cục bộ không thể chứa các thuộc tính, phương thức, lớp khai báo static. Do các interface lồng ngầm định là static, lớp cục bộ không chứa định nghĩa interface lồng.
- Lớp cục bộ không thể khai báo với các bộ từ public, protected, private và static. Những bộ từ này chỉ được sử dụng trên các thành viên của lớp và không được phép trên khai báo lớp cục bộ.
- Lớp cục bộ không thể có cùng tên với bất kỳ lớp bọc ngoài nào.
- Không thể định nghĩa interface cục bộ.
- Lớp cục bộ có thể sử dụng biến cục bộ final hoặc các tham số của phương thức có thể nhìn thấy trong tầm vực mà chúng được định nghĩa.

4. Lớp vô danh (anonymous)

Lớp cục bộ không có tên gọi là lớp nội vô danh (inner anonymous), được dùng khi bạn chỉ cần tạo một thể hiện của lớp cục bộ và thường dùng ngay. Một ví dụ điển hình là dùng lớp vô danh khi tạo thread.

```
new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                sleep(1000);
                System.out.print(".");
            }
        } catch (InterruptedException ex) { }
    }
}).start();
```

Một ví dụ điển hình khác là dùng lớp vô danh trong Java Collection Framework:

```
ArrayList<String> friendList = new ArrayList<String>() { {
    add("Donald Trump");
    add("Barack Obama");
    add("Bill Clinton");
} };
```

Một số điểm quan trọng khi sử dụng lớp vô danh:

- Lớp vô danh không thể có constructor vì không có tên lớp.

Nếu bạn muốn khởi tạo các biến thực thể của lớp vô danh phải dùng một setter:

```
new Thread() {
    String message;

    Thread init(String message) {
        this.message = message;
        return this;
    }

    @Override public void run() {
        System.out.println(message);
    }
}.init("TEST").start();
```

- Lớp vô danh không thể định nghĩa thuộc tính, các phương thức, hoặc các lớp static.

- Không thể định nghĩa interface lồng trong một lớp vô danh vì các interface này ngầm định là static.

- Không thể định nghĩa một interface vô danh.

- Giống như lớp cục bộ, lớp vô danh không thể dùng bộ từ public, protected, private, hoặc static.

- Do không có tên, khi biên dịch lớp EnclosingClassName chứa lớp vô danh, trình biên dịch tạo ra hai lớp: EnclosingClassName.class và EnclosingClassName\$1.class

Biểu thức Lambda

Biểu thức Lambda

Biểu thức Lambda là một cấu trúc nền tảng và phổ biến trong các ngôn ngữ lập trình hàm (functional programming language) như Lisp, Haskell và OCaml. Biểu thức Lambda được đưa vào từ Java 8, cho phép viết một phương thức như một biểu thức.

Cú pháp như sau:

```
( paramlist ) -> { statements }
```

Bên trái của `->` là danh sách tham số (không cần khai báo kiểu dữ liệu) của phương thức.

Bên phải của `->` là thân của phương thức.

Kỹ thuật này cho phép viết ngắn gọn các kiểu lồng (nested), thường sử dụng khi viết các phương thức callback hoặc handler. Trước Java 8, chúng thường được viết bằng lớp nội vô danh nhưng khá phức tạp.

Ví dụ:

```
// (1) lớp nội vô danh, đóng gói nhanh hành vi được định nghĩa vào đối tượng r
```

```
Runnable r = new Runnable() {
    @Override public void run() {
        System.out.println("Hello World");
    }
};
```

```
// (2) biểu thức Lambda tương đương trả về đối tượng r
```

```
Runnable r = () -> System.out.println("Hello World");
```

Biểu thức Lambda làm phương thức trở thành *first-class function*, dễ dàng truyền như tham số. Nhắc lại một số khái niệm của lập trình hàm:

- *first-class function* là hàm giống như đối tượng, có thể sử dụng hàm như tham số, có thể trả hàm về từ một hàm khác và có thể gán hàm cho một biến.

- *higher-order function* là một hàm có thể nhận hàm khác như một tham số hoặc trả về một hàm như một kết quả.

Ví dụ: sắp xếp danh sách với độ dài của tên tăng dần.

```
// (1) truyền phương thức như tham số, dùng lớp nội vô danh
```

```
List<String> names = Arrays.asList("abc", "abcd", "ab");
```

```
Collections.sort(names, new Comparator<String>() {
    @Override public int compare(String o1, String o2) {
        return o1.length() - o2.length();
    }
});
```

```
// (2) truyền phương thức như tham số, dùng biểu thức Lambda
```

```
Collections.sort(names, (first, second) -> first.length() - second.length());
```

```
System.out.println(names); // [ab, abc, abcd]
```

Từ Java 8, trình biên dịch được cải thiện khả năng *suy kiểu* (type inference) từ ngữ cảnh, Trong ví dụ trên, ta không cần cung cấp rõ kiểu cho danh sách tham số ở vế trái của biểu thức Lambda, trình biên dịch sẽ suy ra kiểu cần thiết. Trong trường hợp không thể suy kiểu vì thiếu hoặc không đầy đủ thông tin ngữ cảnh nó sẽ báo lỗi, như trường hợp sau:

```
// phải cung cấp kiểu cho ngữ cảnh, ví dụ: Comparator<String>
```

```
Comparator comparator = (first, second) -> first.length() - second.length();
```

Lưu ý rằng biểu thức Lambda không chỉ là một cách viết ngắn gọn, mà có sự cải tiến thật sự về hiệu suất khi dùng chúng:

- Java 8 không dùng các lớp ẩn danh do hiệu suất hoạt động và những thay đổi có thể có của lớp ẩn danh trong tương lai.

- Sử dụng chỉ thị `invokedynamic`, được bổ sung từ Java 7, khi gọi biểu thức Lambda.

Chuyển đổi biểu thức Lambda

Ví dụ trên cho thấy biểu thức Lambda được *dùng như một trị tham chiếu*, nó tự động chuyển thành đối tượng đúng kiểu, nằm ở vế trái toán tử gán. Trong ví dụ trên là kiểu interface `Comparator`.

Chú ý rằng biểu thức Lambda *không có tên phương thức*, vì vậy nó được xem như một *phương thức vô danh* (anonymous method). Để đảm bảo điều này, biểu thức Lambda phải thỏa mãn:

- Trả về một thực thể kiểu interface được dự kiến trước. Ta gọi việc trả về này là chuyển đổi (conversion) biểu thức Lambda thành thực thể cài đặt kiểu interface dự kiến.

- Kiểu interface dự kiến của thực thể đó phải có *duy nhất một phương thức*, gọi là SAM (Single Abstract Method – interface chỉ có một phương thức trừu tượng duy nhất). Vì chỉ có một phương thức nên ta không quan tâm đến tên phương thức.

- Phương thức duy nhất của interface dự kiến phải có signature phù hợp với các tham số vế trái của biểu thức Lambda.

Nói cách khác, biểu thức Lambda tạo một thực thể cài đặt interface dự kiến trước. Vế trái của biểu thức Lambda là danh sách tham số, vế phải của biểu thức Lambda là cài đặt cho phương thức duy nhất của interface đó.

Functional interface

Biểu thức Lambda chuyển đổi thành thực thể cài đặt *functional interface*. Interface này được đánh dấu (không bắt buộc) bằng annotation `@FunctionalInterface`.

Do interface này chỉ có một phương thức, nếu bạn thử đánh dấu một interface với `@FunctionalInterface`, nhưng khai báo nhiều phương thức hoặc không khai báo phương thức nào (interface đánh dấu) trong interface, bạn sẽ nhận được thông báo lỗi.

```
@FunctionalInterface
```

```
interface Converter<R, T> {
```

```
    T convert(R from);
```

```
}
```

```
// cài đặt cho biểu thức Lambda, chuyển một số nguyên thành chuỗi nhị phân
```

```
Converter<Integer, String> converter = (from) -> Integer.toBinaryString(from);
```

// hoặc dùng tham chiếu phương thức (thảo luận sau)

```
Converter<Integer, String> converter = Integer::toBinaryString;
```

```
System.out.println(converter.convert(10));
```

Java 8 cung cấp nhiều functional interface, một số nâng cấp từ các interface cũ như Runnable, Comparator, ActionListener; một số thuộc gói java.util.function, được trình bày tiếp sau đây.

Các lớp sau cần cho các ví dụ minh họa:

```
enum TaskType {
    READING("Reading"), WRITING("Writing"), CODING("Coding"), SPEAKING("Speaking");
    private String value;
    private TaskType(String value) { this.value = value; }
    public String getValue() { return value; }
}

class Task {
    int id;
    String title;
    TaskType type;
    LocalDate createdOn;
    final DateTimeFormatter df = DateTimeFormatter.ofPattern("dd/MM/yyyy");

    public Task(int id, String title, TaskType type, String date) {
        this.id = id;
        this.title = title;
        this.type = type;
        this.createdOn = LocalDate.parse(date, df);
    }

    public int getId() { return id; }
    public String getTitle() { return title; }
    public TaskType getType() { return type; }
    public LocalDate getCreatedOn() { return createdOn; }
    @Override public String toString() { return String.format("%s, %s", title, createdOn); }
}

List<Task> tasks = Arrays.asList(
    new Task(1, "Master and Margarita", TaskType.READING, "02/09/1990"),
    new Task(2, "My poem", TaskType.WRITING, "30/04/2000"),
    new Task(3, "Les Miserables", TaskType.READING, "19/05/2012"));
```

1. Predicate<T>

Định nghĩa phương thức test() dùng kiểm tra cho một số điều kiện trên đối tượng kiểu T.

Ví dụ, từ danh sách tasks, lọc ra các task có type là READING, trả về title của các task đó.

```
private List<String> taskTitles(List<Task> tasks, Predicate<Task> filter) {
    List<String> results = new ArrayList<>();
    for (Task task : tasks)
        if (filter.test(task)) results.add(task.title);
    return results;
}

// trong danh sách tasks, lọc các task có type là READING, trả về title của các task đó
List<String> titles = taskTitles(tasks, task -> task.type == TaskType.READING);
titles.forEach(System.out::println);

Dùng Stream API:
tasks.stream()
    .filter(task -> task.type == TaskType.READING)
    .map(Task::getTitle)
    .collect(Collectors.toList())
    .forEach(System.out::println);
```

2. Consumer<T>

Định nghĩa phương thức accept() nhận đối tượng kiểu T, và "tiêu thụ" T mà không trả về giá trị gì. Nghĩa là thực hiện hành động mà không sinh bất kỳ đầu ra nào.

Ví dụ, in ra tất cả title của danh sách tasks.

```
Consumer<Task> consumer = task -> System.out.println(task.title);
for (Task task : tasks) consumer.accept(task);
// hoặc gọn hơn
tasks.forEach(task -> System.out.println(task.title));
```

3. Function<T, R>

Định nghĩa phương thức apply() nhận đối tượng kiểu T và rút trích kiểu trả về R từ đối tượng đó. Nếu nhận hai tham số T, U và trả về kiểu R, dùng interface BiFunction<T, U, R>.

Ví dụ, từ danh sách tasks, lọc ra các task có type là READING, trả về title của các task đó.

```
private <R> List<R> taskTitles(List<Task> tasks, Predicate<Task> filterTasks, Function<Task, R> extractor) {
    List<R> results = new ArrayList<>();
    for (Task task : tasks)
        if (filterTasks.test(task)) results.add(extractor.apply(task));
    return results;
}
// trong danh sách tasks, lọc các task có type là READING, trả về title của các task đó
List<String> titles = taskTitles(tasks, task -> task.type == TaskType.READING, task -> task.title);
titles.forEach(System.out::println);
```

Lưu ý, R cho phép ta trả về một danh sách linh hoạt, đặc biệt khi áp dụng tham chiếu chỉ đến phương thức thay cho extractor:

```
// trả về danh sách các title (kiểu String) của task lọc được
tasks.stream()
    .filter(task -> task.type == TaskType.READING)
    .map(Task::getTitle)
    .collect(Collectors.toList())
    .forEach(System.out::println);
// trả về danh sách các createdOn (kiểu LocalDate) của task lọc được
tasks.stream()
    .filter(task -> task.type == TaskType.READING)
    .map(Task::getCreatedOn)
    .collect(Collectors.toList())
    .forEach(System.out::println);
// trả về danh sách các task (kiểu Task) của task lọc được
tasks.stream()
    .filter(task -> task.type == TaskType.READING)
    .map(Function.identity())
    .collect(Collectors.toList())
    .forEach(System.out::println);
```

4. Supplier<R>

Không nhận bất kỳ đầu vào nào (không tham số) nhưng "cung cấp" ra một giá trị R bằng phương thức get().

Ví dụ, tạo các số nhận dạng duy nhất.

```
Supplier<String> generator = () -> UUID.randomUUID().toString();
System.out.println(generator.get());
```

Dùng trong Stream API:

```
Stream.generate(UUID::randomUUID)
    .limit(1)
    .forEach(System.out::println);
```

5. Tạo functional interface tùy biến

Bạn có thể dễ dàng viết functional interface tùy biến hoặc thừa kế functional interface có sẵn.

@FunctionalInterface

```
interface TaxFunction {
    double calcTax(double grossIncome);
}
```

```
class Customer {
    String name;
    double grossIncome;

    public Customer(String name, double grossIncome) {
        this.name = name;
        this.grossIncome = grossIncome;
    }

    public double getTax(TaxFunction f) {
        return f.calcTax(grossIncome);
    }
}
```

```
Customer customer = new Customer("Donald Trump", 50000);
double tax = customer.getTax(g -> (g < 30000 ? 0.05 : 0.06) * g);
System.out.println("Tax: $" + tax);
```

Tham chiếu phương thức (method reference)

Đôi khi bạn tạo biểu thức Lambda chỉ gọi *một phương thức*, như sau:

```
(Task task) -> task.toString()
```

Biểu thức Lambda này sẽ tự động chuyển đổi thành một thực thể cài đặt @FunctionalInterface, có một phương thức duy nhất, nhận tham số kiểu Task và trả về một chuỗi từ phương thức toString() của thực thể tham số nhận vào.

```
Function<Task, String> showFn = (t) -> t.toString();
```


Với biểu thức Lambda như vậy, Java 8 cung cấp một cú pháp dễ đọc/viết và ngắn gọn hơn:

Task::toString

Cách viết tắt của biểu thức Lambda như trên gọi là *tham chiếu phương thức*.

```
Function<Task, String> showFn = Task::toString;
```

```
System.out.println(showFn.apply(task));
```

Ví dụ:

```
List<String> list = Arrays.asList("dog", "cat", "fish", "iguana", "ferret");
```

```
// các cách sắp xếp danh sách không phân biệt chữ hoa chữ thường.
```

```
// (1) tham số dùng lớp nội vô danh
```

```
Collections.sort(list, new Comparator<String>() {
    @Override public int compare(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
    }
});
```

```
// (2) tham số dùng biểu thức Lambda
```

```
Collections.sort(list, (s1, s2) -> s1.compareToIgnoreCase(s2));
```

```
// viết rõ hơn, cho thấy phương thức được dùng như tham số
```

```
Comparator<String> comp = (s1, s2) -> s1.compareToIgnoreCase(s2);
```

```
Collections.sort(list, comp);
```

```
// (3) tham số dùng tham chiếu phương thức
```

```
Collections.sort(list, String::compareToIgnoreCase);
```

Một số cách khác dùng sắp xếp danh sách chuỗi trên, chú ý so sánh không phân biệt chữ hoa chữ thường:

```
// (1) dùng phương thức Comparator.comparing với extractor
```

```
Collections.sort(list, Comparator.comparing(String::toUpperCase));
```

```
// (2) dùng phương thức sort của List
```

```
list.sort(String.CASE_INSENSITIVE_ORDER);
```

Bạn có thể tạo tham chiếu phương thức đến phương thức static hoặc non-static:

- Tham chiếu phương thức đến phương thức static (static method reference)

Ví dụ, viết tham chiếu phương thức maxFn dùng tìm số lớn nhất từ một danh sách số.

```
Function<List<Integer>, Integer> maxFn = (numbers) -> Collections.max(numbers);
```

Biểu thức Lambda ở trên tương đương với:

```
Function<List<Integer>, Integer> maxFn = Collections::max;
```

Trong đó, max là phương thức static của lớp Collections chứa một tham số kiểu List.

Dùng tham chiếu phương thức maxFn như sau:

```
maxFn.apply(Arrays.asList(1, 9, 7, 3, 5));
```

- Tham chiếu phương thức đến phương thức non-static (instance method reference)

Ví dụ, viết tham chiếu phương thức concatFn nhận hai chuỗi và trả về kết quả là chuỗi nối hai chuỗi đó.

```
BiFunction<String, String, String> concatFn = (s1, s2) -> s1.concat(s2);
```

Biểu thức Lambda ở trên tương đương với:

```
BiFunction<String, String, String> concatFn = String::concat;
```

Tham chiếu phương thức gọi phương thức concat() của lớp String, đối tượng gọi là tham số thứ nhất. Dùng tham chiếu phương thức concatFn như sau:

```
concatFn.apply("abc", "def");
```

- Bạn cũng có thể tạo tham chiếu phương thức đến constructor. Nếu lớp có nhiều constructor, trình biên dịch sẽ dựa vào tham số để gọi đúng constructor.

```
class MyDate {
    LocalDate date;
    MyDate(String d) {
        date = LocalDate.parse(d, DateTimeFormatter.ofPattern("dd/MM/yyyy"));
    }
}
```

```
MyDate(Date d) {
    date = new java.sql.Date(d.getTime()).toLocalDate();
}
```

```
LocalDate getDate() {
    return date;
}
```

```
@Override public String toString() {
    return date.format(DateTimeFormatter.ISO_DATE);
}
}
```

```
interface DateFactory<D extends MyDate> {
    D create(Date d);
}
```

```
// gọi constructor MyDate(Date)
```

```
DateFactory<MyDate> factory = MyDate::new;
```

```
MyDate date = factory.create(new Date());
```

```
System.out.println(date);
// gọi constructor MyDate(String)
List<String> dates = Arrays.asList("30/04/1975", "11/09/2001", "02/09/1945");
dates.stream().map(MyDate::new)
    .sorted(Comparator.comparing(MyDate::getDate))
    .forEach(System.out::println);
// tham chiếu phương thức đến constructor cũng áp dụng với mảng
MyDate[] myDates = dates.stream()
    .map(MyDate::new)
    .toArray(MyDate[]::new);
```

Lập trình giao diện đồ họa

Có hai tập Java API cho lập trình giao diện đồ họa: AWT (Abstract Windowing Toolkit) và Swing.

- AWT API được giới thiệu trong JDK 1.0. Hầu hết các component AWT đã trở nên lạc hậu và được thay thế bằng các component Swing tương ứng mới hơn.

- Swing API là một tập các thư viện đồ họa toàn diện hơn tăng cường cho AWT, được giới thiệu như là một phần của JFC (Java Foundation Classes) sau khi phát hành JDK 1.1. JFC gồm Swing, Java2D, Accessibility, Internationalization, và Pluggable Look-and-Feel API. JFC là một add-on cho JDK 1.1 nhưng đã được tích hợp vào trong lõi Java từ JDK 1.2.

Ngoài AWT và Swing cung cấp trong JDK, còn các tập API cung cấp giao diện khác, như SWT (Standard Widget Toolkit) của Eclipse, GWT (Google Web Toolkit) của Google.

AWT

AWT có 12 gói. Tuy nhiên, chỉ có 2 gói thường được sử dụng: `java.awt` và `java.awt.event`.

- Gói `java.awt` chứa các lớp đồ họa AWT:

- + Các lớp GUI Component, như `Button`, `TextField` và `Label`.
- + Các lớp GUI Container, như `Frame`, `Panel`, `Dialog` và `ScrollPane`.
- + Các lớp quản lý cách bố trí, như `FlowLayout`, `BorderLayout` và `GridLayout`.
- + Các lớp đồ họa tùy biến, như `Graphics`, `Color` và `Font`.

- Gói `java.awt.event` hỗ trợ xử lý sự kiện:

- + Các lớp Event (sự kiện), như `ActionEvent`, `MouseEvent`, `KeyEvent` và `WindowEvent`.
- + Các interface Event Listener (lắng nghe sự kiện), như `ActionListener`, `MouseListener`, `KeyListener` và `WindowListener`.
- + Các lớp tiếp hợp Event Listener, như `MouseAdapter`, `KeyAdapter`, và `WindowAdapter`.

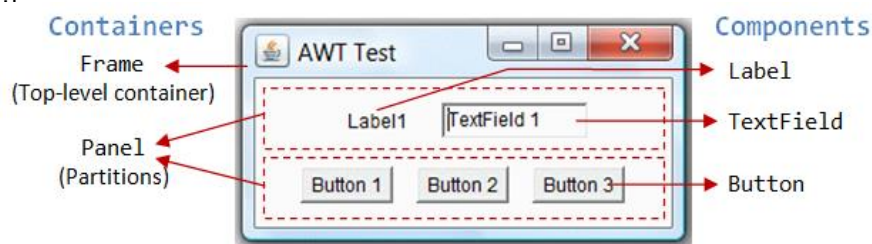
AWT cung cấp một giao diện độc lập nền tảng và độc lập thiết bị để phát triển các chương trình đồ họa chạy trên tất cả các nền tảng như Windows, Mac, và Linux.

GUI có hai thành phần:

- Component: là các đơn thể GUI nhỏ, như `Button`, `Label` và `TextField`.

- Container: chứa các component, như `Frame`, `Panel` và `Applet`. Được sử dụng để giữ các component trong một bố trí cụ thể, thành các dòng (flow) hoặc lưới (grid). Một container cũng có thể chứa container con.

Các component cũng được gọi là control hoặc widget, cho phép người dùng tương tác với ứng dụng thông qua các chúng. Ví dụ, nhấn một nút hoặc nhập văn bản vào.



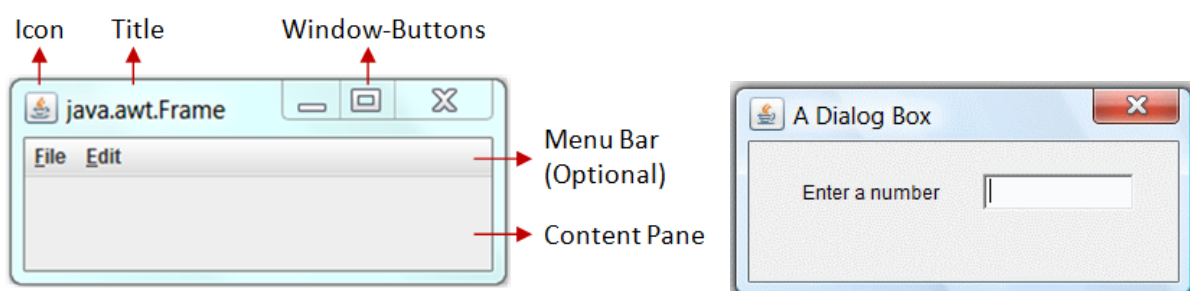
Trong hình trên, có ba container: một `Frame` và hai `Panel`. `Frame` là container cấp cao nhất (top-level), có: thanh tiêu đề (chứa icon, tiêu đề, và ba nút chuẩn Minimize/Maximize/Close), menu-bar tùy chọn và vùng hiển thị nội dung. `Panel` là một vùng hình chữ nhật được sử dụng để nhóm các component có liên quan trong một bố cục nhất định. Trong hình trên, `Frame` chứa hai `Panel`. `Panel` trên có hai component: `Label` cung cấp mô tả về ô nhập, `TextField` dùng nhập văn bản. `Panel` dưới có ba `Button` để người sử dụng để kích hoạt các hành động được lập trình.

Trong một chương trình có dùng GUI, một container có thể chứa các component bằng phương thức `add(Component c)`.

```
Panel panel = new Panel();           // Panel là một container
Button button = new Button("Press me"); // Button là một component
panel.add(button)
```

1. Các lớp container

Mỗi chương trình dùng GUI có một container top-level. Các container top-level thường được sử dụng trong AWT là `Frame`, `Dialog` và `Applet`:



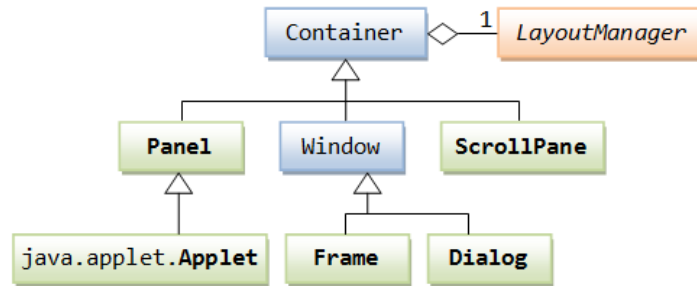
- Một `Frame` cung cấp một "cửa sổ chính" cho các ứng dụng GUI. Để viết một chương trình giao diện, ta thường bắt đầu với một lớp con dẫn xuất từ lớp `java.awt.Frame`.

- Một `Dialog` là một "cửa sổ pop-up" dùng tương tác với người sử dụng. Một `Dialog` có thanh tiêu đề (chứa icon, tiêu đề và nút Close chuẩn) và một vùng hiển thị nội dung.

- Một `Applet` (trong gói `java.applet`) là container top-level cho một applet, đó là một chương trình Java chạy bên trong trình duyệt.

Container thứ cấp được đặt bên trong một container top-level hoặc một container thứ cấp khác. AWT cung cấp các container thứ cấp sau:

- **Panel**: một hộp hình chữ nhật nằm dưới một container cấp cao hơn, dùng bố trí một tập các component có liên quan trong một lưới hoặc dòng vô hình.
 - **ScrollPane**: cung cấp khả năng tự động cuộn ngang và/hoặc cuộn dọc cho component con.
- Hệ thống phân cấp các lớp container AWT như sau. Một container có một layout.

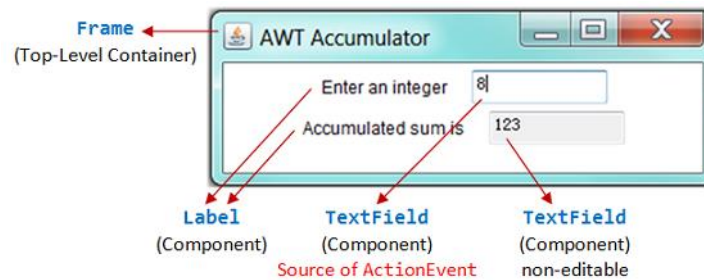


2. Các lớp component

AWT cung cấp nhiều component tạo sẵn và có thể dùng lại. Thường dùng nhất là: Button, TextField, Label, Checkbox, CheckboxGroup (nhóm radio button), List và Choice.



3. Ví dụ minh họa



```

import java.awt.FlowLayout;
import java.awt.Frame;
import java.awt.Label;
import java.awt.TextField;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public class AWTAccumulator extends Frame implements ActionListener {
    private TextField txtInput;
    private TextField txtOutput;
    private int numberIn;
    private int sum = 0;

    public AWTAccumulator() {
        init();
        // xử lý sự kiện click nút Close, đóng cửa sổ ứng dụng
        addWindowListener(new WindowAdapter() {
            @Override public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });
    }

    private void init() {
        setTitle("AWT Accumulator");
    }
}
  
```

```

setSize(350, 120);
setResizable(false);
setLocationRelativeTo(null);
setLayout(new FlowLayout());
add(new Label("Enter an Integer: "));
// khởi gán từng component và thêm chúng vào container
txtInput = new TextField(10);
txtInput.addActionListener(this);
add(txtInput);
add(new Label("The Accumulated Sum is: "));
txtOutput = new TextField(10);
txtOutput.setEditable(false);
add(txtOutput);
}

// lớp giao diện người dùng cũng là listener cho txtInput
@Override public void actionPerformed(ActionEvent e) {
    numberIn = Integer.parseInt(txtInput.getText());
    sum += numberIn;
    txtInput.setText("");
    txtOutput.setText(sum + "");
}

public static void main(String[] args) {
    java.awt.EventQueue.invokeLater(new Runnable() {
        @Override public void run() {
            new AWTAccumulator().setVisible(true);
        }
    });
}
}

```

Swing

Swing là một phần của JFC, được giới thiệu năm 1997, sau khi phát hành của JDK 1.1. Từ JDK 1.2, JFC trở thành bộ phận không thể thiếu của JDK. Swing API khá lớn, 18 gói API (JDK 1.7), là một tập phong phú và toàn diện các component JavaBean dễ hiểu, dễ sử dụng, có thể dùng lại, có thể kéo thả để xây dựng giao diện trong môi trường lập trình trực quan.

Các tính năng chính của Swing:

- Swing được viết thuần Java (trừ một vài lớp) nên có tính khả chuyển cao.
 - Các component Swing sử dụng ít tài nguyên hệ thống hơn các component AWT. Component AWT dựa rất nhiều vào các hệ thống con của sổ (windowing subsystem) của hệ điều hành gốc.
 - Các component Swing hỗ trợ pluggable look-and-feel. Bạn có thể lựa chọn giữa Java look-and-feel và look-and-feel của hệ điều hành bên dưới (Windows, UNIX hay Mac). Một nút Swing chạy trên Windows trông (look) giống như một nút Windows và cảm nhận (feel) giống như một nút Windows.
 - Swing hỗ trợ tác vụ ít dùng mouse hơn, có nghĩa là nó có thể hoạt động hoàn toàn bằng cách sử dụng bàn phím.
 - Các component Swing hỗ trợ "tool-tips".
 - Các component Swing là JavaBeans được dùng trong lập trình trực quan. Bạn có thể kéo thả các component Swing để xây dựng giao diện người dùng trong môi trường lập trình trực quan.
 - Ứng dụng Swing sử dụng các lớp xử lý sự kiện của AWT (gói java.awt.event). Swing thêm một số lớp mới trong gói javax.swing.event, nhưng chúng không được sử dụng thường xuyên.
 - Ứng dụng Swing sử dụng quản lý bố trí mới, chẳng hạn như Springs, Struts, và BoxLayout (gói javax.swing).
- Swing bổ sung các quản lý bố trí mới, chẳng hạn như Springs, Struts, và BoxLayout (gói javax.swing).
- Swing cài đặt vùng đệm kép (double-buffering) giúp vẽ lại màn hình nhanh.
 - Swing giới thiệu JLayeredPane và JInternalFrame để tạo ứng dụng có giao diện nhiều tài liệu (MDI – Multiple Document Interface).
 - Swing hỗ trợ toolbar (JToolBar), splitter, khả năng undo.

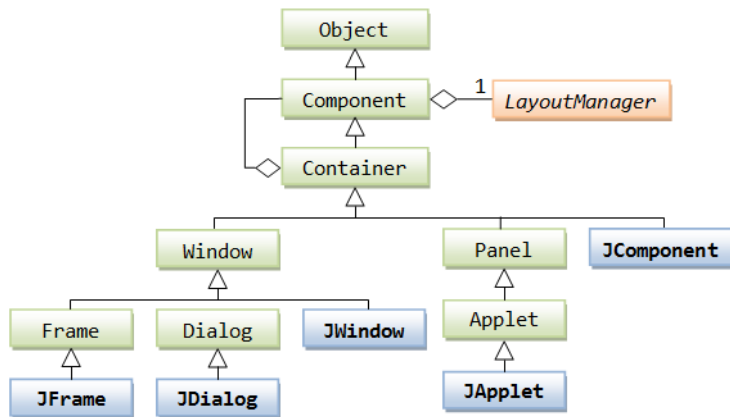
1. Swing với AWT

Nếu bạn hiểu lập trình với AWT, các khái niệm như container/component, event-handling, layout manager; có thể chuyển trực tiếp sang Swing.

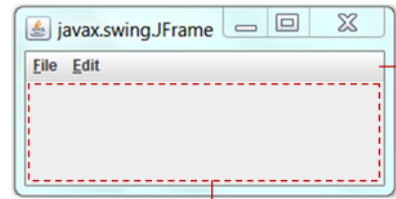
So với các lớp AWT (trong gói java.awt), các lớp của Swing (trong gói javax.swing) bắt đầu với tiền tố "J", ví dụ như JButton, JTextField, JLabel, JPanel, JFrame hoặc JApplet. Tương tự với AWT, Swing cũng có hai nhóm các lớp: container và component. Một container được sử dụng để giữ các component, một container cũng có thể giữ các container khác vì nó là lớp con của component. Chú ý không sử dụng pha trộn giữa các component AWT và component Swing trong cùng một chương trình. Cũng giống như ứng dụng AWT, một ứng dụng Swing đòi hỏi một container cấp cao nhất (top-level). Có ba container top-level trong Swing:

- JFrame: dùng như cửa sổ chính của ứng dụng, có icon, tiêu đề, các nút chuẩn Minimize/Maximize/Close, menu-bar tùy chọn, và vùng nội dung (content-pane).
- JDialog: dùng như các cửa sổ pop-up thứ cấp, có tiêu đề, nút chuẩn Close, và vùng nội dung.
- JApplet: dùng như vùng hiển thị của applet (content-pane) bên trong cửa sổ của trình duyệt.

Cây phân cấp lớp của Swing container như sau:



javax.swing.JFrame



Content Pane

```
Container cp = aJFrame.getContentPane();
aJFrame.setContentPane(aPanel);
```

Tương tự với AWT, có các container thứ cấp như JPanel, có thể được sử dụng để nhóm và bố trí các component có liên quan. Tuy nhiên, không giống như AWT, các component của Swing không được thêm trực tiếp vào container top-level (JFrame, JDialog). Chúng phải được thêm vào *vùng nội dung* (content-pane) của container top-level. Content-pane thực tế là một java.awt.Container có thể được dùng để nhóm và bố trí các component.

Bạn có thể:

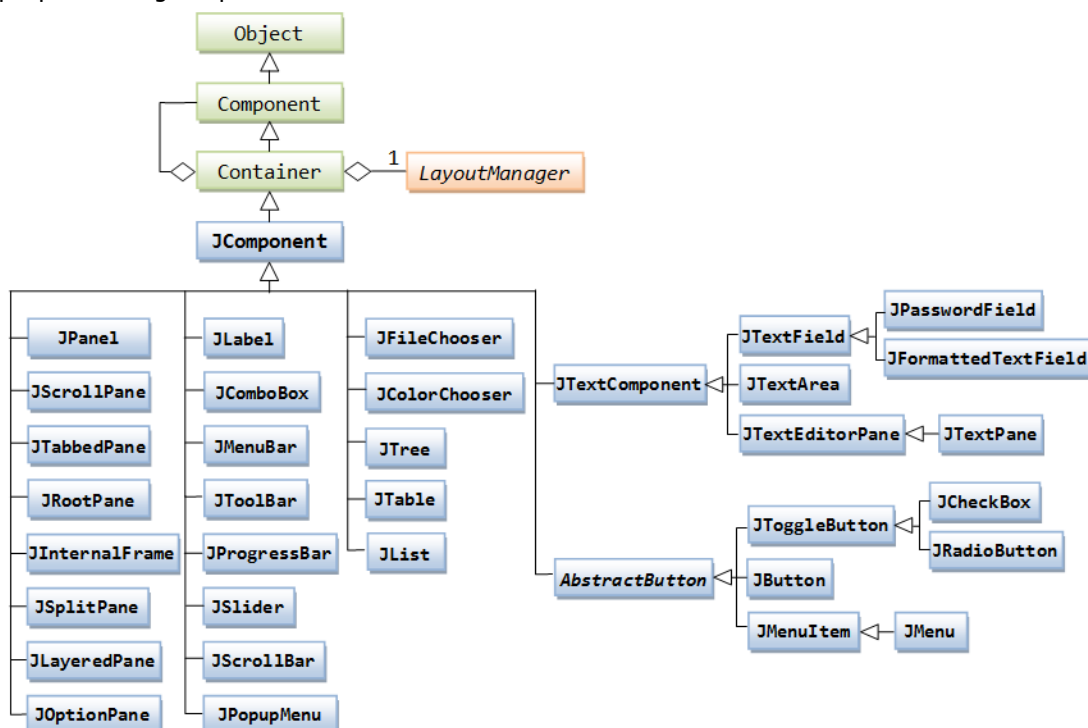
- Lấy vùng nội dung bằng phương thức getContentPane() từ một container top-level, và thêm các component vào nó.

```
Container pane = getContentPane();
pane.setLayout(new FlowLayout());
pane.add(new JLabel("Hello world!"));
pane.add(new JButton("Button"));
```

- Thiết lập vùng nội dung cho một JPanel (panel chính được tạo ra để giữ tất cả các component) bằng phương thức setContentPane() từ một container top-level.

```
JPanel mainPanel = new JPanel(new FlowLayout());
mainPanel.add(new JLabel("Hello world!"));
mainPanel.add(new JButton("Button"));
setContentPane(mainPanel);
```

Cây phân cấp lớp của Swing component như sau:



Swing sử dụng các lớp xử lý sự kiện của AWT (gọi java.awt.event). Swing giới thiệu một vài lớp xử lý sự kiện mới (gọi javax.swing.event) nhưng không được sử dụng thường xuyên.

Hầu hết các component của Swing hỗ trợ các tính năng:

- Có text và icon.
- Phím tắt (gọi là mnemonics).
- Tool tip: hiển thị khi dừng mouse trên component.
- Look-and-feel: tùy chỉnh hiển thị và tương tác người dùng tùy theo hệ nền.
- Bản địa hóa (localization): ngôn ngữ khác nhau cho địa phương khác nhau.

2. Ví dụ minh họa

```
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.Label;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class SwingAccumulator extends JFrame {
    private JTextField txtInput;
    private JTextField txtOutput;
    private int numberIn;
    private int sum = 0;

    public SwingAccumulator() {
        init();
        // xử lý sự kiện click nút Close, đóng cửa sổ ứng dụng
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    private void init() {
        setTitle("Swing Accumulator");
        setSize(350, 120);
        setResizable(false);
        setLocationRelativeTo(this);

        Container pane = getContentPane();
        pane.setLayout(new FlowLayout());
        add(new Label("Enter an Integer: "));
        txtInput = new JTextField(10);
        txtInput.addActionListener(new ActionListener() {
            @Override public void actionPerformed(ActionEvent e) {
                numberIn = Integer.parseInt(txtInput.getText());
                sum += numberIn;
                txtInput.setText("");
                txtOutput.setText(sum + "");
            }
        });
        add(txtInput);
        add(new Label("The Accumulated Sum is: "));
        txtOutput = new JTextField(10);
        txtOutput.setEditable(false);
        add(txtOutput);
    }

    public static void main(String[] args) {
        // chạy trong Event Dispatcher Thread thay vì thread chính, để an toàn thread
        SwingUtilities.invokeLater(new Runnable() {
            @Override public void run() {
                new SwingAccumulator().setVisible(true);
            }
        });
    }
}
```

Chú ý, nếu một component được thêm trực tiếp vào JFrame như trên, nó sẽ được thêm vào vùng nội dung của JFrame. Nghĩa là hai dòng sau tương đương với nhau:

```
add(new JLabel("Hello world!"));
getContentPane().add(new JLabel("Hello world!"));
```

Quản lý bố trí

Một container có một đối tượng gọi là layout manager để quản lý việc bố trí các component trong nó. Layout manager cung cấp một cấp trừu tượng để ánh xạ giao diện trên các hệ thống cửa sổ khác nhau, tùy theo hệ nền. Container có phương thức `setLayout()` để thiết lập đối tượng layout manager cho nó. Layout manager cũng dùng như tham số cho constructor của các container thứ cấp như `Panel` hoặc `JPanel`.

Để tạo các bố trí phức tạp, tổ chức một nhóm các component có liên hệ với nhau, đặt vào container con (container thứ cấp) với layout manager riêng.

1. FlowLayout

Trong `java.awt.FlowLayout`, các component được sắp xếp từ trái sang phải trong container theo thứ tự mà chúng được thêm vào. Khi một hàng được lấp đầy, một hàng mới sẽ được bắt đầu. Hiển thị thực tế phụ thuộc vào chiều rộng của cửa sổ hiển thị.

2. GridLayout

Trong `java.awt.GridLayout`, các thành phần được sắp xếp trong một lưới các hàng và các cột bên trong container. Các thành phần được thêm vào từ trái sang phải, từ trên xuống dưới theo thứ tự chúng được thêm vào.

3. BorderLayout

Trong `java.awt.BorderLayout`, container chia thành 5 vùng: EAST, WEST, SOUTH, NORTH, và CENTER.

Các component được thêm vào bằng cách dùng phương thức `aContainer.add(aComponent, aZone)`, ở đây `aZone` là một trong `BorderLayout.NORTH` (hoặc `PAGE_START`), `BorderLayout.SOUTH` (hoặc `PAGE_END`), `BorderLayout.WEST` (hoặc `LINE_START`), `BorderLayout.EAST` (hoặc `LINE_END`), hoặc `BorderLayout.CENTER` (mặc định nếu không chỉ định `aZone`).

Xử lý sự kiện

1. Sự kiện (event)

Khi người thực hiện một tương tác với UI (user interface - giao diện người dùng), click mouse hoặc nhấn phím, một sự kiện sẽ được phát ra. Sự kiện là một đối tượng, gọi là đối tượng event, mô tả điều gì đã xảy ra. Sự kiện được phân loại thành một vài nhóm, mô tả các loại tương tác khác nhau của người dùng.

Java dùng mô hình ủy nhiệm sự kiện (delegation event model). Ví dụ khi người dùng click lên một nút, JVM tạo đối tượng event và nút được nhấn sẽ gửi đối tượng này đến đối tượng lắng nghe sự kiện (listener) đã đăng ký với nút. Bộ xử lý sự kiện (event handler) cài đặt trong đối tượng listener sẽ xử lý sự kiện với thông tin chứa trong đối tượng event nhận được.

- Nguồn sự kiện (event source)

Nguồn sự kiện chính là đối tượng sinh ra sự kiện. Ví dụ, click mouse lên một Button sẽ sinh ra một thực thể `ActionEvent` với Button là nguồn của sự kiện, đối tượng sự kiện `ActionEvent` chứa thông tin về sự kiện đã xảy ra.

Bạn có thể dùng phương thức `getSource()` có sẵn cho tất cả các đối tượng sự kiện, để lấy tham chiếu đến đối tượng nguồn phát ra sự kiện. Phương thức này cần thiết khi sử dụng một listener chung cho nhiều nguồn sự kiện và bạn phải phân biệt nguồn nào đã phát ra sự kiện.

- Bộ xử lý sự kiện (event handler)

Bộ xử lý sự kiện là các phương thức cài đặt trong lớp listener, chúng nhận đối tượng event chứa thông tin về sự kiện, phân tích nó và xử lý sự kiện do tương tác của người dùng tạo nên.

2. Mô hình ủy nhiệm sự kiện

Trong mô hình ủy nhiệm sự kiện (delegate event model), JVM tạo ra các sự kiện (event) nhưng đơn thể UI không xử lý sự kiện mà ủy nhiệm cho một hoặc nhiều lớp đăng ký với nó xử lý sự kiện. Các lớp này gọi là event listener, chúng có khả năng:

- lắng nghe sự kiện: listener sẽ nhận event tương ứng được chuyển tiếp từ đơn thể UI.

- xử lý sự kiện: listener chứa các bộ xử lý sự kiện (event handler) nhận và xử lý event chuyển đến.

Theo cách này, đối tượng xử lý sự kiện có thể tách biệt với đơn thể UI. Listener là các lớp cài đặt interface `EventListener`.

Các đối tượng event chỉ kích hoạt các listener có đăng ký với đơn thể UI là nguồn sự kiện. Mỗi event có một interface listener tương ứng, chứa các phương thức được định nghĩa để nhận kiểu event. Như vậy lớp cài đặt cho interface listener phải:

- Định nghĩa các phương thức nhận đối tượng event và xử lý sự kiện được gửi đến.

- Được đăng ký như là một listener lắng nghe sự kiện cho đơn thể UI. Bạn không cần phải xử lý tất cả các sự kiện, bằng cách hoặc không đăng ký listener lắng nghe sự kiện, hoặc chỉ lựa chọn các phương thức quan tâm để xử lý.

Viết riêng Lớp listener, chứa phương thức xử lý sự kiện `actionPerformed()`:

```
import java.awt.Component;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JOptionPane;

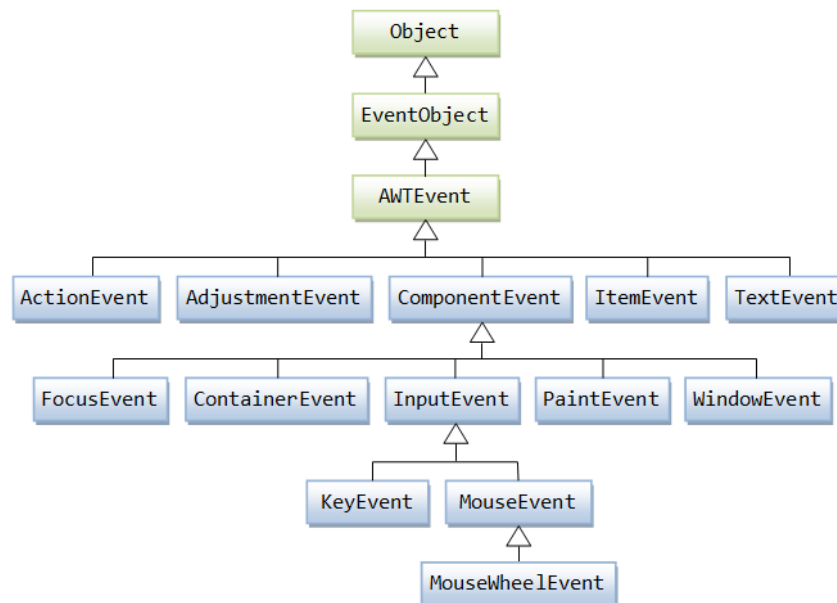
// lớp xử lý sự kiện
public class ButtonHandler implements ActionListener {
    // phương thức xử lý sự kiện, nhận tham số là đối tượng event
    @Override public void actionPerformed(ActionEvent e) {
        // tham số thứ nhất dùng xác định đơn thể cha của JOptionPane
        JOptionPane.showMessageDialog(((Component)e.getSource()).getParent(), e.getActionCommand());
    }
}
```

Đăng ký lớp `ButtonHandler` thành listener cho `JButton` trong lớp giao diện người dùng:

```
JButton btnHello = new JButton("Hello");
// đăng ký đối tượng listener cho đơn thể UI
// khi nhấn nút, ActionEvent sẽ được chuyển cho listener
btnHello.addActionListener(new ButtonHandler());
```

3. Phân loại sự kiện

Thư viện lớp của Java chứa nhiều lớp event, nhưng đa số tập trung trong gói `java.awt.event`. Với mỗi loại sự kiện có một interface được lớp listener cài đặt nếu muốn nhận và xử lý loại sự kiện đó.



Tác động	Loại sự kiện	Interface	Phương thức
Nhấn vào một nút.	Action	ActionListener	actionPerformed(ActionEvent)
Chọn một mục.	Item	ItemListener	itemStateChanged(ItemEvent)
Click một component.	Mouse	MouseListener	mousePressed(MouseEvent) mouseReleased(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mouseClicked(MouseEvent)
Di chuyển mouse.	Chuyển động mouse	MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
Gõ một phím.	Key	KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
Chọn một component	Chọn đơn thể	FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)
	Hiệu chỉnh Component	AdjustmentListener ComponentListener	adjustmentValueChanged(AdjustmentEvent) componentMoved(ComponentEvent) componentHidden(ComponentEvent) componentResized(ComponentEvent) componentShown(ComponentEvent)
Thay đổi window.	Window	WindowListener	windowClosing(WindowEvent) windowOpened(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent)
	Container	ContainerListener	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
Thay đổi văn bản.	Text	TextListener	textValueChanged(TextEvent)

Listener

1. GUI cũng là listener

Trong ví dụ trên, ta viết riêng một *lớp listener* để lắng nghe sự kiện và xử lý sự kiện cho giao diện người dùng (GUI). Tuy nhiên, xử lý sự kiện thường là cập nhật GUI, nên đôi khi lớp GUI được cài đặt *kiêm cả nhiệm vụ listener*, bằng cách:

- Cài đặt một hay nhiều interface listener cho lớp GUI. Như vậy, phải cài đặt cho các phương thức xử lý sự kiện khai báo trong các interface đó. Với những sự kiện không cần xử lý, cài đặt rỗng cho các phương thức xử lý sự kiện tương ứng.
- Sau đó đăng ký *this* như là các listener cho lớp giao diện người dùng.

```

import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import javax.swing.JOptionPane;
import javax.swing.JTextField;

```

```

public class TextFieldTwoListener extends JTextField implements MouseMotionListener, MouseListener {
    public TextFieldTwoListener() {
        this.addMouseListener(TextFieldTwoListener.this);
        this.addMouseMotionListener(TextFieldTwoListener.this);
    }
}

```

```

    }
    // các phương thức xử lý sự kiện của MouseMotionListener
    @Override public void mouseDragged(MouseEvent e) {
        setText("Mouse dragging (" + e.getX() + ", " + e.getY() + ")");
    }
    // các phương thức xử lý sự kiện không dùng đến của MouseMotionListener
    @Override public void mouseMoved(MouseEvent e) { }

    // các phương thức xử lý sự kiện của MouseListener
    @Override public void mouseEntered(MouseEvent e) {
        setText("The mouse entered");
    }
    @Override public void mouseExited(MouseEvent e) {
        setText("The mouse has left");
    }
    // các phương thức xử lý sự kiện không dùng đến của MouseListener
    @Override public void mouseClicked(MouseEvent e) { }
    @Override public void mousePressed(MouseEvent e) { }
    @Override public void mouseReleased(MouseEvent e) { }

    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null, new TextFieldTwoListener());
    }
}

```

2. Listener dùng adapter

Khi cài đặt một interface listener, ví dụ `MouseListener` hoặc `WindowListener`, bạn phải cài đặt cho nhiều phương thức xử lý sự kiện, và bạn mất thời gian để cài đặt rỗng cho các phương thức không dùng đến. Để thuận tiện, Java cung cấp các lớp tiếp hợp (adapter) cài đặt sẵn các interface listener *có nhiều hơn một phương thức*. Trong lớp tiếp hợp này, các phương thức của interface sẽ được *cài đặt rỗng sẵn*. Bạn có thể thừa kế các lớp tiếp hợp này và chỉ cần lựa chọn viết lại (override) đúng các phương thức mà bạn cần.

```

import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

public class MouseClickHandler extends MouseAdapter {
    // chỉ cần cài đặt xử lý sự kiện click mouse, thay vì cài đặt cả 5 phương thức của MouseListener
    @Override public void mouseClicked(MouseEvent e) {
        // xử lý sự kiện click mouse
    }
}

```

3. Listener là lớp nội

a) Lớp nội

Trong ví dụ đầu, lớp xử lý sự kiện là lớp riêng. Rõ ràng bạn khó truy cập đến các thành viên của lớp giao diện người dùng. Nếu lớp xử lý sự kiện là lớp nội (inner class) của lớp giao diện người dùng mà nó đăng ký, nó có thể dễ dàng truy cập đến các thành viên private của lớp chứa nó.

```

import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionAdapter;
import javax.swing.JFrame;
import javax.swing.JLabel;

class MouseClickHandler extends MouseAdapter {
    @Override public void mouseClicked(MouseEvent e) {
        // xử lý click mouse
    }
}

public class InnerFrame extends JFrame {
    private JLabel lblInfo = new JLabel();
    public InnerFrame() {
        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        setSize(200, 75);
        lblInfo.addMouseListener(new MyMouseMotionListener()); // lớp xử lý sự kiện là lớp nội
        lblInfo.addMouseListener(new MouseClickHandler()); // lớp xử lý sự kiện bên ngoài
        add(lblInfo);
    }
}

class MyMouseMotionListener extends MouseMotionAdapter {
    @Override public void mouseDragged(MouseEvent e) {

```

```

        // gọi phương thức của lớp bao nó
        lblInfo.setText("Mouse dragging (" + e.getX() + ", " + e.getY() + ")");
    }
}

public static void main(String[] args) {
    new InnerFrame().setVisible(true);
}
}

```

Lớp nội này còn gọi là lớp thành viên (member class) của lớp bao ngoài. Đôi khi cũng được đánh dấu protected để có thể thừa kế xuống lớp con.

b) Lớp nội vô danh

Bạn cũng có thể đưa toàn bộ định nghĩa lớp xử lý sự kiện vào bên trong tầm vực của một biểu thức. Cách tiếp cận này gọi là dùng lớp nội vô danh (anonymous inner class), lớp xử lý sự kiện được khai báo và tạo thực thể tại một thời điểm. Cách này được sử dụng phổ biến để xử lý sự kiện.

```

import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionAdapter;
import javax.swing.JFrame;
import javax.swing.JLabel;

class MouseClickHandler extends MouseAdapter {
    @Override public void mouseClicked(MouseEvent e) {
        // xử lý click mouse
    }
}

public class AnonymousFrame extends JFrame {
    private JLabel lblInfo = new JLabel();
    public AnonymousFrame() {
        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        setSize(200, 75);
        this.addMouseMotionListener(new MouseMotionAdapter() {
            @Override public void mouseDragged(MouseEvent e) {
                lblInfo.setText("Mouse dragging (" + e.getX() + ", " + e.getY() + ")");
            }
        });
        this.addMouseListener(new MouseClickHandler()); // lớp xử lý sự kiện bên ngoài
        add(lblInfo);
    }

    public static void main(String[] args) {
        new AnonymousFrame().setVisible(true);
    }
}

```

Trong Java 8, các interface listener *chỉ có một phương thức* có thể là functional interface. Thay vì dùng lớp nội vô danh, chúng được viết ngắn gọn bằng biểu thức Lambda:

```

JButton btnHello = new JButton("Hello");
// lớp nội vô danh
btnHello.addActionListener(new ActionListener() {
    @Override public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(rootPane, e.getActionCommand());
    }
});

// biểu thức Lambda tương đương
btnHello.addActionListener((ActionEvent e) -> {
    JOptionPane.showMessageDialog(rootPane, e.getActionCommand());
});

```

4. Form cha và form con

Tình huống thường gặp trong lập trình GUI là chuyển thông tin giữa hai cửa sổ, thường gọi là hai form. Ví dụ:

- Frame StudentGUI hiển thị thông tin của một Student.
- Khi click nút btnCreate trên frame, hộp thoại CreateDialog sẽ mở, hiển thị form nhập thông tin cho một Student mới.
- Sau khi nhập thông tin Student mới, click nút btnOk trên hộp thoại. Hộp thoại CreateDialog đóng lại và thông tin Student mới vừa tạo sẽ được hiển thị trong frame StudentGUI.

Có vài cách tiếp cận để giải quyết vấn đề này:

- a) Form con giữ một tham chiếu chỉ đến form cha của nó.

Khi khởi tạo `CreateDialog`, `StudentGUI` truyền tham chiếu `this` của nó cho constructor của `CreateDialog`. `CreateDialog` sử dụng tham chiếu `parent` này gọi phương thức của `StudentGUI` để cập nhật UI.

```
import java.awt.Container;
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
import static javax.swing.WindowConstants.DISPOSE_ON_CLOSE;

public class StudentGUI extends JFrame {
    JButton btnCreate = new JButton("Create");
    JLabel lblInfo = new JLabel();

    public StudentGUI() {
        initComponents();
    }

    private void initComponents() {
        setTitle("Test");
        setSize(300, 150);
        Container pane = getContentPane();
        pane.setLayout(new FlowLayout());
        pane.add(lblInfo);
        pane.add(btnCreate);
        // gán listener cho nút btnCreate để mở hộp thoại CreateDialog
        btnCreate.addActionListener((java.awt.event.ActionEvent e) -> {
            java.awt.EventQueue.invokeLater(() -> {
                // truyền tham chiếu this cho constructor của CreateDialog
                CreateDialog dialog = new CreateDialog(StudentGUI.this, true);
                dialog.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
                dialog.setLocationRelativeTo(rootPane);
                dialog.setVisible(true);
            });
        });
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
    }

    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(() -> {
            new StudentGUI().setVisible(true);
        });
    }

    public void updateGUI(Student student) {
        // cập nhật UI của StudentGUI với Student nhận được
        lblInfo.setText(String.format("%s (%d)", student.name, student.age));
    }
}

class CreateDialog extends JDialog {
    StudentGUI parent;
    JButton btnOk = new JButton("Ok");
    JTextField txtName = new JTextField(10);
    JTextField txtAge = new JTextField(6);

    public CreateDialog(StudentGUI parent, boolean modal) {
        super(parent, modal);
        initComponents();
        this.parent = parent;
    }

    private void initComponents() {
        setTitle("Dialog");
        setSize(300, 150);
        Container pane = getContentPane();
        pane.setLayout(new FlowLayout());
        pane.add(txtName);
        pane.add(txtAge);
    }
}
```



```

pane.add(btnOk);
// gán listener cho nút btnOk của CreateDialog
btnOk.addActionListener((java.awt.event.ActionEvent e) -> {
    // tạo đối tượng Student từ các input của form nhập
    Student student = new Student();
    student.update(txtName.getText(), Integer.parseInt(txtAge.getText()));
    parent.updateGUI(student);
    dispose();
});
}
}

```

```

class Student {
    String name;
    int age;

    public void update(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

```

b) Dùng callback interface.

CreateDialog dùng đối tượng callback interface để xử lý các sự kiện, StudentGUI cài đặt đối tượng kiểu interface này và gán cho CreateDialog khi khởi tạo nó, bằng cách truyền như tham số cho constructor hoặc bằng cách gọi setter.

```

import java.awt.Container;
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
import static javax.swing.WindowConstants.DISPOSE_ON_CLOSE;

public class StudentGUI extends JFrame implements CreateDialog.CreateDialogListener {
    JButton btnCreate = new JButton("Create");
    JLabel lblInfo = new JLabel();

    public StudentGUI() {
        initComponents();
    }

    private void initComponents() {
        setTitle("Test");
        setSize(300, 150);
        Container pane = getContentPane();
        pane.setLayout(new FlowLayout());
        pane.add(lblInfo);
        pane.add(btnCreate);
        btnCreate.addActionListener((java.awt.event.ActionEvent e) -> {
            java.awt.EventQueue.invokeLater(() -> {
                CreateDialog dialog = new CreateDialog(StudentGUI.this, true);
                dialog.setListener(StudentGUI.this);
                dialog.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
                dialog.setLocationRelativeTo(rootPane);
                dialog.setVisible(true);
            });
        });
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
    }

    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(() -> {
            new StudentGUI().setVisible(true);
        });
    }

    // cài đặt cho listener trung gian
    @Override public void onClickOk(Student student) {
        // cập nhật UI của StudentGUI với Student nhận được
        lblInfo.setText(String.format("%s (%d)", student.name, student.age));
    }
}

```

```

    }
}

class CreateDialog extends JDialog {
    JButton btnOk = new JButton("Ok");
    JTextField txtName = new JTextField(10);
    JTextField txtAge = new JTextField(6);
    private CreateDialogListener listener;
    // callback interface
    interface CreateDialogListener {
        void onClickOk(Student student);
    };

    public CreateDialog(StudentGUI parent, boolean modal) {
        super(parent, modal);
        initComponents();
    }

    public void setListener(CreateDialogListener listener) {
        this.listener = listener;
    }

    private void initComponents() {
        setTitle("Dialog");
        setSize(300, 150);
        Container pane = getContentPane();
        pane.setLayout(new FlowLayout());
        pane.add(txtName);
        pane.add(txtAge);
        pane.add(btnOk);
        btnOk.addActionListener((java.awt.event.ActionEvent e) -> {
            // tạo đối tượng Student từ các input của form nhập
            Student student = new Student();
            student.update(txtName.getText(), Integer.parseInt(txtAge.getText()));
            listener.onClickOk(student);
            dispose();
        });
    }
}

```

```

class Student {
    String name;
    int age;

    public void update(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

```

c) Dùng mẫu thiết kế Observer

Mẫu thiết kế Observer có sẵn trong API của Java, form con sẽ đăng ký form cha (cài đặt Observer) với dữ liệu (thừa kế Observable). Thay đổi của dữ liệu từ form con dẫn đến cập nhật tự động form cha.

```

import java.awt.Container;
import java.awt.FlowLayout;
import java.util.Observable;
import java.util.Observer;
import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
import static javax.swing.WindowConstants.DISPOSE_ON_CLOSE;

// GUI cài đặt Observer, cập nhật khi Observable thay đổi
public class StudentGUI extends JFrame implements Observer {
    JButton btnCreate = new JButton("Create");
    JLabel lblInfo = new JLabel();

    public StudentGUI() {
        initComponents();
    }
}

```

```

private void initComponents() {
    setTitle("Test");
    setSize(300, 150);
    Container pane = getContentPane();
    pane.setLayout(new FlowLayout());
    pane.add(lblInfo);
    pane.add(btnCreate);
    btnCreate.addActionListener((java.awt.event.ActionEvent e) -> {
        java.awt.EventQueue.invokeLater(() -> {
            CreateDialog dialog = new CreateDialog(StudentGUI.this, true);
            dialog.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
            dialog.setLocationRelativeTo(rootPane);
            dialog.setVisible(true);
        });
    });
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLocationRelativeTo(null);
}

// cài đặt phương thức update của Observer, tự động gọi khi Observable có cập nhật
@Override public void update(Observable o, Object arg) {
    Student student = (Student) o;
    lblInfo.setText(String.format("%s (%d)", student.name, student.age));
}

public static void main(String args[]) {
    java.awt.EventQueue.invokeLater(() -> {
        new StudentGUI().setVisible(true);
    });
}
}

class CreateDialog extends JDialog {
    JButton btnOk = new JButton("Ok");
    JTextField txtName = new JTextField(10);
    JTextField txtAge = new JTextField(6);
    Student observable = new Student();

    public CreateDialog(StudentGUI parent, boolean modal) {
        super(parent, modal);
        // dialog đăng ký Observer với Observable
        observable.addObserver(parent);
        initComponents();
    }

    private void initComponents() {
        setTitle("Dialog");
        setSize(300, 150);
        Container pane = getContentPane();
        pane.setLayout(new FlowLayout());
        pane.add(txtName);
        pane.add(txtAge);
        pane.add(btnOk);
        btnOk.addActionListener((java.awt.event.ActionEvent e) -> {
            observable.update(txtName.getText(), Integer.parseInt(txtAge.getText()));
            dispose();
        });
    }
}

// Dữ liệu thừa kế Observable, thông báo cho các Observer đăng ký với nó khi nó thay đổi
class Student extends Observable {
    String name;
    int age;

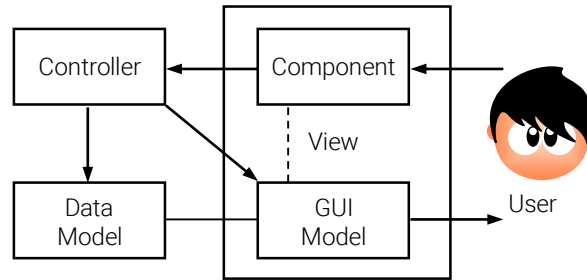
    public void update(String name, int age) {
        this.name = name;
        this.age = age;
        setChanged();
        notifyObservers();
    }
}

```

Mô hình MVC

Mô hình MVC lần đầu tiên được giới thiệu bởi Trygve Reenskaug (1979), giúp tách thao tác nghiệp vụ (truy cập dữ liệu) khỏi thao tác hiển thị dữ liệu.

1. Các thành phần MVC



a. Model

Trong ứng dụng Swing, có hai loại model:

- Data Model, mô hình hóa cho dữ liệu của ứng dụng và các luật liên quan đến truy cập, cập nhật dữ liệu đó. Phần này cung cấp thao tác nghiệp vụ (business logic) cho ứng dụng.

- GUI Model, còn gọi là Display Model, model dùng lưu trữ dữ liệu (trạng thái) của một UI component liên kết với nó.

Một số GUI Model, thường lấy bằng phương thức getModel() hoặc getSelectionModel() từ UI component tương ứng.

BoundedRangeModel Dùng bởi JProgressBar, JScrollbar, JSlider. Lưu trữ 4 trị nguyên: value, extent, min, max liên quan đến việc thiết đặt cho UI component đó.

ButtonModel Dùng bởi tất cả các lớp con của AbstractButton. Lưu trữ các trị boolean thể hiện thao tác được thực hiện trên button đó.

ListModel Dùng bởi JList. Lưu trữ một collection các Object.

ComboBoxModel<E> Dùng bởi JComboBox<E>. Lưu trữ một collection các Object và một Object được chọn.

MutableComboBoxModel<E> Dùng bởi JComboBox<E>. Một phiên bản thay đổi được của ComboBoxModel<E>.

ListSelectionModel Dùng bởi JList, TableColumnModel. Lưu trữ một hoặc nhiều chỉ số các mục được chọn của một danh sách hoặc một bảng.

SpinnerModel Dùng bởi JSpinner. Lưu trữ một collection có thứ tự và thành phần được chọn hiện tại.

SingleSelectionModel Dùng bởi JMenuBar, JPopupMenu, JMenuItem, JTabbedPane. Lưu trữ chỉ số thành phần được chọn.

ColorSelectionModel Dùng bởi JColorChooser. Lưu trữ một Color.

TableModel Dùng bởi JTable. Lưu trữ một mảng hai chiều (dòng và cột) các Object.

TableColumnModel Dùng bởi JTable, lấy từ phương thức getColumnModel(), giúp xử lý chỉ tiết tập cột của JTable.

TreeModel Dùng bởi JTree. Lưu trữ các Object hiển thị trong JTree, các Object này có tổ chức phân cấp, phân biệt các node trong và node lá.

TreeSelectionModel Dùng bởi JTree. Lưu trữ các "dòng" được chọn (chọn một, chọn liên tục hoặc không liên tục).

Document Dùng bởi các component văn bản, lấy từ phương thức getDocument().

b. View

Hiển thị dữ liệu của Model. View cung cấp thao tác trình diễn (presentation logic) cho ứng dụng. Nếu dữ liệu của Model thay đổi, View phải cập nhật hiển thị của nó. Điều này có thể đạt được bằng cách sử dụng:

- + Mô hình đẩy (push model), trong đó View đăng ký chính nó với Model như một listener để nhận được thông báo thay đổi.
- + Mô hình kéo (pull model), trong đó View có nhiệm vụ gọi Model khi cần để lấy dữ liệu mới nhất.

c. Controller

Chuyển tương tác của người dùng trên View thành hành động mà Model sẽ thực hiện. Trong ứng dụng dùng GUI, tương tác của người dùng có thể là click lên một nút, nhấn một phím, hoặc lựa chọn một mục trong menu.

Controller liên kết với View. Như vậy, bất kỳ tương tác nào của người dùng đến View sẽ gọi một phương thức listener đã đăng ký trong lớp Controller. Cài đặt của phương thức này có tham chiếu đến Model bên dưới.

Khi người dùng tương tác với View, các hành động sau đây xảy ra:

- + View nhận biết có tác động trên giao diện (nhấn một nút, kéo thanh cuộn, ...), gọi phương thức listener thích hợp trên Controller.
- + Controller truy cập Model, có thể cập nhật Model tùy hành động của người dùng.
- + Nếu Model thay đổi, nó sẽ thông báo thay đổi cho các listener (chẳng hạn View) quan tâm. Controller có thể chịu trách nhiệm cập nhật View.

2. Ví dụ minh họa

Xem xét các thành phần của MVC trong ví dụ sau:

- Model

- + Data Model: lớp POJO Person, interface PersonService và lớp PersonServiceMock mô hình hóa dữ liệu của ứng dụng và tác vụ lên các dữ liệu đó.

- + GUI Model: DefaultListModel và DefaultListSelectionModel liên kết với JList searchResultList; Document liên kết với JTextField searchField. Liên kết giữa chúng theo kiểu push model. Thay đổi trong Model sẽ dẫn đến thay đổi ở View.

- View

- + Các View có GUI Model: JTextField searchField, JList searchResultList

+ JButton searchButton và JButton showPersonDetailsButton.

- Controller

+ SearchPersonAction: gán cho nút

+ PersonDetailsAction: gán cho nút showPersonDetailsButton, tương tác của người dùng sẽ dẫn đến việc truy cập hai model DefaultListSelectionModel (lấy index) và DefaultListModel (lấy thông tin Person tương ứng).

```
import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.event.ActionEvent;
import java.util.ArrayList;
import java.util.List;
import javax.swing.AbstractAction;
import javax.swing.Action;
import javax.swing.DefaultListModel;
import javax.swing.DefaultListSelectionModel;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextField;
import javax.swing.ListSelectionModel;
import javax.swing.event.ListSelectionEvent;
import javax.swing.text.BadLocationException;
import javax.swing.text.Document;
import javax.swing.text.PlainDocument;

class Person {
    private int id;
    private String firstName;
    private String lastName;
    public Person(int id, String firstName, String lastName) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public int getId() { return id; }
    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    @Override public String toString() { return firstName + ", " + lastName; }
}

interface PersonService {
    List<Person> search(String keyword);
}

class PersonServiceMock implements PersonService {
    private List<Person> personDB = new ArrayList<>();
    public PersonServiceMock() {
        personDB.add(new Person(1, "Barack", "Obama"));
        personDB.add(new Person(2, "Karl", "Marx"));
        personDB.add(new Person(3, "Fidel", "Castro"));
    }

    @Override public List<Person> search(String keyword) {
        List<Person> matches = new ArrayList<>();
        if (keyword != null) {
            keyword = keyword.toLowerCase();
            for (Person p : personDB)
                if (p.getFirstName().toLowerCase().contains(keyword) || p.getLastName().toLowerCase().contains(keyword))
                    matches.add(p);
        }
        return matches;
    }
}

class PersonDetailsAction extends AbstractAction {
    private DefaultListModel personListModel;
    private ListSelectionModel personSelectionModel;
    public PersonDetailsAction(DefaultListModel personListModel, ListSelectionModel personSelectionModel) {
```

```

    this.personListModel = personListModel;
    this.personSelectionModel = personSelectionModel;
    personSelectionModel.addListSelectionListener((ListSelectionEvent e) -> {
        ListSelectionModel listSelectionModel = (ListSelectionModel) e.getSource();
        updateEnablement(listSelectionModel);
    });
    updateEnablement(personSelectionModel);
}

@Override public void actionPerformed(ActionEvent e) {
    int index = personSelectionModel.getMinSelectionIndex();
    Person person = (Person) personListModel.get(index);
    JOptionPane.showMessageDialog(null, getPersonDetails(person));
}

private String getPersonDetails(Person person) {
    return person.getId() + ": " + person.getFirstName() + " " + person.getLastName();
}

private void updateEnablement(ListSelectionModel listSelectionModel) {
    boolean emptySelection = listSelectionModel.isEmptySelection();
    setEnabled(!emptySelection);
}
}

class SearchPersonAction extends AbstractAction {
    private Document searchInput;
    private DefaultListModel searchResult;
    private PersonService personService;
    public SearchPersonAction(Document searchInput, DefaultListModel searchResult, PersonService personService) {
        this.searchInput = searchInput;
        this.searchResult = searchResult;
        this.personService = personService;
    }

    @Override public void actionPerformed(ActionEvent e) {
        String searchString = getSearchString();
        List<Person> matchedPersons = personService.search(searchString);
        searchResult.clear();
        for (Person person : matchedPersons)
            searchResult.addElement(person);
    }

    private String getSearchString() {
        try {
            return searchInput.getText(0, searchInput.getLength());
        } catch (BadLocationException e) {
            return null;
        }
    }
}

public class Main extends JFrame {
    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(() -> {
            new Main().setVisible(true);
        });
    }

    public Main() {
        initComponents();
    }

    private void initComponents() {
        // Models
        PersonService personService = new PersonServiceMock();
        DefaultListModel searchResultListModel = new DefaultListModel();
        DefaultListSelectionModel searchResultSelectionModel = new DefaultListSelectionModel();
        searchResultSelectionModel.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

        // Controllers

```



```

Document searchInput = new PlainDocument();
Action personDetailsAction = new PersonDetailsAction(searchResultListModel, searchResultSelectionModel);
personDetailsAction.putValue(Action.NAME, "Person Details");
Action searchPersonAction = new SearchPersonAction(searchInput, searchResultListModel, personService);
searchPersonAction.putValue(Action.NAME, "Search");

// Views
JPanel searchInputPanel = new JPanel();
searchInputPanel.setLayout(new BorderLayout());
JTextField searchField = new JTextField(searchInput, null, 0);
searchField.addActionListener(searchPersonAction);
JButton searchButton = new JButton(searchPersonAction);
searchInputPanel.add(searchField, BorderLayout.CENTER);
searchInputPanel.add(searchButton, BorderLayout.EAST);

JList searchResultList = new JList();
searchResultList.setModel(searchResultListModel);
searchResultList.setSelectionModel(searchResultSelectionModel);
JPanel searchResultPanel = new JPanel();
searchResultPanel.setLayout(new BorderLayout());
JScrollPane scrollableSearchResult = new JScrollPane(searchResultList);
searchResultPanel.add(scrollableSearchResult, BorderLayout.CENTER);

JPanel selectionOptionsPanel = new JPanel();
JButton showPersonDetailsButton = new JButton(personDetailsAction);
selectionOptionsPanel.add(showPersonDetailsButton);

// GUI
setTitle("MVC");
setSize(300, 200);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setLocationRelativeTo(null);
Container contentPane = getContentPane();
contentPane.add(searchInputPanel, BorderLayout.NORTH);
contentPane.add(searchResultPanel, BorderLayout.CENTER);
contentPane.add(selectionOptionsPanel, BorderLayout.SOUTH);
}
}

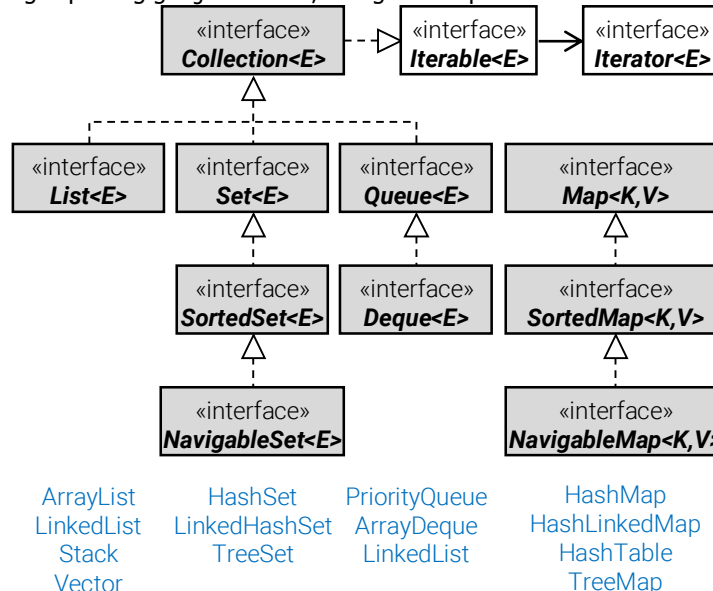
```

Collection Framework

Collection Framework

Collection là tên gọi chung của các cấu trúc dữ liệu dùng để lưu trữ một nhóm đối tượng. Các đối tượng lưu trong collection được gọi là các element. Trong Java, các cấu trúc dữ liệu cấp phát động này được hỗ trợ dựa trên một kiến trúc thống nhất, gọi là Collection Framework. Trong Collection Framework, các cấu trúc dữ liệu được thiết kế sao cho chúng có giao diện thống nhất để người dùng có thể lưu trữ, truy cập và thao tác lên các element của chúng mà không quan tâm đến cài đặt bên dưới. Điều này cho phép người dùng tập trung vào giao diện sử dụng, không quan tâm đến cài đặt cụ thể của các cấu trúc dữ liệu.

Collection Framework được cung cấp trong gói `java.util`, bao gồm các phần chính sau:



Hai bộ phận chính của Collection Framework: Collection và Map

- Các interface

- + `Iterable<E>` khai báo phương thức `iterator()` trả về đối tượng `Iterator<E>` dùng duyệt các element của collection.
- + `Iterator<E>` khai báo các phương thức hoạt động cho đối tượng duyệt iterator.
- + `List<E>` mô hình hóa một mảng cấp phát động cho phép truy cập theo chỉ số, cho phép các element trùng. Các lớp cài đặt phổ biến: `ArrayList`, `LinkedList`, `Vector` và `Stack`.
- + `Set<E>` mô hình hóa một tập hợp, không cho phép element trùng. Các lớp cài đặt phổ biến: `HashSet` và `LinkedHashSet`.
- + Interface con `SortedSet<E>` mô hình hóa một tập hợp các element có thứ tự và được sắp xếp, lớp cài đặt là `TreeSet`.
- + `Queue<E>`: mô hình hóa hàng đợi và hàng đợi ưu tiên. Interface con `Deque<E>` mô hình hóa hàng đợi có thể chèn/xóa tại hai đầu (Double-Ended Queue), lớp cài đặt là `PriorityQueue`, `ArrayDeque` và `LinkedList`.
- + `Map<K,V>` mô hình hóa ánh xạ của các cặp khóa K (duy nhất) và các trị lưu trữ. Các lớp cài đặt phổ biến: `HashMap`, `Hashtable` và `LinkedHashMap`. Interface con `SortedMap<K,V>` mô hình hóa một map có thứ tự và được sắp xếp dựa trên các khóa, lớp cài đặt là `TreeMap`. Chú ý, `Map<K,V>` không phải là interface con của `Collection<E>`.

- Các lớp cài đặt cụ thể.

- Các lớp công cụ, còn gọi là các algorithms.

1. Sử dụng các collection

a) `Collection<E>`

`Collection<E>` là interface gốc của Collection Framework, định nghĩa các hành vi chung cho các lớp của framework.

- Các tác vụ cơ bản

- | | |
|---|--|
| <code>int size()</code> | trả về số element của collection. |
| <code>void clear()</code> | loại bỏ tất cả element của collection. |
| <code>boolean isEmpty()</code> | trả về true nếu không có element trong collection. |
| <code>boolean add(E element)</code> | thêm element vào collection. |
| <code>boolean remove(Object element)</code> | loại bỏ element khỏi collection. |
| <code>boolean contains(Object element)</code> | trả về true nếu collection chứa element chỉ định. |

- Các tác vụ kết hợp với collection khác

- | | |
|--|--|
| <code>boolean containsAll(Collection<?> c)</code> | |
| <code>boolean addAll(Collection<? extends E> c)</code> | |
| <code>boolean removeAll(Collection<?> c)</code> | |
| <code>boolean retainAll(Collection<?> c)</code> | |

- Các tác vụ mảng

- | | |
|---|--|
| <code>Object[] toArray()</code> | chuyển collection thành mảng các Object. |
| <code><T> T[] toArray(T[] a)</code> | chuyển collection thành mảng kiểu T. |

- Làm việc với Collection Framework:

- | | |
|--|---|
| <code>Collection<String> c = new ArrayList<>();</code> | Lớp <code>ArrayList</code> cài đặt interface <code>Collection</code> . |
| <code>c = new TreeSet<String>();</code> | Lớp <code>TreeSet</code> cũng cài đặt interface <code>Collection</code> . |
| <code>int n = c.size();</code> | Lấy kích thước của collection. |

```

c.add("Obama"); c.add("Putin");
System.out.println(c);
boolean b = c.remove("Obama");
b = c.contains("Sally");
for (String s : c) System.out.println(s);
Iterator<String> iter = c.iterator();

```

Thêm các element vào collection.

In chuỗi chứa tất cả các element trong collection [Obama, Putin].

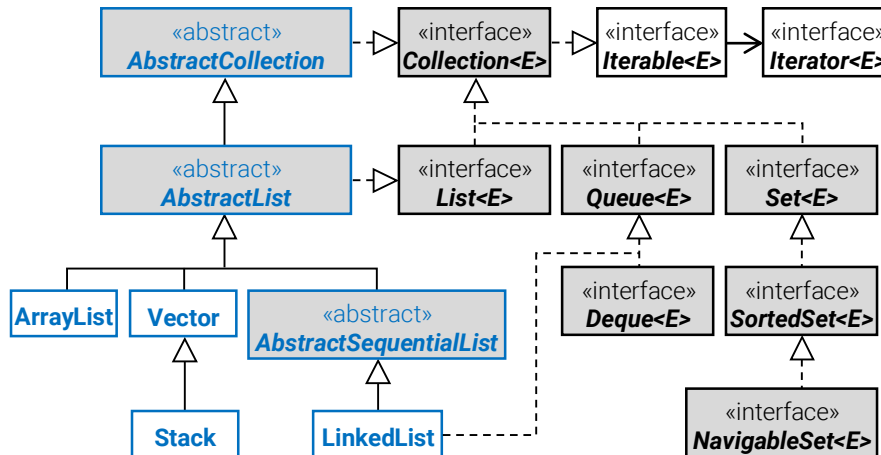
Loại một element của collection, trả về false nếu không tìm thấy element đó.

Kiểm tra collection có chứa element chỉ định hay không?

Dùng vòng lặp for tăng cường, in từng element của collection.

Lấy iterator của collection để duyệt collection.

b) List<E>



List mô hình hóa một mảng tuyến tính cấp phát động, hỗ trợ truy cập bằng chỉ số. Các element có thể chèn/lấy tại vị trí chỉ định trong list, dựa trên chỉ số. Các element có thể chứa trị trùng hoặc trị null.

```
void add(int index, E element)
```

thêm element tại vị trí *index*.

```
E set(int index, E element)
```

thiết lập trị cho element tại vị trí *index*.

```
E get(int index)
```

lấy element tại vị trí *index*, không loại bỏ.

```
E remove(int index)
```

loại bỏ element tại vị trí *index*.

```
int indexOf(Object obj)
```

trả về chỉ số của *obj* tìm được đầu tiên.

```
int lastIndexOf(Object obj)
```

trả về chỉ số của *obj* tìm được đầu tiên kể từ cuối danh sách.

Các lớp cài đặt cụ thể của interface List<E> gồm có:

ArrayList<E>

cài đặt không synchronized của interface List. Sử dụng rất phổ biến.

Vector<E>

cài đặt synchronized của interface List. Đã lạc hậu.

Stack<E>

cấu trúc dữ liệu hoạt động theo nguyên tắc LIFO, thừa kế Vector.

- Làm việc với List<E>

```
LinkedList<String> list = new LinkedList<>();
```

Tạo một list rỗng.

```
list.addLast("Obama");
```

Thêm element vào cuối list.

```
list.addFirst("Putin");
```

Thêm element vào đầu list.

```
list.getFirst();
```

Lấy element lưu ở đầu list.

```
list.getLast();
```

Lấy element lưu ở cuối list.

```
String first = list.removeFirst();
```

Loại bỏ element thứ nhất của list, trả về nó.

```
String last = list.removeLast();
```

Loại bỏ element cuối cùng của list, trả về nó.

```
ListIterator<String> iter = list.listIterator();
```

Trả về đối tượng iterator để duyệt list.

- Làm việc với Stack<E>

```
Stack<Integer> s = new Stack<>();
```

Khởi dựng một stack rỗng.

```
s.push(1);
```

Thêm 1 vào stack.

```
s.push(2);
```

Phương thức toString() in [1, 2], đỉnh stack (top) ở cuối.

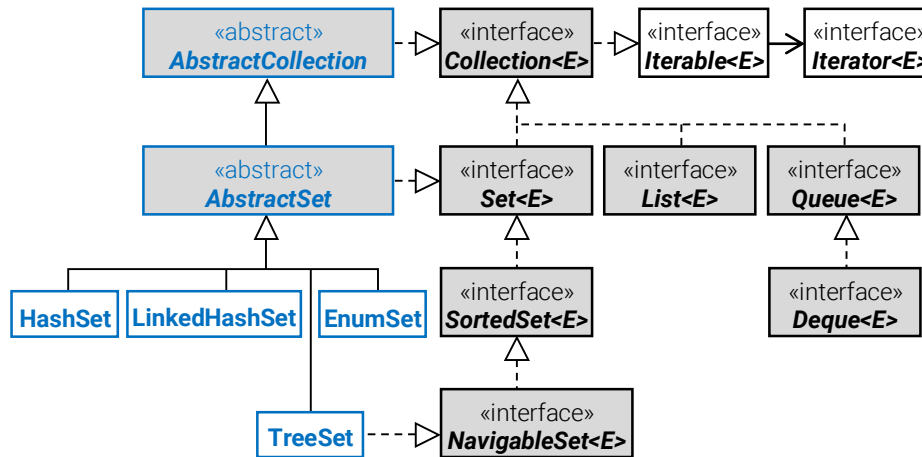
```
int removed = s.pop();
```

Loại bỏ element tại đỉnh stack (2), trả nó về.

```
int head = s.peek();
```

Lấy element tại đỉnh stack (1) nhưng không loại bỏ nó.

c) Set<E>



Set là collection không cho phép các element trùng. Các phương thức trừu tượng của interface Set trả về false khi tác vụ thất bại, thay vì ném ra exception:

`boolean add(E o)` thêm element chỉ định nếu nó không có sẵn.
`boolean remove(Object o)` loại bỏ element chỉ định nếu nó có sẵn.
`boolean contains(Object o)` trả về true nếu có chứa `o`.
`boolean addAll(Collection<? extends E> c)` this sẽ chứa phần hội của this và `c`.
`boolean retainAll(Collection<?> c)` this sẽ chứa phần giao của this và `c`.

Các lớp cài đặt cụ thể của interface Set<E> bao gồm:

`HashSet<E>` Dùng bảng băm, sắp xếp theo thứ tự mã băm. Nên viết `equals()` và `hashCode()` cho element.
`LinkedHashSet<E>` Dùng danh sách liên kết, sắp xếp theo thứ tự chèn vào. Hiệu quả khi thêm/xóa nhiều.
`TreeSet<E>` Dùng cây đỏ-đen, sắp xếp theo cách so sánh element (bằng Comparable hoặc Comparator).
 Cài đặt interface `NavigableSet<E>` và `SortedSet<E>`. Hiệu quả khi tìm kiếm, thêm/xóa nhiều.

Interface con `NavigableSet<E>` bổ sung thêm một số phương thức:

`Iterator<E> descendingIterator()` trả về iterator dùng duyệt theo thứ tự giảm.
`E floor(E e)` trả về element lớn nhất trong set, nhỏ hơn hoặc bằng element chỉ định.
`E ceiling(E e)` trả về element nhỏ nhất trong set, lớn hơn hoặc bằng element chỉ định.
`E lower(E e)` trả về element lớn nhất trong set, nhỏ hơn element chỉ định. Hoặc trả về null.
`E higher(E e)` trả về element nhỏ nhất trong set, lớn hơn element chỉ định. Hoặc trả về null.
`SortedSet<E> headSet(E to)` trả về một phần set có các element nhỏ hơn element `to` chỉ định.
`SortedSet<E> tailSet(E from)` trả về một phần set có các element lớn hơn hoặc bằng element `from` chỉ định.
`SortedSet<E> subSet(E to, E from)` trả về một phần set có các element trong dãy từ `from` đến nhỏ hơn `to`.

Ví dụ sau dùng `TreeSet` với `Comparator`, so sánh theo trường code của element Student.

```

import java.util.Set;
import java.util.TreeSet;

class Student implements Comparable<Student> {
    int code;
    String name;
    public Student(int code, String name) {
        this.code = code;
        this.name = name;
    }

    @Override public String toString() { return code + ":" + name; }
    @Override public int compareTo(Student o) { return name.compareTo(o.name); }
    @Override public int hashCode() { return code ^ 17; }

    @Override
    public boolean equals(Object o) {
        if (o == null || getClass() != o.getClass()) return false;
        if (o == this) return true;
        final Student t = (Student) o;
        return compareTo(t) == 0;
    }
}

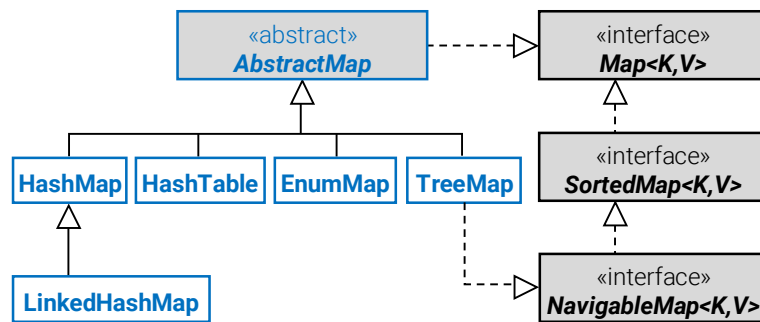
public class Main {
    public static void main(String[] args) {
        Set<Student> students = new TreeSet<Student>((o1, o2) -> Integer.compare(o1.code, o2.code)) {{
            add(new Student(3, "Marx"));
            add(new Student(1, "Obama"));
            add(new Student(2, "Castro"));
        }};
    }
}
  
```

```

        System.out.println(students);
    }
}

```

d) Map<K,V>



Map là collection chứa các cặp key-value, ánh xạ 1-1 một khóa duy nhất với một trị lưu trữ. Như vậy, map lưu hai tập hợp: tập hợp khóa Set<K> lấy bằng phương thức keySet() và tập hợp trị Collection(V) lấy bằng phương thức values().

V get(Object key)	trả về trị lưu với key chỉ định.
V put(K key, V value)	lưu một trị với key chỉ định.
boolean containsKey(Object key)	kiểm tra xem map có key chỉ định.
boolean containsValue(Object value)	kiểm tra xem map có lưu trị value chỉ định.
Set<K> keySet()	trả về tập khóa của map.
Collection<V> values()	trả về tập trị của map.
Set entrySet()	trả về tập key-value của map.

Các lớp cài đặt cụ thể của interface Map<K,V> bao gồm:

HashMap<K,V>	bảng băm cài đặt interface Map<K,V>. Các phương thức không synchronized.
TreeMap<K,V>	cây đồ-đen cài đặt interface SortedMap<K,V>.
LinkedHashMap<K,V>	bảng băm với danh sách liên kết, hiệu quả khi chèn/xóa nhiều.
Hashtable<K,V>	bảng băm cài đặt interface Map<K,V>. Các phương thức synchronized. Đã lạc hậu.

Map<String, Integer> scores;	Khóa kiểu String, trị kiểu Integer. Khai báo biến kiểu interface Map.
scores = new TreeMap<String, Integer>();	Dùng HashMap nếu không cần duyệt có thứ tự.
scores.put("Obama", 90);	Thêm một cặp khóa-trị vào Map.
scores.put("Obama", 100);	Thay đổi trị với khóa tồn tại sẵn.
scores.put("Putin", 101);	Lưu một cặp khóa-trị khác.
int n = scores.get("Obama");	Lấy trị liên kết với khóa, nếu khóa không có mặt, trả về null.
System.out.println(scores);	In map dạng chuỗi {Obama=100, Putin=101}.
for (String key : scores.keySet())	Duyệt tập khóa, xuất các trị tương ứng.
System.out.printf("%d\n", scores.get(key));	
scores.remove("Sally");	Loại bỏ một cặp khóa-trị.

Java 8 bổ sung nhiều phương thức quan trọng cho Map API, làm cho việc sử dụng Map trở nên dễ dàng hơn.

```

class Job {
    int id;
    String title;

    Job(int id, String title) {
        this.id = id;
        this.title = title;
    }

    public int getId() { return id; }
    public String getTitle() { return title; }
    @Override public String toString() { return String.format("[%d] %s", id, title); }
}

```

```

List<Job> jobs = Arrays.asList(new Job(1, "Thinking"), new Job(2, "Coding"), new Job(3, "Debugging"));
- Tạo Map từ List

```

```

Map<Integer, Job> jobHashMap = jobs.stream()
    .collect(Collectors.toMap(Job::getId, identity()));

```

Mặc định, phương thức toMap() tạo ra HashMap, bạn có thể chỉ định toMap() tạo loại Map khác, bằng cách cung cấp một supplier phù hợp như tham số thứ tư của toMap().

```

Map<Integer, Job> jobLinkedMap = jobs.stream()
    .collect(Collectors.toMap(Job::getId, identity(), (v1, v2) -> v1, LinkedHashMap::new));

```

Bạn lưu ý tham số thứ ba, là một BinaryOperator<U> dùng để giải quyết trường hợp có phần tử trùng. Trong ví dụ trên, nếu có hai phần tử trùng thì chỉ dùng phần tử thứ nhất.

- tạo Map từ các bộ (tuples)

```

Map<Integer, Job> jobsMap = Stream.of(
    new SimpleEntry<>(1, new Job(1, "Coding")),
    new SimpleEntry<>(2, new Job(2, "Thinking")))

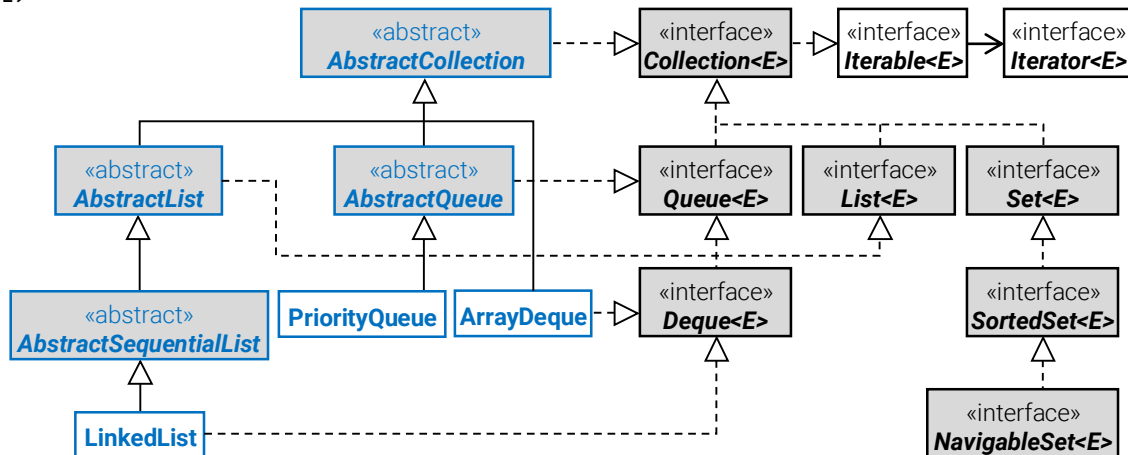
```

```

).collect(Collectors.toMap(SimpleEntry::getKey, SimpleEntry::getValue));

```

e) Queue<E>



Queue (hàng đợi) là collection mà các element thêm/xóa theo một thứ tự cụ thể, thường là FIFO. Các phương thức của queue có hai dạng: ném ra exception nếu tác vụ thất bại, trả về null hoặc false tùy tác vụ.

```

boolean add(E e)      // chèn cuối queue, ném IllegalStateException nếu không đủ không gian.
boolean offer(E e)     // chèn cuối queue, trả về true nếu thêm element thành công.
E remove()            // xóa tại head, ném NoSuchElementException nếu queue rỗng.
E poll()              // xóa tại head, trả về head của queue, hoặc null nếu queue rỗng.
E element()           // xem head, không xóa ném NoSuchElementException nếu queue rỗng.
E peek()              // xem head, không xóa, trả về the head của queue, hoặc null nếu queue rỗng.

```

Deque (double-ended queue) là một queue cho phép thêm/xóa ở cả hai đầu queue.

Deque bổ sung các tác vụ để thao tác tại cả hai đầu queue:

```

- Chèn thêm element
void addFirst(E e)
void addLast(E e)
boolean offerFirst(E e)
boolean offerLast(E e)

- Xóa element
E removeFirst()
E removeLast()
E pollFirst()
E pollLast()

- Lấy element, nhưng không xóa
E getFirst()
E getLast()
E peekFirst()
E peekLast()

```

Một deque có thể dùng như một queue (FIFO) với các phương thức add(e), remove(), element(), offer(e), poll(), peek(); hoặc dùng như một stack (LIFO) với các phương thức push(e), pop(), peek().

Các lớp cài đặt cụ thể của Queue<E> và Deque<E> bao gồm:

```

PriorityQueue<E>      // queue với thứ tự chỉ định, thay vì FIFO.
ArrayDeque<E>        // queue và deque cài đặt như một mảng động, tương tự ArrayList<E>.
LinkedList<E>        // LinkedList<E> cũng cài đặt các interface Queue<E> và Deque<E>, ngoài List<E>.
                        // cung cấp queue và deque cài đặt bằng danh sách liên kết đôi.

```

- Làm việc với Queue

```

Queue<Integer> q = new LinkedList<>();
q.add(1);
q.add(2);
int head = q.remove();
head = q.peek();

```

Lớp LinkedList cài đặt interface Queue.

Thêm 1 vào đuôi (tail) queue.

Phương thức toString in [1, 2].

Loại bỏ element tại đầu (front) queue (1), trả nó về.

Lấy element tại đầu queue (2) nhưng không loại bỏ nó.

- Làm việc với PriorityQueue

```

PriorityQueue<Integer> q = new PriorityQueue<>();
q.add(3); q.add(1); q.add(2);
int first = q.remove();
int second = q.remove();
int next = q.peek();

```

Queue ưu tiên chứa các trị nguyên.

Thêm các trị nguyên vào queue ưu tiên.

Loại bỏ element có trị thấp nhất khỏi queue (1), trả nó về.

Trả về (2).

Lấy element có trị nhỏ nhất mà không loại bỏ nó.

2. Hoán chuyển giữa collection và mảng

Java API còn duy trì một số phương thức lạc hậu, đôi khi chúng cần đổi số là một mảng. Interface Collection cung cấp các phương thức trả về một mảng:

```

Object[] toArray()
<T> T[] toArray(T[] array)

```

trả về mảng các Object.

phiên bản generic, trả về một mảng kiểu T. Tham số dùng chỉ định kiểu trả về.

Ngoài ra, một mảng có thể được sử dụng như một collection, dùng phương thức:

```
public static <T> List<T> asList(T[] array)
class Thing {
    String label;
    String value;
    public Thing(String label, String value) {
        this.label = label;
        this.value = value;
    }
}

public class Main {
    public static void main(String[] args) {
        List<Thing> list = new ArrayList<Thing>() {{
            add(new Thing("100", "Computer"));
            add(new Thing("101", "Boomerang"));
            add(new Thing("102", "Dictionary"));
        }};
        Thing[] array = list.toArray(new Thing[0]);
        for (int i = 0; i < array.length; ++i) {
            System.out.println(array[i].value);
        }
        Collection<Thing> alist = Arrays.asList(array);
        alist.forEach(o -> System.out.println(o.value));
    }
}
```

3. Các lớp công cụ

Collection Framework cung cấp hai lớp công cụ: `java.util.Arrays` và `java.util.Collections` (chú ý, đều có 's'). Hai lớp này cung cấp các thuật toán như sắp xếp và tìm kiếm trên mảng và collection.

a) Comparable và Comparator

Một số collection cài đặt `SortedSet` (`TreeSet`) hoặc `SortedMap` (`TreeMap`), các element đã sắp xếp theo một thứ tự chỉ định. Các collection khác được sắp xếp bằng các phương thức công cụ `Collections.sort()` hoặc `Arrays.sort()`.

Khi sắp xếp các element trong collection. Hai element được so sánh dựa trên một thuộc tính chỉ định nào đó của lớp element.

- Lớp element phải cài đặt interface `Comparable<E>`, trong đó phương thức `compareTo()` được cài đặt bằng cách so sánh thuộc tính chỉ định. Phương thức này trả về -1 nếu thuộc tính của đối tượng `this` nhỏ hơn thuộc tính của đối tượng so sánh, trả về 0 nếu thuộc tính so sánh là bằng nhau; và trả về 1 nếu thuộc tính của đối tượng `this` lớn hơn thuộc tính của đối tượng so sánh.

So sánh hai đối tượng không dùng toán tử so sánh mà bằng cách gọi phương thức `compareTo()`.

- Nếu so sánh hai element chưa cài đặt interface `Comparable<E>`, hoặc muốn so sánh dựa trên thuộc tính khác, phải dùng đối tượng cài đặt interface `Comparator<E>`. Phương thức `compare(E, E)` của nó sẽ cài đặt chi tiết việc so sánh hai đối tượng, trả về giống như phương thức `compareTo()` trình bày ở trên.

Vì đối tượng `Comparator<E>` thường được truyền như một đối số của phương thức cần đến phương thức `compare()` của nó, nên đối tượng `Comparator<E>` còn được gọi là *đối tượng hàm* (function object).

```
class Student implements Comparable<Student> {
    int code;
    String name;
    public Student(int code, String name) {
        this.code = code;
        this.name = name;
    }

    @Override public String toString() {
        return String.format("[%d] %s", code, name);
    }

    @Override public int compareTo(Student o) {
        return Integer.compare(code, o.code);
    }
}

public class Main {
    public static void main(String[] args) {
        ArrayList<Student> students = Arrays.asList(
            new Student(102, "Holland"), new Student(100, "Putin"), new Student(101, "Obama"));
        Collections.sort(students); // sắp xếp theo code
        System.out.println(students);
        Collections.sort(students, new Comparator<Student>() { // sắp xếp theo name
            @Override public int compare(Student o1, Student o2) {
                return o1.name.compareTo(o2.name);
            }
        });
    }
}
```

```

    }
    });
    System.out.println(students);
}
}

```

Sắp xếp theo thứ tự ngược lại:

- Nếu dùng Comparable<E> để so sánh các element trong collection.

```
Collections.sort(students, Collections.reverseOrder());
```

- Nếu dùng đối tượng hàm Comparator<E>, có hai cách.

+ truyền đối tượng hàm cho phương thức Collections.reverseOrder():

```

Collections.sort(students, Collections.reverseOrder(new Comparator<Student>() {
    @Override public int compare(Student o1, Student o2) {
        return o1.name.compareTo(o2.name);
    }
}));

```

+ hoặc gọi phương thức reversed() của đối tượng hàm:

```

Collections.sort(students, new Comparator<Student>() {
    @Override public int compare(Student o1, Student o2) {
        return o1.name.compareTo(o2.name);
    }
}).reversed();

```

b) java.util.Arrays

- Sắp xếp

java.util.Arrays cung cấp các phương thức static dùng để sắp xếp.

Với mỗi kiểu dữ liệu cơ bản (ngoại trừ kiểu boolean) và kiểu Object, có một cặp phương thức sort(). Ví dụ với kiểu int[].

```

public static void sort(int[] a)
public static void sort(int[] a, int fromIndex, int toIndex)

```

Với Object[], các đối tượng phải cài đặt interface Comparable và thứ tự sắp xếp xác định thông qua phương thức compareTo().

```

public static void sort(Object[] a)
public static void sort(Object[] a, int fromIndex, int toIndex)

```

Một cặp phương thức sort() cũng được định nghĩa cho kiểu generic, sẽ sắp xếp dựa trên Comparator chỉ định, thay vì dùng Comparable.

```

public static <T> void sort(T[] a, Comparator<? super T> c)
public static <T> void sort(T[] a, int fromIndex, int toIndex, Comparator<? super T> c)

```

- Tìm kiếm

java.util.Arrays cũng cung cấp các cặp phương thức tìm kiếm cho các kiểu dữ liệu cơ bản (ngoại trừ kiểu boolean) và kiểu Object. Mảng phải được sắp xếp trước khi áp dụng phương thức binarySearch().

```

public static int binarySearch(int[] a, int key)
public static int binarySearch(int[] a, int fromIndex, int toIndex, int key)

```

Tìm kiếm các Object có cài đặt Comparable.

```

public static int binarySearch(Object[] a, Object key)
public static int binarySearch(Object[] a, int fromIndex, int toIndex, Object key)

```

Tìm kiếm với kiểu generic, dựa trên Comparator chỉ định.

```

public static <T> int binarySearch(T[] a, T key, Comparator<? super T> c)
public static <T> int binarySearch(T[] a, T key, int fromIndex, int toIndex, Comparator<? super T> c)

```

- Các tác vụ khác

+ So sánh hai mảng, ví dụ so sánh hai mảng kiểu int[].

```
public static boolean equals(int[] a1, int[] a2)
```

+ Sao chép mảng hoặc một phần mảng, mảng trả về bị xén hoặc đệm thêm 0 nếu cần. Ví dụ với kiểu int[].

```

public static int[] copyOf(int[] original, int newLength)
public static int[] copyOfRange(int[] original, int from, int to)

```

Với kiểu generic:

```

public static <T> T[] copyOf(T[] original, int newLength)
public static <T> T[] copyOfRange(T[] original, int from, int to)
public static <T,U> T[] copyOf(U[] original, int newLength, Class<? extends T[]> newType)
public static <T,U> T[] copyOfRange(U[] original, int from, int to, Class<? extends T[]> newType)

```

+ Đổ đầy mảng hoặc một phần mảng với trị chỉ định. Ví dụ với kiểu int[].

```

public static void fill(int[] a, int value)
public static void fill(int[] a, int fromIndex, int toIndex, int value)

```

+ Trả về thể hiện dạng chuỗi nội dung của mảng chỉ định.

```
public static String toString(int[] a)
```

c) java.util.Collections

- Sắp xếp

Chỉ áp dụng với List, các element cài đặt Comparable. Không áp dụng với Set, Queue và Map. Ngoài ra, SortedSet (TreeSet) và SortedMap (TreeMap) đã được sắp xếp tự động.

+ Các element cài đặt Comparable

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

+ Chỉ định Comparator

```
public static <T> void sort(List<T> list, Comparator<? super T> c)
```

- Tìm kiếm

List phải được sắp xếp trước khi áp dụng tìm kiếm nhị phân.

```
public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)
```

```
public static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)
```

- Các tác vụ khác

+ Tìm element lớn nhất (max) và nhỏ nhất (min). Collection có thứ tự tự nhiên hoặc chỉ định bởi Comparator.

```
public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> c)
```

```
public static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> c)
```

```
public static <T> T max(Collection<? extends T> c, Comparator<? super T> comp)
```

```
public static <T> T min(Collection<? extends T> c, Comparator<? super T> comp)
```

+ Các tác vụ `copy()`, `fill()`, ...

d) Các collection đồng bộ (synchronized)

Đa số các collection được cài đặt như ArrayList, HashMap, HashSet thì không đồng bộ cho multithreaded, ngoại trừ Vector và Hashtable đã lạc hậu. Thay vì dùng chúng, bạn có thể tạo Collection, List, Set, SortedSet, Map và SortedMap đồng bộ (an toàn thread) thông qua các phương thức static của Collections:

```
public static <T> Collection<T> synchronizedCollection(Collection<T> c)
```

```
public static <T> List<T> synchronizedList(List<T> list)
```

```
public static <T> Set<T> synchronizedSet(Set<T> set)
```

```
public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> set)
```

```
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> map)
```

```
public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> map)
```

Ví dụ dùng List đồng bộ trong khối synchronized:

```
List list = Collections.synchronizedList(new ArrayList());
```

// ...

```
synchronized(list) {
```

```
    Iterator iter = list.iterator();
```

```
    while (iter.hasNext())
```

```
        iter.next();
```

// ...

}

Generics

Collection chứa các đối tượng kiểu Object, vì vậy chúng có thể nhận và trả về các kiểu dữ liệu khác nhau. Bạn phải ép kiểu xuống một cách tường minh để lấy đúng kiểu đối tượng bạn cần. Bạn cũng phải dùng lớp bao (wrapper) để đưa kiểu cơ bản vào collection. Điều này không an toàn kiểu, ví dụ có thể ném ClassCastException trong thời gian chạy nếu ép kiểu xuống bị sai.

```
ArrayList list = new ArrayList();
```

```
list.add(0, new Integer(42));
```

```
int total = ((Integer) list.get(0)).intValue();
```

1. Generic cấp lớp (class level)

Từ Java 5, giải pháp cho vấn đề này là generics, một cách viết code độc lập với kiểu dữ liệu. Generics cung cấp cho trình biên dịch thông tin về kiểu dữ liệu của các element trong collection. Kiểu sẽ được xác định tự động trong thời gian chạy, tránh phải ép kiểu tường minh như với collection.

Thông tin về kiểu khi truyền đến collection được đóng trong cặp <>.

Nếu sử dụng generics cho code trên, kết hợp với khả năng autoboxing, code trở nên đơn giản và rõ ràng hơn:

- Lớp ArrayList được thiết kế thành generic với kiểu chung ArrayList<T>.

- Người sử dụng sẽ chỉ định kiểu cụ thể khi dùng:

```
ArrayList<Integer> list = new ArrayList<Integer>();
```

```
list.add(0, 42);
```

```
int total = list.get(0);
```

Kiểu T của một collection thường được gọi là *generic type* (kiểu chung), khai báo giống như sau:

```
interface Box<T> {
```

```
    void box(T t);
```

```
    T unbox();
```

}

<T> gọi là tham số kiểu (type parameter) và T được gọi là kiểu được tham số hóa (parameterized type), hàm ý khi viết một kiểu như List<Student> chúng ta truyền kiểu Student cụ thể đến List như một tham số. Kiểu được tham số hóa T thường đặt theo quy ước: <E> element, <T> type, <K,V> key và value, <N> number, <E,S,U,V>: tham số thứ nhất, thứ hai, thứ ba và thứ tư.

Thực tế, trình biên dịch thay thế các tham chiếu đến kiểu T thành kiểu Object, rồi thực hiện kiểm tra kiểu và chèn vào các toán tử ép kiểu xuống theo yêu cầu. Vì vậy, generic có thể dùng chung cho mọi kiểu và tương thích ngược với collection không dùng generic. Điều này gọi là *type erasure* (xóa kiểu), nghĩa là xóa thông tin kiểu chung T và dùng generic như collection chứa các Object.

Từ Java 7, khi tạo thực thể của một generic type, cho phép thiếu trị của tham số kiểu tại vế phải của phát biểu gán, gọi là cú pháp *diamond*.

```
List<Employee> salesList = new ArrayList<>();
```

Để chuyển collection cũ (gọi là kiểu thô, raw type) thành generic type, dùng ép kiểu:

```
List someThings = getSomeThings();
// ép kiểu không an toàn, nhưng chúng ta biết rằng someThings chứa các element kiểu String
List<String> myStrings = (List<String>)someThings
```

Ví dụ, lớp Stack Generic:

```
import java.math.BigDecimal;
import java.math.MathContext;
class Stack<T> {
    protected T[] stack = (T[]) new Object[100];
    int top = -1;

    void push(T data) {
        if (top < 100) stack[++top] = data;
    }

    T pop() {
        return isEmpty() ? null : (T) stack[top--];
    }

    T peek() {
        return isEmpty() ? null : (T) stack[top];
    }

    boolean isEmpty() {
        return top < 0;
    }
}

public class TestStack {
    public static void main(String[] args) {
        System.out.println("Creating 'BigDecimal' stack:");
        Stack<BigDecimal> bigDecimalStack = new Stack<>();
        bigDecimalStack.push(new BigDecimal("12.5E+7"));
        bigDecimalStack.push(new BigDecimal(125, MathContext.DECIMAL128));
        System.out.println(bigDecimalStack.pop());
        System.out.println(bigDecimalStack.pop());
        System.out.println();
    }
}
```

Bạn cũng có thể tạo interface hoạt động với kiểu dữ liệu generic, tương tự như tạo lớp generic. Ví dụ như interface Map trong Collection Framework:

```
public interface Map<K,V>
```

Lớp HashMap cài đặt interface này:

```
public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable
```

2. Generic cấp phương thức (method level)

Ngoài cách dùng generic cho lớp, gọi là generic cấp lớp, kiểu generic cũng được chỉ định như tham số trong signature và thân của các phương thức, gọi là generic cấp phương thức.

```
public class Main {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        String[] array = { "guava", "mango", "rambutan" };
        TestGeneric.<String>arrayToList(array, list);
        for (String s : list) System.out.println(s);
    }
}

class TestGeneric {
    public static <E> void arrayToList(E[] a, ArrayList<E> list) {
        list.addAll(Arrays.asList(a));
    }
}
```

Tương tự như generic cấp lớp, khi trình biên dịch gặp một generic cấp phương thức, nó cũng thực hiện kiểm tra an toàn kiểu, rồi xóa kiểu E thành Object. Bạn có thể dùng cú pháp tùy chọn để chỉ định kiểu cho một generic cấp phương thức: đưa kiểu cụ thể vào cặp <> rồi đặt giữa toán tử (.) và tên phương thức, như ví dụ trên.

3. Ký tự đại diện (wildcard)

Xem ví dụ sau, do chưa xác định kiểu các element trong list, tạm dùng kiểu Object.

```
public static void printList(List<Object> list) {
    for (Object o : list) System.out.println(o);
}
```

```
public static void main(String[] args) {
    List<String> list = Arrays.asList(new String[] { "banana", "orange", "mango" });
    printList(list); // lỗi biên dịch, List<String> không thể chuyển thành List<Object>
}
```

Để giải quyết vấn đề này, một ký tự đại diện được cung cấp trong generic, ký tự ?, đại diện cho kiểu nào đó không rõ.

```
public static void printList(List<?> list) {
    for (Object o : list) System.out.println(o);
}
```

Ngoài ra, nếu muốn tạo lớp generic chỉ hoạt động trong phạm vi các kiểu quy định quy định, dùng *bounded types* (kiểu ràng buộc), gọi như vậy vì ta đã ràng buộc lớp với một phạm vi chỉ định của các kiểu dữ liệu để hoạt động trên đó.

Danh sách các ký tự đại diện (wildcard) có thể áp dụng cho tham số kiểu như sau:

Type Parameter	Mô tả
<T>	Kiểu không ràng buộc; tương tự <T extends Object>
<T, P>	Kiểu không ràng buộc; tương tự <T extends Object> và <P extends Object>
<T extends P>	Kiểu ràng buộc lên (upper bounded); kiểu chỉ định T phải là kiểu con (subtype) của kiểu P
<T extends P & S>	Kiểu ràng buộc lên; kiểu chỉ định T phải là subtype của kiểu P và cài đặt kiểu S
<T super P>	Kiểu ràng buộc xuống (lower bounded); kiểu chỉ định T phải là kiểu cha (supertype) của kiểu P
<?>	Ký tự đại diện không ràng buộc; kiểu đối tượng bất kỳ, tương tự <? extends Object>
<? extends P>	Ký tự đại diện có ràng buộc; kiểu nào đó phải là subtype của kiểu P
<? extends P & S>	Ký tự đại diện có ràng buộc; kiểu nào đó phải là subtype của kiểu P và cài đặt kiểu S
<? super P>	Ký tự đại diện ràng buộc xuống; kiểu nào đó phải là supertype của kiểu P

Các ký tự đại diện không ảnh hưởng đến nguyên tắc xóa kiểu khi trình biên dịch làm việc với generic.

Khi dùng ký tự đại diện, bạn theo nguyên tắc get/put sau:

- Dùng extends khi chỉ lấy trị ra (get).

```
interface List<E> extends Collection<E> {
    boolean addAll(Collection<? extends E> c);
}
```

```
List<Integer> srcList = new ArrayList<>();
srcList.add(0);
List<Integer> destList = new ArrayList<>();
destList.addAll(srcList);
```

- Dùng super khi chỉ đặt trị vào (put).

```
public class Collections {
    public static<T> boolean addAll(Collection<? super T> c, T... elements) {...}
}
```

```
List<Number> sList = new ArrayList<>();
sList.add(0);
Collections.addAll(sList, (byte)1, (short)2);
```

- Không dùng ký tự đại diện khi thực hiện cả hai get/put.

Khi dùng generic mà không chỉ định tham số kiểu thì kiểu dùng gọi là raw type (kiểu thô). Từ Java 5, trình biên dịch kỳ vọng bạn dùng generic với tham số nên sẽ sinh ra ghi chú (note) "uses unchecked or unsafe operations".

Duyệt collection

1. Iterator

Bạn có thể duyệt một collection bằng cách dùng một *iterator* (bộ lặp). Interface Iterator cơ bản cho phép bạn duyệt đi tới (forward) qua một collection bất kỳ, giống như dùng một con trỏ (cursor). Trong trường hợp duyệt một Set, thứ tự duyệt sẽ không xác định. Đối tượng lớp List hỗ trợ thêm một ListIterator, cho phép duyệt list theo chiều ngược lại.

```
List<String> list = new ArrayList<>();
// thêm vào một số phần tử
Iterator<String> it = list.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

Iterator cung cấp nhiều phương thức xử lý element nơi iterator chỉ đến. Phương thức remove() cho phép loại bỏ element nơi iterator chỉ đến. Nếu việc loại bỏ không được hỗ trợ bởi loại collection đang duyệt, UnsupportedOperationException sẽ được ném ra. Phương thức set() cho phép thay đổi element nơi iterator chỉ đến. Phương thức add() chèn element mới vào collection ngay trước con trỏ của iterator. Nghĩa là nếu bạn gọi previous() ngay sau khi gọi add(), iterator chỉ element mới thêm vào. Khi dùng ListIterator, di chuyển qua list theo hướng tới (dùng next()) hoặc ngược lại (dùng previous()). Nếu dùng previous() ngay sau next() (hoặc ngược lại) bạn sẽ nhận cùng một element.

```
String[] colors = {"gold", "white", "brown", "blue", "gray", "silver"};
List<String> list = new LinkedList<>();
list.addAll(Arrays.asList(colors)); // từ mảng thành list
ListIterator<String> it = list.listIterator();
// chuyển các element thành chữ hoa
while (it.hasNext()) {
    it.set(it.next().toUpperCase());
}
```

```

}
// in list theo thứ tự ngược lại, iterator đang chỉ cuối list
// hoặc dùng: it = list.listIterator(list.size());
while (it.hasPrevious()) {
    System.out.printf("%s ", it.previous());
}
// giữ lại hai phần tử đầu và cuối list
list.subList(1, list.size()-1).clear();
colors = list.toArray(new String[list.size()]); // từ list thành mảng

```

2. Vòng lặp for tăng cường

Iterator hữu dụng nhưng khó hiểu và phức tạp, phải gọi nhiều phương thức với cấu trúc thiếu trực quan. Ví dụ:

```

public void deleteAll(Collection<NameList> c) {
    for (Iterator<NameList> i = c.iterator(); i.hasNext(); i.next()) {
        NameList nl = i.next();
        nl.deleteItem();
    }
}

```

Từ Java 5, bạn có thể duyệt collection bằng cách đơn giản, dễ hiểu và an toàn hơn: dùng vòng lặp for tăng cường.

```

public void deleteAll(Collection<NameList> c) {
    for (NameList nl : c) {
        nl.deleteItem();
    }
}

```

Khi duyệt mảng, bạn có thể duyệt qua các element của mảng mà không cần dùng một biến chỉ số (index) nào.

```

public int sum(int[] array) {
    int result = 0;
    for (int element : array) {
        result += element;
    }
    return result;
}

```

Với vòng lặp lồng, dùng vòng lặp for tăng cường giúp code trở nên đơn giản, dễ hiểu, tránh nhầm lẫn:

```

List<Subject> subjects = ...;
List<Teacher> teachers = ...;
List<Course> courseList = new ArrayList<Course>();
for (Subject subject : subjects)
    for (Teacher teacher : teachers)
        courseList.add(new Course(subject, teacher));

```

3. Tác vụ hàm (functional operation)

Từ Java 8, collections API cung cấp tác vụ duyệt collection theo cách tiếp cận lập trình hàm, phương thức `forEach()`. Phương thức `forEach()` thực hiện code trên từng thành viên của collection.

Java 8 cũng cung cấp phương thức `removeIf()`, dùng loại bỏ một thành phần của collection nếu điều kiện kiểm tra là true.

```

class Thing {
    String label;
    String value;
    public Thing(String label, String value) {
        this.label = label;
        this.value = value;
    }
}

public class Main {
    public static void main(String[] args) {
        List<Thing> list = new ArrayList<>();
        list.add(new Thing("100", "Computer"));
        list.add(new Thing("101", "Boomerang"));
        list.add(new Thing("102", "Dictionary"));
        list.forEach(m -> System.out.println(m.label));
        Map<String, Thing> map = new HashMap<>();
        list.forEach(m -> map.put(m.label, m));
        map.forEach((l, v) -> System.out.println(l + "|" + v.value));
        list.removeIf(m -> m.value.equals("Putin"));
    }
}

```

4. Spliterator

Java 8 giới thiệu một interface mới là `Spliterator`, một iterator đặc biệt có thể duyệt collection. Interface `Collection` được cập nhật phương thức `spliterator()` để trả về `Spliterator`. `Spliterator` có thể phân vùng một số phần tử của nó thành

một `Splitterator` mới. Điều này cho phép xử lý song song các bộ phận khác nhau của collection, nhưng lưu ý rằng chính `Splitterator` không cung cấp hành vi xử lý song song. Thay vào đó, `Splitterator` hỗ trợ duyệt song song của các phần vùng phù hợp của collection. Điều này giải quyết vấn đề phân chia dữ liệu thành các đơn vị con có kích thước phù hợp có thể được xử lý song song.

`Splitterator` cần thiết nếu bạn muốn viết loại collection riêng và có ý định tối ưu hóa các hoạt động song song với chúng.

```
// lấy Splitterator (nguyên collection)
Splitterator<Job> splitterator = jobs.splitterator();
// tách splitterator thành hai và trả về splitterator đầu
Splitterator<Job> newPartition = splitterator.trySplit();
// trả về số phần tử ước lượng của splitterator
System.out.println(" " + newPartition.estimateSize());
// trả về số phần tử chính xác của splitterator
System.out.println(" " + splitterator.getExactSizeIfKnown());
// duyệt splitterator thứ hai
splitterator.forEachRemaining(System.out::println);
// lấy tính chất của splitterator thứ hai, là kết quả OR của nhiều tính chất
System.out.println(splitterator.characteristics());
// xác định tính chất cụ thể, thường là ORDERED | SIZED | SUBSIZED
if (splitterator.hasCharacteristics(Splitterator.ORDERED)) System.out.println("ORDERED");
if (splitterator.hasCharacteristics(Splitterator.DISTINCT)) System.out.println("DISTINCT");
if (splitterator.hasCharacteristics(Splitterator.SORTED)) System.out.println("SORTED");
if (splitterator.hasCharacteristics(Splitterator.SIZED)) System.out.println("SIZED");
if (splitterator.hasCharacteristics(Splitterator.CONCURRENT)) System.out.println("CONCURRENT");
if (splitterator.hasCharacteristics(Splitterator.IMMUTABLE)) System.out.println("IMMUTABLE");
if (splitterator.hasCharacteristics(Splitterator.NONNULL)) System.out.println("NONNULL");
if (splitterator.hasCharacteristics(Splitterator.SUBSIZED)) System.out.println("SUBSIZED");
```

Stream API (Java 8)

1. Stream API

Stream API là API sử dụng Lambda được giới thiệu trong JDK 8, cung cấp thao tác trên các collection của Java. Stream API khắc phục những hạn chế của Collection API:

- Collection API không cung cấp các cấu trúc mức độ cao để truy vấn dữ liệu, các nhà phát triển buộc phải viết rất nhiều mã thủ tục (boilerplate) cho các nhiệm vụ tầm thường.

- Ngôn ngữ gập hạn chế khi xử lý song song dữ liệu trong Collection API.

Mặc dù stream nhìn bên ngoài trông giống collection, nhưng stream rất khác collection:

- Stream không phải là cấu trúc dữ liệu, nó không lưu dữ liệu như collection, mà nghiêng về tính toán.
- Stream chỉ tính toán khi có truy cập (lazy). Điều này cho phép bạn tạo các stream dữ liệu vô hạn.
- Stream không dùng iterator bên ngoài để duyệt các phần tử. Người sử dụng Stream API chỉ cung cấp các tác vụ mà họ muốn áp dụng, và iterator nội áp dụng các tác vụ đó cho mọi phần tử của collection bên dưới.
- Chỉ xử lý stream một lần.

Các tác vụ của Stream API thường thực hiện thành một chuỗi liên tiếp nhau, các tác vụ này có thể là:

- Tác vụ kết thúc (terminal) trả về kết quả là một kiểu nhất định.
- Tác vụ trung gian (intermediate) trả về stream mới để bạn gọi phương thức tiếp, tạo thành dây chuyền (chain) gọi phương thức. API xây dựng như vậy gọi là *fluent API*.

Các tác vụ của Stream API có mức trừu tượng cao, khai báo tương tự như truy vấn SQL. Điều này có nghĩa là khi dùng Stream API, bạn viết những gì bạn muốn làm hơn là chỉ dẫn làm thế nào để thực hiện được điều đó (imperative style). Lưu ý các tác vụ stream là non-mutating, nghĩa là chúng không làm biến đổi collection ban đầu.

Trước hết, phải tạo stream từ một nguồn, ví dụ `java.util.Collection` như danh sách hoặc tập hợp (không hỗ trợ map). Sau đó, áp dụng chuỗi các tác vụ intermediate, mỗi bước sinh ra một stream mới. Cuối cùng, gọi tác vụ terminal để sinh ra kết quả.

Ví dụ dùng Stream API.

Các lớp sau cần cho các ví dụ minh họa:

```
enum TaskType {
    READING("Reading"), WRITING("Writing"), CODING("Coding"), SPEAKING("Speaking");
    private String value;
    private TaskType(String value) { this.value = value; }
    public String getValue() { return value; }
}
```

```
class Task {
    int id;
    String title;
    TaskType type;
    LocalDate createdOn;
    final DateTimeFormatter df = DateTimeFormatter.ofPattern("dd/MM/yyyy");

    public Task(int id, String title, TaskType type, String date) {
        this.id = id;
        this.title = title;
        this.type = type;
        this.createdOn = LocalDate.parse(date, df);
    }
}
```

```

    }

    public int getId() { return id; }
    public String getTitle() { return title; }
    public TaskType getType() { return type; }
    public LocalDate getCreatedOn() { return createdOn; }
    @Override public String toString() { return String.format("%s, %s", title, createdOn); }
}

```

```

List<Task> tasks = Arrays.asList(
    new Task(1, "Master and Margarita", TaskType.READING, "02/09/1990"),
    new Task(2, "My poem", TaskType.WRITING, "30/04/2000"),
    new Task(3, "Les Miserables", TaskType.READING, "19/05/2012"));

```

```

// (1) cách cũ, thao tác dùng Collection API
List<Task> readingTasks = new ArrayList<>();
for (Task task : tasks)
    if (task.type == TaskType.READING) readingTasks.add(task);
Collections.sort(readingTasks, new Comparator<Task>() {
    @Override public int compare(Task t1, Task t2) {
        return t1.getTitle().length() - t2.getTitle().length();
    }
});
for (Task readingTask : readingTasks)
    System.out.println(readingTask.getTitle());

```

```

// (2) thao tác dùng Stream API
tasks.stream()
    .filter(task -> task.type == TaskType.READING)
    .sorted((t1, t2) -> t1.getTitle().length() - t2.getTitle().length())
    .map(Task::getTitle)
    .collect(Collectors.toList())
    .forEach(System.out::println);

```

Dây chuyền tác vụ trong ví dụ trên bao gồm:

- stream** tạo ra một stream bằng cách gọi phương thức stream() trên bộ collection nguồn, tức List<Task> tasks.
- filter** nhận một Predicate là biểu thức Lambda, lọc các phần tử trong stream phù hợp với điều kiện được định nghĩa bởi Predicate này.
- sorted** nhận một Comparator được định nghĩa bởi biểu thức Lambda, trả về một stream với các phần tử được sắp xếp bởi Comparator này.
- map** nhận một Function<T, R>, áp dụng trên mỗi phần tử của stream, dựa trên khóa T tạo ra stream mới chứa các phần tử R (Stream<R>). Trong trường hợp R là Stream<S>, tức map() trả về một Stream<Stream<S>> như ví dụ dưới, "làm phẳng" nó bằng flatMap(). flatMap() thay thế một phần tử trong stream thành một stream, định nghĩa bởi Function<T, R> nhận được, sau đó nối các stream này lại với nhau.


```

Pattern.compile("")
    .splitAsStream(line) // Stream<String>
    .map(s -> s.split("")) // Stream<Stream<String>>
    .flatMap(Arrays::stream) // Stream<String>
    .forEach(System.out::println)
            
```
- collect** thu thập kết quả của các tác vụ thực hiện trên stream vào một đích, ở đây là List. Xem phần Collectors bên dưới.
- forEach** nhận một Consumer là biểu thức Lambda, đây là tác vụ kiểu terminal, thực hiện Consumer với mỗi phần tử của List.

2. Tạo stream

- Từ collection

```

String[] tags = {"java", "git", "lambdas", "machine-learning"};
Arrays.asList(tags).stream().forEach(System.out::println);

```

- Từ mảng

```

String[] tags = {"java", "git", "lambdas", "machine-learning"};
Stream.of(tags).forEach(System.out::println);
Arrays.stream(tags).forEach(System.out::println);
Arrays.stream(tags, 1, 3).forEach(System.out::println);

```

Mảng rỗng:

```

Stream<String> silence = Stream.empty();

```

- Tạo stream vô hạn (infinite stream)

```

Stream<Double> randoms = Stream.generate(Math::random);
randoms.limit(4).forEach(System.out::println);
Stream<BigInteger> integers = Stream.iterate(BigInteger.ZERO, n -> n.add(BigInteger.ONE));
integers.limit(10).forEach(System.out::println);

```

- Từ tập tin

Cho tập tin contents.txt có nội dung như sau, hãy tách lấy phần thứ ba (phân cách bởi dấu :) trên dòng thứ hai.

a:b:c:d

e:f:g:h

i:j:k:l

```
try (Stream<String> lines = Files.lines(Paths.get("contents.txt"))) {
    lines.skip(1)
        .map(line -> line.split(":"))
        .findFirst()
        .ifPresent(a -> System.out.println(a[2]));
}
```

- Từ một chuỗi

String contents = "1 2 3 4 5";

Pattern pattern = Pattern.compile("\\s+");

System.out.println(pattern.splitAsStream(contents).collect(Collectors.joining(", ")));

- Stream với kiểu cơ bản

Với các kiểu cơ bản (primitive) như int, long, double, Java 8 cũng cung cấp các stream đặc biệt IntStream, LongStream, DoubleStream tương ứng, gọi là các stream số (numeric stream).

```
IntStream.rangeClosed(1, 10)
    .filter(e1 -> e1 % 2 == 0)
    .limit(12)
    .forEach(System.out::println);
```

3. Tác vụ terminal

Một số tác vụ terminal khác:

count trả về số phần tử (long) trong stream.

```
private long countReadingTasks(List<Task> tasks) {
    return tasks.stream()
        .filter(task -> task.type == TaskType.READING)
        .count();
}
```

anyMatch nhận một Predicate, trả về true nếu có bất kỳ phần tử nào so trùng điều kiện được định nghĩa bởi Predicate.

```
System.out.println(
    tasks.stream().anyMatch(task -> task.createdOn.isAfter(LocalDate.ofYearDay(2000, 1)))
);
```

allMatch nhận một Predicate, trả về true nếu tất cả phần tử đều so trùng điều kiện được định nghĩa bởi Predicate.

```
System.out.println(
    tasks.stream().allMatch(task -> task.createdOn.isAfter(LocalDate.ofYearDay(2000, 1)))
);
```

Các tác vụ cùng nhóm noneMatch (không so trùng), findAny, findFirst (lấy phần tử trong stream).

reduce nhận một biểu thức Lambda, thu giảm stream thành một trị. Nghĩa là thực hiện một thao tác tích lũy lên các phần tử của stream (nối chuỗi, tính tổng, ...). Thao tác tích lũy này mô tả bởi biểu thức Lambda:

```
private String joinAllTaskTitles(List<Task> tasks) {
    return tasks.stream()
        .map(Task::getTitle)
        .reduce((first, second) -> first + " , " + second)
        .get();
}
```

Ví dụ khác, tính tổng các trị trong mảng.

```
int count = Stream.of(1, 3, 5, 7, 9).reduce(0, (acc, element) -> acc + element);
```

4. Tác vụ intermediate

Một số tác vụ intermediate khác:

distinct trả về danh sách các phần tử phân biệt của stream.

```
private List<Task> allDistinctTasks(List<Task> tasks) {
    return tasks.stream().distinct().collect(Collectors.toList());
}
```

Phương thức equals() của phần tử được dùng để xác định hai phần tử trùng.

limit giới hạn stream kết quả với số phần tử chỉ định, thường dùng khi phân trang (pagination).

Ví dụ, lấy n task thuộc trang page:

```
List<String> tasksOfPage = tasks.stream()
    .sorted(Comparator.comparing(Task::getCreatedOn).reversed())
    .map(Task::getTitle)
    .skip(page * n)
    .limit(n)
    .collect(Collectors.toList());
```

concat nối hai stream, trả về stream mới.

peek sinh ra stream có cùng phần tử với stream gốc, gọi một hàm cho mỗi phần tử lấy được. Thường dùng để xem các phần tử của một stream.

```
Double[] powers = Stream.iterate(1.0, p -> p * 2)
    .peek(System.out::println)
    .limit(5)
```

```

        .toArray(Double[]::new);
skip    nhận một số n, trả về stream với n phần tử đầu tiên bị bỏ qua.

```

5. Collectors

Trong Stream API, tác vụ terminal collect thu giảm một dòng stream thành một trị, trị đó có thể là Collection, Map hoặc một đối tượng giá trị (value object). Phương thức collect nhận tham số là một Collector. Một số các Collector hữu dụng được lấy bằng cách gọi các phương thức static của lớp Collectors.

Ta dùng collect để:

- Thu giảm stream thành một giá trị duy nhất: kết quả thực hiện stream có thể được thu giảm thành một giá trị. Giá trị đó có thể là một Collection hoặc số như int, double, ... hoặc một đối tượng giá trị chỉ định.

Thu thập các title của các phần tử Task trong stream vào List.

```

tasks.stream()
    .map(Task::getTitle).collect(Collectors.toList())
    .forEach(System.out::println);

```

Thu thập các title của các phần tử Task trong stream vào Set.

```

tasks.stream()
    .map(Task::getTitle).collect(Collectors.toSet())
    .forEach(System.out::println);

```

Thu thập các title của các phần tử Task trong stream vào Map.

```

tasks.stream()
    .collect(Collectors.toMap(Task::getId, identity(), (v1, v2) -> v2))
    .forEach((k, v) -> System.out.println "[" + k + "]: " + v));

```

Lấy task mới nhất.

```

Task task = tasks.stream()
    .collect(Collectors.collectingAndThen(Collectors.maxBy(Comparator.comparing(Task::getCreatedOn)),
        Optional::get));

```

Đếm số task, summingInt nhận tham số là một ToIntFunction trả về int, trong ví dụ dưới là đếm từng task, tiện dụng nếu Task có tác vụ đếm nào đó, ví dụ đếm các tag được gán cho cho một task.

```

int count = tasks.stream().collect(Collectors.summingInt(t -> 1));

```

Lấy title của các task và nối lại thành một danh sách.

```

String info = tasks.stream()
    .map(Task::getTitle)
    .collect(Collectors.joining(", "));

```

- Nhóm các phần tử trong một stream: nhóm tất cả T trong một stream theo R. Kết quả sẽ là một Map<R, List<T>> với mỗi mục có chứa một R và các danh sách các T liên quan. Bạn có thể sử dụng collection bất kỳ thay cho List. Nếu bạn không cần tất cả các T liên quan đến một R, chẳng hạn nếu chỉ chọn T đầu tiên, bạn có thể lựa chọn tạo ra một Map<R, T>.

Nhóm các task theo type.

```

tasks.stream()
    .collect(Collectors.groupingBy(Task::getType))
    .forEach((k, v) -> System.out.println "[" + k + "]: " + v));

```

Nhóm các task theo type, trong một type, nhóm theo createdOn.

```

tasks.stream()
    .collect(Collectors.groupingBy(Task::getType, Collectors.groupingBy(Task::getCreatedOn)))
    .forEach((k, v) -> System.out.println "[" + k + "]: " + v));

```

- Phân nhóm (partitioning) các phần tử trong một stream.

Đôi khi bạn muốn phân chia một tập dữ liệu thành hai tập dữ liệu dựa trên một predicate. Ví dụ, chúng ta có thể phân chia các task thành hai nhóm bằng cách định nghĩa một hàm phân vùng - một chứa các task với ngày tạo (createdOn) sau ngày chỉ định, và một chứa các task còn lại.

```

tasks.stream()
    .collect(Collectors.partitioningBy(task -> task.getCreatedOn().isAfter(LocalDate.of(2000, 1, 1))))
    .forEach((k, v) -> System.out.println "[" + k + "]: " + v));

```

6. Stream song song (parallel streams)

Các stream có thể tuần tự hay song song, các tác vụ trên các stream tuần tự được thực hiện trong một thread đơn, trong lúc các tác vụ song song được thực hiện đồng hành trên nhiều thread. Ví dụ sau cho thấy hiệu suất khi dùng stream song song:

```

List<String> values = Stream.generate(UUID::randomUUID)
    .limit(1000000)
    .map(String::valueOf)
    .collect(Collectors.toList());
long t0 = System.nanoTime();
// stream tuần tự
long count = values.stream().sorted().count();
System.out.println(count);
long t1 = System.nanoTime();
long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
System.out.printf("Time elapsed: %d ms\n", millis);
Thay thế bằng stream song song và so sánh kết quả.
long count = values.parallelStream().sorted().count();

```

Nếu đã khởi tạo stream tuần tự:

```
long count = values.stream().parallel().sorted().count();
```

Các stream song song, dùng fork/join của Java 7, đòi hỏi chi phí cao so với stream tuần tự và chỉ thuận lợi trên kiến trúc máy tính đa nhân. Vì vậy bạn cần cân nhắc các yếu tố sau khi quyết định lựa chọn sử dụng stream song song.

- Số phần tử * chi phí trên mỗi phần tử phải lớn.
- Collection nguồn phải được phân chia (split) dễ dàng và hiệu quả.
- Thao tác trên từng phần tử là độc lập, không ảnh hưởng đến các phần tử khác.
- Máy tính sử dụng nên có kiến trúc đa nhân.

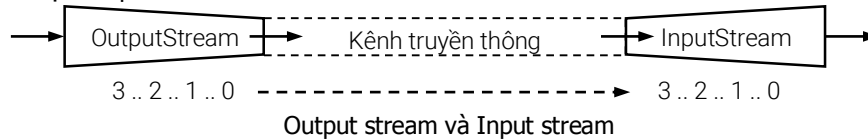
I/O Stream

Khái niệm

I/O Stream là đối tượng thể hiện một kết nối đến một kênh truyền thông (communication channel), như một kết nối TCP/IP hoặc một kết nối đến vùng đệm bộ nhớ. Các đối tượng chứa tin cũng được xem như kênh truyền thông: tập tin, thiết bị I/O, chương trình khác, mảng bộ nhớ, ... I/O Stream có thể đọc từ kênh truyền thông hoặc ghi đến nó.

Java API cung cấp một số lớp I/O stream cơ bản, có thể chia thành hai nhóm: input stream (stream nhập) và output stream (stream xuất).

I/O Stream thể hiện như một điểm đầu cuối (endpoint) hoặc một kênh truyền thông một chiều. Dữ liệu được ghi đến output stream sẽ được đọc tuần tự từ input stream.



Tính chất của I/O Stream:

- FIFO (first-in first-out): chuỗi dữ liệu được ghi đến output stream sẽ được đọc theo *đúng thứ tự* từ input stream tương ứng.
- Truy cập liên tục (sequential): chuỗi dữ liệu trong I/O stream được truy cập liên tục từng byte một nối tiếp nhau.
- Chỉ đọc (read-only) và chỉ ghi (write-only): output stream ghi dữ liệu vào kênh truyền thông và input stream đọc dữ liệu từ kênh truyền thông, không có lớp I/O Stream thực hiện cả đọc lẫn ghi.
- Blocking: khi I/O Stream đang đọc/ghi, nó tạm thời bị khóa (blocking) cho đến khi xong tác vụ, không cho truy cập từ tiến trình khác.

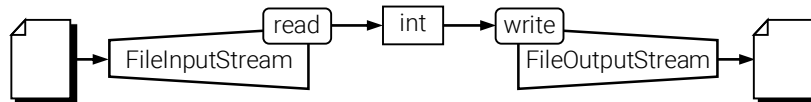
I/O Stream API được thiết kế theo design pattern Decorator. Điều này cho phép "lồng stream": gọi constructor của I/O Stream có tính năng tăng cường với đối số là đối tượng I/O Stream thích hợp có tính năng cơ bản, nhằm tăng tính năng của I/O Stream cơ bản. Thường bạn lồng nhiều I/O Stream liên tiếp nhau (chaining streams) cho đến khi nhận được I/O Stream có các phương thức phù hợp với nhu cầu sử dụng.

Các loại stream chính

1. Byte Stream

Các chương trình thường dùng byte stream để thực hiện xuất nhập dữ liệu 8-bit. Tất cả các lớp byte stream thừa kế hai lớp chính của gói java.io là InputStream và OutputStream. Đây là các lớp trừu tượng, dẫn xuất thành nhiều lớp con với những chức năng đặc thù khác nhau.

Các tác vụ chính của byte stream có thể tham khảo từ FileInputStream và FileOutputStream. Xem ví dụ sau:



```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class CopyBytes {
    public static void main(String[] args) throws IOException {
        try (FileInputStream in = new FileInputStream("source.txt");
            FileOutputStream out = new FileOutputStream("target.txt")) {
            int c;
            while ((c = in.read()) != -1)
                out.write(c);
        }
    }
}
```

Đôi khi ta cần đẩy dữ liệu ra khỏi I/O Stream, không cần đợi I/O Stream đầy. Điều này gọi là "súc" (flushing) I/O Stream. Một số I/O Stream hỗ trợ tự động "súc" (autoflush) khi có sự kiện nào đó, thiết lập bằng đối số trong constructor. Có thể "súc" I/O Stream bằng phương thức flush(), thường chỉ hiệu quả với I/O Stream có vùng đệm (buffer).

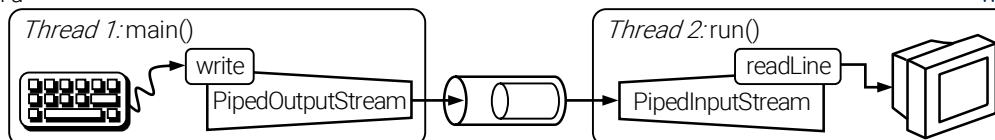
Khi sử dụng xong I/O Stream, cần đóng nó bằng phương thức close(), thường dùng trong khối finally. Đóng I/O Stream cũng là đóng kênh truyền thông liên kết với nó. Từ Java 7, I/O Stream được đóng tự động bằng cách dùng try-with-resources.

Hai loại byte stream thường được dùng và một số lớp của chúng:

- Các I/O Stream cơ bản: được cung cấp các hành vi cơ bản của InputStream và OutputStream để có thể thao tác với nhiều loại kênh truyền thông khác nhau. Để có thể thao tác linh hoạt hơn với loại kênh truyền thông gắn với chúng, các lớp dẫn xuất được cung cấp nhiều phương thức phù hợp hơn.

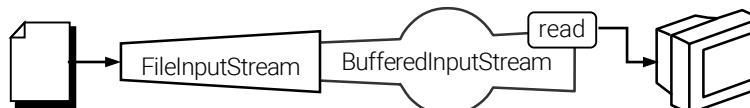
- ByteArrayOutputStream và ByteArrayInputStream dùng đọc/ghi dữ liệu từ một mảng byte. Thường dùng để chuyển dữ liệu vào một mảng, xử lý nó trước khi chuyển đi. Vì vậy từ lớp này có thể dùng để sinh ra các lớp I/O Stream phức tạp, như I/O Stream xử lý thông điệp, I/O Stream mã hóa dữ liệu, ...

- PipeOutputStream và PipeInputStream thường tạo thành từng cặp tương ứng. Dùng để truyền thông tin giữa các thread trong một ứng dụng đơn bằng I/O Stream.

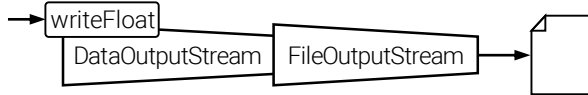


- Các I/O Stream lọc: cho phép thêm các chức năng tăng cường cho các I/O Stream của Java: `FilterInputStream` lồng vào một `InputStream` có sẵn, cung cấp các tính năng mở rộng cho `InputStream`; tương tự, một `FilterOutputStream` lồng vào một `OutputStream` có sẵn để tăng cường các chức năng của nó.

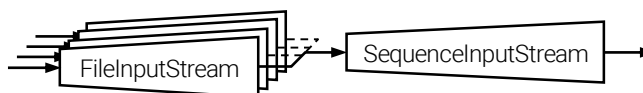
• `BufferedOutputStream` và `BufferedInputStream` cung cấp xuất/nhập có vùng đệm cho I/O Stream, tăng thêm hiệu quả cho xử lý xuất nhập. Constructor cần I/O Stream mà nó lồng vào và kích thước buffer mong muốn.



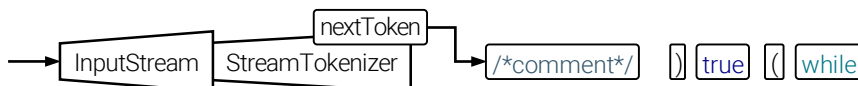
• `DataOutputStream` và `DataInputStream` tăng cường khả năng cho các `OutputStream` và `InputStream` bằng cách cho phép xử lý nhiều kiểu dữ liệu hơn như chuỗi, integer, số thực, ...



• `SequenceInputStream` kết nối với nhiều `InputStream` cùng lúc và nhận dữ liệu từ chúng theo thứ tự kết nối. Hình dưới minh họa một `SequenceInputStream` nhận một `Vector<FileInputStream>` làm đối số của constructor.



• `StreamTokenizer` cung cấp khả năng tách chuỗi nhận từ `InputStream` thành các token.

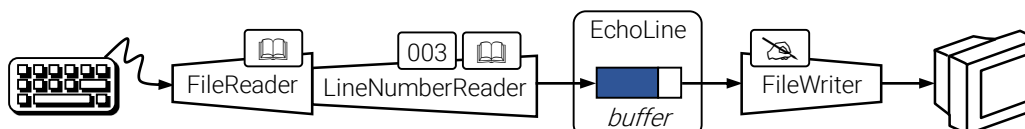


2. Character Stream

Java lưu trữ các ký tự bằng cách dùng Unicode. Các character stream tự động chuyển đổi qua lại giữa định dạng lưu trữ và tập ký tự vào/ra. Đa số các ứng dụng dùng character stream vì người dùng quen thuộc với chuỗi hơn, phần lớn các byte stream có các character stream tương ứng, một số byte stream đã lạc hậu và hiện được thay thế bằng character stream.

Output		Input	
byte stream	character stream	byte stream	character stream
<code>OutputStream</code>	<code>Writer</code>	<code>InputStream</code>	<code>Reader</code>
<code>FileOutputStream</code>	<code>FileWriter</code>	<code>FileInputStream</code>	<code>FileReader</code>
<code>ByteArrayOutputStream</code>	<code>CharArrayWriter</code>	<code>ByteArrayInputStream</code>	<code>CharArrayReader</code>
-	<code>StringWriter</code>	<code>StringBufferInputStream</code>	<code>StringReader</code>
<code>PipedOutputStream</code>	<code>PipedWriter</code>	<code>PipedInputStream</code>	<code>PipedReader</code>
<code>FilterOutputStream</code>	<code>FilterWriter</code>	<code>FilterInputStream</code>	<code>FilterReader</code>
<code>BufferedOutputStream</code>	<code>BufferedWriter</code>	<code>BufferedInputStream</code>	<code>BufferedReader</code>
<code>PrintStream</code>	<code>PrintWriter</code>	<code>PushbackInputStream</code>	<code>PushbackReader</code>
		<code>LineNumberInputStream</code>	<code>LineNumberReader</code>
<code>DataOutputStream</code>	-	<code>SequenceInputStream</code>	-
<code>ObjectOutputStream</code>	-	<code>DataInputStream</code>	-
-	<code>OutputStreamWriter</code>	<code>ObjectInputStream</code>	-
		-	<code>InputStreamReader</code>

Tất cả các character stream thừa kế `Reader` và `Writer`, cũng giống như byte stream (thừa kế `InputStream` và `OutputStream`). Các tác vụ chính của các lớp có thể tham khảo từ `FileReader` và `FileWriter`. Xem ví dụ sau:



```
import java.io.FileDescriptor;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.LineNumberReader;

public class EchoLine {
    public static void main(String[] args) throws IOException {
        // đầu vào: bàn phím, đầu ra: màn hình
        FileReader in = new FileReader(FileDescriptor.in);
```

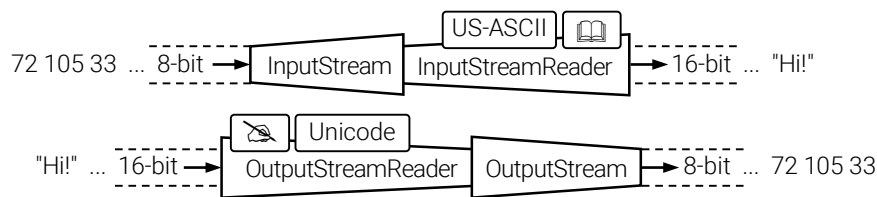
```

    FileWriter out = new FileWriter(FileDescriptor.out);
    // lồng stream
    LineNumberReader lineIn = new LineNumberReader(in);
    char[] buffer = new char[256];
    int numberRead;
    // đọc các dòng nhập từ Reader, đánh số, viết hoa toàn dòng, ghi vào Writer
    while ((numberRead = lineIn.read(buffer)) > -1) {
        String upper = new String(buffer, 0, numberRead).toUpperCase();
        out.write(lineIn.getLineNumber() + ": " + upper);
    }
    out.flush();
}
}

```

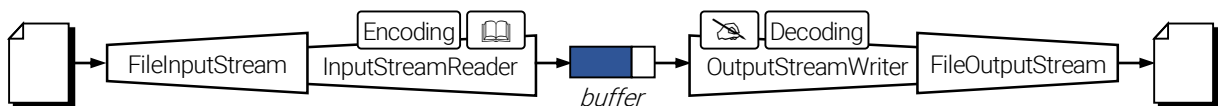
Các character stream có thể lồng (wrapper) các byte stream, cầu nối byte-character chủ yếu là:

- `InputStreamReader`: lồng với một `InputStream` (đọc các byte), nó sẽ chuyển các byte này thành ký tự Unicode theo bảng mã chỉ định.
- `OutputStreamWriter`: nhận các ký tự từ chương trình, chuyển nó thành byte bằng cách dùng bảng mã chỉ định, rồi chuyển các byte này ra `OutputStream` mà nó lồng vào.



Khi cầu nối byte-character được thiết lập, cần chỉ định tập ký tự được dùng bởi byte stream. Các tập ký tự hỗ trợ: latin1, latin2, latin3, latin4, cirillic, arabic, greek, hebrew, latin5, ASCII, Unicode, UnicodeBig, UnicodeBigUnmarked, UnicodeLittle, UnicodeLittleUnmarked, UTF8. Tập ký tự mặc định được xác định tùy theo hệ thống.

Ví dụ chuyển đổi tập tin từ tập ký tự này sang tập ký tự khác:

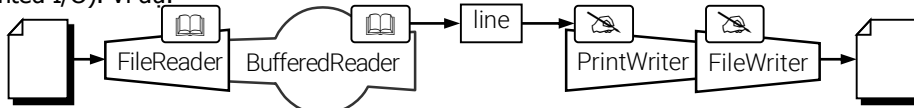


```

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
// cú pháp dùng, ví dụ: java Convert ASCII source.txt Unicode target.txt
public class Convert {
    public static void main(String[] args) throws IOException {
        if (args.length != 4)
            throw new IllegalArgumentException("Convert <srcEnc> <source> <destEnc> <dest>");
        try (InputStreamReader in = new InputStreamReader(new FileInputStream(args[1]), args[0]);
             OutputStreamWriter out = new OutputStreamWriter(new FileOutputStream(args[3]), args[2])) {
            char[] buffer = new char[16];
            int numberRead;
            while ((numberRead = in.read(buffer)) > -1) out.write(buffer, 0, numberRead);
        }
    }
}

```

Thao tác trên ký tự đôi khi không thuận tiện, ví dụ trường hợp ký tự kết thúc dòng: có thể là một cặp carriage-return/line-feed ("`\r\n`"), một carriage-return ("`\r`"), hoặc một line-feed ("`\n`") tùy hệ thống. Các chương trình có khuynh hướng thao tác trên từng dòng (line-oriented I/O). Ví dụ:



```

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
public class CopyLines {
    public static void main(String[] args) throws IOException {
        try (BufferedReader in = new BufferedReader(new FileReader("source.txt"));
             PrintWriter out = new PrintWriter(new FileWriter("target.txt"))) {
            String line;
            while ((line = in.readLine()) != null) out.println(line);
        }
    }
}

```

```

    }
}
}

```

Xuất/nhập dữ liệu

1. Xuất/nhập từ thiết bị I/O chuẩn

Java hỗ trợ ba thiết bị xuất/nhập chuẩn, do đó các I/O Stream gắn với nó được gọi là các I/O Stream chuẩn:

- Standard Input truy cập thông qua `System.in`, thông thường là bàn phím.
- Standard Output truy cập thông qua `System.out`, thông thường là màn hình.
- Standard Error truy cập thông qua `System.err`, thông thường là màn hình.

Các I/O Stream này được định nghĩa sẵn, không cần phải mở.

Ví dụ sau dùng `Scanner` gắn với thiết bị nhập chuẩn. Ngày, tháng và năm sẽ được nhập từ bàn phím (cách nhau bằng dấu space và không kiểm tra tính hợp lệ), ghi ra màn hình thứ trong tuần của ngày đó.

```

import java.util.Calendar;
import java.util.Scanner;
public class DayOfWeek {
    public static void main(String[] args) throws Exception {
        Scanner keyboard = new Scanner(System.in);
        Calendar calendar = Calendar.getInstance();
        calendar.set(Calendar.DAY_OF_MONTH, keyboard.nextInt());
        calendar.set(Calendar.MONTH, keyboard.nextInt() - 1);
        calendar.set(Calendar.YEAR, keyboard.nextInt());
        System.out.printf("%1$tA%n", calendar);
    }
}

```

Java cũng cung cấp đối tượng `Console`, trả về từ lời gọi đối tượng `System.console()`. Đối tượng `Console` được cung cấp các I/O Stream (character stream), thường dùng cho việc kiểm tra password.

2. Quét (scanning) và định dạng (formatting)

Lập trình I/O thường yêu cầu chuyển đổi và định dạng dữ liệu phù hợp yêu cầu sử dụng. Java cung cấp hai tập API cho vấn đề này: `scanner` API cho phép tách dữ liệu nhập thành các đơn vị ngữ nghĩa (token), `formatting` API lắp ráp dữ liệu thành định dạng phù hợp.

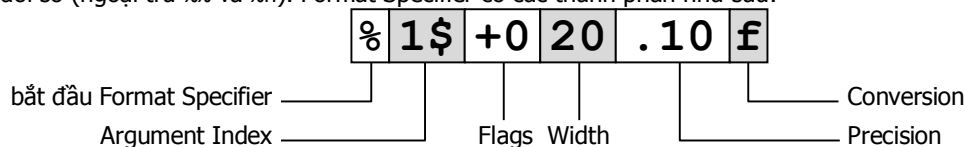
- Scanning: dòng dữ liệu nhập cần phải tách thành các *token* rồi chuyển đổi thành các kiểu dữ liệu phù hợp. Trước đây việc này được thực hiện bởi `java.util.StringTokenizer`, hiện nay được thực hiện tốt hơn với `java.util.Scanner`. Đối tượng thuộc các lớp này tách token bằng cách dùng *delimiter* (ký tự phân cách) mặc định là space, hoặc dùng ký tự phân cách từ chuỗi *delimiter* cung cấp cho nó. Chúng cũng hỗ trợ tách lấy token kế tiếp với định dạng yêu cầu.

```

import java.io.BufferedReader;
import java.io.FileReader;
import java.util.Scanner;
public class DayOfWeek {
    public static void main(String[] args) throws Exception {
        try (Scanner scanner = new Scanner(new BufferedReader(new FileReader("data.txt")))) {
            scanner.useDelimiter(",\\s*");
            while (scanner.hasNext())
                System.out.println(scanner.next());
        }
    }
}

```

- Formatting: việc định dạng dữ liệu xuất được thực hiện bởi phương thức của các đối tượng thuộc lớp `PrintWriter` hoặc `PrintStream` (đã lạc hậu). Các phương thức `format()`, `printf()` hỗ trợ mạnh định dạng xuất theo chuỗi định dạng chỉ định. Cũng có thể dùng phương thức static `String.format()` để trả về chuỗi được định dạng theo cách tương tự. Các định dạng giống hàm `printf()` của C nhưng linh hoạt hơn nhiều. Chuỗi định dạng gồm nhiều *Format Specifier*, mỗi *Format Specifier* phải so trùng với một đối số (ngoại trừ `%` và `%n`). *Format Specifier* có các thành phần như sau:



- Conversion: chỉ loại định dạng xuất (d: số nguyên, f: số thực, n: xuống dòng, x: số hex, s: chuỗi ...)
- Precision: chỉ định độ chính xác cho số thực, với s và các định dạng khác: chiều rộng tối đa, nếu lớn hơn sẽ bị xén phải.
- Width: chỉ chiều rộng tối thiểu, nếu chưa đủ sẽ đệm thêm từ trái với ký tự đệm mặc định là space.
- Flags: là tùy chọn định dạng thêm (+: xuất có dấu, 0: ký tự đệm là 0, -: đệm phải ...)
- Argument Index: cho biết chỉ số đối số sẽ được so trùng, một đối số có thể so trùng nhiều *Format Specifier*.

3. Các lớp công cụ

Khi làm việc với tập tin bạn cần một số tác vụ như kiểm tra một tập tin có tồn tại hay không, đổi tên tập tin, ... Java cung cấp các lớp công cụ để thực hiện các tác vụ này.

a) Lớp File

Lớp `java.io.File` cho phép bạn đổi tên hoặc xóa một tập tin, kiểm tra tồn tại, tạo một thư mục, kiểm tra kích thước tập tin. Trước tiên bạn phải tạo một thực thể của lớp `File` để gọi các phương thức trên, điều này không có nghĩa là tạo tập tin. Khi muốn tạo tập tin vật lý thật sự, dùng phương thức `createNewFile()`.

```
public static void traversal(String name) {
    File file = new File(name);
    if (file.exists()) {
        if (file.isDirectory()) {
            System.out.printf("[%s]\n", name);
            for (String subname : file.list()) traversal(name + "\\ " + subname);
        } else if (file.isFile()) {
            System.out.printf("%s\n", name);
        }
    }
}
// gọi hàm đệ quy duyệt cây thư mục
traversal(".");
```

Một số phương thức của lớp `File`:

<code>createNewFile()</code>	Tạo một tập tin rỗng mới, tên theo tên được truyền cho constructor.
<code>delete()</code>	Xóa một tập tin hoặc thư mục.
<code>renameTo()</code>	Đổi tên một tập tin.
<code>length()</code>	Trả về kích thước của tập tin tính bằng byte.
<code>exists()</code>	Kiểm tra tập tin với tên chỉ định có tồn tại hay không.
<code>list()</code>	Trả về mảng chuỗi chứa tên các tập tin hoặc thư mục.
<code>lastModified()</code>	Trả về thời điểm tập tin thay đổi lần cuối.
<code>mkdir()</code>	Tạo một thư mục.

b) Các lớp của NIO.2

Java 7 cung cấp một số lớp và interface mới để làm việc hiệu quả hơn với tập tin và thư mục, được tham chiếu như NIO.2. Các lớp và interface mới này tập trung trong các gói `java.nio` và `java.nio.file`.

- interface `Path` Thể hiện vị trí (đường dẫn) của tập tin hoặc thư mục.

- class `Paths` Factory các đối tượng `Path` với hệ thống tập tin mặc định.

```
Path path = Paths.get("."); // lấy là đường dẫn đến tập tin hoặc thư mục
```

```
path.getFileName(); // lấy tên tập tin hoặc thư mục
```

```
if (!path.isAbsolute()) // kiểm tra path là đường dẫn tuyệt đối hay không
```

```
path = path.toAbsolutePath(); // chuyển thành đường dẫn tuyệt đối
```

- class `Files` Lớp chứa các phương thức static dùng thao tác trên tập tin và thư mục.

```
Path path = Paths.get("customers.txt");
```

```
if (Files.exists(path) && Files.isRegularFile(path)) // kiểm tra path tồn tại và là tập tin
```

```
System.out.printf("Size: %d bytes", Files.size(path)); // kích thước tập tin hoặc thư mục
```

Đọc tập tin văn bản:

```
public void readData() throws IOException {
    try (BufferedReader in = Files.newBufferedReader(Paths.get("text.txt"))) {
        String s;
        while ((s = in.readLine()) != null) System.out.println(s);
    }
}
```

```
public void readAll() throws IOException {
    List<String> lines = Files.readAllLines(Paths.get("text.txt"));
    for (String line : lines) System.out.println(line);
}
```

- interface `DirectoryStream` Dùng duyệt xuyên qua nội dung của thư mục.

```
public static void traversal(Path path) throws IOException {
    if (Files.exists(path)) {
        if (Files.isDirectory(path)) {
            System.out.printf("[%s]\n", path);
            DirectoryStream<Path> ds = Files.newDirectoryStream(path);
            for (Path p : ds) traversal(p);
        } else if (Files.isRegularFile(path)) {
            System.out.printf("%s\n", path);
        }
    }
}
```

// gọi hàm đệ quy duyệt cây thư mục

```
Path path = Paths.get(".");
```

```
traversal(path);
```

- class `FileSystem` Thể hiện một hệ thống tập tin, kể cả hệ thống tập tin nén, định danh bằng URI.

- class `FileSystems` Chứa các phương thức để tạo và mở một `FileSystem`.

```
import java.io.*;
```

```

import java.util.*;
import java.net.URI;
import java.net.URISyntaxException;
import java.nio.charset.Charset;
import java.nio.file.FileSystem;
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;
import java.nio.file.StandardOpenOption;

public class Main {
    private static FileSystem openZip(Path zipPath) throws URISyntaxException, IOException {
        Map<String, String> providerProps = new HashMap<>();
        providerProps.put("create", "true");
        URI zipUri = new URI("jar:file", zipPath.toUri().getPath(), null);
        return FileSystems.newFileSystem(zipUri, providerProps);
    }

    private static void copyToZip(FileSystem zipFS) throws IOException {
        Path sourceFile = FileSystems.getDefault().getPath("file1.txt");
        Path destFile = zipFS.getPath("/file1.bak");
        Files.copy(sourceFile, destFile, StandardCopyOption.REPLACE_EXISTING);
    }

    private static void writeToFileInZip(FileSystem zipFS, String[] data) throws IOException {
        try (BufferedWriter in = Files.newBufferedWriter(zipFS.getPath("/file2.txt"))) {
            for (String s : data) {
                in.write(s);
                in.newLine();
            }
        }
    }

    private static void writeToFileInZip1(FileSystem zipFS, String[] data) throws IOException {
        Files.write(zipFS.getPath("file3.txt"), Arrays.asList(data), Charset.defaultCharset(), StandardOpenOption.CREATE);
    }

    public static void main(String[] args) {
        String[] a = { "He shot me down", "I hit the ground", "That awful sound", "My baby shot me down" };
        try (FileSystem zipFS = openZip(Paths.get("data.zip"))) {
            copyToZip(zipFS);
            writeToFileInZip(zipFS, a);
            writeToFileInZip1(zipFS, a);
        } catch (IOException | URISyntaxException e) {
            e.printStackTrace(System.err);
        }
    }
}

```

Một số I/O Stream đặc biệt

1. Data Stream

Data stream hỗ trợ xuất nhập nhị phân cho trị có các kiểu dữ liệu cơ bản (boolean, char, byte, short, int, long, float và double) cũng như kiểu String. Các data stream đều cài đặt một trong hai interface DataInput và DataOutput. Các lớp chính thường được sử dụng là DataInputStream và DataOutputStream.

```

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class DataStreams {
    static final double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
    static final int[] units = { 12, 8, 13, 29, 50 };
    static final String[] descs = { "T-shirt", "Mug", "Dolls", "Pin", "Key Chain" };
    public static void main(String[] args) throws IOException {
        try (DataOutputStream out = new DataOutputStream(new BufferedOutputStream(new FileOutputStream("data.dat")))) {
            for (int i = 0; i < prices.length; ++i) {
                out.writeDouble(prices[i]);
            }
        }
    }
}

```

```

        out.writeInt(units[i]);
        out.writeUTF(descs[i]);
    }
}
// đọc lại tập tin đã lưu
double total = 0.0;
try (DataInputStream in = new DataInputStream(new BufferedInputStream(new FileInputStream("data.dat")))) {
    try { // data stream kiểm tra EOF bằng cách bắt exception, thay vì kiểm tra trị không hợp lệ
        while (true) {
            double price = in.readDouble();
            int unit = in.readInt();
            String desc = in.readUTF();
            System.out.printf("You ordered %d unit(s) of %s at $%.2f\n", unit, desc, price);
            total += unit * price;
        }
    } catch (EOFException e) { }
    System.out.format("For a TOTAL of: $%.2f\n", total);
}
}
}

```

2. Object Stream

Trong lúc data stream hỗ trợ xuất nhập với nhiều kiểu dữ liệu cơ bản, object stream lại hỗ trợ xuất nhập các đối tượng. Các đối tượng này phải được cài đặt interface `java.io.Serializable`. Object stream đặc biệt tiện dụng trong lập trình mạng.

Các lớp chính `ObjectInputStream` và `ObjectOutputStream` cài đặt interface `ObjectInput` và `ObjectOutput`, là các interface dẫn xuất từ `DataInput` và `DataOutput`. Điều này có nghĩa là các phương thức cơ bản cho xuất nhập cho data stream cũng được cài đặt cho object stream. Như vậy object stream có thể chứa vừa kiểu dữ liệu cơ bản, vừa kiểu đối tượng.

Trong ví dụ minh họa ta tạo lớp `Dummy` cài đặt interface `Serializable`. Các đối tượng thuộc lớp này sẽ được ghi vào tập tin rồi đọc trở lại thông qua các object stream của lớp `Main`. Các phương thức (private) `readObject` và `writeObject` của `Dummy` hỗ trợ thêm khi dùng object stream đọc hoặc ghi đối tượng lớp `Dummy`.

```

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectInputStream;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.util.Date;
class Dummy implements Serializable {
    java.util.Date date;
    transient String str; // biến tạm, bỏ qua trong quá trình serialization
    transient boolean restored; // không serialization, nhưng gán true mỗi khi đọc đối tượng từ stream
    private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
        in.defaultReadObject();
        restored = true; // gán lại cho biến tạm restored
    }

    private void writeObject(ObjectOutputStream out) throws IOException {
        out.defaultWriteObject();
    }
}

public class Main {
    public static void main(String[] args) {
        try (ObjectOutput out = new ObjectOutputStream(new FileOutputStream("dummy.ser"))) {
            // lưu đối tượng xuống tập tin
            Dummy dummy = new Dummy();
            dummy.date = new Date();
            out.writeObject(dummy);
            out.flush();
        } catch (IOException e) {
            e.printStackTrace(System.err);
        }
        // đọc lại đối tượng (deserialization), ép kiểu đối tượng thành kiểu ban đầu
        try (ObjectInput in = new ObjectInputStream(new FileInputStream("dummy.ser"))) {
            Dummy dummy = (Dummy) in.readObject();
            System.out.println("dummy.date " + dummy.date);
            System.out.println("dummy.restored " + dummy.restored);
        } catch (IOException | ClassNotFoundException e) {

```



```
e.printStackTrace(System.err);  
}  
}
```

Thread

Khái niệm

1. Thread

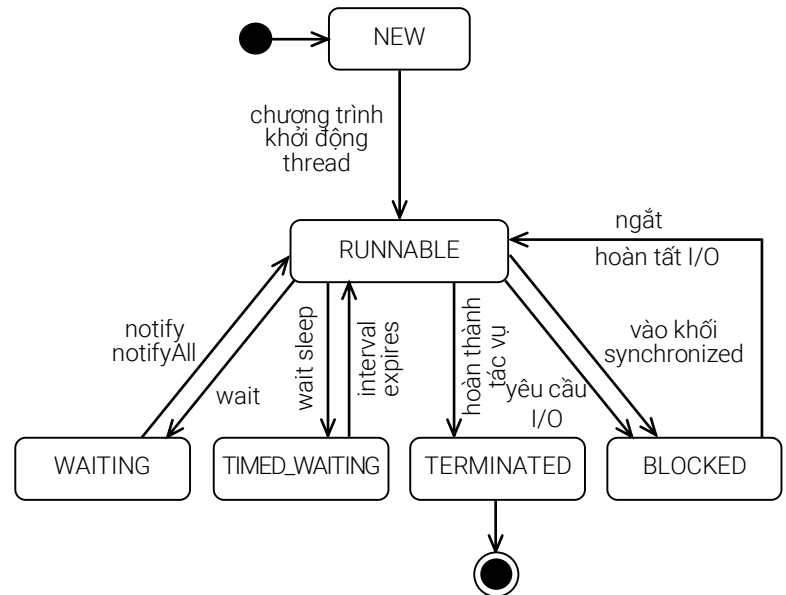
Ứng dụng Java thực thi thông qua các thread (tuyến trình), đó là những tuyến thực thi độc lập. Khi có nhiều thread, tuyến thực thi của thread này có thể khác tuyến thực thi của thread khác. Mỗi ứng dụng Java có một thread chính mặc định, thực thi phương thức main() của ứng dụng.

2. Vòng đời của thread

Các trạng thái của thread được định nghĩa trong enum Thread.State và lấy bằng phương thức getState(). Một thread được tạo ra ở trạng thái NEW. Nó giữ trạng thái này cho đến khi chương trình khởi động (start) thread và chuyển nó sang trạng thái RUNNABLE. Trạng thái RUNNABLE là trạng thái mà thread thực thi các tác vụ của nó.

Đôi khi một thread ở trạng thái RUNNABLE có thể:

- Chuyển sang trạng thái WAITING để chờ thread khác hoạt động. Thread ở trạng thái WAITING quay về trạng thái RUNNABLE khi một thread khác báo cho nó tiếp tục hoạt động.
- Chuyển sang trạng thái TIMED_WAITING nếu bị đưa vào trạng thái "ngủ" (sleep). Khi hết thời gian "ngủ" chỉ định, nó quay trở lại trạng thái RUNNABLE.
- Chuyển sang trạng thái BLOCKED, bị "khóa" bởi monitor để chờ đợi truy cập tài nguyên.
- Chuyển sang trạng thái TERMINATED khi thực thi xong các tác vụ của nó.



Tạo và thực hiện Thread

Trong Java, có vài cách cơ bản để tạo một lớp hoạt động như một thread:

- Thừa kế lớp java.lang.Thread, lớp Thread này cài đặt interface java.lang.Runnable.
- Cài đặt interface java.lang.Runnable.
- Cài đặt interface java.util.concurrent.Callable.

1. Lớp Thread

Với thread thuộc lớp thừa kế lớp Thread, các tác vụ trong phương thức run() được xem như chạy trên một thread riêng. Phương thức run() không thể gọi trực tiếp, khi phương thức start() của thread được gọi, thread chuyển sang trạng thái runnable và tự động thực thi run(). Nếu không viết lại run(), phương thức run() mặc định sẽ chạy và không làm gì cả.

```

import java.util.Random;
class PrintTask extends Thread {
    PrintTask(String printTask) {
        super(printTask);
    }

    @Override public void run() {
        for (int i = 0; i < 10; ++i) {
            try {
                System.out.printf("[%s]::run()%" , getName());
                sleep(new Random().nextInt(2000)); // dừng ngẫu nhiên 0 - 2 giây
            } catch (InterruptedException e) {
                System.err.println(e.getMessage());
            }
        }
    }
}
  
```

```

public class RunnableTester {
    public static void main(String[] args) {
        new PrintTask("print").start();
        System.out.printf("[%s]::main()%" , Thread.currentThread().getName());
    }
}
  
```

Thread sẽ kết thúc khi thực thi hết các lệnh trong run().

2. Interface Runnable

Cách thích hợp nhất để tạo ứng dụng multithreaded là cài đặt interface java.lang.Runnable. Interface Runnable chỉ có một phương thức đơn run(), ta cài đặt cho phương thức này các tác vụ cần thực hiện song hành.

```

import java.util.Random;
  
```

```

public class RunnableTester {
    public static void main(String[] args) {
        new Thread(
            new Runnable() {
                @Override public void run() {
                    for (int i = 0; i < 10; ++i) {
                        try {
                            System.out.printf("[%s]::run()\n", Thread.currentThread().getName());
                            Thread.sleep(new Random().nextInt(2000)); // dừng ngẫu nhiên 0 - 2 giây
                        } catch (InterruptedException e) {
                            System.err.println(e.getMessage());
                        }
                    }
                }
            }
        ).start();
        System.out.printf("[%s]::main()\n", Thread.currentThread().getName());
    }
}

```

Mỗi đối tượng vô danh cài đặt interface Runnable sẽ thực hiện song hành các tác vụ trong phương thức run(). Sau đó truyền nó cho constructor của một Thread, rồi khởi động thread đó.

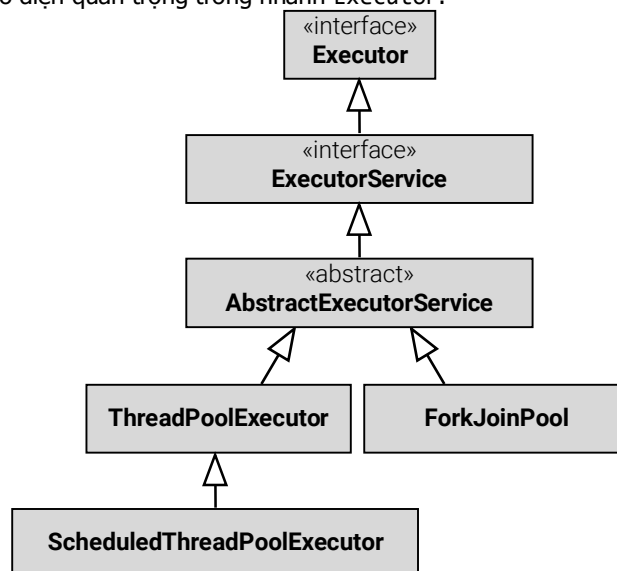
3. Framework Executor

Thread cài đặt Runnable còn được thực thi bởi một đối tượng cài đặt interface java.util.concurrent.Executor.

Một đối tượng Executor thường tạo và quản lý một nhóm các thread có sẵn, gọi là thread pool. Các thread cài đặt interface Runnable được truyền như tham số đến phương thức duy nhất của interface Executor: phương thức execute(). Đối tượng Executor gán mỗi Runnable vào một thread có sẵn trong thread pool. Nếu không còn thread có sẵn, Executor tạo một thread mới hoặc chờ thread có sẵn để gán cho nó đối tượng Runnable được truyền đến.

Interface java.util.concurrent.ExecutorService là interface con của Executor khai báo một số phương thức quản lý vòng đời của Executor. Một đối tượng cài đặt interface ExecutorService có thể được tạo bằng cách dùng các phương thức static khai báo trong lớp java.util.concurrent.Executors.

Hình dưới trình bày các lớp và giao diện quan trọng trong nhánh Executor.



```

import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
public class RunnableTester {
    public static void main(String[] args) {
        // tạo và đặt tên thread
        PrintTask task = new PrintTask("thread");
        System.out.println("Starting Executor");
        // tạo ExecutorService từ Executors để quản lý các thread
        ExecutorService threadExecutor = Executors.newCachedThreadPool();
        // khởi động thread và đưa nó vào trạng thái chạy
        threadExecutor.execute(task); // khởi động task
        threadExecutor.shutdown(); // ngưng các thread
        // phương thức main cũng chạy trong một thread, main thread; sẽ terminated khi kết thúc chương trình
        System.out.println("Threads started, main ends");
    }
}

```

4. Interface Callable và interface Future

Đối tượng cài đặt interface Runnable có hai khuyết điểm: phương thức run() không có trị trả về và không cho phép khai báo ném exception, exception phải giải quyết bên trong run() và không thể truyền đến thread gọi.

Interface java.util.concurrent.Callable giải quyết vấn đề này. Lớp cài đặt interface Callable phải cài đặt phương thức call(), tương tự như phương thức run() của Runnable.

```
import java.util.concurrent.Callable;
import java.util.Random;
public class PrintTask implements Callable<Integer> {
    private String threadName;    // tên thread

    public PrintTask(String name) {
        threadName = name;
    }
    // phương thức call() chứa code thực hiện bởi thread mới
    @Override public Integer call() {
        int sleepTime = new Random().nextInt(5000);
        try {
            System.out.printf("%s going to sleep for %d milliseconds.%n", threadName, sleepTime);
            Thread.sleep(sleepTime);    // cho thread ngủ
        } catch (InterruptedException e) {
            e.printStackTrace(System.err);
        }
        System.out.printf("%s done sleeping%n", threadName);
        return sleepTime;
    }
}
```

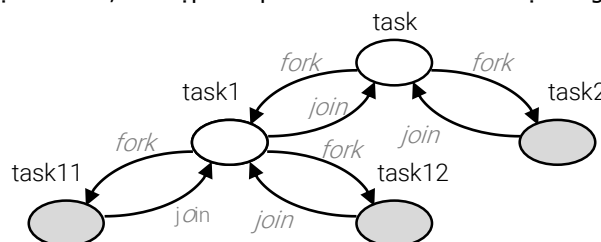
Interface ExecutorService cung cấp phương thức submit() nhận đối tượng Callable như đối số và chạy phương thức call() của nó trong thread. Phương thức submit() trả về đối tượng kiểu java.util.concurrent.Future, interface này có phương thức get() trả kết quả từ đối tượng Callable về cũng như cung cấp các phương thức khác quản lý hoạt động của Callable.

```
public class RunnableTester {
    public static void main(String[] args) {
        Callable<Integer> task = new PrintTask("thread");
        System.out.println( "Starting Executor" );
        ExecutorService threadExecutor = Executors.newSingleThreadExecutor();
        Future<Integer> future = threadExecutor.submit(task);
        threadExecutor.shutdown();
        System.out.printf("Sleep time: %d%n", future.get());
    }
}
```

5. Framework Fork/Join

Java 7 cung cấp framework Fork/Join, là một cài đặt của interface ExecutorService, giúp đơn giản việc tính toán song hành. Framework Fork/Join rất dễ dùng khi mô hình hóa các vấn đề "chia để trị" (divide-and-conquer), thích hợp với các tác vụ có thể chia đệ quy thành các tác vụ con để tính toán song hành.

Chia tác vụ thành các tác vụ con gọi là "fork", kết hợp kết quả tính toán từ các tác vụ con gọi là "join".



Framework Fork/Join dùng "chia để trị" hoàn thành tác vụ, các tác vụ màu xám là đã đủ nhỏ để xử lý

Các lớp sau giữ vai trò quan trọng trong framework Fork/Join:

- ForkJoinPool, lớp quan trọng nhất, là một thread pool cho các tác vụ fork/join đang chạy. Phương thức invoke() của nó thực thi một ForkJoinTask và trả về kết quả tính toán được.
- ForkJoinTask<V> là một thực thể tương tự thread định nghĩa các phương thức fork() và join().
- RecursiveTask<V>, thừa kế ForkJoinTask, là một tác vụ có thể chạy trong ForkJoinPool, phương thức compute() của nó trả về trị có kiểu V mà tác vụ tính toán được.
- RecursiveAction, tương tự như RecursiveTask nhưng phương thức compute() của nó không trả trị về.

Các bước dùng framework Fork/Join:

- Kiểm tra xem vấn đề cần giải quyết có thích hợp cho việc dùng framework Fork/Join hay không. Các loại vấn đề sau là thích hợp để dùng framework:

- + Vấn đề thiết kế như một tác vụ đệ quy mà tác vụ đó có thể chia thành các đơn vị nhỏ hơn và các kết quả thu được có thể được kết hợp với nhau.
- + Các tác vụ chia nhỏ là độc lập và chúng có thể tính toán tách biệt mà không cần liên lạc giữa các tác vụ khi tính toán đang tiến hành. Sau khi tính toán, bạn cần kết hợp các kết quả với nhau.

- Nếu vẫn đề bạn muốn giải quyết theo mô hình đệ quy, định nghĩa lớp tác vụ thừa kế RecursiveTask (nếu trả về kết quả) hoặc thừa kế RecursiveAction (không trả về kết quả).
- Viết lại (override) phương thức compute() của lớp tác vụ. Phương thức này xử lý tác vụ nếu tác vụ đủ nhỏ để thực hiện, hoặc chia tác vụ thành các tác vụ con và triệu gọi chúng. Các tác vụ con có thể được triệu gọi bằng phương thức invokeAll() hoặc fork(). Dùng fork() khi tác vụ con trả về một trị. Dùng phương thức join() để lấy các kết quả tính toán được nếu dùng fork() trước đó.
- Kết hợp các kết quả, nếu chúng được tính toán từ các tác vụ con.
- Tạo thực thể ForkJoinPool và thực thể lớp tác vụ, và bắt đầu thực hiện tác vụ bằng cách dùng phương thức invoke() của ForkJoinPool.

Ví dụ minh họa tính tổng các số từ 1 đến N (với N là số lớn).

```
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;
public class SumOfNUsingForkJoin {
    private static long N = 1000_000;
    private static final int NUM_THREADS = 10;

    static class RecursiveSumOfN extends RecursiveTask<Long> {
        long from, to; // lớp tác vụ, tính tổng từ from đến to
        public RecursiveSumOfN(long from, long to) {
            this.from = from;
            this.to = to;
        }

        @Override public Long compute() {
            if ((to - from) <= N / NUM_THREADS) {
                // tác vụ đủ nhỏ để xử lý với một thread, tính tổng bằng vòng lặp, dùng đệ quy
                long localSum = 0;
                for (long i = from; i <= to; ++i) localSum += i;
                System.out.printf("\tSum of value range %d to %d is %d\n", from, to, localSum);
                return localSum;
            } else {
                // tác vụ quá lớn để xử lý với một thread, chia thành hai tác vụ con
                long mid = (from + to) / 2;
                System.out.printf("Forking computation into two ranges: %d to %d and %d to %d\n",
                                from, mid, mid, to);
                // tác vụ con thứ nhất tính từ from đến mid
                RecursiveSumOfN firstHalf = new RecursiveSumOfN(from, mid);
                firstHalf.fork();
                // tác vụ con thứ hai tính từ mid+1 đến to
                RecursiveSumOfN secondHalf = new RecursiveSumOfN(mid + 1, to);
                long resultSecond = secondHalf.compute();
                // chờ tác vụ con thứ nhất tính xong, cộng kết quả với tổng tính từ tác vụ con thứ hai
                return firstHalf.join() + resultSecond;
            }
        }
    }

    public static void main(String[] args) {
        // tạo một fork-join pool chứa NUM_THREADS thread
        ForkJoinPool pool = new ForkJoinPool(NUM_THREADS);
        // chuyển tác vụ tính toán đến fork-join pool
        long computedSum = pool.invoke(new RecursiveSumOfN(0, N));
        // tính đối chứng bằng công thức tính tổng dãy 1..N
        long formulaSum = (N * (N + 1)) / 2;
        // so sánh với kết quả tính bằng framework Fork/Join
        System.out.printf("Sum for range 1..%d; computed sum = %d, "
                        + "formula sum = %d\n", N, computedSum, formulaSum);
    }
}
```

6. Parallel Stream

Ngoài stream tuần tự (sequential), Java 8 cung cấp stream song song (parallel). Stream song song sẽ chia các phần tử thành nhiều "chunk", xử lý mỗi "chunk" trong các thread khác nhau, và kết hợp các kết quả từ những thread này để tính toán kết quả cuối cùng nếu cần. Stream song song sử dụng framework Fork/Join bên trong.

Ví dụ minh họa đếm các số nguyên tố từ 2 đến 10000, đầu tiên là phiên bản tuần tự.

```
import java.util.stream.LongStream;
public class PrimeNumbers {
    private static boolean isPrime(long val) {
        for (long i = 2; i <= val / 2; ++i)
            if ((val % i) == 0) return false;
    }
}
```

```

    return true;
}

public static void main(String[] args) {
    long numOfPrimes = LongStream.rangeClosed(2, 100_000)
        .filter(PrimeNumbers::isPrime)
        .count();
    System.out.println(numOfPrimes);
}
}

```

rất dễ chuyển thành phiên bản song song: sau khi gọi `parallel()`, stream thành song song và các số được chia và chuyển đến các thread trong framework Fork/Join để kiểm tra, tốc độ thực thi tăng lên rất nhiều.

```

long numOfPrimes = LongStream.rangeClosed(2, 100_000)
    .parallel()
    .filter(PrimeNumbers::isPrime)
    .count();

```

Khi gọi phương thức `stream()` của lớp `Collection`, bạn nhận được stream tuần tự. Nếu gọi phương thức `parallelStream()` bạn sẽ nhận được stream song song. Ví dụ, đếm các số chẵn trong mảng.

```

List<Integer> ints = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
System.out.println(ints.parallelStream().filter(i -> (i % 2) == 0).count());

```

Một số vấn đề liên quan đến thread

1. Một số phương thức thread

- Độ ưu tiên của thread

Để bộ điều phối thread xác định được thứ tự chạy thread, các thread có độ ưu tiên từ `Thread.MIN_PRIORITY` (giá trị 1) đến `Thread.MAX_PRIORITY` (giá trị 10), giá trị mặc định là `Thread.NORM_PRIORITY` (giá trị 5). Thread có độ ưu tiên càng cao thì càng sớm được thực hiện và hoàn thành. Chú ý là bộ điều phối phụ thuộc vào hệ nền.

Một thread mới sẽ thừa kế độ ưu tiên từ thread tạo ra nó.

```

final int getPriority();           thiết lập trị ưu tiên chỉ định cho thread
final void setPriority(int priority); trả về trị ưu tiên của thread hiện hành

```

- Các phương thức static

```

void sleep(long t);               thread tạm dừng (ngủ) trong thời gian t mili giây.
void yield();                     nhường điều khiển cho thread khác.
void interrupt();                 ngắt hoạt động của thread.

```

- Phương thức `join()`

Một thread (thường là thread chính) khởi động các thread khác (gọi là worker thread) để thực hiện các tác vụ dưới nền, tốn nhiều thời gian. Sau khi worker thread kết thúc, thread khởi động worker thread sẽ xử lý kết quả do worker thread trả về và chờ cho worker thread kết thúc. Điều này được thực hiện bằng phương thức `join()`.

```

public class ThreadDemo {
    private static long result;
    public static void main(String[] args) {
        Runnable r = () -> { result = sum(50000); }; // worker thread
        Thread t = new Thread(r);
        t.start();
        try {
            t.join();
        } catch (InterruptedException ie) { }
        System.out.println(result);
    }
}

```

```

public static long sum(int n) {
    long t = 0;
    for (int i = 0; i < n; ++i) t += i;
    return t;
}
}

```

- Các phương thức lạc hậu

`suspend()`, `resume()`, `stop()` đã lạc hậu.

Do phương thức `stop()` lạc hậu, khi dừng đột ngột một thread có thể gây mất nhất quán trong cấu trúc dữ liệu của đối tượng. Muốn dừng thread thường ta đặt cờ logic để dừng vòng lặp trong phương thức `run()`.

```

public class StoppableThread extends Thread {
    private boolean done = false;
    @Override public void run() {
        while (done != true) {
            // tác vụ của thread
        }
    }

    public void shutDown() { done = true; }
}

```


2. Thread dịch vụ (daemon thread)

Daemon thread thường là các thread hỗ trợ môi trường thực thi của các thread (non-daemon) khác. Nó tự động kết thúc khi thread non-daemon cuối cùng của ứng dụng kết thúc. Ví dụ: garbage collector của Java là một daemon thread.

Các phương thức dùng cho daemon thread:

<code>void setDaemon(boolean isDaemon);</code>	đặt một thread trở thành daemon thread
<code>boolean isDaemon();</code>	kiểm tra một thread có phải là daemon thread không

3. Nhóm thread

Các thread có thể được đưa vào trong cùng một nhóm thông qua lớp ThreadGroup. Một ThreadGroup chỉ có thể xử lý trên các thread trong nhóm, ví dụ ngắt tất cả các thread thuộc nhóm.

Có thể tạo ra các nhóm thread là nhóm con của một nhóm thread khác.

Một số nhóm thread đặc biệt: system, main

Đoạn code sau liệt kê các thread của chương trình:

```
public static void listThreads() {
    ThreadGroup root = Thread.currentThread().getThreadGroup().getParent();
    while (root.getParent() != null)
        root = root.getParent();
    visitGroup(root, 0);
}

public static void visitGroup(ThreadGroup group, int level) {
    int numThreads = group.activeCount();
    Thread[] threads = new Thread[numThreads];
    group.enumerate(threads, false);
    for (int i = 0; i < numThreads; ++i) {
        Thread thread = threads[i];
        printThreadInfo(thread);
    }
    int numGroups = group.activeGroupCount();
    ThreadGroup[] groups = new ThreadGroup[numGroups];
    numGroups = group.enumerate(groups, false);
    for (int i = 0; i < numGroups; ++i)
        visitGroup(groups[i], level + 1);
}

private static void printThreadInfo(Thread t) {
    System.out.println("Thread: " + t.getName() + " Priority: " + t.getPriority() +
        (t.isDaemon()? " Daemon":"") + (t.isAlive()? "":" Not Alive"));
}
```

4. Lớp Timer

Khi xử lý công việc có liên quan đến định thời, thường dùng hai lớp chuyên dụng sau:

javax.swing.Timer đơn giản, thường dùng tạo hoạt hình trên GUI, applet.

java.util.Timer cho phép một tác vụ chạy:

- Tại một thời điểm chỉ định, dùng phương thức `schedule(TimerTask, int)`.

- Chạy lặp đi lặp lại trong một khoảng thời gian chỉ định, dùng phương thức `scheduleAtFixedRate(TimerTask, int, int)`.

Tác vụ chạy định thời được cài đặt trong phương thức `run()` của lớp `java.util.TimerTask` truyền cho các phương thức trên.

```
import java.util.Timer;
import java.util.TimerTask;
// ...

int delay = 5000;
int period = 1000; // run() của TimerTask sẽ gọi mỗi lần period miligiây, gọi là timer interval
Timer timer = new Timer();
timer.scheduleAtFixedRate(new TimerTask() {
    @Override public void run() {
        // các tác vụ cần định thời
    }
}, delay, period);
```

Đồng bộ thread (thread synchronyzation)

Thông thường các thread có thể thao tác bất kỳ lên tài nguyên dùng chung. Điều này đưa đến việc nhiều thread thay đổi cùng lúc, một cách không kiểm soát đến tài nguyên dùng chung, gọi là vấn đề tranh đoạt điều khiển (*race condition*).

Vấn đề này có thể được giải quyết nếu cho mỗi thread một khoảng thời gian độc quyền chạy đoạn code chia sẻ tài nguyên. Lúc này, các thread khác muốn truy cập tài nguyên dùng chung sẽ bị giữ lại chờ đợi đến lượt mình. Điều này gọi là độc quyền truy cập (*mutual exclusion*), cho phép lập trình viên thực hiện đồng bộ thread (*thread synchronization*), phối hợp nhiều thread song hành cùng truy cập tài nguyên dùng chung.

Bài toán điển hình là Producer – Consumer: Producer đặt trị vào buffer (chứa một trị nguyên) nếu buffer còn trống, Consumer lấy trị từ buffer nếu buffer không rỗng. Tại một thời điểm chỉ có Producer hoặc Consumer được truy cập buffer dùng chung.

// interface Buffer chỉ định các phương thức được gọi bởi Producer và Consumer

```
interface Buffer {
    void set(int value);    // đặt trị vào Buffer
    int get();              // lấy trị ra khỏi Buffer
}
```

1. Lock và đối tượng Condition

Đối tượng nào cũng có thể chứa một đối tượng cài đặt interface `java.util.concurrent.locks.Lock`. Một thread sẽ tham khảo phương thức `lock()` của đối tượng `Lock` để lấy khóa truy cập vào tài nguyên, khi đó đối tượng `Lock` sẽ không cho phép thread khác có được khóa, cho đến khi thread đang chạy giải phóng khóa (bằng cách gọi phương thức `unlock()`). Lớp `ReentrantLock` là cài đặt cơ bản của interface `Lock`.

Khi thread đã dùng đối tượng `Lock` khóa tài nguyên dùng chung để truy cập, nó có thể chờ một điều kiện nào đó thành true, điều này được thực hiện bằng *đối tượng điều kiện*, cài đặt interface `Condition`. Đối tượng `Lock` cho phép khai báo tường minh các đối tượng `Condition` bằng phương thức `newCondition()`.

Để chờ đối tượng `Condition`, thread gọi phương thức `await()` của đối tượng `Condition`. Điều này sẽ tách thread ra khỏi đối tượng `Lock` và đưa nó vào trạng thái waiting trên đối tượng `Condition` này, lúc này một thread khác có thể thử dùng đối tượng `Lock`. Khi một thread ở trạng thái runnable hoàn thành một tác vụ và xác định có thread đang waiting, nó gọi phương thức `signal()` của đối tượng `Condition` cho phép thread đang waiting trở về trạng thái runnable.

// SynchronizedBuffer đồng bộ truy cập đến buffer dùng chung

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;
interface Buffer {
    void set(int value);    // đặt trị vào Buffer
    int get();              // lấy trị ra khỏi Buffer
}
```

```
public class SynchronizedBuffer implements Buffer {
    // tạo đối tượng Lock để đồng bộ cho buffer này
    private Lock accessLock = new ReentrantLock();
    // các điều kiện đọc và ghi (cờ hiệu)
    private Condition canWrite = accessLock.newCondition();
    private Condition canRead = accessLock.newCondition();
    private int buffer = -1;    // tài nguyên dùng chung cho thread Producer và Consumer
    private boolean occupied = false; // báo buffer đầy dữ liệu (true) hay trống (false)

    // đặt một trị vào buffer
    @Override public void set(int value) {
        accessLock.lock();    // khóa buffer để ghi
        try {
            while (occupied) { // trong lúc buffer đầy, đưa thread ghi (Producer) vào trạng thái chờ
                System.out.println("Producer tries to write. Buffer full. Producer waits.");
                canWrite.await(); // chờ cho đến khi buffer có chỗ trống
            }
            buffer = value;    // nếu buffer có chỗ trống, đặt một trị mới vào
            occupied = true;
            System.out.printf(" Producer writes %d\n", buffer);
            canRead.signal();  // báo cho thread đang chờ là có thể đọc buffer (buffer có dữ liệu)
        } catch (InterruptedException e) {
            e.printStackTrace(System.err);
        } finally {
            accessLock.unlock(); // mở khóa buffer, cho phép thread khác truy cập buffer
        }
    }

    // lấy một trị từ buffer
    @Override public int get() {
        int readValue = 0;    // khởi tạo biến sẽ chứa trị đọc được từ buffer
        accessLock.lock();    // khóa buffer để đọc
        try {
            while (!occupied) { // trong lúc buffer rỗng, đưa thread đọc (Consumer) vào trạng thái chờ
                System.out.println("Consumer tries to read. Buffer empty. Consumer waits.");
                canRead.await(); // chờ cho đến khi buffer có dữ liệu
            }
            occupied = false;
            readValue = buffer; // nếu buffer có dữ liệu, lấy trị từ nó
            System.out.printf("Consumer reads %d\n", readValue);
            canWrite.signal();  // báo cho thread đang chờ là có thể ghi vào buffer
        } catch (InterruptedException e) {
            e.printStackTrace(System.err);
        }
    }
}
```

```

    } finally {
        accessLock.unlock(); // mở khóa buffer, cho phép thread khác truy cập buffer
    }
    return readValue;
}
}

```

Producer: thread (nhà cung cấp) đặt trị vào buffer.

// phương thức run() của Producer lưu các trị từ 1 đến 10 vào buffer

```

import java.util.Random;
public class Producer implements Runnable {
    private static Random generator = new Random();
    private Buffer sharedLocation; // tham chiếu đến tài nguyên dùng chung
    public Producer(Buffer shared) {
        sharedLocation = shared;
    }

    // lưu trữ từ 1 đến 10 vào buffer dùng chung
    @Override public void run() {
        int sum = 0;
        for (int count = 1; count <= 10; ++count) {
            try {
                Thread.sleep(generator.nextInt(3000)); // ngủ 0 đến 3 giây, rồi đặt trị vào buffer
                sharedLocation.set(count); // đặt trị vào buffer
                sum += count;
            } catch (InterruptedException e) {
                e.printStackTrace(System.err);
            }
        }
        System.out.printf("\n%s\n%s\n", "Producer done producing.", "Terminating Producer.");
    }
}

```

Consumer: thread (nhà tiêu thụ) lấy trị khỏi buffer.

// phương thức run() của Consumer lặp 10 lần để đọc các trị từ buffer

```

import java.util.Random;
public class Consumer implements Runnable {
    private static Random generator = new Random();
    private Buffer sharedLocation; // tham chiếu đến tài nguyên dùng chung
    public Consumer(Buffer shared) {
        sharedLocation = shared;
    }

    // đọc trị từ buffer dùng chung
    @Override public void run() {
        int sum = 0;
        for (int count = 1; count <= 10; ++count) {
            try {
                Thread.sleep(generator.nextInt(3000)); // ngủ 0 đến 3 giây, rồi đọc trị từ buffer
                sum += sharedLocation.get(); // đọc trị từ buffer
            } catch (InterruptedException exception) {
                exception.printStackTrace(System.err);
            }
        }
        System.out.printf("\n%s %d.\n%s\n", "Consumer read values totaling", sum, "Terminating Consumer.");
    }
}

```

Kiểm tra truy cập đồng bộ đến tài nguyên dùng chung:

// Hai thread cùng thao tác vào một buffer đồng bộ

```

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

```

```

public class SharedBufferTest {
    public static void main(String[] args) {
        // Tạo một thread pool mới chứa 2 thread
        ExecutorService application = Executors.newCachedThreadPool(2);
        // tạo một SynchronizedBuffer chứa buffer chia sẻ
        Buffer sharedLocation = new SynchronizedBuffer();
        System.out.printf("%-40s%s\t\t%s\n%-40s%s\n\n",
            "Operation", "Buffer", "Occupied", "-----", "-----\t\t-----");
        try {
            application.execute(new Producer(sharedLocation));
            application.execute(new Consumer(sharedLocation));
        }
    }
}

```

```

    } catch (Exception exception) {
        exception.printStackTrace(System.err);
    }
    application.shutdown();
}
}

```

Một số lớp khóa khác có hiệu suất cải tiến so với ReentrantLock: ReentrantReadWriteLock, StampedLock.

2. Monitor và synchronized

Một cách thực hiện đồng bộ khác là dùng các monitor (bộ giám sát) có sẵn của Java. Đối tượng nào cũng có một monitor. Monitor của một đối tượng chỉ cho phép một thread được thực hiện khối synchronized của đối tượng đó tại một thời điểm. Điều này được thực hiện bằng cách khóa đối tượng khi chương trình vào khối synchronized:

```

synchronized (object) {
    statements
}

```

Java cũng hỗ trợ các phương thức synchronized. Bộ điều phối đảm bảo tại mỗi thời điểm chỉ có một thread được gọi phương thức synchronized. Khi một thread gọi phương thức synchronized, đối tượng chứa phương thức sẽ bị khóa. Khi thread đó thực hiện xong phương thức, đối tượng chứa phương thức sẽ được mở khóa.

Một phương thức synchronized *tương đương với một khối* synchronized cho toàn bộ code của phương thức đó. Code sau:

```

public void foo() {
    synchronized(this) { // gọi code trong khối sẽ khóa this
        System.out.println("synchronized");
    }
}

```

là tương đương với:

```

public synchronized void foo() { // gọi phương thức sẽ khóa đối tượng chứa phương thức
    System.out.println("synchronized");
}

```

Bạn có thể khai báo phương thức static là synchronized, khóa sẽ được thực hiện trên đối tượng kiểu Class, truy cập bằng cách dùng cú pháp ClassName.class, ví dụ như sau:

```

class SomeClass {
    private static int val;
    public static void assign(int i) {
        synchronized(SomeClass.class) {
            val = i;
        }
    }
}

```

Trong khi thực thi phương thức synchronized, một thread có thể gọi phương thức wait() của lớp Object để chuyển sang trạng thái chờ cho đến khi một điều kiện nào đó xảy ra. Một thread khác có thể sử dụng đối tượng và khi thực hiện xong tác vụ thread đó có thể dùng phương thức notify() thông báo cho thread đang chờ truy cập đối tượng.

```

interface Buffer {
    void set(int value);
    int get();
}

```

```

public class SynchronizedBuffer implements Buffer {
    private int buffer = -1; // tài nguyên dùng chung cho thread Producer và Consumer
    private boolean occupied = false; // báo buffer đầy (true) hoặc rỗng (false)

    // đặt một trị vào buffer
    @Override public synchronized void set(int value) {
        while (occupied) {
            try {
                System.out.println("Producer tries to write. Buffer full. Producer waits.");
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace(System.err);
            }
        }
        buffer = value;
        occupied = true;
        System.out.printf(" Producer writes %d\n", buffer);
        notify(); // báo cho thread đang chờ có thể vào trạng thái running
    } // giải phóng lock trên SynchronizedBuffer

    // lấy một trị từ buffer
    @Override public synchronized int get() {
        while (!occupied) {
            try {

```

```

        System.out.println("Consumer tries to read. Buffer empty. Consumer waits.");
        wait();
    } catch (InterruptedException e) {
        e.printStackTrace(System.err);
    }
}
occupied = false;
int readValue = buffer;
System.out.printf("Consumer reads %d\n", readValue);
notify(); // báo cho thread chờ có thể vào trạng thái chạy
return readValue;
} // giải phóng lock trên SynchronizedBuffer
}

```

3. Biến atomic

Thường tác vụ trong khối đồng bộ là tác vụ cơ bản *lên một biến*. Khóa và giải phóng khóa cho tác vụ đó không hiệu quả. Java cung cấp một cách hiệu quả hơn, dùng các biến atomic trong gói `java.util.concurrent.atomic`:

`AtomicBoolean`, `AtomicInteger`, `AtomicIntegerArray`, `AtomicLong`, `AtomicLongArray`,
`AtomicReference<V>`, `AtomicReferenceArray<E>`.

Một tác vụ lên biến atomic (khởi tạo, get/set, tăng/giảm, cập nhật) là một tác vụ không thể phân chia được (tác vụ atomic).

`import java.util.concurrent.atomic.AtomicInteger;`

```

class Counter {
    public static Integer integer = new Integer(0);
    public static AtomicInteger atomicInteger = new AtomicInteger(0);
}

public class AtomicVariableTest {
    static class Incremter extends Thread {
        @Override public void run() {
            Counter.integer++;
            Counter.atomicInteger.incrementAndGet();
        }
    }

    static class Decrementer extends Thread {
        @Override public void run() {
            Counter.integer--;
            Counter.atomicInteger.decrementAndGet();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread incremeterThread[] = new Incremter[1000];
        Thread decremeterThread[] = new Decrementer[1000];
        for (int i = 0; i < 1000; ++i) {
            incremeterThread[i] = new Incremter();
            decremeterThread[i] = new Decrementer();
            incremeterThread[i].start();
            decremeterThread[i].start();
        }
        for (int i = 0; i < 1000; ++i) {
            incremeterThread[i].join();
            decremeterThread[i].join();
        }
        System.out.printf("Integer value = %d AtomicInteger value = %d ", Counter.integer, Counter.atomicInteger.get());
    }
}

```

Trong ví dụ trên, lớp `Counter` có hai thành viên: kiểu `Integer` và kiểu `AtomicInteger`, đều khởi tạo bằng 0. Có hai lớp `Thread`:

- `Incremter` tăng trị của `Integer` và `AtomicInteger`.
- `Decrementer` giảm trị của `Integer` và `AtomicInteger`.

Do số lần tăng và giảm bằng nhau nên trị cuối cùng của `Integer` và `AtomicInteger` phải luôn bằng 0. Tuy nhiên, quan sát kết quả, ta thấy đôi khi trị `Integer` bằng 0 (không có race condition) và đôi khi khác 0 (có race condition). Nhưng trị `AtomicInteger` luôn bằng 0, nghĩa là thao tác trên `AtomicInteger` là an toàn mà không cần dùng khóa.

4. API đồng bộ cấp cao

Ngoài các cấu trúc cấp thấp như `synchronized`, giao diện `Runnable`, nhiều lớp và interface trong gói `java.util.concurrent` cung cấp API cấp cao cho lập trình song hành. Các thread thường cần phối hợp hành động để hoàn thành các tác vụ cấp cao hơn, điều này mở ra nhu cầu xây dựng các khái niệm trừu tượng cấp cao cho việc đồng bộ hai hoặc nhiều thread, gọi là các *synchronizers*. Các `Synchronizers` dùng API cấp thấp bên trong để đồng bộ thread.

Từ Java 5, các synchronizers được cung cấp trong gói `java.util.concurrent`, bao gồm:

- Semaphore kiểm soát truy cập đến tài nguyên dùng chung, Semaphore giữ một bộ đếm để xác định số lượng tài nguyên nó kiểm soát.

Một semaphore có thể tự động tăng hoặc giảm trị đếm tài nguyên để điều khiển truy cập đến tài nguyên dùng chung. Các chương trình song hành dùng chúng để đồng bộ tương tác giữa nhiều thread. Có hai kiểu semaphore: counting semaphore và binary semaphore (trị đếm chỉ chứa 0 hoặc 1).

- CountdownLatch cho phép chờ vài thread hoàn thành trước khi bắt đầu xử lý thread chỉ định. Các thread được chờ sẽ giảm bộ đếm xuống, trong khi thread chỉ định chờ bộ đếm về 0 mới thực hiện.

```
import java.util.concurrent.CountDownLatch;
class Service implements Runnable {
    private final String name;
    private final int timeToStart;
    private final CountDownLatch latch;

    public Service(String name, int timeToStart, CountDownLatch latch) {
        this.name = name;
        this.timeToStart = timeToStart;
        this.latch = latch;
    }

    @Override public void run() {
        try {
            Thread.sleep(timeToStart);
        } catch (InterruptedException ex) {
            ex.printStackTrace(System.err);
        }
        System.out.println(name + " is Up");
        latch.countDown(); // giảm count của CountDownLatch xuống 1 đơn vị
    }
}

public class CountdownLatchDemo {
    public static void main(String args[]) {
        final CountDownLatch latch = new CountDownLatch(2); // khởi tạo bộ đếm với 2
        new Thread(new Service("CacheService", 1000, latch)).start();
        new Thread(new Service("ValidationService", 1200, latch)).start();
        try {
            latch.await(); // thread main chờ CountDownLatch đếm xuống đến 0 mới thực hiện
            System.out.println("All services are up, Application is starting now");
        } catch (InterruptedException ex) {
            ex.printStackTrace(System.err);
        }
    }
}

- Exchanger dùng trao đổi dữ liệu giữa hai thread.
import java.util.concurrent.Exchanger;
class MakeString implements Runnable {
    private Exchanger<String> ex;
    public MakeString(Exchanger<String> ex) {
        this.ex = ex;
    }

    @Override public void run() {
        String s = "create string";
        try {
            s = ex.exchange(s);
            System.out.printf("%s now has [%s]%n", Thread.currentThread().getName(), s);
        } catch (InterruptedException e) {
            e.printStackTrace(System.err);
        }
    }
}

class UseString implements Runnable {
    private Exchanger<String> ex;
    public UseString(Exchanger<String> ex) {
        this.ex = ex;
    }

    @Override public void run() {
```



```

    String s = "use string";
    try {
        s = ex.exchange(s);
        System.out.printf("%s now has [%s]%n", Thread.currentThread().getName(), s);
    } catch (InterruptedException e) {
        e.printStackTrace(System.err);
    }
}
}

```

```

public class ExchangerDemo {
    public static void main(String[] args) {
        Exchanger<String> exchanger = new Exchanger<>();
        new Thread(new MakeString(exchanger), "MakeString").start();
        new Thread(new UseString(exchanger), "UseString").start();
    }
}

```

- CyclicBarrier cung cấp một "rào chắn" (barrier), mà một thread phải chờ đợi cho đến khi các thread nó chờ đợi đều đến "rào chắn" đó.

```

import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.BrokenBarrierException;
class MixedDoubleTennisGame extends Thread {
    // phương thức run() chỉ được gọi khi các Player đều đến barrier
    @Override public void run() {
        System.out.println("All four players ready, game starts\n Love all...");
    }
}

```

// khi một Player đến, nó chờ các Player khác đến barrier

```

class Player extends Thread {
    CyclicBarrier waitPoint;
    public Player(CyclicBarrier barrier, String name) {
        this.setName(name);
        waitPoint = barrier;
    }

    @Override public void run() {
        System.out.println("Player " + getName() + " is ready ");
        try {
            waitPoint.await(); // chờ cho đến khi các Player đến barrier
        } catch (BrokenBarrierException | InterruptedException e) {
            System.err.println("An exception occurred while waiting... " + e);
        }
    }
}

```

// Tạo đối tượng CyclicBarrier, truyền số thread và thread sẽ chạy khi các thread đều đến barrier

```

public class CyclicBarrierTest {
    public static void main(String[] args) {
        // trận đấu đôi tennis cần chờ đủ bốn người chơi
        System.out.println("Reserving tennis court as soon as four players arrive, game will start");
        CyclicBarrier barrier = new CyclicBarrier(4, new MixedDoubleTennisGame());
        new Player(barrier, "Roger Federer").start();
        new Player(barrier, "Novak Djokovic").start();
        new Player(barrier, "Rafael Nadal").start();
        new Player(barrier, "Andy Murray").start();
    }
}

```

Networking

TCP

1. Địa chỉ Socket

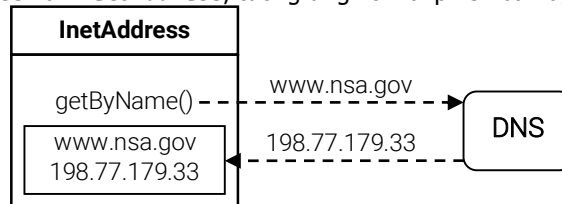
Khi client bắt đầu liên lạc với server, nó phải chỉ định địa chỉ IP của host đang chạy chương trình server. Hạ tầng mạng sẽ dùng *địa chỉ đích* này để chuyển thông tin của client đến một máy cụ thể.

Các địa chỉ được chỉ định trong Java bằng cách dùng một trong các chuỗi sau:

- Địa chỉ dạng số, tùy theo phiên bản IP đang dùng, ví dụ `192.0.2.27` cho IPv4 hoặc `fe20:12a0::0abc:1234` cho IPv6.

- Địa chỉ dạng tên, ví dụ `server.example.com`. Trong trường hợp này, địa chỉ dạng tên sẽ được dịch vụ DNS *phân giải* thành địa chỉ dạng số trước khi dùng.

Lớp trừu tượng `InetAddress` thể hiện một địa chỉ trên mạng, đóng gói trong nó thông tin địa chỉ dạng tên và địa chỉ dạng số. Lớp này có hai lớp con: `Inet4Address` và `Inet6Address`, tương ứng với hai phiên bản địa chỉ IP đang dùng.



Để lấy các địa chỉ của một host cục bộ, ta dùng lớp trừu tượng `NetworkInterface`. Lớp này cung cấp truy cập đến thông tin của tất cả các giao diện mạng (network interface) của một host.

```
import java.util.Enumeration;
import java.net.*;
```

```
public class InetAddressExample {
    public static void main(String[] args) {
        // Lấy các interface và địa chỉ liên kết với chúng cho host này
        try {
            Enumeration<NetworkInterface> interfaceList = NetworkInterface.getNetworkInterfaces();
            if (interfaceList == null) {
                System.out.println("--No interfaces found--");
            } else {
                while (interfaceList.hasMoreElements()) {
                    NetworkInterface iface = interfaceList.nextElement();
                    System.out.println("Interface " + iface.getName() + ":");
                    Enumeration<InetAddress> addrList = iface.getInetAddresses();
                    if (!addrList.hasMoreElements()) {
                        System.out.println("\t(No addresses for this interface)");
                    } while (addrList.hasMoreElements()) {
                        InetAddress address = addrList.nextElement();
                        System.out.print("\tAddress " + ((address instanceof Inet4Address ? "(v4)"
                            : (address instanceof Inet6Address ? "(v6)" : "(?)"))));
                        System.out.println(": " + address.getHostAddress());
                    }
                }
            }
        } catch (SocketException se) {
            System.out.println("Error getting network interfaces:" + se.getMessage());
        }
        // Lấy tên/địa chỉ cho (các) host liệt kê như tham số dòng lệnh
        for (String host : args) {
            try {
                System.out.println(host + ":");
                InetAddress[] addressList = InetAddress.getAllByName(host);
                for (InetAddress address : addressList) {
                    System.out.println("\t" + address.getHostAddress() + "/" + address.getHostAddress());
                }
            } catch (UnknownHostException e) {
                System.out.println("\tUnable to find address for " + host);
            }
        }
    }
}
```

InetAddress: Khởi tạo, truy cập

```
static InetAddress[] getAllByName(String host)
static InetAddress getByName(String host)
static InetAddress getLocalHost()
byte[] getAddress()
```

Các phương thức static factory trả về các thực thể, chúng được dùng như đối số chỉ định host cho các phương thức thuộc lớp Socket. Chuỗi *host* có thể là địa chỉ dạng tên hoặc địa chỉ dạng số. Với các địa chỉ IPv6 dạng số, dạng rút gọn có thể được dùng. Một tên có thể liên kết với một hay nhiều địa chỉ dạng số; phương thức `getAllByName` trả về mảng các thực thể `InetAddress` liên kết với một tên.

Phương thức `getAddress` trả về địa chỉ dạng nhị phân như một mảng byte có chiều dài tương ứng (4 byte cho `Inet4Address` và 16 byte cho `Inet6Address`). Phần tử đầu tiên của mảng byte là byte trái nhất của địa chỉ.

Một thực thể `InetAddress` có nhiều cách chuyển thành một String.

`InetAddress`: Chuyển thành String

```
String toString()
String getHostAddress()
String getHostName()
String getCanonicalHostName()
```

Các phương thức này trả về địa chỉ dạng tên hoặc dạng số của một host, hoặc kết hợp giữa hai dạng theo định dạng chỉ định. Phương thức `toString` trả về chuỗi có định dạng, ví dụ *hostname.example.com/192.0.2.127*. Phương thức `getHostAddress` trả về địa chỉ dạng số, với IPv6 chuỗi thể hiện luôn chứa đủ 8 nhóm (7 dấu ":"), tránh nhầm lẫn khi có số hiệu port kèm sau cuối. Hai phương thức cuối trả về tên của host, khác nhau như sau: nếu địa chỉ cho trước có dạng tên, `getHostName` trả về tên (không cần phân giải địa chỉ), ngược lại sẽ phân giải địa chỉ bằng cách dùng cơ chế phân giải của hệ thống; `getCanonicalHostName` luôn thử phân giải địa chỉ để lấy tên miền đầy đủ, ví dụ *bam.example.com*. Cả hai phương thức đều trả về địa chỉ dạng số nếu phân giải không hoàn tất.

Lớp `InetAddress` cũng hỗ trợ kiểm tra chi tiết các thuộc tính của địa chỉ chứa trong thực thể của nó.

`InetAddress`: Kiểm tra thuộc tính

```
boolean isAnyLocalAddress()
boolean isLinkLocalAddress()
boolean isLoopbackAddress()
boolean isMulticastAddress()
boolean isMCGlobal()
boolean isMCLinkLocal()
boolean isMCNodeLocal()
boolean isMCOrgLocal()
boolean isMCSiteLocal()
boolean isReachable(int timeout)
boolean isReachable(NetworkInterface netif, int ttl, int timeout)
```

Các phương thức này làm việc với cả IPv4 lẫn IPv6. Ba phương thức đầu dùng kiểm tra có phải là địa chỉ "any", "link-local" hoặc "loopback". Phương thức thứ tư kiểm tra có phải là địa chỉ "multicast", và các phương thức `isMC...` kiểm tra tầm vực (scope) của địa chỉ multicast.

Hai phương thức cuối kiểm tra xem có thể trao đổi packet với host chỉ định bởi `InetAddress` này. Khác với các phương thức trên (chỉ kiểm tra cú pháp địa chỉ), các phương thức này gửi packet kiểm tra. Phương thức cuối gửi các packet thông qua `NetworkInterface` chỉ định, với trị TTL (time-to-live) chỉ định để xác định `InetAddress` này có thể kết nối được hay không.

Lớp `NetworkInterface` cũng cung cấp một số lớn các phương thức.

`NetworkInterface`: Khởi tạo, lấy thông tin

```
static Enumeration<NetworkInterface> getNetworkInterfaces();
static NetworkInterface getByInetAddress(InetAddress address);
static NetworkInterface getName(String name);
Enumeration<InetAddress> getInetAddresses();
String getName();
String getDisplayName();
```

Thường dùng `getNetworkInterfaces` để lấy danh sách các interface của host (kể cả loopback, interface ảo, ...), rồi dùng phương thức `getInetAddress` để lấy địa chỉ của từng interface.

Phương thức `getName` trả về tên của interface (không phải tên host),

2. TCP Socket

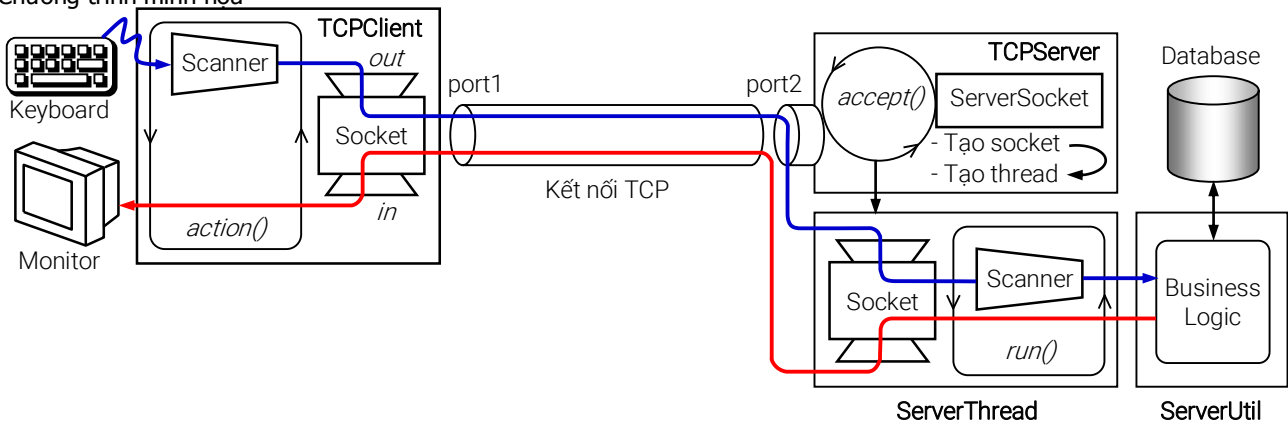
Java cung cấp hai lớp cho TCP: `Socket` và `ServerSocket`.

Một kết nối TCP là một kênh truyền thông hai chiều mà mỗi đầu cuối được xác định bởi một Socket, bao gồm địa chỉ IP và một số hiệu port. Trước khi được dùng để truyền thông, một kết nối TCP phải được hình thành:

- Bắt đầu bằng việc TCP client gửi một yêu cầu kết nối đến TCP server khi hình thành thực thể Socket phía Client.
- Trên TCP server, một thực thể `ServerSocket` lắng nghe yêu cầu tạo kết nối TCP này tại port chỉ định. Khi client kết nối, server tạo một thực thể Socket phía server để xử lý kết nối nhận được.

Như vậy, một kết nối TCP được hình thành với hai thực thể Socket thể hiện hai đầu cuối của kết nối TCP.

3. Chương trình minh họa



a) TCP Client

Server thụ động lắng nghe tại một port chỉ định (gọi là port dịch vụ), client kết nối đến server tại port dịch vụ đó. Chương trình client điển hình có ba bước:

- Sinh ra một đối tượng Socket, kết nối đến host (tên host hoặc địa chỉ IP) và port chỉ định.
- Trao đổi thông tin bằng cách dùng các stream I/O của Socket. Đối tượng Socket là một đối tượng "hiểu stream", ta có thể lấy các stream I/O cơ bản (InputStream và OutputStream) từ nó. Sau đó lồng thêm các stream hữu dụng hơn cho đến khi nhận được các phương thức nhập/xuất như ý. Các stream lồng thêm thường là character stream hoặc object stream.

Thông qua các stream này ta gửi/nhận dữ liệu với phía server.

- Đóng kết nối bằng cách dùng phương thức close của Socket hoặc dùng try-with-resources (từ Java 7) để tự đóng kết nối.

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.NoSuchElementException;
import java.util.Scanner;
```

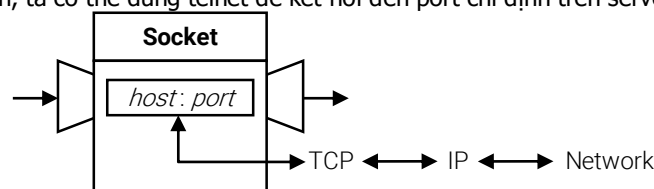
```
public class TCPClient {
    private static final int PORT = 6868;
    private static final String HOST = "localhost";

    public TCPClient() { }

    public void action() {
        Scanner keyboard = new Scanner(System.in);
        String s;
        // tạo Socket kết nối với server tại host (IP hoặc hostname) và port chỉ định
        try (Socket socket = new Socket(HOST, PORT)) {
            // lấy stream cơ bản từ Socket và lồng thêm stream để có các stream hữu dụng hơn
            try (Scanner in = new Scanner(socket.getInputStream());
                PrintWriter out = new PrintWriter(socket.getOutputStream(), true)) {
                while (true) {
                    s = keyboard.nextLine();           // nhận một dòng từ bàn phím
                    out.println(s);                   // ghi vào stream xuất để chuyển đến server
                    System.out.println(in.nextLine()); // nhận từ stream nhập rồi xuất ra màn hình
                }
            } catch (NoSuchElementException e) {
                System.err.println("Server has departed");
            }
        } catch (IOException e) {
            System.err.println("Connect refused!");
        }
    }

    public static void main(String[] args) {
        new TCPClient().action();
    }
}
```

Trong phương thức action(), vòng lặp sẽ dừng khi chờ nhận một dòng từ bàn phím hoặc khi chờ nhận dòng trả về từ server. Thay cho chương trình client trên, ta có thể dùng telnet để kết nối đến port chỉ định trên server: telnet localhost 6868



Socket: Khởi tạo

```

Socket(InetAddress remoteAddr, int remotePort)
Socket(String remoteHost, int remotePort)
Socket(InetAddress remoteAddr, int remotePort, InetAddress localAddr, int localPort)
Socket(String remoteHost, int remotePort, InetAddress localAddr, int localPort)
Socket()

```

Bốn constructor đầu tạo một TCP socket và kết nối nó đến một địa chỉ từ xa và một port chỉ định. Hai constructor đầu không chỉ định địa chỉ và port cục bộ (local, nghĩa là của client); sẽ dùng địa chỉ cục bộ mặc định và một port nào đó có thể sử dụng được. Chỉ định địa chỉ cục bộ cần thiết khi client có nhiều interface. Địa chỉ chỉ định có thể là một thực thể `InetAddress` hoặc một chuỗi với định dạng được chấp nhận bởi các phương thức khởi tạo của `InetAddress`.

Constructor cuối dùng tạo một socket không kết nối, sẽ được kết nối tường minh sau (thông qua phương thức `connect`), trước khi dùng để truyền thông.

Socket: Tác vụ

```

void connect(SocketAddress destination)
void connect(SocketAddress destination, int timeout)
InputStream getInputStream()
OutputStream getOutputStream()
void close()
void shutdownInput()
void shutdownOutput()

```

Các phương thức `connect` tạo một kết nối TCP đến một đầu cuối chỉ định đã mở sẵn. Lớp trừu tượng `SocketAddress` thể hiện một dạng chung của địa chỉ một socket; lớp con của nó là `InetSocketAddress`.

Truyền thông giữa hai bên client và server thông qua các I/O stream lấy từ thực thể `Socket` liên kết với chúng, dùng các phương thức `getInputStream` và `getOutputStream`.

Phương thức `close` dùng đóng socket và các I/O stream liên kết với chúng. Phương thức `shutdownInput` đóng đầu nhập (nhận về) của một TCP stream. Phương thức `shutdownOutput` đóng đầu xuất (gửi đi).

Socket: Thuộc tính

```

InetAddress getInetAddress()
int getPort()
InetAddress getLocalAddress()
int getLocalPort()
SocketAddress getRemoteSocketAddress()
SocketAddress getLocalSocketAddress()

```

Các phương thức này trả về các thuộc tính của socket. Các phương thức trả về một `SocketAddress` thực tế trả về một thực thể `InetSocketAddress`. `InetSocketAddress` đóng gói trong nó một `InetAddress` và một số hiệu port.

Lớp `Socket` thực tế có một số lượng lớn các thuộc tính liên kết được tham chiếu như là các tùy chọn socket (socket option).

InetSocketAddress

```

InetSocketAddress(InetAddress addr, int port)
InetSocketAddress(int port)
InetSocketAddress(String hostname, int port)
static InetSocketAddress createUnresolved(String hostname, int port)
boolean isUnresolved()
InetAddress getAddress()
int getPort()
String getHostName()
String toString()

```

Lớp `InetSocketAddress` cung cấp một kết hợp giữa địa chỉ host với port. Constructor với đối số là port được dùng để chỉ một địa chỉ đặc biệt nào đó, thường dùng lập trình phía server. Constructor với đối số là một chuỗi `hostname` sẽ thử phân giải tên host thành một địa chỉ IP; phương thức static `createUnresolved` cho phép một thực thể khởi tạo mà không thử phân giải địa chỉ. Phương thức `isUnresolved` trả về true nếu thực thể được tạo theo cách trên, hoặc nếu việc thử phân giải địa chỉ trong constructor thất bại. Các getter cung cấp truy cập đến các thành phần chỉ định của `InetSocketAddress`.

b) TCPServer

b.1. Server

Nhiệm vụ chính của server là lắng nghe tại một port chỉ định. Khi client kết nối đến port này, server sẽ tạo một socket phía server để liên lạc với socket phía client, tạo thành một kết nối TCP. Chương trình server điển hình có hai bước:

- Sinh ra một đối tượng `ServerSocket` với đối số là `port` chỉ định. Đối tượng `ServerSocket` này sẽ lắng nghe những kết nối đến `port` chỉ định từ phía client.
- Trong vòng lặp lắng nghe xử lý:

+ Dùng phương thức `accept` của `ServerSocket` để lắng nghe kết nối đến `port` chỉ định, vòng lặp sẽ dừng chờ. Khi có một client kết nối đến, phương thức `accept` trả về một đối tượng `Socket`, đại diện cho socket phía server kết nối với socket phía client tương ứng. Để dễ hình dung, bạn tưởng tượng có hai `Socket` ở hai đầu của kết nối TCP.

+ Server sinh một thread và trao `Socket` vừa tạo ra cho thread đó, để tiến hành giao tiếp với client trong một thread riêng.

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class TCPServer {
    private static final int PORT = 6868;
    private ServerSocket server;

    public TCPServer() {
        try {
            this.server = new ServerSocket(PORT); // tạo ServerSocket với port lắng nghe chỉ định
            System.out.println("Server listening...");
        } catch (IOException e) {
            System.err.println("Server start error!");
        }
    }

    public void action() {
        Socket socket;
        ExecutorService executor = Executors.newCachedThreadPool();
        try {
            // khi có kết nối, accept() trả về một Socket, kết nối Socket tương ứng phía client
            while ((socket = server.accept()) != null) {
                System.out.println("Thread generating...");
                // trao Socket cho ServerThread để xử lý riêng kết nối trong một thread,
                // rồi đặt thread mới vào thread pool
                executor.execute(new ServerThread(socket));
            }
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }

    public static void main(String[] args) {
        new TCPServer().action();
    }
}
```

b.2. Server Thread

Để tránh giao tiếp bị khóa (blocked) vì phải chờ xử lý yêu cầu, server giải quyết từng yêu cầu trên một thread riêng (multi-threaded server). Công việc của mỗi thread gồm:

- Kiểm tra dữ liệu nhận được qua stream nhập của `Socket`: thường là phân tích cú pháp (parse) chuỗi nhận được để xác nhận hợp lệ và tách chuỗi thành những token xử lý riêng.
- Xử lý chuỗi với các phương thức nghiệp vụ (business method), các phương thức này hoặc được viết dưới dạng phương thức hỗ trợ (helper method), hoặc tập trung trong một lớp riêng (business object). Kết quả xử lý được chuyển về client qua stream xuất của `Socket`.
- Đóng socket khi xử lý xong yêu cầu hoặc khi mất kết nối với client.

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.NoSuchElementException;
import java.util.Scanner;

public class ServerThread implements Runnable {
    private Socket socket;

    public ServerThread(Socket socket) {
        this.socket = socket;
    }

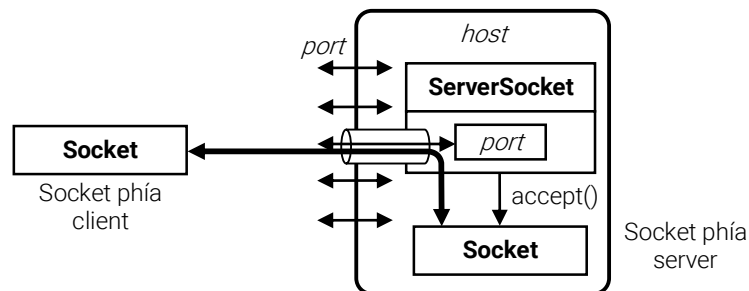
    @Override
    public void run() {
        String command;
        // từ Socket (phía Server) lấy stream cơ bản và lồng thêm stream để có các stream hữu dụng hơn
```



```

try (Scanner in = new Scanner(socket.getInputStream());
    PrintWriter out = new PrintWriter(socket.getOutputStream(), true)) {
    while (true) {
        command = in.nextLine();
        if (command.trim().matches("KW \\d+")) {
            // xử lý dữ liệu nhập nhờ lớp công cụ, trả kết quả về client bằng stream xuất
            int kw = Integer.parseInt(command.substring(2).trim());
            out.printf("Pending: %s%n", ServerUtil.getMoney(kw));
        } else {
            out.println("Syntax error!");
        }
    }
} catch (NoSuchElementException e) {
    System.err.println("Client has departed!");
} catch (IOException e) {
    System.err.println(e.getMessage());
}
}
}

```



ServerSocket: Khởi tạo

```

ServerSocket(int LocalPort)
ServerSocket(int LocalPort, int queueLimit)
ServerSocket(int LocalPort, int queueLimit, InetAddress LocalAddr)
ServerSocket()

```

Đầu cuối TCP trên server phải liên kết với một port chỉ định (gọi là port dịch vụ) để client trực tiếp kết nối đến. Ba constructor đầu tạo một đầu cuối TCP liên kết với port cục bộ chỉ định và sẵn sàng accept kết nối đi vào. Số hiệu port hợp lệ trong dãy 0 – 65535 (nếu số hiệu port là 0, một số hiệu port tùy ý trong dãy sẽ được chọn). Một cách tùy chọn, kích thước của queue (hàng đợi) kết nối và địa chỉ cục bộ có thể được thiết lập. Địa chỉ cục bộ, nếu chỉ định, phải là địa chỉ của một interface của host. Nếu không chỉ định địa chỉ, socket sẽ chấp nhận kết nối đến địa chỉ IP bất kỳ. Điều này áp dụng cho host có nhiều interface và server muốn nhận kết nối trên một trong những interface của nó.

Constructor thứ tư tạo một ServerSocket không liên kết với bất kỳ port cục bộ nào, trước khi dùng nó, phải liên kết (bound) nó với một port, bằng phương thức bind.

ServerSocket: Tác vụ

```

void bind(int port)
void bind(int port, int queueLimit)
Socket accept()
void close()

```

Phương thức bind liên kết socket này với một port cục bộ. Một ServerSocket chỉ có thể liên kết với một port. Nếu thực thể này liên kết sẵn với một port hoặc nếu port chỉ định đang được sử dụng, phương thức này sẽ ném ra IOException.

Phương thức accept trả về một thực thể Socket, đầu cuối phía server của kết nối TCP. Khi chưa có kết nối vào port đang lắng nghe, phương thức accept sẽ khóa cho đến khi có kết nối hoặc timeout.

Phương thức close dùng đóng socket.

ServerSocket: Thuộc tính

```

InetAddress getInetAddress()
SocketAddress getLocalSocketAddress()
int getLocalPort()

```

Các phương thức này trả về địa chỉ và port cục bộ của ServerSocket. Chú ý rằng khác với Socket, một ServerSocket không liên kết với các I/O stream.

c) Server Util

Lớp ServerUtil tách phần thao tác nghiệp vụ (business logic) ra khỏi phần giao tiếp mạng. Vì lớp này được xem như lớp công cụ nên các phương thức đều static. Ví dụ về một phương thức nghiệp vụ:

```

class ServerUtil {
    public static String getMoney(int kw) {
        int money = kw * 500;
        if (kw > 100) money += (kw - 100) * 200;
        if (kw > 200) money += (kw - 200) * 300;
        return String.format("Spending: %d", money);
    }
}

```

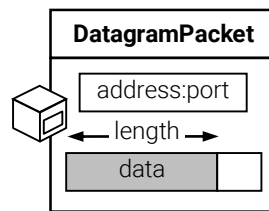
UDP

UDP (**U**ser **D**atagram **P**rotocol) là giao thức không tạo kết nối (connectionless) thuộc lớp Transport trong mô hình ISO/OSI. Với các gói có kích thước lớn, khi gửi bị phân thành các segment, giao thức UDP không bảo đảm các segment được nhận đầy đủ và lắp ráp đúng thứ tự phía nhận.

1. DatagramPacket và DatagramSocket

a) DatagramPacket

Thay vì dùng các byte stream để chuyển dữ liệu thông qua socket như TCP, hai bên đầu cuối dùng UDP trao đổi các thông điệp được đóng gói, gọi là datagram, thể hiện trong Java như các thực thể của lớp DatagramPacket.



Ngoài dữ liệu, thực thể lớp DatagramPacket còn chứa thông tin về địa chỉ và port tùy theo packet gửi hoặc nhận. Khi DatagramPacket được gửi đi, địa chỉ và port chỉ đích đến, khi DatagramPacket được nhận, địa chỉ và port chỉ nguồn gửi. Bên trong DatagramPacket cũng có các trường offset và length, mô tả vị trí byte đầu tiên và kích thước dữ liệu trong vùng đệm.

DatagramPacket: Khởi tạo

```

DatagramPacket(byte[] data, int length)
DatagramPacket(byte[] data, int offset, int length)
DatagramPacket(byte[] data, int length, InetAddress remoteAddr, int remotePort)
DatagramPacket(byte[] data, int offset, int length, InetAddress remoteAddr, int remotePort)
DatagramPacket(byte[] data, int length, SocketAddress sockAddr)
DatagramPacket(byte[] data, int offset, int length, SocketAddress sockAddr)

```

Các constructor này tạo một datagram với dữ liệu chứa trong một mảng byte. Hai constructor đầu thường dùng để nhận vì ta không chỉ định địa chỉ đích, mặc dù có thể chỉ định sau bằng các phương thức setAddress, setPort hoặc setSocketAddress. Bốn constructor sau thường dùng để gửi.

Khi chỉ định offset, dữ liệu chuyển vào mảng byte bắt đầu tại vị trí chỉ định bởi offset. Đối số length chỉ định số byte sẽ được chuyển từ mảng byte khi gửi, hoặc số byte tối đa đã được chuyển khi nhận; nó có thể nhỏ hơn nhưng không vượt quá data.length.

Địa chỉ và port đích có thể chỉ định riêng hoặc chung trong một thực thể SocketAddress.

DatagramPacket: Địa chỉ

```

InetAddress getAddress()
void setAddress(InetAddress address)
int getPort()
void setPort(int port)
SocketAddress getSocketAddress()
void setSocketAddress(SocketAddress sockAddr)

```

Bổ sung cho constructor, các phương thức này cung cấp một cách khác để truy cập và thay đổi địa chỉ của một DatagramPacket. Chú ý là phương thức receive của DatagramSocket thiết lập địa chỉ và port của gói nhận thành địa chỉ và port của người gửi.

DatagramPacket: Xử lý dữ liệu

```

int getLength()
void setLength(int length)
int getOffset()
byte[] getData()
void setData(byte[] data)
void setData(byte[] buffer, int offset, int length)

```

Hai phương thức đầu trả về/thiết lập chiều dài dữ liệu của datagram. Chiều dài của dữ liệu có thể thiết lập bởi constructor hoặc phương thức setLength. Thiết lập chiều dài lớn hơn vùng đệm sẽ ném ra IllegalArgumentException. Phương thức receive của DatagramSocket dùng length này theo hai cách: khi tạo gói, nó chỉ định số byte tối đa của thông điệp nhận sẽ được sao chép vào vùng đệm, và khi trả về nó chỉ định số byte thực sự được đặt vào vùng đệm.

Phương thức `getOffset` trả về vị trí của byte dữ liệu đầu tiên trong vùng đệm dùng gửi/nhận. Không có phương thức `setOffset`, tuy nhiên ta có thể thiết lập bằng `setData`.

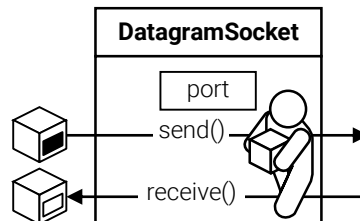
Phương thức `getData` trả về mảng byte liên kết với datagram. Các phương thức `setData` đặt dữ liệu vào mảng byte chỉ định, phương thức đầu dùng trọn mảng byte, phương thức sau đặt dữ liệu vào từ `offset` đến `offset + length - 1`.

b) DatagramSocket

UDP dùng thực thể thuộc lớp `DatagramSocket` như một đối tượng gửi và nhận các `DatagramPacket`.

Để gửi, chương trình Java xây dựng một thực thể `DatagramPacket` chứa dữ liệu muốn gửi rồi truyền nó như là đối số đến phương thức `send` của một đối tượng `DatagramSocket`.

Để nhận, chương trình Java xây dựng một thực thể `DatagramPacket` chứa một vùng đệm cấp phát trước, tùy theo IP header kích thước tối đa khoảng từ 65467 đến 65507 byte, rồi truyền nó như là đối số đến phương thức `receive` của một đối tượng `DatagramSocket`. Nếu có packet vào, vùng đệm này sẽ chứa nội dung thông điệp nhận được.



Tất cả `DatagramSocket` đều liên kết với một port cục bộ. Khi viết client, thường không quan tâm đến port này. Trên server, port liên kết này dùng để lắng nghe datagram đi vào.

DatagramSocket: Khởi tạo

```
DatagramSocket()
DatagramSocket(int LocalPort)
DatagramSocket(int LocalPort, InetAddress LocalAddr)
DatagramSocket(SocketAddress interface)
```

Các constructor này tạo một socket UDP. Nếu không chỉ định port cục bộ hoặc chỉ định port là 0, socket sẽ kết nối với một port bất kỳ dùng được, lắng nghe trên port đó. Port TCP và UDP không liên quan nhau, hai server không liên quan có thể dùng cùng một số hiệu port nếu một server dùng TCP và server còn lại dùng UDP.

Constructor thứ ba mở một socket với một port và địa chỉ (interface) biết trước. Java 1.4 thêm constructor với đối số là `SocketAddress`, gồm chung port và địa chỉ.

Chú ý là port và địa chỉ từ xa (remote) lưu trong `DatagramPacket`, không trong `DatagramSocket`. Như vậy một `DatagramSocket` có thể gửi/nhận dữ liệu cho nhiều host và port từ xa.

DatagramSocket: Kết nối và Đóng kết nối

```
void connect(InetAddress remoteAddr, int remotePort)
void connect(SocketAddress remoteSockAddr)
void disconnect()
void close()
```

Các phương thức `connect` dùng thiết lập địa chỉ và port kết nối từ xa cho socket. Một khi kết nối, socket chỉ liên lạc với địa chỉ và port chỉ định; thử gửi datagram với địa chỉ và port khác (chỉ định bởi `DatagramPacket`) sẽ ném ra exception.

Phương thức `disconnect` hủy thiết lập địa chỉ và port từ xa. Phương thức `close` đóng socket và không sử dụng tiếp.

DatagramSocket: Địa chỉ

```
InetAddress getInetAddress()
int getPort()
SocketAddress getRemoteSocketAddress()
InetAddress getLocalAddress()
int getLocalPort()
SocketAddress getLocalSocketAddress()
```

Phương thức đầu trả về một thực thể `InetAddress` thể hiện địa chỉ của socket từ xa mà socket này sẽ kết nối đến, trả về `null` nếu không kết nối được. Tương tự, `getPort` sẽ trả về số hiệu port socket này sẽ kết nối đến, trả về -1 nếu không kết nối được. Phương thức thứ ba trả về cả hai địa chỉ và port, đóng gói trong một thực thể `SocketAddress`, trả về `null` nếu không kết nối được. Ba phương thức sau cung cấp cùng một dịch vụ như trên, nhưng với địa chỉ và port cục bộ.

DatagramSocket: Gửi và Nhận

```
void send(DatagramPacket packet)
void receive(DatagramPacket packet)
```

Phương thức `send` dùng gửi `DatagramPacket`. Packet được gửi tới địa chỉ được chỉ định bởi `DatagramPacket`.

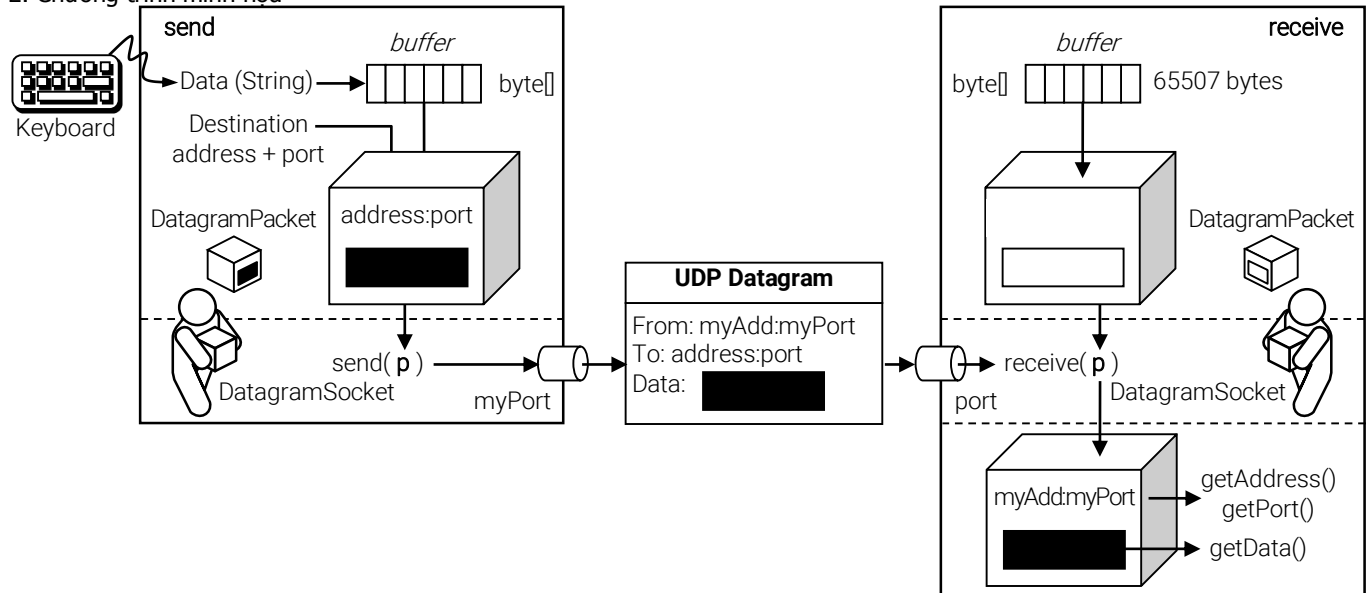
Phương thức `receive` khóa (block) cho đến khi nhận được một datagram, dữ liệu datagram nhận được sẽ được sao chép vào `DatagramPacket` cho trước (là đối số của phương thức `receive`).

DatagramSocket: Tùy chọn

```
int getSoTimeout()
void setSoTimeout(int timeoutMillis)
```

Các phương thức này trả về/thiết lập khoảng thời gian tối đa một lời gọi receive sẽ khóa (block) socket này. Nếu quá hạn thời gian này mà không nhận được dữ liệu, một `InterruptedIOException` được ném ra. Trị timeout này tính bằng miligiây.

2. Chương trình minh họa



a) UDP Client

UDP client bắt đầu gửi một datagram đến server đang thụ động chờ kết nối. UDP client điển hình thực hiện ba bước:

- Sinh ra một thực thể `DatagramSocket`. Tùy chọn có thể chỉ định địa chỉ và port cục bộ (tức địa chỉ và port của client).
- `DatagramSocket` gửi và nhận các thực thể `DatagramPacket` bằng cách dùng các phương thức `send` và `receive`.
- Khi chấm dứt, giải phóng socket bằng cách dùng phương thức `close` của `DatagramSocket`.

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;
import java.net.UnknownHostException;
import java.util.Scanner;
```

```
public class UDPClient {
    private static final int PORT = 1234;
    private static final String HOST = "127.0.0.1";
    private DatagramSocket socket;
    private InetAddress host;

    public UDPClient() {
        try {
            socket = new DatagramSocket();
            host = InetAddress.getByName(HOST); // địa chỉ đích, thực thể lớp InetAddress
        } catch (UnknownHostException | SocketException e) {
            e.printStackTrace(System.err);
        }
    }

    public void action() {
        Scanner keyboard = new Scanner(System.in);
        try {
            while (true) {
                String s = keyboard.nextLine(); // nhận chuỗi từ bàn phím, gửi đến server
                send(s, host, PORT);
                // lấy dữ liệu từ datagram nhận được, chuyển thành String rồi xuất ra màn hình
                String r = new String(receive().getData()).trim();
                System.out.println(r);
            }
        } catch (Exception e) {
```

```

        e.printStackTrace(System.err);
    } finally {
        socket.close();
    }
}

private void send(String s, InetAddress host, int port) throws IOException {
    byte[] buffer = s.getBytes();
    DatagramPacket packet = new DatagramPacket(buffer, buffer.length, host, port);
    socket.send(packet);
}

private DatagramPacket receive() throws IOException {
    byte[] buffer = new byte[65507];
    DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
    socket.receive(packet);
    return packet;
}

public static void main(String[] args) {
    new UDPCClient().action();
}
}

```

UDP client và UDP server đều dùng các phương thức send và receive, mô tả cách chuẩn bị, gửi và nhận datagram:

- Gửi: chuyển dữ liệu thành mảng byte, tạo DatagramPacket gửi đi với địa chỉ và port đích. Gửi bằng phương thức send của DatagramSocket.
- Nhận: chuẩn bị DatagramPacket nhận với vùng đệm (rỗng) có kích thước tối đa (65507 byte), gọi phương thức receive của DatagramSocket với datagram vừa tạo là đối số. Lấy dữ liệu từ DatagramPacket nhận được.

b) UDP Server

Giống như TCP server, nhiệm vụ của UDP server là thiết lập một đầu cuối liên lạc và thụ động chờ client khởi tạo kết nối đến. UDP server điển hình thực hiện ba bước:

- Khởi tạo một thực thể DatagramSocket, chỉ định port lắng nghe cục bộ. Có thể chỉ định cả địa chỉ cục bộ. Bây giờ server sẵn sàng nhận datagram từ client bất kỳ.
- Nhận một thực thể DatagramPacket bằng cách dùng phương thức receive của DatagramSocket. Khi phương thức receive thực hiện xong, datagram là đối số của phương thức receive sẽ chứa địa chỉ của client cần thm chiều để gửi datagram trả lời.
- Gửi datagram trả lời bằng cách dùng phương thức send của DatagramSocket.

```

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class UDPServer {
    private static final int PORT = 1234;
    private DatagramSocket socket;

    public UDPServer() {
        try {
            socket = new DatagramSocket(PORT);
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }

    public void action() {
        try {
            System.out.println("Server listening...");
            while (true) {
                DatagramPacket packet = receive();
                // phân tích datagram nhận, lấy dữ liệu, lấy địa chỉ nguồn và port nguồn để gửi đáp trả
                InetAddress host = packet.getAddress();
                int port = packet.getPort();
                String s = new String(packet.getData()).trim();
                // xử lý dữ liệu nhập nhờ business object,
                // đóng gói kết quả vào datagram và gửi về bằng phương thức send
                if (s.matches("^KW \\d+$")) {
                    int kw = Integer.parseInt(s.substring(2, s.length()).trim());
                    send(ServerUtil.getMoney(kw), host, port);
                } else {
                    send("Syntax error!", host, port);
                }
            }
        }
    }
}

```

```

    }
}
} catch (IOException | NumberFormatException e) {
    e.printStackTrace(System.err);
} finally {
    socket.close();
}
}

private void send(String s, InetAddress host, int port) throws IOException {
    byte[] buffer = s.getBytes();
    DatagramPacket packet = new DatagramPacket(buffer, buffer.length, host, port);
    socket.send(packet);
}

private DatagramPacket receive() throws IOException {
    byte[] buffer = new byte[65507];
    DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
    socket.receive(packet);
    return packet;
}

public static void main(String[] args) {
    new UDPServer().action();
}
}

```

3. Broadcast và Multicast

Có hai kiểu dịch vụ one-to-many (một gửi cho nhiều): broadcast và multicast. Với broadcast, mọi host trên mạng (LAN) sẽ nhận một bản sao của thông điệp. Với multicast, thông điệp được gửi đến một địa chỉ multicast, và mạng (WAN) sẽ phân phối nó chỉ cho những host (trên WAN) đăng ký nhận các thông điệp được chuyển đến địa chỉ này. TCP socket là unicast, chỉ có UDP socket cho phép broadcast hoặc multicast.

a) Broadcast

Broadcast các UDP datagram tương tự như unicast các datagram đó, ngoại trừ địa chỉ đích là một *địa chỉ broadcast quy ước* thay vì một địa chỉ (unicast) thông thường. IPv6 không cung cấp địa chỉ broadcast, thay vào đó là địa chỉ link-local multicast (FF02::1). Địa chỉ IPv4 broadcast cục bộ (255.255.255.255) cho phép gửi thông điệp đến mọi host trên cùng một mạng LAN, thông điệp gửi đến địa chỉ này không bao giờ được router gateway chuyển tiếp ra ngoài mạng.

Trong Java, viết code cho unicast và broadcast là như nhau, ta chỉ cần dùng địa chỉ broadcast quy ước khi muốn broadcast thông điệp đến tất cả các host trên cùng mạng.

b) Multicast

Một địa chỉ IPv4 lớp D (224.0.0.0 đến 239.255.255.255) hoặc một địa chỉ IPv6 bắt đầu bằng FF (ngoại trừ một số trường hợp) được dùng làm địa chỉ multicast. Địa chỉ này thể hiện một nhóm multicast, thông điệp gửi đến địa chỉ này sẽ được chuyển đến tất cả thành viên tham gia (join) nhóm.

Trong Java, ứng dụng kết nối đến địa chỉ multicast dùng thực thể của lớp MulticastSocket, là dẫn xuất của lớp DatagramSocket. MulticastSocket sau khi khởi tạo, dùng kết nối (join) với một địa chỉ multicast (đại diện cho nhóm) bằng phương thức joinGroup. Sau đó dùng MulticastSocket như một DatagramSocket để gửi nhận các UDP datagram đến địa chỉ multicast của nhóm.

MulticastSocket: Khởi tạo

```

MulticastSocket()
MulticastSocket(int LocalPort)
MulticastSocket(SocketAddress bindaddr)

```

Các constructor này tạo một UDP socket có khả năng multicast. Nếu không chỉ định port cục bộ hoặc chỉ định bằng 0, socket sẽ kết nối với port cục bộ bất kỳ có sẵn. Nếu chỉ định địa chỉ, socket chỉ nhận với địa chỉ này.

MulticastSocket: Quản lý nhóm

```

void joinGroup(InetAddress groupAddress)
void joinGroup(SocketAddress mcastaddr, NetworkInterface netIf)
void leaveGroup(InetAddress groupAddress)
void leaveGroup(SocketAddress mcastaddr, NetworkInterface netIf)

```

Các phương thức joinGroup và leaveGroup của MulticastSocket dùng quản lý các thành viên của nhóm multicast. Kết nối đến một nhóm nghĩa là trở thành thành viên của nhóm multicast này.

MulticastSocket: Getter/Setter

```

int getTimeToLive()

```



```

void setTimeToLive(int ttl)
boolean getLoopbackMode()
void setLoopbackMode(boolean disable)
InetAddress getInterface()
NetworkInterface getNetworkInterface()
void setInterface(InetAddress inf)
void setNetworkInterface(NetworkInterface netIf)

```

Phương thức `getTimeToLive` và `setTimeToLive` là getter/setter trị TTL cho tất cả datagram gửi bởi socket này.

Một socket với chế độ loopback cho phép nhận các datagram do nó gửi đi. Chế độ này có thể được kiểm tra hoặc thiết lập bằng các phương thức `getLoopbackMode` và `setLoopbackMode`.

Bốn phương thức cuối dùng với host có nhiều interface, để get/set các interface đi ra ngoài khi gửi các multicast packet.

c) Chương trình minh họa

Chương trình phải chạy trên WAN (máy có kết nối Internet). Kết nối với nhóm: `java MulticastChat 224.0.0.1:6868 Funny`

```

import java.awt.BorderLayout;
import java.awt.event.*;
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;
import javax.swing.*;

public class MulticastChat extends WindowAdapter implements Runnable, ActionListener {
    protected InetAddress group;
    protected int port;
    protected String nick;

    protected JFrame frame;
    protected JTextArea output;
    protected JScrollPane scroll;
    protected JTextField input;
    protected Thread listener;
    protected MulticastSocket socket;
    protected DatagramPacket outgoing, incoming;

    public MulticastChat(InetAddress group, int port, String nick) throws IOException {
        this.group = group;
        this.port = port;
        this.nick = nick;
        init();
    }

    protected void initNetwork() throws IOException {
        socket = new MulticastSocket(port);
        socket.setTimeToLive(1);
        socket.joinGroup(group);
        outgoing = new DatagramPacket(new byte[1], 1, group, port);
        incoming = new DatagramPacket(new byte[65507], 65507);
    }

    protected final void init() {
        frame = new JFrame("MulticastChat [" + group.getHostAddress() + ":" + port + "] - " + nick);
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        frame.setSize(360, 250);
        frame.setResizable(false);
        frame.setLayout(new BorderLayout());
        frame.add(scroll, "Center");
        frame.add(input, "South");
        frame.addWindowListener(this);

        output = new JTextArea();
        output.setLineWrap(true);
        output.setWrapStyleWord(true);
        output.setEditable(false);
        scroll = new JScrollPane(output);

        input = new JTextField();
        input.addActionListener(this);
    }
}

```

```

public synchronized final void start() throws IOException {
    if (listener == null) {
        initNetwork();
        listener = new Thread(this);
        listener.start();
        frame.setVisible(true);
    }
}

public synchronized void stop() throws IOException {
    frame.setVisible(false);
    if (listener != null) {
        listener.interrupt();
        listener = null;
        try {
            socket.leaveGroup(group);
        } finally {
            socket.close();
        }
    }
}

// WindowAdapter
@Override public void windowOpened(WindowEvent e) {
    super.windowOpened(e);
    input.requestFocus();
}

// ActionListener
@Override public void actionPerformed(ActionEvent event) {
    try {
        byte[] utf = (nick + ": " + event.getActionCommand()).getBytes("UTF8");
        outgoing.setData(utf);
        outgoing.setLength(utf.length);
        socket.send(outgoing);
        input.setText("");
    } catch (IOException e) {
        handleIOException(e);
    }
}

protected synchronized void handleIOException(IOException e) {
    if (listener != null) {
        output.append(e + "\n");
        input.setVisible(false);
        frame.validate();
        if (listener != Thread.currentThread())
            listener.interrupt();
        listener = null;
        try {
            socket.leaveGroup(group);
        } catch (IOException ignored) {
            ignored.printStackTrace(System.err);
        }
        socket.close();
    }
}

// Runnable
@Override public void run () {
    try {
        while (!Thread.interrupted()) {
            incoming.setLength(incoming.getData().length);
            socket.receive(incoming);
            String message = new String(incoming.getData(), 0, incoming.getLength (), "UTF8");
            output.append(message + "\n");
        }
    } catch (IOException e) {
        handleIOException(e);
    }
}

```

```

    }

    public static void main(String[] args) throws IOException {
        if ((args.length != 2) || (!args[0].contains(":")))
            throw new IllegalArgumentException("Syntax: MulticastChat <group>:<port> <nick>");
        int index = args[0].indexOf(":");
        InetAddress group = InetAddress.getByName(args[0].substring(0, index));
        int port = Integer.parseInt(args[0].substring(index + 1));
        new MulticastChat(group, port, args[1]).start();
    }
}

```

RMI

RMI (Remote Method Invocation) cung cấp khả năng triệu gọi các phương thức của một đối tượng từ xa (remote object) thông qua giao thức JRMP. Bằng cách này, chúng ta có thể liên lạc với đối tượng nghiệp vụ từ xa, gọi các phương thức của chúng xuyên qua nhiều hệ thống trên mạng.

Trước đây, việc cài đặt RMI khá phức tạp, cần phải xử lý một loạt khái niệm: RMI Registry, export đối tượng, Stub và Skeleton, ... Hiện nay RMI được cài đặt đơn giản hơn rất nhiều, phần này trình bày các bước cài đặt chủ yếu của RMI.

1. Đối tượng triệu gọi từ xa

Đối tượng triệu gọi từ xa, ta gọi tắt là *đối tượng remote*, chính là đối tượng nghiệp vụ phân tán phía RMIServer.

Cài đặt đối tượng remote như sau:

- Khai báo interface remote.
- Định nghĩa lớp thực thi của đối tượng remote, cài đặt interface remote.

Interface remote bộc lộ các phương thức sẽ được triệu gọi của đối tượng remote. Interface này sẽ được chuyển cho phía client để triệu gọi. Vì vậy, nếu bạn đã có sẵn đối tượng nghiệp vụ, để chuyển nó thành đối tượng triệu gọi từ xa, đơn giản tạo interface remote chứa các phương thức cần triệu gọi rồi điều chỉnh cho đối tượng nghiệp vụ có sẵn, cài đặt interface remote vừa tạo.

a) Khai báo interface remote

Interface remote có nhiệm vụ *bộc lộ các phương thức nghiệp vụ của đối tượng remote* để RMIClient có thể tham chiếu và triệu gọi từ xa. Các yêu cầu của interface remote:

- Phải được khai báo public để RMIClient có thể "nhìn thấy" các phương thức nghiệp vụ khai báo trong interface, vì RMIClient thường không cùng gói với interface remote.
- Thừa kế interface java.rmi.Remote.
- Các phương thức triệu gọi từ xa phải ném ra exception java.rmi.RemoteException. Exception này được ném khi gặp các lỗi liên quan đến kết nối mạng hoặc lỗi server.

```

public interface ServerUtilInterface extends Remote {
    String getMoney(int kw) throws RemoteException;
}

```

b) Định nghĩa lớp thực thi của đối tượng remote

Lớp thực thi của đối tượng remote sẽ cài đặt interface remote vừa được khai báo ở trên.

Trong lớp thực thi của đối tượng remote, ta *cài đặt cụ thể các phương thức nghiệp vụ* đã được khai báo trong interface remote, nghĩa là các phương thức mà đối tượng remote phải thực hiện.

```

class ServerUtil implements ServerUtilInterface {
    @Override public String getMoney(int kw) {
        int money = kw * 500;
        if (kw > 100) money += (kw - 100) * 200;
        if (kw > 200) money += (kw - 200) * 300;
        return String.format("Spending: %d", money);
    }
}

```

Mặc định trong RMI, các đối tượng cục bộ được truyền bằng trị (ngoại trừ các đối tượng static hoặc transient). Tất cả dữ liệu *kiểu đối tượng* là tham số hoặc trị trả về của phương thức triệu gọi từ xa, nếu muốn truyền qua mạng phải cài đặt một trong hai interface java.rmi.Remote (truyền bằng tham chiếu) hoặc java.io.Serializable (truyền bằng trị).

Như vậy, nếu bạn truyền một POJO qua mạng, phải cài đặt interface Serializable cho POJO đó.

Quá trình đóng gói các đối số của phương thức triệu gọi từ xa trên RMIClient để chuyển đi được lớp Stub thực hiện. Quy trình này trong RMI gọi là marshaling data (sắp xếp tuần tự dữ liệu).

2. RMIServer

Sau khi khai báo interface remote và cài đặt cho đối tượng nghiệp vụ triệu gọi từ xa, RMIServer thực hiện các công việc sau:

a) Khởi động dịch vụ naming

Đối tượng remote có bản chất phân tán, nghĩa là vị trí của chúng phải trong suốt đối với client. RMIClient phải định vị được đối tượng remote để có thể triệu gọi từ xa các phương thức của nó. Điều này được thực hiện dưới sự hỗ trợ của dịch vụ Remote Object Registry (RMI Registry). RMI Registry là một dịch vụ đặt tên (naming service), cho phép đăng ký (bind, rebind) đối tượng remote dưới một tên, sau đó RMIClient có thể truy tìm (lookup) tham chiếu đến đối tượng remote đó thông qua tên đã đăng ký. Các tác vụ liên quan đến dịch vụ naming (bind, rebind, unbind, lookup) thực hiện thông qua JNDI, dùng java.rmi.Naming hoặc java.rmi.Registry.

Bạn có thể khởi động RMI Registry của Java trước khi chạy RMIServer bằng dòng lệnh, với cổng mặc định 1099:

```
start rmiregistry
```

Hiện nay, ta đơn giản khởi động dịch vụ naming từ constructor của RMIServer:

```
LocateRegistry.createRegistry(1099);
Registry registry = LocateRegistry.getRegistry(1099);
```

b) Tạo, kết xuất (export) và đăng ký đối tượng remote

RMIServer quản lý các đối tượng remote. RMIServer sẽ:

- Tạo các thể hiện (instantiate) cho lớp thực thi đối tượng remote.
- Kết xuất (export) các đối tượng remote.
- Đăng ký các đối tượng remote với dịch vụ naming.

Khi đó đối tượng remote mới sẵn sàng đáp ứng cho việc triệu gọi từ xa của RMIClient.

Trước đây, việc cài đặt khá phức tạp:

- Lớp thực thi của đối tượng remote hoặc phải thừa kế lớp `java.rmi.server.UnicastRemoteObject`, hoặc tự kết xuất trong constructor bằng cách gọi phương thức static `exportObject` của lớp `UnicastRemoteObject` với tham số là chính nó (this).

Điều này làm cho việc tạo đối tượng remote trở nên phức tạp.

- RMIServer tạo đối tượng remote và đăng ký đối tượng remote với RMI registry.
- Cần phải tạo trước lớp Stub khi thực thi bằng công cụ dòng lệnh `rmic`. Phần lớn nguyên nhân thất bại khi cài đặt RMI trước đây là do không tìm thấy lớp Stub do đặt sai vị trí.

Hiện nay, cách cài đặt đơn giản hơn rất nhiều:

- Lớp thực thi của đối tượng remote (`ServerUtil`) chỉ cài đặt interface remote (`ServerUtilInterface`) với các phương thức nghiệp vụ đã được cài đặt cụ thể.
- KHÔNG cần tự kết xuất.
- KHÔNG cần tạo lớp Stub bằng `rmic`. Điều này giúp tránh rất nhiều phức tạp khi phải tạo Stub.

RMIServer sẽ chịu trách nhiệm tạo đối tượng remote, *kết xuất đối tượng remote* thành stub và *đăng ký stub* với RMI Registry.

```
ServerUtil remoteObj = new ServerUtil();
ServerUtilInterface stub = (ServerUtilInterface)UnicastRemoteObject.exportObject(remoteObj, 0);
registry.rebind("rmi://127.0.0.1/kw_service", stub);
```

3. RMIClient

Hai bước quan trọng khi viết RMIClient là:

- Nhận tham chiếu của đối tượng remote bằng cách "lookup" thông qua dịch vụ naming.
- Triệu gọi các phương thức từ xa từ tham chiếu nhận được.

a) Nhận tham chiếu của đối tượng remote

RMIClient sẽ lấy tham chiếu đến đối tượng remote từ dịch vụ naming. Phương thức `lookup()` của lớp `Naming` hoặc lớp `Registry` nhận đối số là tên đăng ký của đối tượng remote và trả về đối tượng được đăng ký dưới tên đó.

```
Registry registry = LocateRegistry.getRegistry(1099);
ServerUtilInterface remoteObject = (ServerUtilInterface)registry.lookup("rmi://127.0.0.1/kw_service");
```

Chú ý, `remoteObject` có kiểu là interface remote.

b) Triệu gọi các phương thức từ xa

RMIClient triệu gọi phương thức từ xa của đối tượng remote giống như triệu gọi phương thức của một đối tượng bình thường. Ngoại trừ chúng có khả năng ném ra exception `RemoteException` nếu triệu gọi không thành công.

Việc triệu gọi từ xa được thực hiện thông qua interface remote, còn thực thi lời triệu gọi được thực hiện ngầm bên dưới do lớp Stub đảm nhiệm.

```
int kw = 123;
System.out.println(remoteObject.getMoney(kw));
```

Kết nối cơ sở dữ liệu

JDBC

1. SQL

Dữ liệu trong cơ sở dữ liệu quan hệ được lưu trữ trên nhiều bảng. Mỗi bảng bao gồm các hàng và cột, thường mô tả một thực thể hoặc mối quan hệ giữa hai thực thể. Mỗi hàng trong bảng là một bản ghi được xác định duy nhất với một định danh gọi là khóa chính (primary key).

SQL (Structured Query Language) là một ngôn ngữ khai báo trừu tượng, dùng để tạo các truy vấn và thực hiện thao tác dữ liệu lên một cơ sở dữ liệu quan hệ. SQL là một chuẩn, không gắn liền với một cơ sở dữ liệu cài đặt cụ thể. Hầu hết các hệ quản trị cơ sở dữ liệu hiện nay hỗ trợ SQL.

SQL bao gồm:

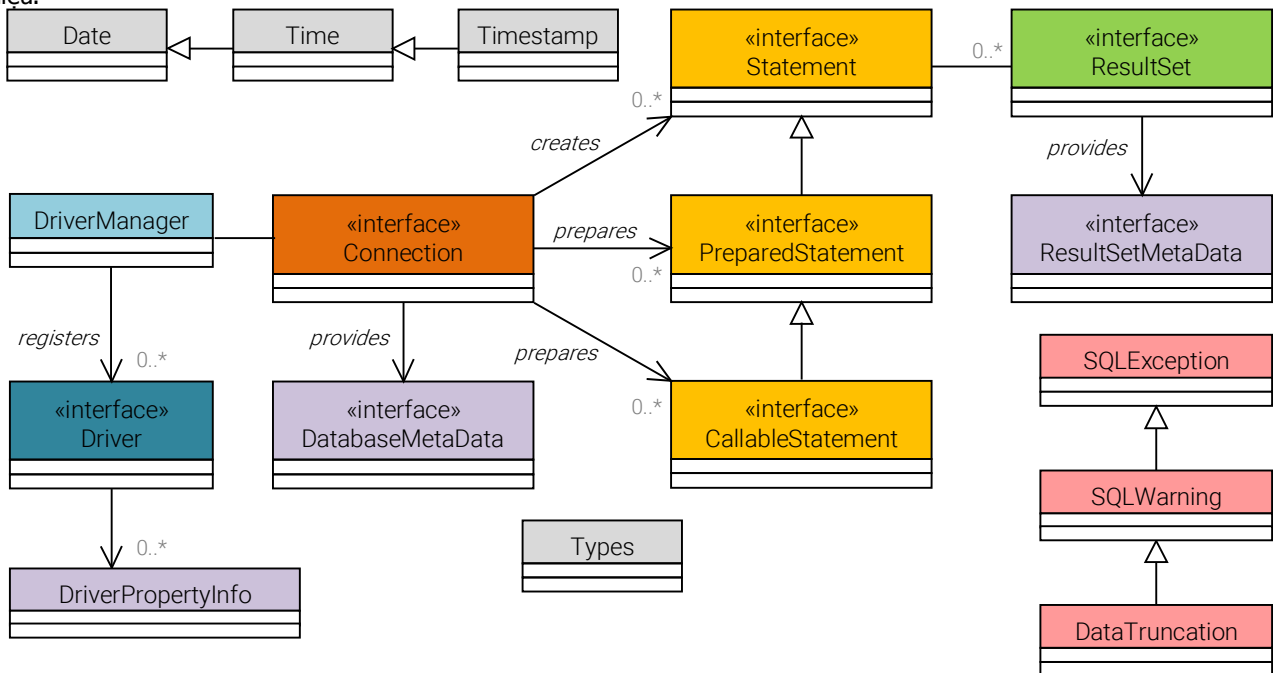
- Ngôn ngữ thao tác dữ liệu (DML – Data Manipulation Language) sử dụng các từ khóa SELECT, INSERT, UPDATE, DELETE để truy vấn và thao tác lên dữ liệu trong bảng.
- Ngôn ngữ định nghĩa dữ liệu (DDL – Data Definition Language) được dùng để tạo và thao tác lên cấu trúc của bảng: CREATE TABLE, ALTER TABLE, ADD COLUMN, DROP COLUMN, ADD FOREIGN KEY, ...

Stored procedure được định nghĩa như một tập các phát biểu SQL được lưu trữ ngay trong cơ sở dữ liệu và sau đó, được triệu gọi bởi một chương trình, một trigger hoặc từ một stored procedure khác.

Bạn có thể tham khảo SQL và stored procedure từ các tài liệu khác.

2. JDBC

JDBC (Java DataBase Connectivity) API cung cấp giao diện chuẩn, độc lập với cơ sở dữ liệu, để tương tác với nguồn dữ liệu dạng bảng. Trong hầu hết các ứng dụng dùng JDBC, bạn làm việc với hệ quản trị cơ sở dữ liệu quan hệ (RDBMs), tuy nhiên JDBC cho phép làm việc với bất kỳ nguồn dữ liệu dạng bảng nào, như bảng tính Excel, các tập tin, v.v... Thường bạn dùng JDBC API kết nối với cơ sở dữ liệu, truy vấn và cập nhật dữ liệu bằng SQL. JDBC API cũng cho phép thực thi stored procedure SQL trong cơ sở dữ liệu.



a) JDBC driver

Do có nhiều DBMS khác nhau (Oracle database, Microsoft SQL Server, Sybase database, DB2, MySQL, Java DB, v.v...), để độc lập với DBMS, hầu hết JDBC API được định nghĩa bằng cách dùng interface, các nhà cung cấp DBMS (hoặc bên thứ ba) cung cấp các cài đặt cho các interface này. Tập hợp các lớp cài đặt được cung cấp bởi nhà cung cấp dùng tương tác với một cơ sở dữ liệu cụ thể được gọi là *JDBC driver*. Có nhiều loại JDBC driver cho các cơ sở dữ liệu khác nhau (hoặc cho cùng một cơ sở dữ liệu).

Bạn có thể dùng một trong bốn kiểu JDBC driver trong chương trình Java để kết nối đến RDBMs:

- JDBC-ODBC bridge driver. JDBC driver sẽ tạo "cầu nối" với ODBC driver (ODBC – Open Database Connectivity) là giao diện lập trình ứng dụng mở dùng truy cập cơ sở dữ liệu, tạo bởi SQL Access Group, đứng đầu là Microsoft. ODBC cho phép truy cập đến một số cơ sở dữ liệu như Access, dBase, DB2, Excel, và text. ODBC xử lý truy vấn SQL và chuyển nó thành yêu cầu đến loại cơ sở dữ liệu cụ thể. Tuy nhiên, từ Java 8, cầu nối JDBC-ODBC đã được loại bỏ.
- JDBC native API Driver. JDBC driver dùng thư viện gốc (native library) của DBMS để thực hiện các truy cập cơ sở dữ liệu. Nó sẽ dịch lời gọi JDBC thành lời gọi DBMS, và thư viện gốc sẽ kết nối với cơ sở dữ liệu. Bạn phải cài đặt phần mềm riêng của DBMS.
- JDBC-network protocol driver. JDBC driver được viết bằng Java, driver dịch lời gọi JDBC thành một giao thức mạng và truyền lời gọi đến server. Server dịch lời gọi của giao thức mạng thành các tác vụ JDBC do server cung cấp.
- JDBC native protocol driver. Driver cài đặt hoàn toàn bằng Java, do nhà cung cấp cơ sở dữ liệu cung cấp, cho phép kết nối trực tiếp và hiệu suất cao đến cơ sở dữ liệu.

Khi sử dụng JDBC để kết nối với cơ sở dữ liệu, bạn phải chắc chắn rằng gói cài đặt JDBC của nhà cung cấp có trên classpath. Với JDBC 3.0, driver cho JDBC của nhà cung cấp cơ sở dữ liệu được nạp bởi cơ chế classloader, bằng cách gọi lệnh `Class.forName("<vendor_class>")`. Từ JDBC 4.0, điều này hiện đã được thực hiện tự động với khả năng dò tìm driver (driver discovery) từ thư mục META-INF/services của gói JAR chứa JDBC driver.

JVM nạp lớp DriverManager của JDBC API. Lớp DriverManager nạp tất cả các thực thể nó tìm thấy trong tập tin META-INF/services/java.sql.Driver của các tập tin JAR/ZIP trên classpath. Các lớp driver gọi DriverManager.register(this) để tự đăng ký nó với DriverManager.

b) Connection

RDBM Server là một server có tính năng mạng, kết nối đến chúng nghĩa là tạo một kết nối TCP đến địa chỉ và port chỉ định. Để có được kết nối, bạn cần phải cung cấp JDBC với các thông số kết nối, trong hình thức một URL cho phương thức static getConnection() của DriverManager.

Khi phương thức DriverManager.getConnection() được triệu gọi, DriverManager triệu gọi phương thức connect() của từng thực thể Driver đăng ký với chuỗi URL. Driver đầu tiên tạo kết nối thành công sẽ trả về thực thể Connection.

```
Connection connection = DriverManager
    .getConnection("jdbc:mysql://localhost:3306/company_db", "username", "password");
```

Trong URL, giao thức kết nối phụ (sau jdbc:) khác nhau với các RDBMs khác nhau, tham chiếu tài liệu của driver cụ thể:

MySQL	jdbc:mysql://hostname:3306/databaseName
Oracle	jdbc:oracle:thin:@hostname:1521:databaseName
DB2	jdbc:db2:hostname:portNumber/databaseName
PostgreSQL	jdbc:postgresql://hostname:5432/databaseName
Java DB/Apache	jdbc:derby:databaseName(embedded)
Derby	jdbc:derby://hostname:1527/databaseName(network)
Microsoft SQL Server	jdbc:sqlserver://hostname:1433;databaseName=databaseName
Sybase	jdbc:sybase:Tds:hostname:6262/databaseName

Các số hiệu port được định là số hiệu port mặc định của RDBM Server.

Một khi kết nối đến cơ sở dữ liệu hình thành, bạn có thể thực hiện các tác vụ SQL để truy vấn hoặc thao tác lên dữ liệu.

Bạn có thể kiểm tra khả năng của cơ sở dữ liệu bên dưới bằng cách lấy DatabaseMetaData từ đối tượng Connection.

```
DatabaseMetaData metaData = connection.getMetaData();
System.out.println("Database is: " + metaData.getDatabaseProductName() + " "
    + metaData.getDatabaseProductVersion());
System.out.println("Driver is: " + metaData.getDriverName() + metaData.getDriverVersion());
System.out.println("The URL for this connection is: " + metaData.getURL());
System.out.println("User name is: " + metaData.getUserName());
System.out.println("Maximum no. of rows you can insert is: " + metaData.getMaxRowSize());
```

c) Statement

Statement

Từ thực thể Connection, bạn tạo một thực thể cài đặt interface Statement, được dùng để gửi truy vấn SQL đến cơ sở dữ liệu. Tùy theo truy vấn SQL, việc thực thi Statement có thể trả về một tập kết quả (ResultSet) hoặc trả về một số int cho biết số dòng trong bảng cơ sở dữ liệu bị ảnh hưởng bởi truy vấn.

```
Connection connection = DriverManager.getConnection(url, user, pwd);
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery("SELECT * FROM Customer");
```

Statement cung cấp vài phương thức khác nhau để thực hiện truy vấn SQL:

executeQuery() được dùng khi biết truy vấn loại query, như SELECT, sẽ trả về một tập kết quả (ResultSet).

executeUpdate() được dùng với các truy vấn loại non-query, như INSERT, UPDATE, DELETE và truy vấn DDL. Trả về số dòng bị ảnh hưởng bởi truy vấn.

execute() dùng để xác định loại truy vấn. Nếu trả về true, câu truy vấn sẽ trả về một ResultSet. Chú ý là trước khi trả về boolean được trả về, truy vấn *đã được thực hiện*. Thông thường, phương thức này dùng để gọi một stored procedure.

```
ResultSet resultSet;
int numRows;
boolean status = statement.execute(""); // true nếu trả về ResultSet
if (status) {
    resultSet = statement.getResultSet(); // lấy ResultSet
    // xử lý ResultSet ...
} else {
    numRows = statement.getUpdateCount(); // lấy số dòng bị ảnh hưởng
    if (numRows == -1) { // nếu trả về -1, không có kết quả
        System.out.println("No results");
    } else {
        System.out.println(numRows + " rows affected.");
    }
}
```

Đối tượng Statement có thể dùng lại được, nếu bạn thực hiện một chuỗi các truy vấn, bạn chỉ cần một thực thể Statement.

PreparedStatement

Truy vấn SQL có thể nhận các tham số, việc chèn trực tiếp tham số vào câu truy vấn sẽ phức tạp do phân biệt tham số dạng chuỗi và dạng số, đồng thời dễ dẫn đến các lỗi SQL injection. PreparedStatement, dẫn xuất từ Statement, dùng ký tự "giữ chỗ" dấu (?) để nhận các tham số, gọi là phát biểu SQL *biên dịch trước*. Các tham số sau đó được thiết lập bằng cách gọi các phương thức setXxx(index, value); index là số thứ tự của ký tự (?) tính từ 1, value là trị của tham số có kiểu dữ liệu xxx.

```
PreparedStatement statement = connection.prepareStatement("SELECT AVG(GPA) FROM Student WHERE age >= ? AND age < ?");
```



```
// thiết lập các tham số thực:
statement.setInt(1, age);
statement.setInt(2, age + 10);
ResultSet resultSet = statement.executeQuery();
```

Phương thức `setNull()` đặt tham số tương ứng thành null và phương thức `clearParameters()` xóa trị của tất cả tham số. Do `PreparedStatement` có thể dùng nhiều lần, JDBC 2.1 cung cấp khả năng tạo và thực hiện các phát biểu SQL theo lô (batch SQL statements), cho phép bạn thực hiện hàng loạt các truy vấn SQL tương tự nhau nhưng có tham số thực khác nhau:

```
PreparedStatement statement = connection.prepareStatement("INSERT INTO OrderItems VALUES(?, ?, ?, ?)");
for (Product product : productList) {
    // thiết lập các tham số thực cho một truy vấn trong lô:
    statement.setInt(1, orderId);
    statement.setInt(2, product.getId());
    statement.setInt(3, product.getQuantity());
    statement.setBigDecimal(4, product.getPrice());
    statement.addBatch();
}
statement.executeBatch();
```

CallableStatement

Bạn có thể gọi stored procedure lưu trong cơ sở dữ liệu, có tham số hoặc không, bằng cách dùng `CallableStatement`.

```
CallableStatement statement = connection.prepareCall("{CALL findByPrice (?, ?)}");
// thiết lập các tham số thực cho stored procedure:
```

```
statement.setInt(1, age);
statement.setInt(2, age + 10);
ResultSet resultSet = statement.executeQuery();
```

Có ba loại tham số:

- IN, tham số có giá trị chưa biết khi tạo truy vấn SQL, sau đó bạn gán trị cho tham số IN với phương thức `setXxx()`. Dùng như tham số đầu vào của stored procedure. Các đối tượng `PreparedStatement` chỉ nhận tham số IN.
- OUT, tham số có giá trị được cung cấp khi truy vấn SQL trả về, sau đó bạn lấy trị từ tham số OUT với phương thức `getXxx()`. Dùng như biến lưu trữ trị trả về cho stored procedure.
- INOUT, tham số cung cấp cả hai trị nhập và xuất, bạn gán trị cho tham số với phương thức `setXxx()` và lấy trị từ tham số với phương thức `getXxx()`.

d) ResultSet

Khi truy vấn trả về một tập kết quả, một thực thể cài đặt interface `ResultSet` được trả về. Dùng các phương thức định nghĩa trong interface `ResultSet`, bạn có thể đọc và thao tác lên dữ liệu kết quả. `ResultSet` là một bản sao của dữ liệu từ cơ sở dữ liệu khi thực hiện truy vấn.

Di chuyển một chiều trong ResultSet

Đối tượng `ResultSet` duy trì một con trỏ chỉ đến dòng hiện hành trong kết quả. Khi truy vấn trả về `ResultSet`, con trỏ nằm ngay *trước dòng đầu tiên* của kết quả, dùng `next()` để di chuyển con trỏ tới từng dòng và kiểm tra có dữ liệu tại con trỏ.

Tại dòng kết quả, để lấy dữ liệu của các cột theo thứ tự bất kỳ, bạn có thể:

- Lấy dữ liệu các cột theo tên, dùng các phương thức `getXxx()` với tham số là tên cột của `ResultSet` và `xxx` là kiểu dữ liệu Java tương ứng với kiểu dữ liệu SQL của cột.

- Lấy dữ liệu các cột theo chỉ số, `ResultSet` cũng cung cấp các phương thức nạp chồng `getXxx()`, cho phép lấy dữ liệu từ các cột, với tham số là chỉ số (tính từ 1) của cột trong tập kết quả và `xxx` là kiểu dữ liệu Java tương ứng với kiểu dữ liệu SQL của cột. Bạn có thể lấy số cột từ đối tượng `ResultSetMetaData`.

Mỗi kiểu dữ liệu SQL có một phương thức getter tương ứng cho `ResultSet`. Chuyển đổi kiểu dữ liệu từ SQL sang Java thuộc về ba loại:

- Tương đương trực tiếp, ví dụ SQL `INTEGER` sang kiểu `int` của Java.
- Tương đương, ví dụ SQL `CHAR`, SQL `VARCHAR`, SQL `LONGVARCHAR` sang kiểu `String` của Java.
- Chuyển sang kiểu Java đặc biệt, ví dụ SQL `DATE` có thể đọc vào đối tượng `java.sql.Date`, dẫn xuất từ `java.util.Date`, phương thức `toString()` của nó trả về chuỗi có định dạng "yyyy mm dd".

```
ArrayList<Book> books = new ArrayList<>();
String query = "SELECT Title, PubDate, UnitPrice from Book";
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery(query);
while (resultSet.next()) {
    String title = resultSet.getString("Title");
    java.util.Date pubDate = new java.util.Date(resultSet.getDate("PubDate").getTime());
    float price = resultSet.getFloat("Price");
    books.add(new Book(title, pubDate, price));
}
```

Lấy thông tin về ResultSet

Khi bạn cho phép người dùng tùy biến truy vấn, có thể không biết số cột, kiểu và tên các cột trả về. Các phương thức của đối tượng thuộc lớp `ResultSetMetaData` sẽ cung cấp các thông tin này.

```
String query = "SELECT AuthorID FROM Author";
Statement statement = connection.createStatement();
```

```

ResultSet resultSet = statement.executeQuery(query);
ResultSetMetaData rsMetaData = resultSet.getMetaData();
resultSet.next();
int colCount = rsMetaData.getColumnCount(); // số cột trong ResultSet
System.out.println("Column Count: " + colCount);
for (int i = 1; i <= colCount; ++i) {
    System.out.println("Table Name : " + rsMetaData.getTableName(i));
    System.out.println("Column Name : " + rsMetaData.getColumnName(i));
    System.out.println("Column Size : " + rsMetaData.getColumnDisplaySize(i));
}

```

Di chuyển tới lui trong ResultSet

Mặc định với Statement, con trỏ chỉ di chuyển tới (forward) và không hỗ trợ thay đổi ResultSet. Interface ResultSet định nghĩa các đặc tính này trong các hằng: TYPE_FORWARD_ONLY và CONCUR_READ_ONLY. Tuy nhiên, JDBC cho phép di chuyển con trỏ tới, lui, nhảy đến vị trí chỉ định trong tập kết quả. JDBC cũng cung cấp khả năng thay đổi ResultSet trong lúc mở và những thay đổi thay đổi này được ghi xuống cơ sở dữ liệu.

Từ JDBC 2.1, JDBC hỗ trợ một trong các kiểu con trỏ sau:

- TYPE_FORWARD_ONLY, mặc định cho ResultSet, con trỏ chỉ di chuyển tới trong tập kết quả.
- TYPE_SCROLL_INSENSITIVE, cho phép con trỏ di chuyển tới, lui, nhảy đến vị trí chỉ định trong tập kết quả. Những thay đổi của cơ sở dữ liệu bên dưới *không* ảnh xạ đến tập kết quả.
- TYPE_SCROLL_SENSITIVE, cho phép con trỏ di chuyển tới, lui, nhảy đến vị trí chỉ định trong tập kết quả. Những thay đổi của cơ sở dữ liệu bên dưới sẽ ảnh xạ đến tập kết quả đang mở.

JDBC cũng cung cấp hai tùy chọn cho truy cập dữ liệu song hành:

- CONCUR_READ_ONLY, mặc định một tập kết quả đang mở là chỉ đọc (read-only) và không thể thay đổi.
- CONCUR_UPDATABLE, tập kết quả có thể thay đổi thông qua các phương thức của ResultSet.

Bạn kiểm tra cơ sở dữ liệu và JDBC driver có hỗ trợ các tùy chọn trên bằng DatabaseMetaData.

```

Connection connection = DriverManager.getConnection(...);
DatabaseMetaData dbMetaData = connection.getMetaData();
if (dbMetaData.supportsResultSetType(ResultSet.TYPE_FORWARD_ONLY)) {
    System.out.print("Supports TYPE_FORWARD_ONLY");
    if (dbMetaData.supportsResultSetConcurrency(ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE)) {
        out.println(" and supports CONCUR_UPDATABLE");
    }
}

if (dbMetaData.supportsResultSetType(ResultSet.TYPE_SCROLL_INSENSITIVE)) {
    System.out.print("Supports TYPE_SCROLL_INSENSITIVE");
    if (dbMetaData.supportsResultSetConcurrency(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE)) {
        System.out.println(" and supports CONCUR_UPDATABLE");
    }
}

if (dbMetaData.supportsResultSetType(ResultSet.TYPE_SCROLL_SENSITIVE)) {
    System.out.print("Supports TYPE_SCROLL_SENSITIVE");
    if (dbMetaData.supportsResultSetConcurrency(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE)) {
        System.out.println("Supports CONCUR_UPDATABLE");
    }
}

```

Để tạo ResultSet với TYPE_SCROLL_INSENSITIVE và CONCUR_UPDATABLE, Statement dùng để tạo ResultSet phải được khởi tạo từ Connection với kiểu con trỏ và chế độ truy cập bạn muốn.

```
Connection connection = DriverManager.getConnection(...);
```

```
Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
```

Sau đó bạn có thể dùng các phương thức định vị con trỏ để di chuyển tới lui trong ResultSet:

resultSet.first()	resultSet.absolute(1)
	resultSet.absolute(2)
	resultSet.relative(-2)
resultSet.previous()	resultSet.relative(-1)
Con trỏ đang chỉ dòng hiện tại	
resultSet.next()	resultSet.relative(1)
	resultSet.relative(2)
	resultSet.absolute(-2)
resultSet.last()	resultSet.absolute(-1)

Cập nhật ResultSet

Bên cạnh việc trả về kết quả truy vấn, ResultSet có thể dùng để thay đổi nội dung của bảng cơ sở dữ liệu, như cập nhật các dòng, xóa dòng, chèn thêm dòng mới.

```
Connection connection = DriverManager.getConnection(...);
Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
// ...
// thử cập nhật
resultset.absolute(2); // chuyển đến dòng thứ 2 của ResultSet
resultset.updateFloat("balance", 42.55f);
resultset.updateRow();
resultset.cancelRowUpdates(); // bỏ qua các cập nhật trong ResultSet
// thử chèn
resultset.moveToInsertRow(); // chuyển đến dòng đặc biệt trong ResultSet dùng insert
resultset.updateInt("acctNum", 999999);
resultset.updateString("surname", "Harrison");
resultset.updateString("firstName", "Christine Dawn");
resultset.updateFloat("balance", 25.00f);
resultset.insertRow();
// thử xóa
resultset.moveToCurrentRow(); // chuyển đến dòng vừa chèn
resultset.deleteRow
```

e) Transaction

Transaction (giao tác) là một hay nhiều phát biểu SQL được nhóm thành *một đơn vị hoạt động duy nhất*. Nếu tất cả các phát biểu được thực hiện thành công, transaction sẽ *hoàn tất* (commit) và cơ sở dữ liệu thực sự thay đổi, cập nhật. Nếu chỉ một phát biểu thất bại, transaction sẽ *quay lại* (rollback), cơ sở dữ liệu sẽ quay trở về trạng thái trước khi thực hiện transaction.

Trong SQL, transaction được thực hiện bằng phát biểu COMMIT và ROLLBACK. Trong JDBC, chúng là các phương thức commit() và rollback() của interface Connection.

Mặc định, JDBC thiết lập transaction *cho từng phát biểu SQL*, gọi là chế độ autocommit, bạn cần bật hoạt chế độ này để nhóm các phát biểu SQL vào một transaction:

```
connection.setAutoCommit(false);
try {
    // ba phát biểu được nhóm thành transaction.
    statement.executeUpdate(update1);
    statement.executeUpdate(update2);
    statement.executeUpdate(update3);
    connection.commit();
} catch (SQLException ex) {
    connection.rollback();
    System.err.println("SQL error! Changes aborted...");
}
```

Ngoài ra, bạn có thể thiết lập mức độ cô lập (isolation level) của cơ sở dữ liệu khi thực hiện transaction. Mức độ mặc định, TRANSACTION_SERIALIZABLE bảo đảm thực hiện các phát biểu SQL tuần tự, nhưng làm giảm hiệu suất truy cập cơ sở dữ liệu.

```
connection.setTransactionIsolationLevel(TRANSACTION_REPEATABLE_READ);
```

Các mức độ cho phép: TRANSACTION_SERIALIZABLE, TRANSACTION_REPEATABLE_READ, TRANSACTION_READ_COMMITTED và TRANSACTION_READ_UNCOMMITTED.

Bạn cũng có thể thiết lập transaction là đọc ghi (mặc định, false) hoặc chỉ đọc (true):

```
connection.setReadOnly(true);
```

f) Xử lý lỗi

Các phương thức của JDBC API ném ra SQLException khi có lỗi truy cập cơ sở dữ liệu. SQLException chứa:

- Mô tả về lỗi, gọi phương thức getMessage().
- Mã lỗi dạng chuỗi, từ nhà cung cấp, gọi phương thức getSQLState().
- Mã lỗi dạng integer, từ nhà cung cấp, gọi phương thức getErrorCode().
- Chuỗi các exception, cho phép truy cập SQLException kế tiếp, gọi phương thức getNextException().

```
Logger logger = Logger.getLogger("com.example.MyClass");
try {
    // JDBC code
    // ...
} catch (SQLException se) {
    while (se != null) {
        logger.log(Level.SEVERE, "----- SQLException -----");
        logger.log(Level.SEVERE, "SQLState: " + se.getSQLState());
        logger.log(Level.SEVERE, "Vendor Error code: " + se.getErrorCode());
        logger.log(Level.SEVERE, "Message: " + se.getMessage());
        se = se.getNextException();
    }
}
```

JDBC cũng cung cấp `SQLWarning`, dẫn xuất từ `SQLException`. Các đối tượng `SQLWarning` có thể nhận từ `Connection`, `Statement`, `ResultSet`, bằng cách gọi phương thức `getWarnings()`, gọi `getNextWarning()` trên đối tượng `warning` để lấy đối tượng `warning` kế tiếp. Chúng được xem như "báo cáo" im lặng của các đối tượng JDBC.

```
ResultSet resultset = statement.executeQuery(someQuery);
SQLWarning warning = statement.getWarnings();
while (warning != null) {
    System.err.println("Message      : " + warning.getMessage());
    System.err.println("SQLState     : " + warning.getSQLState());
    System.err.println("Vendor Error: " + warning.getErrorCode());
    warning = warning.getNextWarning();
}
while (resultset.next()) {
    int value = resultset.getInt(1);
    SQLWarning warning = resultset.getWarnings();
    while (warning != null) {
        System.err.println("Message:      " + warning.getMessage());
        System.err.println("SQLState:     " + warning.getSQLState());
        System.err.println("Vendor Error: " + warning.getErrorCode());
        warning = warning.getNextWarning();
    }
}
```

Java DB

Apache Derby là một hệ quản trị cơ sở dữ liệu quan hệ mã nguồn mở, có nguồn gốc từ Cloudscape. Oracle đóng gói Apache Derby trong Java (từ Java 7) và gọi nó là Java DB. Do Java DB là một phần của Java, bạn có thể phát triển các ứng dụng Java sử dụng Java DB mà không cần phải cài đặt hệ cơ sở dữ liệu quan hệ của bên thứ ba. Java DB là lý tưởng cho ứng dụng dùng cơ sở dữ liệu nhỏ.

Để sử dụng các tiện ích dòng lệnh của Java DB, bạn cần thêm vào biến môi trường Path chuỗi `%JAVA_HOME%\db\bin`.

Chọn một thư mục, nơi bạn sẽ lưu trữ cơ sở dữ liệu, ví dụ: `C:\derby`, chạy console rồi dùng tiện ích `ij`.

```
C:\derby>ij
ij version 10.10
ij>
```

1. Tạo và kết nối đến Java DB

Lệnh sau sẽ tạo cơ sở dữ liệu tên `javadb` với cặp `username:password` là `root:toor`, rồi kết nối đến nó.

```
ij> connect 'jdbc:derby:javadb;create=true;user=root;password=toor';
```

Chú ý, mỗi lệnh của `ij` phải kết thúc bằng dấu `;`

Để kết nối đến cơ sở dữ liệu có sẵn, dùng lệnh:

```
ij> connect 'jdbc:derby:javadb;user=root;password=toor';
```

Sau khi kết nối đến cơ sở dữ liệu, bạn có thể dùng các phát biểu SQL để truy cập và thao tác trên cơ sở dữ liệu. Thay vì thực hiện nhiều phát biểu SQL, bạn có thể chạy một tập tin script, ví dụ `javadb.sql` có nội dung sau:

```
CREATE TABLE Student(code INT PRIMARY KEY, firstname VARCHAR(30), lastname VARCHAR(30));
INSERT INTO Student VALUES(100, 'Karl', 'Marx');
INSERT INTO Student VALUES(101, 'Fidel', 'Castro');
```

rồi chạy:

```
ij> run 'javadb.sql';
ij> CREATE TABLE Student(code INT PRIMARY KEY, firstname VARCHAR(30), lastname VARCHAR(30));
```

```
0 rows inserted/updated/deleted
```

```
ij> INSERT INTO Student VALUES(100, 'Karl', 'Mark');
```

```
1 row inserted/updated/deleted
```

```
ij> INSERT INTO Student VALUES(101, 'Fidel', 'Castro');
```

```
1 row inserted/updated/deleted
```

```
ij> SELECT * FROM Student;
```

CODE	FIRSTNAME	LASTNAME
100	Karl	Mark
101	Fidel	Castro

```
2 rows selected
```

```
ij> disconnect;
```

```
ij> exit;
```

2. Chế độ nhúng (embedded)

Phần trước, ta vừa tạo cơ sở dữ liệu nhúng `javadb`, có thể truy cập trực tiếp cơ sở dữ liệu nhúng này bằng JDBC.

Trong NetBeans, cần đưa thư viện Java DB Driver vào nhánh Libraries của dự án. Chú ý rằng bạn sẽ gặp lỗi không tương thích nếu cơ sở dữ liệu và driver sử dụng các phiên bản Java DB khác nhau.

- Kiểm tra phiên bản cơ sở dữ liệu được tạo bằng tiện ích `ij`.

- Kiểm tra phiên bản driver sử dụng trong NetBeans: vào Tools > Libraries, chọn Java DB Driver.

```
import java.sql.Connection;
import java.sql.DriverManager;
```

```

import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBCUsingJavaDB {
    public static void main(String[] args) throws SQLException, ClassNotFoundException {
        try (Connection connection = DriverManager.getConnection("jdbc:derby:c:/derby/javadb;user=root;password=toor");
            Statement statement = connection.createStatement()) {
            ResultSet resultSet = statement.executeQuery("SELECT * FROM Student");
            while (resultSet.next()) {
                System.out.println(resultSet.getString(2) + " " + resultSet.getString(3));
            }
        }
    }
}

```

3. Chế độ mạng

Java DB có thể chạy trong chế độ mạng. Từ *thư mục chứa cơ sở dữ liệu* nhưng vừa tạo ở trên, mở console và gọi tiện ích:

`startNetworkServer`

Server Java DB sẽ khởi động với cặp host:port mặc định là localhost:1527. Host và port mặc định có thể thay đổi bằng các tùy chọn -h và -p. Ví dụ:

`startNetworkServer -p 6868 -h sampleserver.sampledomain.com`

Sau đó, từ một console khác, dùng tiện ích `ij` kết nối vào server Java DB.

D:\working>ij

ij version 10.10

ij> connect 'jdbc:derby://localhost:1527/javadb;user=root;password=toor';

NetBeans hỗ trợ tốt chế độ mạng. Bạn vào Window > Services (Ctrl+5), nhánh Databases > Java DB. Hiệu chỉnh Properties của server Java DB sao cho:

- Java DB Installation tương thích với phiên bản Java DB dùng tạo cơ sở dữ liệu nhưng trên, tức %JAVA_HOME%\db

- Database Location chỉ đến thư mục chứa cơ sở dữ liệu nhưng trên, tức C:\derby

Sau đó, khởi động server Java DB, bạn sẽ thấy cơ sở dữ liệu javadb trên server, kết nối đến nó.

Bạn cũng có thể thêm vào cơ sở dữ liệu mới, chỉ định cụ thể vị trí lưu trữ của nó trong phần Database Location.

Khi dùng JDBC với chế độ mạng, chuỗi url là: `jdbc:derby://localhost:1527/javadb`.

JPA

JPA (Java Persistence API) phát triển từ Hibernate. JPA 2.1 (JSR 338) được cung cấp trong JavaEE 7 (2013). Tài liệu này dùng JPA 2. Các đặc điểm chính:

- Cung cấp một lớp trừu tượng để dùng trên đỉnh của JDBC.

- Dùng annotation (metadata) cài đặt ánh xạ đối tượng/quan hệ (ORM – Object-to-Relational Mapping).

- Thông qua Entity Manager thực hiện các thao tác CRUD (Create, Read, Update, Delete), bằng cách dùng ngôn ngữ truy vấn hướng đối tượng (Object-Oriented Query Language) gọi là JP QL.

Cài đặt của JSR 338 được cung cấp bởi EclipseLink, Hibernate và Open JPA. Các gói chính:

<code>javax.persistence</code>	Core API
<code>javax.persistence.criteria</code>	Criteria API
<code>javax.persistence.metamodel</code>	Metamodel API
<code>javax.persistence.spi</code>	SPI của nhà cung cấp JPA

1. ORM (Object-to-Relational Mapping)

a) Entity

Trong JPA, thực thể entity là đối tượng Java đơn giản (POJO – Plain Old Java Object).

Khi được quản lý bởi đối tượng Entity Manager cung cấp dịch vụ bền vững (persistence), entity trở thành đối tượng dữ liệu bền vững, gọi là đối tượng persistent. Những thay đổi trạng thái của đối tượng persistent sẽ được đồng bộ với dữ liệu tương ứng trong cơ sở dữ liệu.

Thực thể entity chỉ là POJO (các trường không public) có ít nhất một constructor không đối số và có các getters/setters để truy cập các trạng thái của nó. Khác với POJO bình thường, entity dùng nhiều annotation (meta-data mapping):

`@Entity` dùng để chỉ định rằng POJO này là một entity, có thể tham gia các truy vấn và các quan hệ với entity khác. Mỗi entity có một thuộc tính name, được dùng khi phát biểu truy vấn JPQL tham chiếu đến entity đó.

`@Table` chỉ định tên bảng ánh xạ với entity. Nếu thiếu `@Table`, tên bảng mặc định cùng tên với tên lớp entity.

`@Column` chỉ định tên cột ánh xạ với trường tương ứng của entity, trong đó khai báo rõ các ràng buộc cấp cột.

`@Basic` cung cấp ánh xạ kiểu tự động giữa kiểu dữ liệu của cột trong bảng và kiểu dữ liệu trường của POJO.

`@Temporal` cung cấp ánh xạ kiểu dựa trên thời gian, giữa kiểu JDBC (enum `TemporalType`: DATE, TIME, TIMESTAMP) và kiểu Java (`java.sql`: Date, Time, Timestamp).

Một số trường của POJO có thể dùng `@Transient` đánh dấu đó là trường tạm thời, sẽ bỏ qua khi thực hiện persistence.

Có hai cách đặt các annotation cấp thành viên:

- Field access: đặt annotation trên các trường của POJO. Entity Manager sẽ *truy cập trực tiếp* dữ liệu của entity thông qua các trường này. Cách này thường được dùng hơn:

```

@Entity
@Table(name="EMPLOYEE")
public class Employee {

```



```

@Id private Long id;
private String name;
private long salary;
// constructors, getters & setters
}

```

- Property access: đặt annotation trên accessor (getter/setter) của trường tương ứng. Entity Manager sẽ *truy cập gián tiếp* dữ liệu của entity thông qua các accessor.

```

@Entity
@Table(name="EMPLOYEE")
public class Employee {
    private Long id;
    private String name;
    private long salary;
    // constructors
    @Id public long getId() { return id; }
    public void setId(long id) { this.id = id; }
    // getters & setters khác
}

```

Ngoài ra, có thể kết hợp field access với property access, gọi là mixed access, nhưng ít dùng.

b) Khóa chính

Khóa chính cần dùng khi truy vấn đối tượng entity, chúng thường là tham số khi tìm hoặc lấy một entity.

@Id chỉ định trường của POJO là khóa chính.

Ta cũng có thể định nghĩa một khóa tự sinh bằng @GeneratedValue, báo rằng trường khóa chính của POJO sẽ nhận trị tự sinh từ bộ sinh tự động. Có nhiều bộ sinh khóa (enum GenerationType: AUTO, TABLE, SEQUENCE và IDENTITY). Nếu cơ sở dữ liệu hỗ trợ cơ chế sinh khóa (gọi là sequence), dùng GenerationType.SEQUENCE.

```

@Entity
@SequenceGenerator(name="CustomerSequence", sequenceName="CUSTOMER_SEQ", initialValue=100, allocationSize=50)
public class Customer implements Serializable {
    @Id @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="CustomerSequence")
    private Integer id;
}

```

Nếu khóa chính gồm nhiều trường (composite PK hoặc khóa phức), ta cần có lớp khóa chính. Lớp khóa chính là một POJO cài đặt giao diện Serializable, có constructor mặc định không đối số, có nạp chồng các phương thức hashCode() và equals().

```

public class CustomerPK implements Serializable {
    private String lastName;
    private Long ssn;
    public String getLastName() { return lastName; }
    public void setLastName(String lastName) { this.lastName = lastName; }
    public Long getSsn() { return ssn; }
    public void setSsn(Long ssn) { this.ssn = ssn; }
    @Override public int hashCode() { }
    @Override public boolean equals(Object obj) { }
}

```

Sau khi định nghĩa lớp khóa chính, có hai cách dùng:

- Dùng @IdClass: khi dùng @IdClass định nghĩa lớp khóa chính cho một entity. Entity đó phải có các trường giống lớp khóa chính, khai báo @Id cho mỗi trường thành phần này.

```

@Entity
@IdClass(CustomerPK.class)
public class Customer implements Serializable {
    @Id private String lastName;
    @Id private Long ssn;
}

```

- Dùng @EmbeddedId: dùng annotation @Embeddable cho lớp khóa chính.

```

@Embeddable
public class CustomerPK implements Serializable {
    @Column private String lastName;
    @Column private Long ssn;
    public String getLastName() { return lastName; }
    public void setLastName(String lastName) { this.lastName = lastName; }
    public Long getSsn() { return ssn; }
    public void setSsn(Long ssn) { this.ssn = ssn; }
    @Override public int hashCode() { }
    @Override public boolean equals(Object obj) { }
}

```

Sau đó, bạn có thể dùng trực tiếp đối tượng lớp khóa chính trong entity với annotation @EmbeddedId

```

@Entity
public class Customer implements Serializable {
    @EmbeddedId
    private CustomerPK pk;
}

```

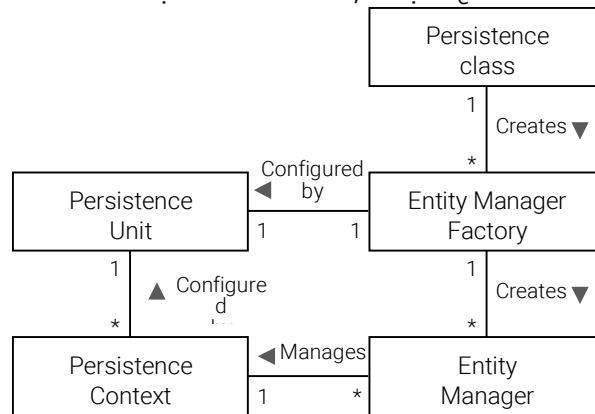

2. Các thành phần trong JPA

a) Persistence Unit

Gói JAR chứa các entity, gọi là *persistence archive*, cần có tập tin *persistence.xml* trong META-INF của nó. Tập tin cấu hình này định nghĩa một hoặc nhiều *persistence unit*, đơn vị cấu hình được đặt tên này mô tả nguồn dữ liệu liên kết với các entity trong gói. Như vậy, một Persistence Unit là tập các entity ánh xạ với cùng một cơ sở dữ liệu.

```
<persistence>
  <persistence-unit name="PersistenceUnit" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>samples.Project</class>
    <class>samples.Employee</class>
    <class>samples.CreditCard</class>
    <properties>
      <property name="javax.persistence.jdbc.driver"
        value="com.microsoft.sqlserver.jdbc.SQLServerDriver"/>
      <property name="javax.persistence.jdbc.url"
        value="jdbc:sqlserver://localhost:1433;databaseName=MOON"/>
      <property name="javax.persistence.jdbc.user" value="sa"/>
      <property name="javax.persistence.jdbc.password" value=""/>
    </properties>
  </persistence-unit>
</persistence>
```

Thuộc tính name của <persistence-unit> sẽ được tham chiếu đến, ví dụ từ @PersistenceContext.



Mối quan hệ giữa các thành phần trong JPA

b) Persistence Context

Entity Manager duy trì một vùng nhớ chứa tập các entity mà nó quản lý, gọi là *persistence context*. Persistence Context cho phép Entity Manager theo dõi các entity được tạo ra để thay đổi, loại bỏ chúng và chuyển các thay đổi này đến cơ sở dữ liệu ánh xạ với chúng. Có hai kiểu Persistence Context:

- Transaction-scoped: Persistence Context nằm trong một transaction, sẽ đóng khi transaction hoàn tất.
- Extended: entity kèm theo extended context vẫn còn được quản lý (managed) sau khi transaction hoàn tất.

c) Entity Manager

Entity Manager cung cấp các dịch vụ quản lý entity theo framework JPA: tạo truy vấn, tìm đối tượng, CRUD, đồng bộ hóa, ... nó cũng quản lý tương tác với dịch vụ transaction (JTA). Bạn phải có được đối tượng Entity Manager khi muốn tương tác với các entity. Có thể nhận được Entity Manager từ JavaEE (bên trong EJB Container) hoặc JavaSE.

Dùng các cách sau để có được Entity Manager:

- Trong session bean, "tiêm" EntityManager liên kết với <persistence-unit> vào EJB Container.

@Stateless

```
public class CustomerManager {
    @PersistenceContext(unitName="PersistenceUnit")
    private EntityManager em;

    public void createCustomer() {
        final Customer customer = new Customer();
        customer.setName("Karl Marx");
        em.persist(customer);
    }
}
```

- Trong JavaSE client, dùng EntityManagerFactory lấy từ lớp "khởi động" Persistence.

```
public class CustomerService {
    public static void main(String[] args) {
        final EntityManagerFactory emf = Persistence.createEntityManagerFactory("PersistenceUnit");
        final EntityManager em = emf.createEntityManager();
    }
}
```

```

    final Customer customer = new Customer();
    em.persist(customer);
    emf.close();
}
}

```

- Trong JavaEE, "tiêm" EntityManagerFactory trực tiếp vào trường hoặc setter của session bean, dùng @PersistenceUnit. EntityManagerFactory sẽ tự động đóng khi hủy bean.

@Stateless

```

public class MyBean implements MyBusinessInterface {
    @PersistenceUnit(unitName="PersistenceUnit")
    private EntityManagerFactory emf;

    public void createCustomer() {
        final EntityManager em = emf.createEntityManager();
        final Customer customer = new Customer();
        em.persist(customer);
    }
}

```

- Dò tìm (lookup) EntityManager từ dịch vụ JNDI, áp dụng cả cho JavaEE lẫn JavaSE.

@Stateless

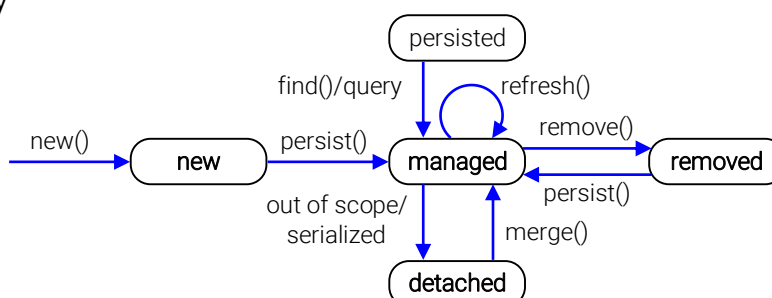
```

@PersistenceContext(unitName="PersistenceUnit")
public class CustomerServiceBean {
    @Resource
    SessionContext ctx;

    public void performService() {
        EntityManager em = (EntityManager)ctx.lookup("PersistenceUnit");
    }
}

```

d) Các trạng thái của entity



- New: entity được tạo mới đơn giản bằng một trong các constructor của POJO, không tham số hoặc có tham số. Trong trạng thái này, nó chưa ánh xạ với cơ sở dữ liệu (chưa có persistent identity) và do đó chưa liên kết với Persistence Context nào.

- Managed: để entity trở thành một thực thể persistent, đồng bộ với cơ sở dữ liệu, client phải gọi phương thức persist() của EntityManager. Khi đó entity gọi là được quản lý (managed) hoặc được gắn vào (attached). Khi được quản lý, entity liên kết với một Persistence Context, mọi thay đổi trên nó sẽ được theo dõi bởi Entity Manager thông qua Persistence Context đó. Entity ở trạng thái được quản lý không nhất thiết có từ trạng thái New, thực tế nó thường là kết quả của các truy vấn từ cơ sở dữ liệu hoặc bằng cách gọi phương thức find() của EntityManager.

Nếu một entity được quản lý trong Persistence Context, phương thức contains() của EntityManager với đối số là entity đó, sẽ trả về true.

- Detached: khi tách entity khỏi Persistence Context, entity không còn đồng bộ với cơ sở dữ liệu nữa. Dùng merge() hoặc refresh() của EntityManager để chuyển entity trở lại trạng thái Managed. Để tách tất cả các entity khỏi Persistence Context, gọi phương thức clear() của EntityManager.

- Removed: muốn loại bỏ entity khỏi cơ sở dữ liệu, gọi phương thức remove() của EntityManager. Dòng tương ứng trong cơ sở dữ liệu sẽ bị loại bỏ.

	New	Managed	Detached	Removed
persist()	Managed	Ignored	IllegalArg	Managed
remove()	Ignored	Removed	IllegalArg	Ignored
refresh()	Ignored	Removed	IllegalArg	Ignored
merge()	Managed	Ignored	Managed	IllegalArg

Cột: trạng thái của entity. Hàng: phương thức của ManagerEntity.

e) CRUD

Create

Tác vụ bền vững hóa (persisting) một entity là hành động chèn entity đó vào cơ sở dữ liệu, được thực hiện bằng cách gọi phương thức persist() của EntityManager.

```

Employee bill = new Employee(ID_BILL, "Bill Gates");
em.persist(bill);

```

Phương thức `persist()` có thể gọi trong transaction, entity được chèn vào cơ sở dữ liệu ngay lập tức hoặc đưa vào hàng đợi để xử lý tùy theo chế độ flush, hoặc có thể ép thực hiện bằng cách gọi phương thức `EntityManager.flush()`. Phương thức `persist()` chỉ được gọi ngoài transaction nếu Persistence Context là Extended.

Find và Query

Entity Manager cung cấp hai cơ chế để truy tìm entity trong cơ sở dữ liệu:

- Find: tìm entity theo khóa chính. Dùng phương thức `find()` hoặc `getReference()` của `EntityManager`. Nếu không tìm thấy entity trong cơ sở dữ liệu, `find()` trả về null trong lúc `getReference()` sẽ ném exception.

```
em.find(Employee.class, ID_BILL);
```

- Query: tạo và thực hiện truy vấn bằng ngôn ngữ JP QL, query được tạo bằng cách gọi các phương thức `createQuery()`, `createNamedQuery()`, hoặc `createNativeQuery()` của `EntityManager`. Giống Prepared Statement trong JDBC, query cũng có tham số và được thiết lập bằng phương thức `setParameter()` của nó.

```
Query query = entityManager.createQuery("FROM Employees c WHERE c.id = :id");
query.setParameter("id", ID_BILL);
Employee employee = (Employee)query.getSingleResult();
```

Update

Phương thức `merge()` của `EntityManager` cho phép một entity ở trạng thái Detached kết nối trở lại cơ sở dữ liệu. Đây là cách cập nhật một entity có trong cơ sở dữ liệu.

```
@PersistenceContext(unitName="PersistenceUnit")
private EntityManager em;
```

```
Employee bill = new Employee(ID_BILL, "Bill Sharp"); // cập nhật tên mới cho entity (detached)
Employee managedBill = em.merge(bill); // áp dụng thay đổi này vào cơ sở dữ liệu
bill.setName("Bill Sharp"); // áp dụng thay đổi này với entity
```

Delete

Một entity có thể được loại khỏi cơ sở dữ liệu bằng cách gọi phương thức `remove()` của `EntityManager`. Dòng dữ liệu tương ứng không bị xóa ngay lập tức; khi Entity Manager quyết định flush, truy vấn DELETE mới thực hiện.

```
@PersistenceContext(unitName="PersistenceUnit")
private EntityManager em;
```

```
Employee bill = em.find(Employee.class, ID_BILL);
if (bill != null) em.remove(bill);
```

Sau khi gọi `remove()`, entity có trạng thái Detached, tác vụ này có thể khôi phục lại (undo) bằng cách tạo lại entity với phương thức `persist()`.

Flush và Cascade

Khi gọi `persist()`, `merge()` hoặc `remove()`, những thay đổi trên entity không đồng bộ ngay với cơ sở dữ liệu cho đến khi Entity Manager quyết định flush. Mặc định, flush tiến hành trước khi truy vấn tương ứng thực hiện và khi transaction hoàn tất (commit). Để thay đổi chế độ mặc định, dùng `javax.persistence.FlushModeType`. Ngoài ra, có thể ép flush bằng cách gọi tường minh phương thức `flush()`.

Khi gọi `remove()` hoặc `refresh()`, tác động xóa hoặc cập nhật mới sẽ đổ xuống các entity có quan hệ với entity hiện hành.

Các tác vụ CRUD thường được gọi trong transaction:

```
em.getTransaction().begin();
Employee bill = new Employee(ID_BILL, "Bill Gates");
em.persist(bill);
em.getTransaction().commit();
System.out.println("Persisted " + bill);
```

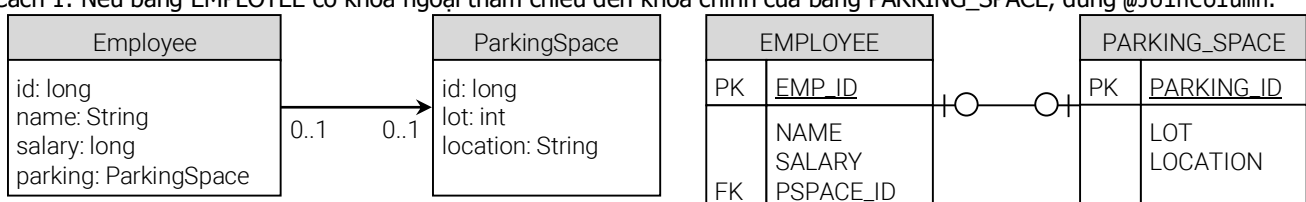
3. Quan hệ giữa các Entity

Có bốn kiểu quan hệ giữa các entity : one-to-one, one-to-many, many-to-one, và many-to-many. Ngoài ra, mỗi mối quan hệ có thể là một chiều (unidirectional) hoặc hai chiều (bidirectional). Vì quan hệ one-to-many và many-to-one hai chiều là giống nhau, nên chỉ có bảy kiểu quan hệ có thể có giữa các entity.

a) One-to-one

Quan hệ 1-1 một chiều, ví dụ như quan hệ giữa một Employee (nhân viên) với một ParkingSpace (chỗ đậu xe). Với mỗi nhân viên, ta xác định được chỗ đậu xe của họ, nhưng ta không quan tâm một chỗ đậu xe thuộc về nhân viên nào. Nói cách khác, Employee là bên sở hữu mối quan hệ, bên ngược lại không lưu thông tin mối quan hệ.

- Cách 1: Nếu bảng EMPLOYEE có khóa ngoại tham chiếu đến khóa chính của bảng PARKING_SPACE, dùng `@JoinColumn`.



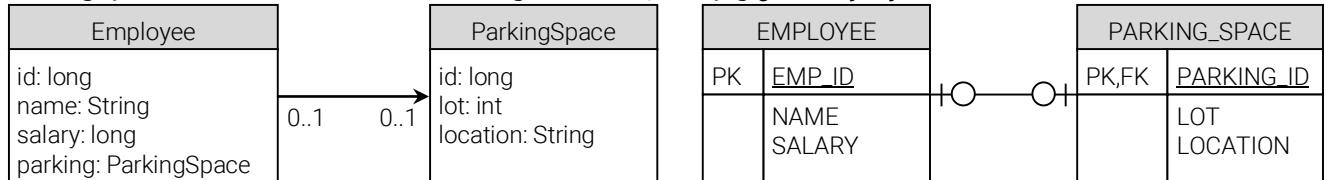
```
@Entity
```

```
@Table(name="EMPLOYEE")
```

```
public class Employee implements Serializable {
    // ...
    @OneToOne
    @JoinColumn(name="PSPACE_ID", referencedColumnName="PARKING_ID")
    private ParkingSpace parking; // foreign key
    // ...
}
```

@JoinColumn định nghĩa cột trong bảng EMPLOYEE tham chiếu đến khóa chính của bảng PARKING_SPACE. Thuộc tính name chỉ tên cột là khóa ngoại trong bảng EMPLOYEE, thuộc tính referencedColumnName chỉ tên cột cần tham chiếu trong bảng PARKING_SPACE, tức khóa chính của bảng PARKING_SPACE, thuộc tính này không cần đến nếu giống thuộc tính name. Nếu tham chiếu đến một thực thể có khóa chính phức tạp, dùng @JoinColumns.

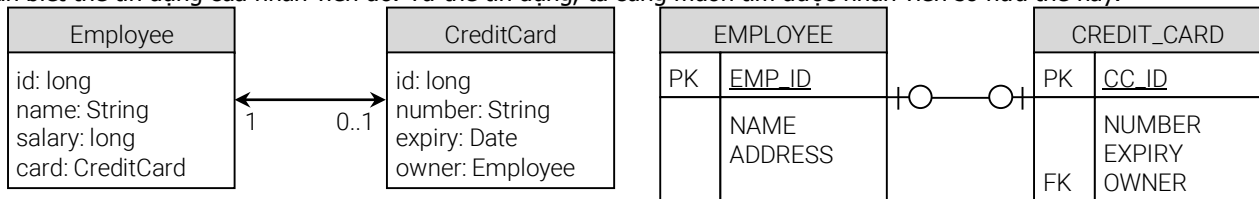
- Cách 2: Nếu hai bảng EMPLOYEE và PARKING_SPACE dùng chung khóa chính, nghĩa là khóa chính của PARKING_SPACE cũng là khóa ngoại tham chiếu đến khóa chính của bảng EMPLOYEE, sử dụng @PrimaryKeyJoinColumn.



```
@Entity
@Table(name="EMPLOYEE")
public class Employee implements Serializable {
    // ...
    @OneToOne
    @PrimaryKeyJoinColumn(name="EMP_ID", referencedColumnName="PARKING_ID")
    private ParkingSpace parking; // parent primary key join
    // ...
}
```

Trong cả hai cách, từ thực thể ParkingSpace ta không thể tham chiếu đến Employee sở hữu nó.

Quan hệ 1-1 hai chiều, ví dụ như quan hệ giữa một Employee (nhân viên) và một CreditCard (thẻ tín dụng). Từ một nhân viên, ta cần biết thẻ tín dụng của nhân viên đó. Từ thẻ tín dụng, ta cũng muốn tìm được nhân viên sở hữu thẻ này.



Trong quan hệ hai chiều, cần chọn bên sở hữu mỗi quan hệ, thường là bên khởi đầu của hướng được tìm nhiều hơn. Trong trường hợp trên là CreditCard, vì ta thường từ CreditCard tìm ra Employee. Trong lớp CreditCard đánh dấu @OneToOne cho tham chiếu chỉ đến Employee.

```
@Entity
@Table(name="CREDIT_CARD")
public class CreditCard implements Serializable {
    // ...
    @OneToOne
    @JoinColumn(name = "OWNER", referencedColumnName = "EMP_ID")
    private Employee owner;
    // ...
}
```

Bên CreditCard, bạn cũng có thể dùng @PrimaryKeyJoinColumn thay cho @JoinColumn.

Thực thể bên ngược lại (inverse) là Employee, cũng chứa tham chiếu đến lớp CreditCard, thiết lập thuộc tính mappedBy.

```
@Entity
@Table(name="EMPLOYEE")
public class Employee implements Serializable {
    // ...
    @OneToOne(mappedBy="owner")
    private CreditCard card;
    // ...
}
```

Thuộc tính mappedBy thiết lập mỗi quan hệ hai chiều và báo cho EntityManager biết thông tin cho việc ánh xạ mỗi quan hệ giữa các bảng này được chỉ định trong lớp CreditCard, ánh xạ bởi trường owner.

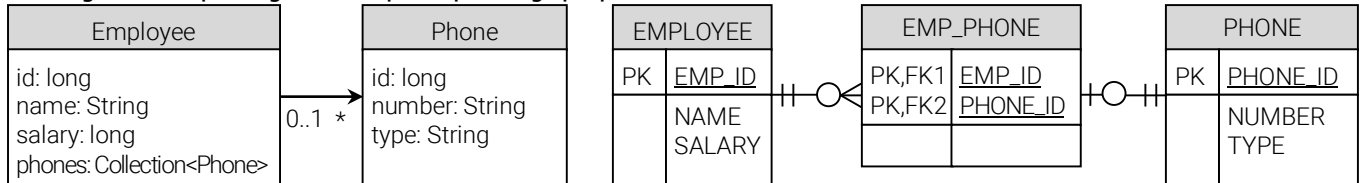
Sau đây là ví dụ thiết lập mỗi quan hệ hai chiều:

```
// tạo một CreditCard
CreditCard card = new CreditCard();
card.setNumber("1234 5678 1101 1001");
card.setName("Rowan Atkinson");
// tạo một Employee
Employee rowan = new Employee("Rowan Atkinson");
```

```
// persist
em.persist(rowan);
em.persist(card);
// thiết lập liên kết hai bên cho mỗi quan hệ hai chiều
rowan.setCard(card);
card.setOwner(rowan);
```

b) One-to-many

Quan hệ 1-n một chiều, ví dụ như quan hệ giữa một Employee (nhân viên) và Phone (số điện thoại) thuộc nhân viên đó. Một nhân viên có thể có nhiều số điện thoại (cơ quan, nhà, cá nhân, ...). Ta cần phải tìm các số điện thoại của một nhân viên, nhưng ta không cần từ một trong các số điện thoại tìm ngược lại nhân viên sở hữu số đó.



Trong lớp Employee, tạo quan hệ dựa trên một collection các Phone:

```
@Entity
@Table(name="EMPLOYEE")
public class Employee implements Serializable {
    // ...
    @OneToMany(orphanRemoval=true)
    @JoinTable(name="EMP_PHONE",
        joinColumns=@JoinColumn(name="EMP_ID"),
        inverseJoinColumns=@JoinColumn(name="PHONE_ID"))
    private Collection<Phone> phones;
    // ...
}
```

Trong entity Phone, không dùng annotation nào mô tả mối quan hệ ngược lại.

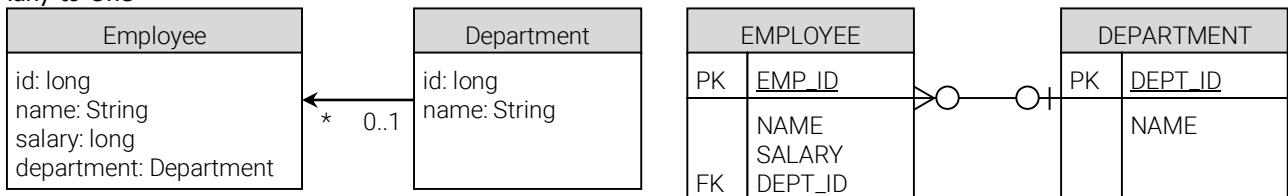
Ta có thể sử dụng các cấu trúc dữ liệu khác từ java.util như Collection, List, Map hoặc Set. Collection có thể chứa các tham chiếu chỉ đến cùng một entity (tham chiếu trùng lặp) trong lúc Set chỉ chứa các tham chiếu chỉ đến các entity riêng biệt.

Sau đây là mã tương tác với EntityManager:

```
// tạo một Employee
Employee steve = new Employee("Steve Jobs");
// tạo hai Phone
Phone home = new Phone("800-USE-EJB3", "Home");
Phone office = new Phone("8675309", "Office");
// persist
em.persist(steve);
em.persist(home);
em.persist(office);
// liên kết
steve.getPhones().add(home);
steve.getPhones().add(office);
```

Nếu muốn loại bỏ một Phone khỏi mối quan hệ trên, cần loại bỏ Phone khỏi collection lẫn cơ sở dữ liệu. Nếu không Phone sẽ bị "mồ côi" (orphan). Nếu thiết lập true cho thuộc tính orphanRemoval của @OneToMany, Phone "mồ côi" sẽ được loại bỏ tự động.

c) Many-to-One

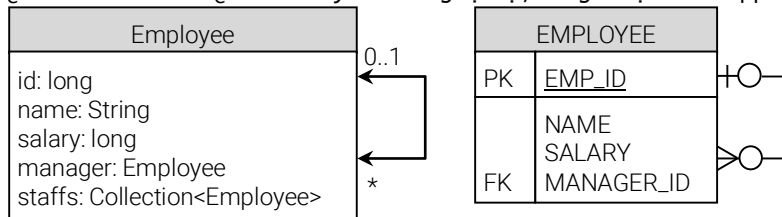


Quan hệ n-1 một chiều, ví dụ như quan hệ giữa Department (phòng ban) và Employee (nhân viên). Một phòng ban có nhiều nhân viên, các nhân viên đều biết mình thuộc phòng ban nào, nhưng ta không cần liệt kê nhân viên của một phòng ban. Khóa chính từ bảng DEPARTMENT được tham chiếu như khóa ngoại trong bảng EMPLOYEE. Bên @ManyToOne, tức Employee luôn là bên sở hữu mối quan hệ.

```
@Entity
@Table(name="EMPLOYEE")
public class Employee implements Serializable {
    // ...
    @ManyToOne
    @JoinColumn(name="DEPT_ID")
    private Department department;
    // ...
}
```

Trong entity `Department`, không dùng annotation nào mô tả mối quan hệ ngược lại.

Quan hệ 1-n hai chiều và quan hệ n-1 hai chiều là như nhau. Ví dụ, một nhân viên cũng là một quản lý. Bên `@ManyToOne` luôn sở hữu mối quan hệ, dùng `@JoinColumn`. Bên `@OneToOne` là bên ngược lại, dùng thuộc tính `mappedBy`.

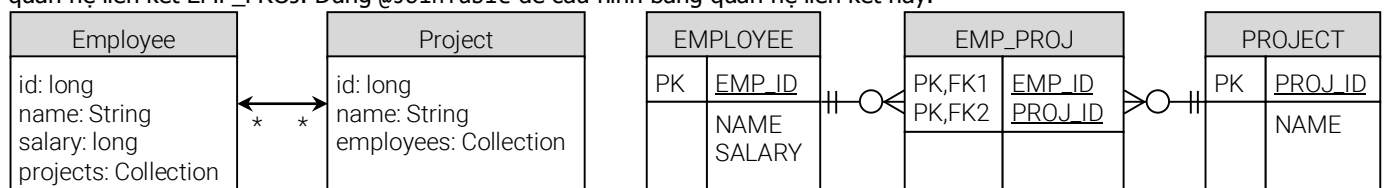


```
@Entity
@Table(name="EMPLOYEE")
public class Employee implements Serializable {
    // ...
    @OneToOne(mappedBy="manager")
    private Collection<Employee> staffs;
    // ...
    @ManyToOne
    @JoinColumn(name="MANAGER_ID", referencedColumnName="EMP_ID")
    private Employee manager;
    // ...
}
```

Theo mặc định, vì lý do hiệu suất, tất cả các trường tham chiếu được chỉ định kiểu "lấy" (fetch) là `FetchType.LAZY`. Điều này dễ thấy ở quan hệ 1-n, ví dụ khi bạn có `Employee` thì chỉ tiết các `Phone` trong collection `phones` chưa được nạp. Để truy cập ngay các `Phone`, phải thiết lập `FetchType.EAGER`.

d) Many-to-Many

Quan hệ n-n, ví dụ quan hệ giữa `Employee` (nhân viên) và `Project` (dự án). Một nhân viên có thể tham gia nhiều dự án, một dự án có thể có nhiều nhân viên tham gia. Một nhân viên cụ thể tham gia một dự án cụ thể phải được ghi nhận trong một bảng quan hệ liên kết `EMP_PROJ`. Dùng `@JoinTable` để cấu hình bảng quan hệ liên kết này.



Bên sở hữu mỗi quan hệ:

```
@Entity
@Table(name="EMPLOYEE")
public class Employee implements Serializable {
    // ...
    @ManyToMany
    @JoinTable(name="EMP_PROJ",
        joinColumns={@JoinColumn(name="EMP_ID")},
        inverseJoinColumns={@JoinColumn(name="PROJ_ID")})
    private Collection<Project> projects;
    // ...
}
```

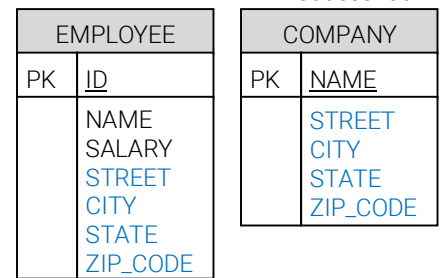
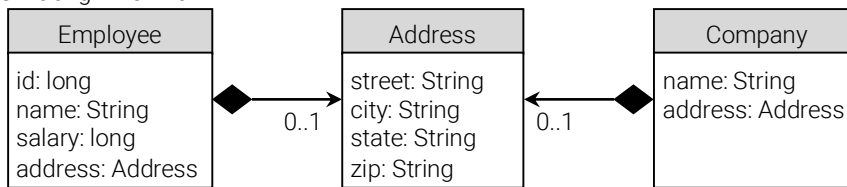
Bên ngược lại:

```
@Entity
@Table(name="PROJECT")
public class Project implements Serializable {
    // ...
    @ManyToMany(mappedBy="projects")
    private Collection<Employee> employees;
    // ...
}
```

4. Đối tượng nhúng

Đối tượng nhúng là một phần trạng thái của một entity, được tách ra và lưu trữ trong một đối tượng Java riêng biệt. Trong Java, đối tượng nhúng thể hiện như đối tượng có quan hệ phụ thuộc với entity. Trong cơ sở dữ liệu, trạng thái của entity và trạng thái của đối tượng nhúng lưu chung trong một dòng của bảng thực thể.

Xem ví dụ dưới, đối tượng nhúng `Address` được chia sẻ giữa `Employee` với `Company`. Xem trong các bảng `EMPLOYEE` và `COMPANY`, ta thấy bộ các cột `STREET`, `CITY`, `STATE`, `ZIP_CODE` kết hợp thành đối tượng nhúng `Address`.



```

@Entity
@Access(AccessType.FIELD)
public class Address {
    private String street;
    private String city;
    private String state;
    @Column(name="ZIP_CODE")
    private String zip;
    // ...
}
  
```

Sử dụng đối tượng nhúng trong Employee và Company:

```

@Entity
@Table(name="Employee")
public class Employee {
    @Id private long id;
    private String name;
    private long salary;
    @Embedded private Address address;
    // ...
}
  
```

```

@Entity
@Table(name="Company")
public class Company {
    @Id private String name;
    @Embedded private Address address;
    // ...
}
  
```

5. JP QL

Có nguồn gốc từ EJB QL trong đặc tả EJB 2.0, JP QL là ngôn ngữ truy vấn hướng đối tượng. Cách viết tựa như SQL-92 nhưng làm việc với các entity, các thuộc tính của entity, trả về collection các entity hoặc collection các thuộc tính.

- Lọc kết quả:

```

SELECT e
FROM Employee e
WHERE e.department.name = 'NA42' AND e.address.state IN ('NY', 'CA')
  
```

- Nối (join) giữa các entity:

```

SELECT p.number
FROM Employee e JOIN e.phones p
WHERE e.department.name = 'NA42' AND p.type = 'Cell'
  
```

- Gọi các hàm SQL:

```

SELECT d, COUNT(e), MAX(e.salary), AVG(e.salary)
FROM Department d JOIN d.employees e
GROUP BY d
HAVING COUNT(e) >= 5
  
```

- Truy vấn có tham số: tham số có dấu ":" phía trước

```

SELECT e
FROM Employee e
WHERE e.department = :dept AND e.salary > :base
  
```

JPA cung cấp các interface Query và TypedQuery để cấu hình và thực thi các truy vấn. Query được dùng khi kiểu trả về là Object hoặc chưa biết rõ, TypedQuery thừa kế Query, làm việc với kiểu trả về biết rõ.

Có ba cách định nghĩa một truy vấn JP QL:

- Dynamic query, chỉ định trong thời gian chạy.

```

public class QueryService {
    private static final String QUERY =
        "SELECT e.salary " +
        "FROM Employee e " +
        "WHERE e.department.name = :deptName AND e.name = :empName ";
    @PersistenceContext(unitName="QueriesUnit")
    EntityManager em;
    public long queryEmpSalary(String deptName, String empName) {
  
```

```

        return em.createQuery(QUERY, Long.class)
            .setParameter("deptName", deptName)
            .setParameter("empName", empName)
            .getSingleResult();
    }
}

```

- Named query, query định nghĩa trước kèm theo entity, có đặt tên.

```

@NamedQueries({
    @NamedQuery(name="Employee.findAll", query="SELECT e FROM Employee e"),
    @NamedQuery(name="Employee.findByPrimaryKey", query="SELECT e FROM Employee e WHERE e.id = :id"),
    @NamedQuery(name="Employee.findByName", query="SELECT e FROM Employee e WHERE e.name = :name")})

```

```

public class Employee implements Serializable { }

```

Thực thi named query:

```

public class EmployeeService {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    public Employee findEmployeeByName(String name) {
        return em.createNamedQuery("Employee.findByName", Employee.class)
            .setParameter("name", name)
            .getSingleResult();
    }
}

```

- Dynamic named query, kết hợp của hai loại truy vấn trên.

Truy vấn được tạo rồi được EntityManagerFactory đưa vào context với tên chỉ định bằng cách gọi phương thức addNamedQuery().

```

TypedQuery<Long> query = em.createQuery(QUERY, Long.class);
emf.addNamedQuery("findSalaryForNameAndDepartment", query);

```

Khi thực thi truy vấn, chọn trả kết quả về bằng cách gọi getSingleResult() nếu chỉ có một kết quả, hoặc getResultList() nếu có nhiều kết quả. Tuy nhiên, hai phương thức này xử lý khác nhau khi gặp kết quả không mong đợi. Khi truy vấn không kết quả, getResultList() trả về collection rỗng, trong lúc getSingleResult() ném NoResultException.

Phương thức executeUpdate() cũng được gọi nếu truy vấn chỉ thay đổi cơ sở dữ liệu (update, delete).

Trường hợp có nhiều kiểu trả về trong kết quả, dùng construction expression như sau:

- Tạo một POJO để gom các kiểu trả về:

```

public class EmpMenu {
    private String employeeName;
    private String departmentName;
    public EmpMenu(String employeeName, String departmentName) {
        this.employeeName = employeeName;
        this.departmentName = departmentName;
    }
    public String getEmployeeName() { return employeeName; }
    public String getDepartmentName() { return departmentName; }
}

```

- Xử lý nhiều kiểu trả về:

```

public void displayProjectEmployees(String projectName) {
    List<EmpMenu> results =
        em.createQuery(
            "SELECT NEW samples.EmpMenu(e.name, e.department.name) " +
            "FROM Project p JOIN p.employees e " +
            "WHERE p.name = ?1 " +
            "ORDER BY e.name", EmpMenu.class)
            .setParameter(1, projectName)
            .getResultList();

    int count = 0;
    for (EmpMenu menu : results) {
        System.out.println(++count + ": " + menu.getEmployeeName() + ", " + menu.getDepartmentName());
    }
}

```

Công cụ Java

Ant

Ant là một build-system dùng xây dựng (build) nhanh ứng dụng. Ant được sử dụng phổ biến, là nền tảng của build-system cho các IDE phổ biến. Ant là một công cụ không thể thiếu khi: phát triển theo nhóm, thực hiện project phát triển ứng dụng Enterprise, triển khai ứng dụng JavaEE, đóng gói sản phẩm, làm việc nhóm qua mạng, các dự án mã nguồn mở, ...

1. Cài đặt Ant

Tải về Ant từ <http://jakarta.apache.org/ant/index.html>, giải nén để cài đặt vào thư mục chỉ định, ví dụ: C:\java\ant.

Đặt biến môi trường ANT_HOME có trị C:\java\ant

Thêm vào biến môi trường PATH chuỗi %ANT_HOME%\bin.

Ant dùng như một tiện ích dòng lệnh dùng một tập tin XML (build file) để mô tả quá trình build. Tập tin XML này mặc định là build.xml. Ant lấy các tập tin đầu vào từ *cấu trúc thư mục lưu trữ*, sau đó Ant có thể thực hiện hàng trăm tác vụ cài đặt sẵn (các lệnh thư mục và tập tin, biên dịch, đóng gói...), đưa các tập tin đầu ra vào *cấu trúc thư mục triển khai* rồi đóng gói hoặc chạy. Khi gọi từ dòng lệnh: ant, build.xml sẽ được tìm kiếm để thực hiện.

Trong NetBeans, build.xml chứa tập tin build thật sự là nbproject/build-impl.xml.

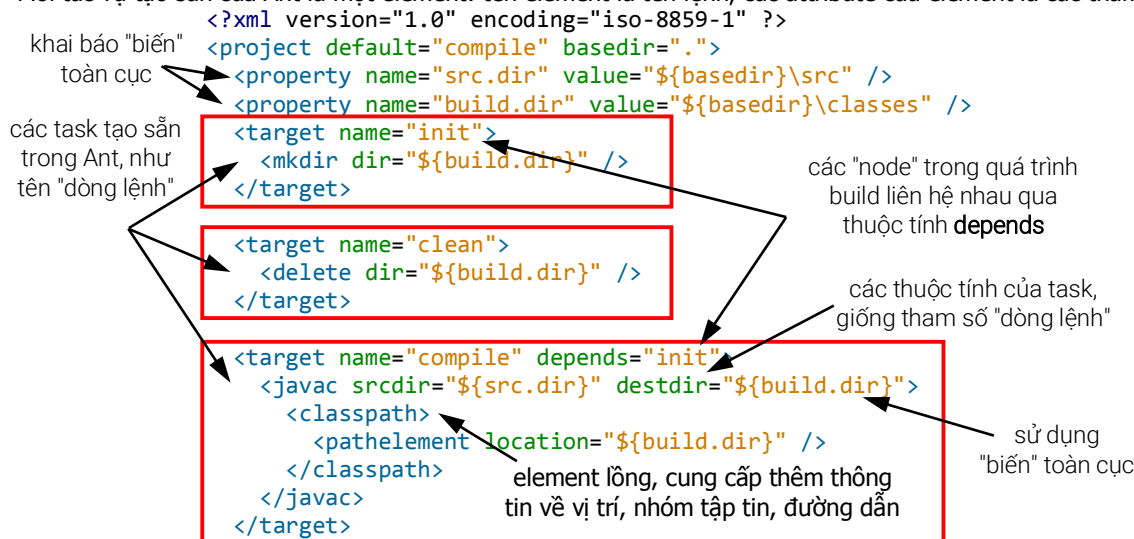
2. Tập tin build

Tập tin build.xml điển hình bao gồm:

- Các "biến" toàn cục được khai báo bằng element property, một biến toàn cục chẳng hạn src.dir, sau khi khai báo sẽ dùng như \${src.dir}.

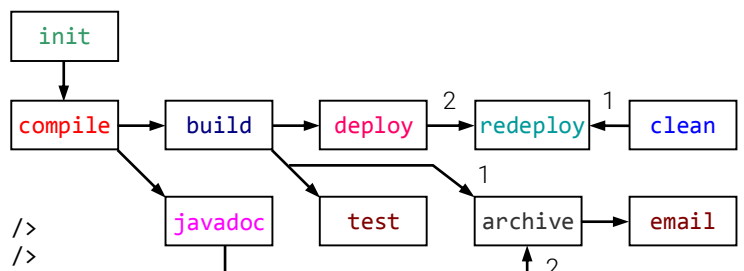
- Các tác vụ chính được khai báo bằng element target, chúng tạo thành các "node" trong cây build dùng để xây dựng ứng dụng. Mỗi tác vụ chính này gọi một hoặc nhiều tác vụ tạo sẵn trong hàng trăm tác vụ tạo sẵn của Ant.

Mỗi tác vụ tạo sẵn của Ant là một element: tên element là tên lệnh, các attribute của element là các tham số dòng lệnh.



Các tác vụ chính, tức các "node" trong cây build, phụ thuộc với nhau thông qua thuộc tính depends. Tác vụ B phụ thuộc (depends) tác vụ A, nghĩa là tác vụ A *phải thực hiện trước* khi tác vụ B được thực hiện.

```
<target name="clean" />
<target name="compile" depends="init" />
<target name="javadoc" depends="compile" />
<target name="build" depends="compile" />
<target name="test" depends="build" />
<target name="deploy" depends="build" />
<target name="email" depends="archive" />
<target name="archive" depends="build, javadoc" />
<target name="redeploy" depends="clean, deploy" />
```



Trong ví dụ trên, chẳng hạn để thực hiện tác vụ chính redeploy, trước hết cần thực hiện các tác vụ chính clean, sau đó thực hiện chuỗi các tác vụ chính theo thứ tự init → compile → build → deploy.

Khi một "node" trong chuỗi tác vụ chính được gọi, các "node" trước nó sẽ được thực hiện trước, kể từ "node" đầu chuỗi.

Bạn có thể gọi một tác vụ chính bất kỳ, ví dụ clean, có trong build.xml bằng dòng lệnh: ant clean

Tác vụ khai báo trong thuộc tính default của element project là tác vụ chính mặc định, sẽ được gọi khi dùng Ant không có đối số với dòng lệnh: ant



Tập tin build.xml được sử dụng lại cho các ứng dụng tương tự, chỉ cần tùy biến một số tác vụ chính hay quy trình build. Ví dụ tập tin build-impl.xml của NetBeans.

Để sử dụng được các tác vụ tạo sẵn trong Ant, dùng một tài liệu tra cứu tác vụ có sẵn (thường có ví dụ tổng quát kèm theo), hoặc tra cứu trên <http://ant.apache.org/index.html>

3. Thực hành với Ant

Viết một tập tin mã nguồn (ví dụ: HelloWorld.java)

```
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello World!");
    }
}
```

Viết build.xml đơn giản, yêu cầu ba tác vụ:

- compile: tự động biên dịch HelloWorld.java.
- compress: đóng gói thành project.jar.
- main: chạy ứng dụng dưới dạng JAR file.

```
<?xml version="1.0" ?>
<project default="main">
    <target name="main" depends="compile, compress">
        <echo>Execute...</echo>
        <java jar="project.jar" fork="true" />
    </target>
    <target name="compile">
        <echo>Compile...</echo>
        <javac srcdir="." />
    </target>
    <target name="compress">
        <echo>Bundle...</echo>
        <jar jarfile="project.jar" basedir="." includes="*.class">
            <manifest>
                <attribute name="Main-Class" value="HelloWorld" />
            </manifest>
        </jar>
    </target>
</project>
```

Chạy Ant trong console và theo dõi kết quả build: ant

JUnit

1. Test-Driven Develop (TDD)

a) Khái niệm

Kiểm thử (testing) là một vấn đề quan trọng trong quy trình phát triển phần mềm. Sau đây gọi tắt kiểm thử là test. Phát triển ứng dụng dưới test, gọi tắt là TDD, là một framework cho test, thường được áp dụng trong quy trình phát triển ứng dụng nhanh. TDD mang lại nhiều lợi ích thiết thực:

- Bao quát được nhiều trường hợp, nhất là các trường hợp đặc biệt. Bảo đảm code chạy đúng trong nhiều tình huống.
- Ví dụ: viết "đơn vị" phương thức getAverageNegative(), tính trung bình cộng các số nguyên âm của một mảng, bạn dễ gặp lỗi do bỏ sót các trường hợp: mảng được cung cấp là null, mảng rỗng, mảng không có phần tử nguyên âm.

```
public class NullArrayException extends Exception {
    public NullArrayException(String message) {
        super(message);
    }
}

public class MyArray {
    public double getAverageNegative(int[] a) throws NullArrayException {
        // sau khi có bộ test, lập trình viên sẽ cài đặt để cho phương thức này "vượt" các test
        if (a == null) throw new NullArrayException("Input array is null.");
        int count = 0;
        double result = 0.0;
        for (int n : a)
            if (n < 0) { result += n; count++; }
        return count == 0 ? 0 : result / count;
    }
}
```

- Ngăn chặn lỗi (bug) ngay từ đầu.

Cùng ví dụ trên, nếu mảng không có phần tử nguyên âm, dẫn đến khi tính trung bình cộng bạn có thể gặp lỗi chia cho 0.

- Đơn giản, dễ áp dụng, dễ đưa vào quy trình.
- Dễ bảo trì code, code được phân thành các đơn vị ổn định do đã được test cẩn thận.
- Dễ cấu trúc lại code (refactoring), sau khi cấu trúc lại code, chỉ cần kiểm tra đơn vị code đó đã vượt các test viết sẵn.

b) Cơ chế

TDD được thực hiện qua ba bước: "code once - test twice"

- TEST (RED): nhân viên kiểm thử (tester) viết các trường hợp test, việc chạy phải cho kết quả sai, vì test chạy trên code chưa viết. Khung sườn code được tạo từ khâu thiết kế, các test được viết trước (gọi là "test first") trên khung sườn code đó.
- CODE: lập trình viên cài đặt cho code, gọi là viết logic.

Lập trình viên viết code đầy đủ vào khung sườn code được cung cấp, mục tiêu là vượt qua các trường hợp test.

- TEST (GREEN): lập trình viên chạy test một lần nữa để chắc code mình viết đã vượt qua các trường hợp test.

Việc vượt qua nhiều trường hợp test có thể làm cho code trở nên rườm rà, phức tạp. Sau đó, lập trình viên tiến hành cấu trúc lại code (refactoring), thường bằng các công cụ do IDE cung cấp, để code đạt hiệu năng cao.

Nhiều tài liệu gọi ba bước của TDD là: RED - GREEN - Refactor.

2. JUnit

Được Erich Gamma và Kent Beck phát triển, JUnit là framework kiểm thử đơn vị (unit test) của Java. Khái niệm *đơn vị* (unit) thường là một lớp Java hoặc một phần của lớp. JUnit đã được tích hợp vào nhiều IDE: Eclipse, IntelliJ IDEA, NetBeans hoặc dễ dàng tích hợp vào dự án như một dependency từ Maven. Tài liệu này giới thiệu JUnit 4.x.

a) Thứ tự test

- Môi trường test gọi là test fixture. Một tác vụ test phải độc lập với tác vụ test khác, vì vậy, môi trường test cần được khởi tạo trước mỗi tác vụ test và được dọn dẹp sau khi tác vụ đó hoàn tất. Các annotation được thiết kế để làm việc với môi trường test:

@BeforeClass Chạy một lần trong lớp test, trước khi chạy các phương thức test của lớp test.

@AfterClass Chạy một lần trong lớp test, sau khi chạy các phương thức test của lớp test.

@Before Chạy trước mỗi phương thức test, khởi tạo cho phương thức kiểm thử. Thường cấp phát và khởi gán cho các đối tượng.

@After Chạy sau mỗi phương thức test, dọn dẹp cho phương thức kiểm thử.

- Phương thức test, đánh dấu bằng @Test, sẽ chạy giữa hai phương thức @Before và @After. Nếu phương thức test được đánh dấu là @Ignore, phương thức này sẽ được bỏ qua khi test.

```
import org.junit.After;
import org.junit.AfterClass;
import static org.junit.Assert.assertEquals;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class MyArrayTest {
    private MyArray o;

    @BeforeClass public static void before() {
        System.out.println("Before class.");
    }

    @AfterClass public static void after() {
        System.out.println("After class.");
    }

    @Before public void setUp() {
        System.out.println("Before test. Object initialization.");
        o = new MyArray();
    }

    @After public void tearDown() {
        System.out.println("After test. Clean up.");
    }

    @Test public void testGetAverageNegativeNormal() throws NullPointerException {
        int[] a = new int[] {-2, 3, 4, -5};
        double expected = -3.5;
        assertEquals("Not equals!", expected, o.getAverageNegative(a), 1E-5);
    }

    @Test public void testGetAverageNegativeNoneNegative() throws NullPointerException {
        int[] a = new int[] {2, 3, 4, 5};
        double expected = 0;
        assertEquals("Not equals!", expected, o.getAverageNegative(a), 1E-5);
    }
}
```

b) Phương thức test và các assert

Phương thức test sử dụng các khẳng định (assert) để test, các khẳng định sai sẽ được ghi nhận lại. Các assert này đều thuộc lớp org.junit.Assert. Để test phương thức foo() chẳng hạn, phương thức testFoo() thường được viết như sau:

- Kiểm tra các tiên điều kiện (precondition) của foo() (có thể bỏ qua).

- Triệu gọi foo().

- Kiểm tra các hậu điều kiện (postcondition) của foo().

JUnit cung cấp một tập khá lớn các assert, quan trọng nhất là:

```
import static org.junit.Assert.*;
assertArrayEquals(expected, actual); // kiểm tra xem hai mảng expected và actual có bằng nhau.
assertEquals(expected, actual); // kiểm tra expected và actual có bằng nhau.
```

```

assertTrue(condition);           // kiểm tra condition có true.
assertFalse(condition);          // kiểm tra condition có false.
assertNull(object);              // kiểm tra object có null.
assertNotNull(object);           // kiểm tra object có khác null.
assertSame(expected, actual);    // kiểm tra expected và actual có tham chiếu đến cùng một đối tượng.

```

Những assert phức tạp hơn được cung cấp trong các gói khác, hoặc có thể tùy biến, giúp các khẳng định trở nên linh động. Ngoài ra, phương thức fail() thường dùng để báo hiệu một exception được ném ra:

```

public void testIndexOutOfBoundsException() {
    int[] a = new int[10];
    try {
        int n = a[a.length];
        fail("Should raise an ArrayIndexOutOfBoundsException");
    } catch (ArrayIndexOutOfBoundsException e) { }
}

```

c) Chạy lớp test

Để chạy các phương thức test của lớp test, dùng phương thức runClasses() của lớp JUnitCore. Kết quả test đặt trong đối tượng Result. Nhận các trường hợp thất bại bằng cách dùng phương thức getFailures() của đối tượng Result. Nhận kết quả thành công bằng cách dùng phương thức wasSuccessful() của đối tượng Result.

```

import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(MyArrayTest.class);
        for (Failure failure : result.getFailures())
            System.out.println(failure.toString());
        System.out.println(result.wasSuccessful());
    }
}

```

Ta cũng chạy TestSuite bằng cách trên. Tuy nhiên, do JUnit được tích hợp với IDE, việc chạy và cho kết quả được thực hiện trong GUI của IDE đó. Ngoài ra, @RunWith cho phép làm việc với các Runner của bên thứ ba như SpringJUnit4ClassRunner, MockitoJUnitRunner, ...

3. TestSuite

a) TestSuite

Một TestSuite (bộ test) bao gồm một hoặc nhiều lớp test, hoặc các TestSuite khác. TestSuite giúp phân khối hệ thống phần mềm cần test. Chạy TestSuite bằng Runner được khai báo trong @RunWith.

```

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({ATest.class, BTest.class})
public class ATestSuite {
}

```

b) Category

Các phương thức test trong các lớp test thuộc một TestSuite có thể phân thành nhiều loại. JUnit cho phép test riêng từng loại, gọi là test theo category.

Ví dụ: TestSuite ATestSuite chứa lớp ATest. Lớp ATest có một số phương thức test, thuộc về một trong hai loại ACategory và BCategory hay cả hai.

```

public interface ACategory { }
public interface BCategory { }

import org.junit.Test;
import org.junit.experimental.categories.Category;
public class ATest {
    @Test
    @Category(ACategory.class)
    public void testMethod1() { }

    @Test
    @Category(BCategory.class)
    public void testMethod2() { }

    @Test
    @Category({ACategory.class, BCategory.class}) // phương thức thuộc cả hai loại category
    public void testMethod3() { }
}

```


Bây giờ ta chỉ kiểm tra các phương thức thuộc category ACategory trong TestSuite:

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.experimental.categories.Categories;
import org.junit.experimental.categories.Categories.ExcludeCategory;
import org.junit.experimental.categories.Categories.IncludeCategory;

@RunWith(Categories.class)
@IncludeCategory(ACategory.class)
@ExcludeCategory(BCategory.class) // không kiểm tra phương thức testMethod3()
@Suite.SuiteClasses({ATest.class})
public class ATestSuite {
}
```

c) Test với tham số (parameterized test)

JUnit4 cho phép chạy test nhiều lần với các bộ dữ liệu đầu vào khác nhau, gọi là test với tham số.

- Lớp test được đánh dấu bằng @RunWith(Parameterized.class).
- Một phương thức static đánh dấu với @Parameters trả về các bộ dữ liệu đầu vào.
- Constructor nhận tham số tương đương một "dòng" dữ liệu đầu vào, lưu vào biến lớp.
- Phương thức test, dùng các biến lớp làm dữ liệu đầu vào.

```
import static org.junit.Assert.assertEquals;
import java.util.*;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)
public class ParameterizedTests {
    private MyArray o;
    private int[] input;
    private double expected;

    @Parameters
    public static List<Object[]> data() {
        return Arrays.asList(new Object[][] {
            { new int[] { -2, 3, 4, -5 }, -3.5 },
            { new int[] { -2, 3, -4, -5 }, -3.66667 },
            { new int[] { 2, 3, 4, 5 }, 0 }
        });
    }

    public ParameterizedTests(int[] input, double expected) {
        this.input = input;
        this.expected = expected;
    }

    @Before public void setUp() {
        o = new MyArray();
    }

    @Test public void test() throws NullArrayException {
        assertEquals(expected, o.getAverageNegative(input), 1E-5);
    }
}
```

d) Theory

JUnit4 cung cấp khái niệm theory cho phép kiểm tra các assert trong điều kiện linh động hơn, ví dụ dưới một số giả định cho trước. Các bước dùng theory:

- Lớp test được đánh dấu bằng @RunWith(Theories.class).
- Một phương thức static đánh dấu với @DataPoints trả về các bộ dữ liệu đầu vào.
- Phương thức test, đánh dấu với @Theory, dùng các phương thức của Assume để đặt giả định và dùng assert để kiểm tra với từng bộ dữ liệu đầu vào.

```
import org.junit.Assume;
import org.junit.experimental.theories.DataPoints;
import org.junit.experimental.theories.Theories;
import org.junit.experimental.theories.Theory;
import org.junit.runner.RunWith;
import static org.junit.Assert.assertTrue;
```

```
@RunWith(Theories.class)
public class TheoryTests {
    @DataPoints
    public static int[][] data() {
        return new int[][] {{-2, 3, 4, -5}, {2, 3, 4, 5}, {2, -3, 4, 5}};
    }

    @Theory
    public void averageIsNotPositive(int[] a) throws NullArrayException {
        System.out.println("Test " + Arrays.toString(a));
        MyArray o = new MyArray();
        Assume.assumeTrue(a != null && a.length > 0);
        assertTrue(o.getAverageNegative(a) <= 0);
    }
}
```

4. Exception và Timeout

Ngoài việc khẳng định một phương thức hoạt động bình thường, JUnit còn cho phép khẳng định một phương thức có ném ra exception như mong đợi hay không, trong tình huống đặc biệt. Có hai cách:

- Dùng tham số expected của @Test

```
@Test (expected=NullArrayException.class)
public void testGetAverageNegativeNullArray() throws NullArrayException {
    o.getAverageNegative(null);
}
```

Nếu phương thức được test ném ra exception, phương thức test xem như thành công.

- Dùng @Rule với ExpectedException, bạn có thể kiểm tra thông tin chi tiết hơn, như thông điệp lỗi được ném ra.

```
import org.junit.Rule;
import org.junit.rules.ExpectedException;
// ...
@Rule
public ExpectedException thrown = ExpectedException.none();
@Test
public void testGetAverageNegativeNullArray() throws NullArrayException {
    thrown.expect(NullArrayException.class);
    thrown.expectMessage("Input array is null.");
    o.getAverageNegative(null);
}
```

Nếu một phương thức test chạy quá lâu, bạn có thể thiết lập để xem như nó tự động thất bại. Có hai cách:

- Áp dụng cho phương thức test: dùng tham số timeout của @Test, chỉ định thời gian timeout, tính bằng miligiây. Phương thức test chạy vượt quá thời gian này xem như thất bại, sẽ ném một exception.

- Áp dụng cho lớp test: dùng @Rule với Timeout

Luật Timeout áp dụng cùng một timeout cho tất cả phương thức test của lớp test, ưu tiên hơn tham số timeout của @Test nếu có thiết lập trước.

```
import org.junit.Rule;
import org.junit.rules.Timeout;
// ...
@Rule
public Timeout timeout = Timeout.millis(50); // 50 miligiây tối đa cho mỗi phương thức test

@Test (timeout=60)
public void testGetAverageNegativeTimeout() throws NullArrayException {
    a = new int[1_000_000];
    for (int i = 0; i < a.length; i++)
        a[i] = i * (int)Math.pow(-1, i);
    double expected = 0;
    assertEquals("Not equals!", expected, o.getAverageNegative(a), eps);
}
```

Log

Đôi khi chúng ta cần lưu các hành vi của ứng dụng, để ghi nhận các tình huống bất thường hoặc lỗi, theo dõi thông tin sử dụng và gỡ lỗi (debug). Mức độ chi tiết của thông tin ghi nhận tùy theo mức độ yêu cầu công việc. Hệ thống ghi nhận như vậy gọi là hệ thống nhật ký (log system). Java cung cấp một giải pháp có sẵn, trong gói java.util.logging.

1. LogManager và Logger

Hệ thống log được quản lý tập trung thông qua lớp LogManager, truy cập bởi phương thức: LogManager.getLogManager().

Các Logger được tạo từ phương thức Logger.getLogger(), tổ chức thành cây phân nhánh theo namespace. LogManager truy cập các thực thể Logger này bằng phương thức LogManager.getLogger().

Trong ví dụ sau, LogManager truy cập Logger toàn cục có sẵn cho ứng dụng.

```
public class Main {
    static final Logger logger = LogManager.getLogger().getLogger(Logger.GLOBAL_LOGGER_NAME);
    public static void main(String[] args) {
        logger.log(Level.INFO, "My log message");
    }
}
July 7, 2016 2:40:12 PM com.twe.samples.Main main
INFO: My log message
```

2. Các mức độ log

Mức độ log (logging level) quyết định chi tiết của thông tin ghi nhận. Mỗi mục log liên kết với một mức độ, được thiết lập bằng phương thức `setLevel()` của `Logger`.

```
logger.setLevel(Level.INFO);
```

Mỗi mức độ có một trị số, bao gồm 7 mức độ log cơ bản và 2 mức độ đặc biệt cho `Logger`. Java cũng cho phép định nghĩa tùy biến mức độ log, nhưng nói chung nên tránh.

Level	Trị	Mô tả
OFF	<code>Integer.MAX_VALUE</code>	Logger không ghi nhận
SEVERE	1000	Thiệt hại nghiêm trọng
WARNING	900	Vấn đề tiềm tàng
INFO	800	Thông tin tổng quát
CONFIG	700	Thông tin cấu hình
FINE	500	Thông tin tổng quát cho người phát triển
FINER	400	Thông tin chi tiết cho người phát triển
FINEST	300	Thông tin chuyên biệt cho người phát triển
ALL	<code>Integer.MIN_VALUE</code>	Logger ghi nhận mọi thứ

3. Các phương thức log

`Logger` cung cấp các phương thức log.

- Phương thức đơn giản

```
logger.log(Level.SEVERE, "Whoops!");
```

```
July 7, 2016 2:40:12 PM com.twe.samples.Main main
```

```
SEVERE: Whoops!
```

- Phương thức tiện dụng (convenience)

Tên phương thức ngầm chỉ mức độ log:

Phương thức	Level
<code>severe</code>	<code>Level.SEVERE</code>
<code>warning</code>	<code>Level.WARNING</code>
<code>info</code>	<code>Level.INFO</code>
<code>config</code>	<code>Level.CONFIG</code>
<code>fine</code>	<code>Level.FINE</code>
<code>finer</code>	<code>Level.FINER</code>
<code>finest</code>	<code>Level.FINEST</code>

```
logger.severe("Whoops!");
```

```
July 7, 2016 2:40:12 PM com.twe.samples.Main main
```

```
SEVERE: Whoops!
```

- Phương thức chính xác (precise)

Thường dùng khi muốn truyền tên lớp và phương thức gọi log.

```
logger.logp(Level.SEVERE, "com.twe.samples.Main", "main", "It broke!");
```

```
July 7, 2016 2:40:12 PM com.twe.samples.Main main
```

```
SEVERE: It broke!
```

- Phương thức chính xác và tiện dụng

Ghi nhận một thông điệp được định nghĩa trước. Luôn được ghi nhận như cấp độ `Level.FINER`

Phương thức	Thông điệp
<code>entering</code>	<code>ENTRY</code>
<code>exiting</code>	<code>RETURN</code>

```
ConsoleHandler handler = new ConsoleHandler();
```

```
logger.addHandler(handler);
```

```
logger.setLevel(Level.FINER);
```

```
handler.setLevel(Level.ALL);
```

```
logger.entering("com.twe.samples.Main", "main");
```

```
logger.logp(Level.WARNING, "com.twe.samples.Main", "main", "Empty function!");
```

```
logger.exiting("com.twe.samples.Main", "main");
```

```
July 7, 2016 2:40:12 PM com.twe.samples.Main main
```

```
FINER: ENTRY
```

```
July 7, 2016 2:40:12 PM com.twe.samples.Main main
```

```
WARNING: Empty function!
```

```
July 7, 2016 2:40:12 PM com.twe.samples.Main main
```

```
FINER: RETURN
```

- Phương thức có thông điệp tham số hóa (parameterized message)

Một số phương thức như `log()`, `logp()` có thông điệp được tham số hóa. Các tham số chỉ định các "giữ chỗ" xuất hiện bên trong thông điệp, chúng có dạng `{N}` với chỉ số N tính từ 0.

```
logger.log(Level.INFO, "{0} is {1} days from {2}.", new Object[] {"Wed", 2, "Fri"});
July 7, 2016 2:40:12 PM com.twe.samples.Main main
INFO: Wed is 2 days from Fri.
```

Ví dụ khác.

```
public static void doStuff(String first, String last) {
    ConsoleHandler handler = new ConsoleHandler();
    logger.addHandler(handler);
    logger.setLevel(Level.FINER);
    handler.setLevel(Level.ALL);
    logger.entering("com.twe.samples.Main", "main", new Object[]{first, last});
    String result = "[" + first + " " + last + "]";
    logger.exiting("com.twe.samples.Main", "main", result);
}
```

gọi: `doStuff("Barack", "Obama");`

```
July 7, 2016 2:40:12 PM com.twe.samples.Main main
FINER: ENTRY Barack Obama
July 7, 2016 2:40:12 PM com.twe.samples.Main main
FINER: RETURN [Barack Obama]
```

4. Tổ chức hệ thống log

Để cung cấp tính linh hoạt, hệ thống log được chia thành các đơn thể. Mỗi đơn thể xử lý một tác vụ chỉ định.

Có ba đơn thể chính:

- Logger

Chịu trách nhiệm đón nhận các thông điệp cần ghi nhận từ ứng dụng.

- Handler

Chịu trách nhiệm xuất thông tin ghi nhận cho Logger. Mỗi Logger có thể có nhiều Handler. Java cung cấp các Handler có sẵn để thêm vào Logger bằng phương thức `Logger.addHandler()`.

- Formatter

Chịu trách nhiệm định dạng thông tin ghi nhận cho Handler. Mỗi Handler có một Formatter. Java cung cấp các Formatter có sẵn để thêm vào Handler bằng phương thức `Handler.setFormatter()`.

```
public class Main {
    static final Logger logger = LogManager.getLogger().getLogger(Logger.GLOBAL_LOGGER_NAME);
    public static void main(String[] args) {
        Handler handler = new ConsoleHandler();
        Formatter formatter = new SimpleFormatter();
        handler.setFormatter(formatter);
        logger.addHandler(handler);
        logger.setLevel(Level.INFO);
        logger.log(Level.INFO, "We're Logging!");
    }
}
July 7, 2016 2:40:12 PM com.twe.samples.Main main
INFO: We're Logging!
```

a) Các handler có sẵn

Các handler có sẵn thừa kế trực tiếp hoặc gián tiếp từ Handler:

- ConsoleHandler: viết đến System.err

- StreamHandler: viết đến OutputStream chỉ định.

- SocketHandler: viết đến socket chỉ định.

- FileHandler: viết đến một hoặc nhiều tập tin (nhóm xoay vòng các tập tin). Hỗ trợ đặt tên tập tin theo mẫu chỉ định.

Trị	Ý nghĩa
/	Dấu phân cách \, khác nhau tùy theo hệ nền
%t	Thư mục tạm
%h	Thư mục của người dùng (home)
%g	Sinh log xoay vòng

Ví dụ: `FileHandler fh = new FileHandler("%h/foo_%g.log", 1000, 4);`

Xoay vòng trong nhóm 4 tập tin `foo_0.log`, `foo_1.log`, `foo_2.log` và `foo_3.log`. Mỗi tập tin tối đa 1000 bytes. Chúng được lưu trữ như sau:

Trên Unix: `/var/users/barack/foo_0.log`

Trên Windows: `C:\Users\barack\foo_0.log`

b) Các formatter có sẵn

Java cung cấp hai formatter có sẵn, thừa kế trực tiếp từ Formatter:

+ XMLFormatter: định dạng nội dung như XML với thành phần root là `<log>`.

+ SimpleFormatter: định dạng nội dung như văn bản đơn giản. Định dạng có thể tùy biến.

```
System.setProperty("java.util.logging.SimpleFormatter.format",
    "[%1$tb %1$td, %1$tY %1$tL:%1$tM:%1$tS %1$Tp] %2$s %n %4$s: %5$s%6$s%n");
Handler handler = new ConsoleHandler();
Formatter formatter = new SimpleFormatter();
handler.setFormatter(formatter);
logger.addHandler(handler);
logger.setLevel(Level.INFO);
logger.log(Level.INFO, "We're Logging!");
```

Ý nghĩa của chuỗi định dạng:

```
[%1$tb %1$td, %1$tY %1$tL:%1$tM:%1$tS %1$Tp] %2$s %n %4$s: %5$s%6$s%n
```

1: date, 2: source, 3: logger, 4: level, 5: message, 6: thrown

kết quả:

```
[July 7, 2016 2:43:12 PM] com.twe.samples.Main main
Info: This is the message.
```

5. Tập tin cấu hình log

Thay vì cấu hình trong code như trên, bạn có thể thiết lập thông tin cấu hình trong một tập tin, theo định dạng tập tin properties chuẩn, ví dụ tập tin log.properties

```
# Global logging properties
handlers = java.util.logging.ConsoleHandler, java.util.logging.FileHandler
.level = ALL
# Handlers
# --- ConsoleHandler ---
java.util.logging.ConsoleHandler.level = ALL
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
# --- FileHandler ---
java.util.logging.FileHandler.level = ALL
java.util.logging.FileHandler.pattern = ./main_%g.log
# Formatters
java.util.logging.SimpleFormatter.format = %1$tY-%1$tM-%1$tD %1$tH:%1$tM:%1$tS.%1$tL %4$-7s [%3$s] (%2$s) %5$s %6$s%n
Tập tin này được đặt trong thư mục gốc của dự án. Thuộc tính java.util.logging.config.file chỉ định tập tin cấu hình này.
public class Main {
    public static void main(String[] args) {
        System.setProperty("java.util.logging.config.file", "log.properties");
        Logger logger = Logger.getLogger(Main.class.getSimpleName());
        logger.info("We're Logging!");
    }
}
2016-07-07 02:43:12.490 INFO [Main] (com.twe.samples.Main main) We're Logging!
```

Regular Expression

1. Khái niệm

Biểu thức chính quy (Regular expression, viết tắt regex) là tập các ký hiệu định nghĩa một mẫu văn bản (pattern) dùng so trùng chuỗi trong một văn bản. Chuỗi so trùng là chuỗi theo đúng quy luật được mô tả trong regex.

Để dễ trình bày, chuỗi regex đặt trong cặp «», chuỗi nhập đặt trong cặp "", còn chuỗi so trùng đặt trong cặp [].

Ví dụ: «reg(ular expressions)?ex(ples)?» so trùng [regular expressions], [regular expression], [regex], [regexp], [regexes] nếu chúng có trong chuỗi nhập.

- Ký tự thường dùng

.	Một ký tự bất kỳ, ngoại trừ ký tự newline.
\d	Một ký tự số (digit). Tương đương [0-9]
\w	Một ký tự word, chữ cái, số hoặc ký tự gạch dưới. Tương đương [a-zA-Z0-9_]
\s	Một ký tự khoảng trắng (whitespace). Tương đương [\t\n\x0B\f\r]
\D	Một ký tự không phải số. Tương đương [^0-9]
\W	Không phải ký tự word. Tương đương [^a-zA-Z0-9_]
\S	Một ký tự không phải khoảng trắng. Tương đương [^ \t\n\x0B\f\r]

- Các lớp ký tự theo chuẩn POSIX

Java cũng hỗ trợ các lớp ký tự theo chuẩn POSIX nhưng sử dụng cú pháp khác.

POSIX	Java	Ý nghĩa
[[:digit:]]	\p{Digit}	[0-9]
[[:lower:]]	\p{Lower}	[a-z]
[[:upper:]]	\p{Upper}	[A-Z]
[[:alpha:]]	\p{Alpha}	[\p{Lower}\p{Upper}]
[[:alnum:]]	\p{Alnum}	[\p{Alpha}\p{Digit}]
[[:punct:]]	\p{Punct}	dấu câu, !"#%&'()*+,-./:;<=>?@[\]^_`{ }~
[[:space:]]	\p{Space}	[\t\n\x0B\f\r]
[[:blank:]]	\p{Blank}	space, tab. [\t]
[[:print:]]	\p{Print}	ký tự in được, space. [\p{Alnum}\p{Punct}]\b
[[:graph:]]	\p{Graph}	ký tự in được, không dấu câu. [\p{Alnum}\p{Punct}]
[[:cntrl:]]	\p{Cntrl}	ký tự điều khiển, không in được. [\x00-\x1F\x7F]

`[:xdigit:]` `\p{XDigit}` `[0-9A-Fa-f]` (hex)

- Ký hiệu số lượng (quantifiers)

* So trùng 0 hoặc nhiều lần. Tương đương `{0,}`
 + So trùng 1 hoặc nhiều lần. Tương đương `{1,}`
 ? So trùng 0 hoặc 1 lần. Tương đương `{0,1}`
`{n}` So trùng đúng *n* lần (*n* dương).
`{min,}` So trùng ít nhất *min* lần (*min* dương).
`{min,max}` So trùng tối thiểu *min* lần, tối đa *max* lần (*min* và *max* dương).

- Quy tắc viết

`^regex` So trùng biên, so trùng tại đầu chuỗi
`regex$` So trùng biên, so trùng tại cuối chuỗi
`[a-c0-2]` So trùng một ký tự trong tập ký tự: dãy từ a đến c và dãy từ 0 đến 2
`[^aeiou]` Không so trùng các ký tự nguyên âm
`[[0-9][A-F][a-f]]` Phần hội của `[0-9]`, `[A-F]` và `[a-f]`, so trùng một ký tự hex. Tương đương `[0-9A-Fa-f]`
`[a-c&&[c-f]]` Phần giao của `[a-c]` và `[c-f]`, nghĩa là so trùng c
`[a-z&&[^aeiou]]` Phần trừ của `[a-z]` và `[aeiou]`, nghĩa là so trùng các phụ âm
`x|y` So trùng x hoặc y
`xy` So trùng x theo ngay sau là y
`\b` So trùng biên của từ. `\B` không so trùng biên
`\A` So trùng đầu văn bản. `\Z` so trùng cuối văn bản
`\u00A9` Unicode escaped ©
`\x` Dùng khi x là một trong các metacharacter sau: `\. * + - { } [] ^ $ | ? () : ! =`

Chú ý, metacharacter trong tập ký tự giữa `[]` xem như đã có ký tự escape, ngoại trừ `]-^\\` (tùy vị trí)

- Ví dụ minh họa

Để viết regex đúng, bạn cần xác định được quy luật chung từ các chuỗi cần so trùng.

+ Tìm một từ trong chuỗi nhập, cho dù viết sai: `«li[cs]en[cs]e»`

+ Hexadecimal: `«0[xX]\p{XDigit}+»`

+ Số thực dương `«\d+(\. \d+)?»`

+ Với chuỗi nhập "This island is beautiful"

`«\bis\b»` so trùng "This island [is] beautiful"

`«\Bis\b»` so trùng "Th[is] island is beautiful"

`«\bis\B»` so trùng "This [is]land is beautiful"

+ `«(sport|music|book|movie)»` so trùng một trong các chuỗi: sport, music, book, movie, trong chuỗi nhập.

- Một số lỗi thường gặp

+ `«Iraq[^u]»` không so trùng [Iraq], nhưng so trùng [Iraq] trong chuỗi nhập

Giải thích: regex được hiểu là "Iraq" theo sau bởi (một ký tự) không phải là u; không phải: "Iraq" không theo sau bởi u.

+ `«[x^]»` so trùng x hoặc ^

`«[]x]»` so trùng] hoặc x

+ `«[^\d\s]»` không tương đương `«[\D\S]»`.

`«[^\d\s]»` không (số hoặc space), tương đương (không số) và (không space)

`«[\D\S]»` (không số) hoặc (không space).

+ `«[50-99]»` so trùng các số từ 0 đến 9, tương đương `[0-9]`, không phải so trùng các số từ 50 đến 99.

+ `«[abc[u-z]]»` so trùng một ký tự trong tập (a, b, c, u, v, x, y, z); khác `«[abc][u-v]»` so trùng 2 ký tự.

2. Regex engine

Một regex engine phải được cài đặt để thực thi việc so trùng. Trong trường hợp dùng Java, regex engine được tích hợp trong Java NIO API.

Có hai loại engine: *text-directed* (DFA) và *regex-directed* (NFA). Loại regex-directed phổ biến hơn, cung cấp các tính năng hữu ích như *lazy quantifier* và *backreference*. Regex-directed luôn trả về kết quả so trùng trái nhất (leftmost) và sớm nhất (eager) thấy được. Nếu so trùng không thành công, một quá trình truy ngược (backtracking) diễn ra để việc so trùng tiến hành từ ký tự kế tiếp trong chuỗi. Bạn có thể tham khảo ứng dụng **The Regex Coach** (www.weitz.de/regex-coach/), công cụ này cho phép chạy regex engine từng bước, giúp bạn hiểu rõ hoạt động của regex engine.

- Greedy và lazy

Khi dùng các ký hiệu số lượng, hãy nhớ là regex engine chạy với chế độ greedy. Nghĩa là regex engine thử so trùng tiếp phần có ký hiệu số lượng, không dừng lại sau khi so trùng thành công.

Tắt chế độ này và chuyển sang chế độ lazy (còn gọi là ungreedy hay reluctant), bằng cách đặt dấu ? sau ký hiệu số lượng.

Ví dụ:

+ Dùng regex `«Nov(ember)?»` với "November" sẽ so trùng [November], không phải [Nov] trong chuỗi nhập. Để chỉ so trùng [Nov], dùng regex `«Nov(ember)??»`

+ Dùng regex `«<.+>»` với chuỗi nhập "The first test" sẽ so trùng [first], không phải []. Để chỉ so trùng [], dùng regex `«<.+?>»`

Giải thích:

Trong chế độ greedy, regex engine sẽ so trùng `«<»` với ký tự [<] đầu tiên thấy được, so trùng tiếp `«.+»` cho đến cuối chuỗi. Tiếp theo, do không so trùng được `«>»`, regex engine truy ngược lại và dừng khi so trùng `«>»` với ký tự [>] đầu tiên tìm thấy tính từ cuối chuỗi.

Trong chế độ lazy, regex engine sẽ so trùng «<» với ký tự [<] đầu tiên thấy được, so trùng tiếp «.» với [E]. Tiếp theo, so trùng «>» nhưng thất bại, regex engine truy ngược lại so trùng tiếp «.» với [M]. Cuối cùng, regex engine dừng khi so trùng «>» với ký tự [>] của .

Bạn cũng có thể dùng cách khác tốt hơn, ví dụ regex «<[>]+>»

Để phân biệt các chế độ, các ký hiệu số lượng (quantifiers) chia thành ba nhóm:

+ greedy (? , * hoặc +) thử so trùng dài nhất có thể.

+ lazy (reluctant): (? , *? hoặc +?) thử so trùng ngắn nhất có thể.

+ possessive (?+ , *+ , hoặc ++) giống greedy nhưng chỉ thử một lần để tìm thấy so trùng dài nhất, trong lúc greedy dùng nhiều lần thử.

Ví dụ: với chuỗi nhập "wend rend end"

greedy «.*end» so trùng "[wend rend end]"

lazy «.*?end» so trùng "[wend][rend][end]"

possessive «.*+end» không xuất kết quả, nó tiêu thụ hết chuỗi nhập, không truy ngược như greedy nên không so trùng phần «end».

Ví dụ: với chuỗi nhập "1011"

greedy «1?» lazy «1??» possessive «1?+»

[1] 0 -> 0 [] 0 -> -1 [1] 0 -> 0

[] 1 -> 0 [] 1 -> 0 [1] 2 -> 2

[1] 2 -> 2 [] 2 -> 1 [1] 3 -> 3

[1] 3 -> 3 [] 3 -> 2

[] 4 -> 3 [] 4 -> 3

Nhận xét: lazy so trùng ngắn nhất có thể, trong trường hợp này là không so trùng được gì; possessive không so trùng chuỗi rỗng (zero-length).

3. Sử dụng regex

a) Kiểm tra dữ liệu nhập hợp lệ

Dùng phương thức matches(regex) của String là cách thường dùng nhất để so trùng chuỗi. Phương thức này chỉ so trùng chuỗi nhập với regex chỉ định.

```
String s;
Scanner keyboard = new Scanner(System.in);
while (true) {
    System.out.print("Enter grade: ");
    s = keyboard.nextLine();
    if (!s.matches("\\d+(\\.\\d+)?")) System.err.out("Not positive real number!");
    else if (Double.parseDouble(s) > 10) System.err.out("Invalid grade!");
    else break;
}
System.out.printf("Grade: %.1f%n", Double.parseDouble(s));
```

b) Tách dữ liệu

Chuỗi nhập là tổ hợp các token và delimiter (ký tự phân cách). Vì delimiter có thể có nhiều loại khác nhau, tiện nhất là dùng regex mô tả chúng.

- Dùng phương thức useDelimiter(regex) của Scanner. Các token được tách bằng các phương thức next() của Scanner.

```
String input = "Stop! Are we at the end?";
Scanner scanner = new Scanner(input);
scanner.useDelimiter("[\\p{Punct}\\s]+");
while (scanner.hasNext()) System.out.println(scanner.next());
- Dùng phương thức split(regex) của String. Các token được tách vào một mảng các String.
String input = "Stop! Are we at the end?";
String[] list = input.split("[\\p{Punct}\\s]+");
for (String s : list) System.out.println(s);
```

c) Tìm và thay thế

Phương thức replaceAll(regex, replace) của String thay thế tất cả regex so trùng trong chuỗi nhập (chuỗi gọi phương thức replaceAll) với chuỗi replace. Chuỗi replace có thể tham chiếu đến các phần của chuỗi nhập bằng cách dùng tham chiếu ngược (backreference) thông qua \$1, \$2, ... \$0 (toàn chuỗi so trùng). Trong chuỗi thay thế, \$ không kèm với số theo sau sẽ ném IllegalArgumentException hoặc nếu kèm theo số lớn hơn số nhóm sẽ ném IndexOutOfBoundsException.

4. Pattern và Matcher

Các lớp quan trọng thuộc gói java.util.regex:

- Pattern mô hình hóa một mẫu (pattern), các phương thức:

```
static Pattern compile(String regex, int flags)
```

Phương thức static, biên dịch regex thành đối tượng Pattern. Nếu regex có cú pháp không chính xác, ném ra exception PatternSyntaxException. Cờ flags tùy chọn dùng thiết lập chế độ so trùng: CANON_EQ, CASE_INSENSITIVE, COMMENTS, DOTALL, LITERAL, MULTILINE, UNICODE_CASE, và UNIX_LINES.

Bạn cũng có thể chỉ định chế độ so trùng bằng cách thêm vào đầu regex: (?i) cho CASE_INSENSITIVE, (?m) cho MULTILINE, (?s) cho DOTALL và (?u) cho UNICODE_CASE.

```
Matcher matcher(CharSequence input)
```

Trả về đối tượng Matcher, sẽ so trùng chuỗi nhập input với regex đã biên dịch.

`static boolean matches(String regex, CharSequence input)`

Biên dịch `regex` và thử so trùng chuỗi nhập `input` với `regex` đã biên dịch.

`String[] split(CharSequence input)`

Tách chuỗi nhập `input` thành mảng chuỗi chứa các chuỗi so trùng với `regex` đã biên dịch.

- `Matcher`, phương thức `matcher(String input)` của `Pattern` nhận chuỗi nhập và trả về đối tượng `Matcher`. Các phương thức của `Matcher`, thường sử dụng trong vòng lặp, giúp định vị các group so trùng được.

`boolean find()` thử tìm group so trùng kế tiếp trong chuỗi nhập.

`String group()` trả về group so trùng.

`String group(int index)` trả về group so trùng theo chỉ số group.

`String group(String name)` trả về group so trùng theo tên group.

`int start()` trả về vị trí đầu của group so trùng.

`int end()` trả về vị trí cuối của group so trùng.

`int groupCount()` đếm số group so trùng được.

`boolean matches()` giống `matches()` của `String`. Xem chuỗi nhập trong `Matcher` có so trùng `regex` chỉ định.

`boolean lookingAt()` giống `startsWith()` của `String`. Xem chuỗi nhập trong `Matcher` có bắt đầu bằng `regex`.

5. Nhóm (group)

Một `regex` có thể tách thành vài nhóm bằng cặp `()`, nhóm có thể lồng nhau.

Các group được đánh chỉ số. Toàn bộ `regex` là group 0, chỉ số các group tăng từ trái sang phải, từ ngoài vào trong.

Java 7 cho phép đặt tên cho group với cú pháp `(?<group_name>regex)`.

`(abc)` capture group

`\n` tham chiếu ngược (backreference) đến group `#n`, tham chiếu ngược là so trùng với nhóm `regex` trước đó

`(?:abc)` non-capturing group, không tính là một group

`(?=abc)` positive lookahead, ưu tiên kiểm tra trước `regex abc`, nếu không tìm thấy sẽ dừng

`(?!abc)` negative lookahead, ưu tiên kiểm tra trước không tồn tại `regex abc`, nếu có sẽ dừng

Ví dụ: dùng tham chiếu ngược để phát hiện trùng từ.

```
String input = "The is is a string";
Pattern pattern = Pattern.compile("\\b(\\w+)\\s+\\1\\b");
Matcher matcher = pattern.matcher(input);
while( matcher.find())
System.out.println("Found duplicate: " + matcher.group());
```

Ví dụ:

```
String input = "int a = 100; double b = 10.2;";
String regex = "(?<declare>(int|double)\\s+[a-z]\\s*)=(?<value>\\s*\\d+(\\.\\d+)?)";
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(input);
while (matcher.find()) {
    String group = matcher.group();
    System.out.println(group);
    System.out.println("declare: " + matcher.group("declare"));
    System.out.println("value: " + matcher.group("value"));
    System.out.println("-----");
}
```

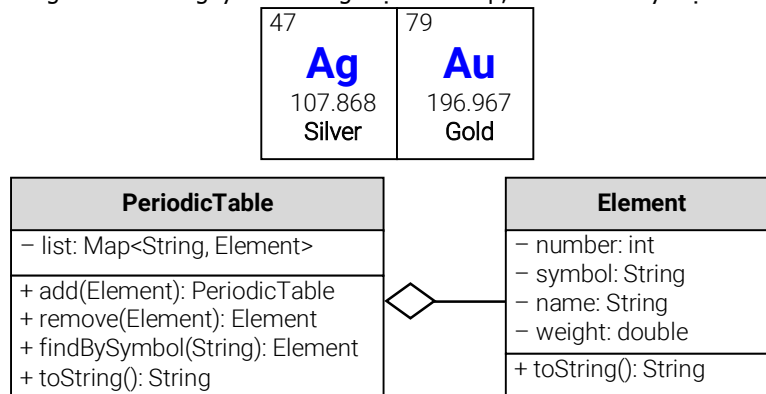
Kết quả:

```
int a = 100;
declare: int a
value: 100;
-----
double b = 10.2;
declare: double b
value: 10.2;
-----
```

Bài tập

[Encapsulation]

Lớp Element trừu tượng hóa một nguyên tố hóa học, chứa các thông tin: số hiệu nguyên tử (number), ký hiệu nguyên tố (symbol), tên nguyên tố (name) và nguyên tử khối (weight). Lớp PeriodicTable trừu tượng hóa bảng tuần hoàn các nguyên tố hóa học Mendeleev, nó đơn giản lưu các nguyên tố trong một HashMap, với khóa là ký hiệu của nguyên tố.



Chương trình cho phép nhập đến 110 nguyên tố, ngưng nhập khi nhập số thứ tự 0. Hiển thị thông tin các nguyên tố được nhập vào.

Kết quả chạy mẫu:

```

Enter element number: 79
Enter symbol: Au
Enter full name: Gold
Enter element weight: 196.967
  
```

```

Enter element number: 47
Enter symbol: Ag
Enter full name: Silver
Enter element weight: 107.868
  
```

```

Enter element number: 29
Enter symbol: Cu
Enter full name: Copper
Enter element weight: 63.546
  
```

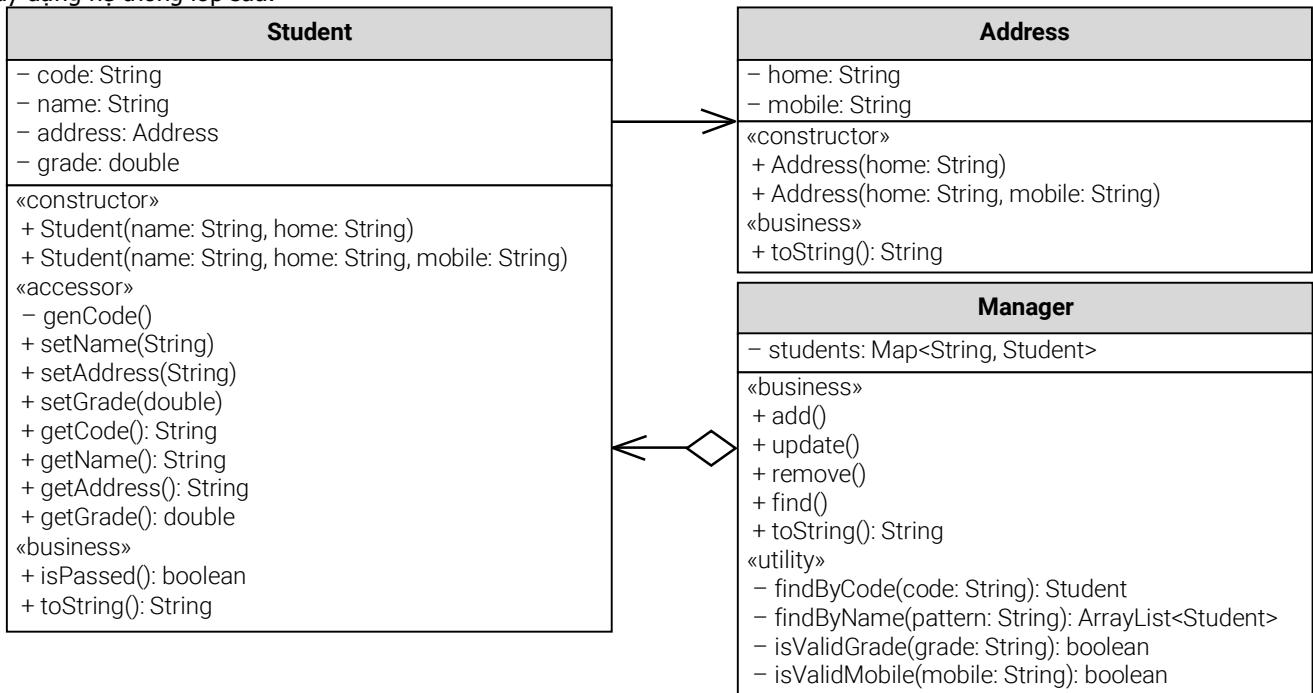
```

Enter element number: 0
  
```

No	Symbol	Name	Weight
79	Au	Gold	196.967
47	Ag	Silver	107.868
29	Cu	Copper	63.546

[Composition]

Xây dựng hệ thống lớp sau:



Lớp Address trừu tượng hóa một địa chỉ, thông tin gồm địa chỉ nhà (home) và số điện thoại (mobile, tùy chọn).

Lớp Student trừu tượng hóa thông tin một sinh viên:

- Phương thức genCode() dùng sinh mã tự động khi tạo một sinh viên, mã tự động gồm: ba ký tự đầu của tên sinh viên viết hoa (ký tự space thay bằng x) và 5 ký tự số lấy từ 5 ký tự cuối trong chuỗi thời gian hiện tại (số giây tính từ 1/1/1970).

- Phương thức isPassed() xác định điểm của sinh viên đã đạt (≥ 5) hay chưa.

Lớp Manager thao tác trên một Map chứa các đối tượng lớp Student. Lớp Manager có thể:

- Thêm một đối tượng mới thuộc lớp Student bằng phương thức add(), đối tượng mới chỉ cần các thông tin (name, address), điểm (grade) sẽ được cập nhật sau và code sẽ sinh tự động bằng phương thức genCode().

- Loại bỏ một đối tượng Student chỉ định bởi code bằng phương thức remove().

- Cập nhật một đối tượng Student chỉ định bởi code bằng phương thức update(). Khi cập nhật, để tiện dụng, các thông tin không muốn thay đổi chỉ cần nhấn Enter để bỏ qua.

- Dùng phương thức find() để tìm các đối tượng Student bằng cách nhập một phần của tên của sinh viên đó. Kết quả là ArrayList lưu các đối tượng Student mà tên có một phần giống với từ khóa chỉ định.

- Xuất danh sách sinh viên bằng phương thức toString() của lớp Manager.

Lớp Manager có một số phương thức công cụ giúp xây dựng các phương thức nghiệp vụ trên:

- findByCode(): tìm một Student theo code.

- findByName(): trả về ArrayList các Student mà tên có chứa chuỗi pattern chỉ định. Phương thức find() sẽ gọi phương thức công cụ này.

- isValidGrade(): khi cập nhật điểm, yêu cầu kiểm tra điểm nhập phải là số thực dương và hợp lệ ($grade \leq 10$).

- isValidMobile(): khi nhập số điện thoại, yêu cầu kiểm tra số điện thoại phải là chuỗi số 10 hoặc 11 ký tự, bắt đầu bằng 0.

Test driver dùng một menu hiển thị các chức năng của chương trình. Nếu người dùng nhập lựa chọn cho menu khác với một trong các ký tự {C,R,U,D,S,Q} (không cần viết hoa), chương trình sẽ báo lỗi và yêu cầu nhập lại.

Kết quả chạy mẫu như sau:

```

----- MENU -----
C Create a student
R Retrieve a student
U Update a student
D Delete a student
S Show student list
Q Quit
-----
Your choice: c
Name? Lionel Messi
Home? Argentina
Mobile (Enter to skip)? 
Create successful [LI04430]!

----- MENU -----
C Create a student
R Retrieve a student
U Update a student
D Delete a student
S Show student list

```

Q Quit

Your choice: u

Code? LI04430

Hint: reuse old value, press Enter

New name? ↵

New home? ↵

New mobile? ↵

New grade? 8.6

Update successful!

----- MENU -----
C Create a student
R Retrieve a student
U Update a student
D Delete a student
S Show student list
Q Quit

Your choice: r

Keyword? e

Found 1 student

[LI04934] Lionel Messi

----- MENU -----
C Create a student
R Retrieve a student
U Update a student
D Delete a student
S Show student list
Q Quit

Your choice: s

[LI04934] Lionel Messi (Argentina - N/A): 8.6

----- MENU -----
C Create a student
R Retrieve a student
U Update a student
D Delete a student
S Show student list
Q Quit

Your choice: d

Code? LI04430

Are you sure (y/n)? y

Remove successful!

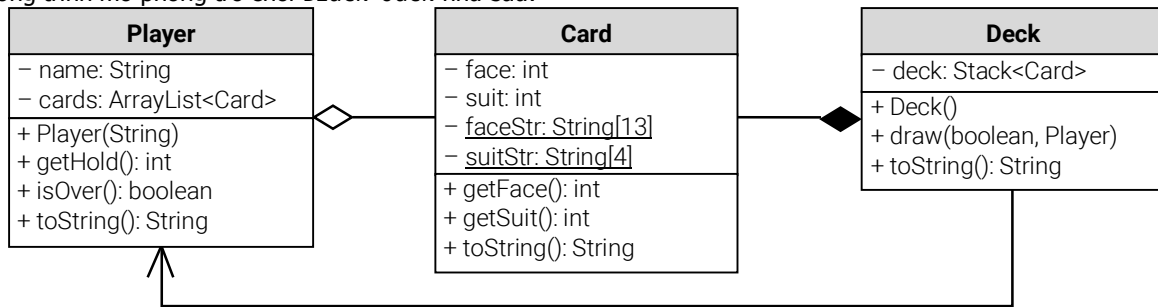
----- MENU -----
C Create a student
R Retrieve a student
U Update a student
D Delete a student
S Show student list
Q Quit

Your choice: q

Bye bye!

[Composition]

Viết chương trình mô phỏng trò chơi Black Jack như sau:



Lớp Card trừu tượng hóa một lá bài tây (poker) chứa thông tin: nước (face: Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King) và chất (suit: hearts, diamonds, spades, clubs).

Lớp Deck trừu tượng hóa một bộ bài có 52 lá bài. Khi khởi tạo, bộ bài được khởi gán rồi trộn ngẫu nhiên bằng phương thức công cụ shuffle() của lớp Collections. Lớp Deck có phương thức draw() dùng để rút bài, phương thức này trả về lá bài đầu tiên trong phần còn lại của bộ bài đã trộn và đang chia. Tham số kiểu boolean của phương thức draw() xác định có bao lá bài được chia hay không, nếu tham số này là false, lá bài được chia không hiển thị sau khi chia.

Lớp Player trừu tượng hóa người chơi bài, mỗi người chơi bài có tên name. Sau mỗi ván, từng người chơi sẽ mở bài (hiển thị các lá bài được chia) bằng phương thức toString(). Phương thức getHold() dùng tính tổng điểm nhận được, bằng cách tính tổng số nút (face) trên các lá bài. Lưu ý:

- Các lá bài: Jack, Queen và King đều có trị 10.
- Lá bài Ace được lựa chọn một trong ba cách tính trị: bằng 1, 10 hoặc 11.

Phương thức isOver() của lớp Player cho biết tổng điểm nhận được có vượt 21 điểm hay không.

Phương thức main() của lớp Main dùng lớp Player và lớp Deck để tiến hành trò chơi giữa hai người chơi như sau:

- Chia bài cho dealer (nhà cái, là máy tính) và người chơi, mỗi bên hai lá bài.
- Người chơi quyết định rút tiếp một số lá bài sao cho tổng điểm nhận được càng lớn càng tốt nhưng không vượt quá 21 điểm.
- Dealer rút một số lá bài theo chiến lược giống như người chơi. Dealer có một ngưỡng an toàn để dùng rút bài tiếp.

Điểm được tính như sau:

- Nếu hai bên hòa điểm hoặc cùng vượt quá 21 điểm, ván bài xem như hòa, không tính điểm.
- Dealer thắng: dealer có tổng điểm không vượt quá 21 điểm và, hoặc người chơi có tổng điểm vượt quá 21 điểm hoặc tổng điểm của dealer lớn hơn tổng điểm của người chơi.
- Người chơi thắng: người chơi có tổng điểm không vượt quá 21 điểm và, hoặc dealer có tổng điểm vượt quá 21 điểm hoặc tổng điểm của người chơi lớn hơn tổng điểm của dealer.

Khi được hỏi, nếu người chơi nhập khác "y" hoặc "n", chương trình báo lỗi và yêu cầu nhập lại.

Kết quả chạy mẫu như sau:

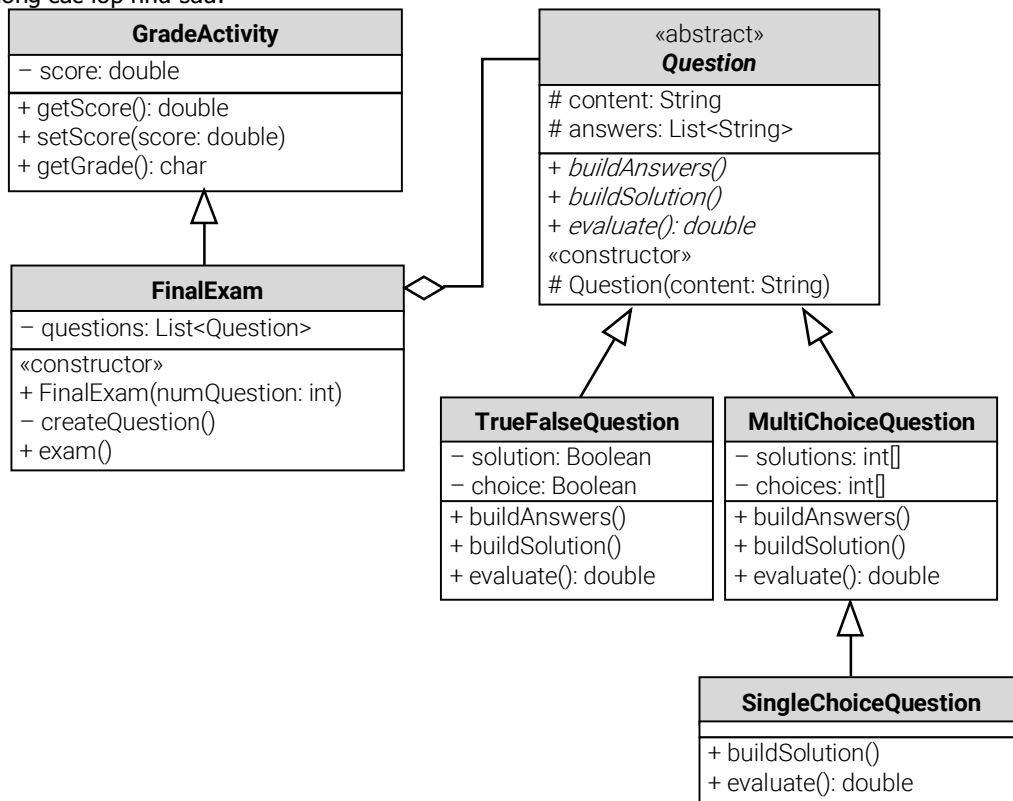
```

[----- Black Jack -----]
Dealer drew.
You drew: four of hearts, Ace of spades
Draw? y
You drew: six of clubs [21]
Draw? n
You   : four of hearts, Ace of spades, six of clubs [21]
Dealer: ten of clubs, Jack of spades [20]
You win, dealer lost!
[Computer: 0 - Human: 1]
Another game? y

[----- Black Jack -----]
Dealer drew.
You drew: two of diamonds, six of spades
Draw? Y
You drew: Ace of hearts [19]
Draw? N
Dealer drew: King of spades
You   : two of diamonds, six of spades, Ace of hearts [19]
Dealer: six of diamonds, four of hearts, King of spades [20]
Dealer win, you lost!
[Computer: 1 - Human: 1]
Another game? n
  
```


[Inheritance]

Xây dựng hệ thống các lớp như sau:



Lớp GradeActivity ghi nhận điểm có được từ một hoạt động học tập (Quiz, Workshop, FinalExam, ...) của một môn học. Điểm (score, thang điểm 10) sẽ chuyển thành thang điểm đánh giá (grade) như sau:

- A: $9 \leq \text{score} \leq 10$.
- B: $8 \leq \text{score} < 9$.
- C: $7 \leq \text{score} < 8$.
- D: $6 \leq \text{score} < 7$.

F: rớt môn (failed), $\text{score} < 6$.

Để đơn giản, chỉ có một hoạt động học tập là FinalExam. Constructor của lớp này cho phép tạo đề thi trắc nghiệm với số câu chỉ định, bằng cách gọi phương thức hỗ trợ createQuestion(). Phương thức exam() sẽ tiến hành thi và lấy điểm kết quả.

Lớp Question trừu tượng hóa một câu hỏi trắc nghiệm, bao gồm hai phần: nội dung câu hỏi (content) và danh sách tùy chọn (answers). Có nhiều loại câu hỏi:

TrueFalseQuestion: câu hỏi trắc nghiệm chỉ có hai tùy chọn "False" hoặc "True".

MultiChoiceQuestion: câu hỏi trắc nghiệm cho phép chọn một hoặc nhiều câu đúng trong các tùy chọn đưa ra.

SingleChoiceQuestion: câu hỏi trắc nghiệm có nhiều tùy chọn, chỉ chọn một trong các tùy chọn là đúng.

Các phương thức trừu tượng của lớp Question:

- buildAnswers(): dùng bởi người tạo đề thi. Phần tùy chọn của mỗi loại câu hỏi trắc nghiệm là khác nhau, nên chỉ cài đặt phương thức này trong lớp con.

TrueFalseQuestion: có hai tùy chọn bắt buộc, "True" và "False".

MultiChoiceQuestion và SingleChoiceQuestion: phải tạo đủ 4 tùy chọn.

- buildSolution(): dùng cho người tạo đề thi. Thiết lập đáp án cho mỗi câu trắc nghiệm, cách thiết lập khác nhau cho mỗi loại câu trắc nghiệm.

TrueFalseQuestion: thiết lập cho solution là true hoặc false.

MultiChoiceQuestion: thiết lập trị 0 hoặc 1 cho các phần tử của mảng solutions, các phần tử này tương ứng (theo chỉ số) với các tùy chọn. Thiết lập 0 tùy chọn sai và 1 cho tùy chọn đúng.

SingleChoiceQuestion: thiết lập chỉ số tùy chọn đúng cho solutions[0].

- evaluate(): dùng cho người thi. Cho phép người thi nhập lựa chọn của họ:

TrueFalseQuestion: người thi chỉ trả lời 0 cho "False" và 1 cho "True".

MultiChoiceQuestion: người thi nhập chuỗi trả lời cho 4 tùy chọn, 0 cho không chọn và 1 cho chọn.

SingleChoiceQuestion: người thi nhập số thứ tự (tính từ 1) của tùy chọn đúng.

Sau đó phương thức evaluate() sẽ so sánh kết quả chọn của người thi với đáp án và tính điểm. Chú ý rằng, điểm của mỗi câu phụ thuộc vào số câu thi, để bảo đảm thang điểm 10.

Kết quả chạy mẫu chương trình:

```

Number of question: 3
Question #1
Kind [1.T/F 2.Single 3.Multi]: 1
Content? A class can implement only one interface.
Solution [0 / 1]: 0
  
```

Question #2

Kind [1.T/F 2.Single 3.Multi]: 2

Content? This keyword refers to an object's superclass.

Option #1: base

Option #2: super

Option #3: this

Option #4: parent

Solution [1 - 4]: 2

Question #3

Kind [1.T/F 2.Single 3.Multi]: 3

Content? These are two of the most useful members of the Object class.

Option #1: toString

Option #2: compareTo

Option #3: equals

Option #4: valueOf

String of 4 solutions (0 / 1): 1 0 1 0

----- Final Exam -----

[01]. A class can implement only one interface.

1. True

2. False

Your choice [0 / 1]: 0

[02]. This keyword refers to an object's superclass.

1. Base

2. Super

3. This

4. Parent

Your choice [1 - 4]: 2

[03]. These are two of the most useful members of the Object class.

1. toString

2. compareTo

3. equals

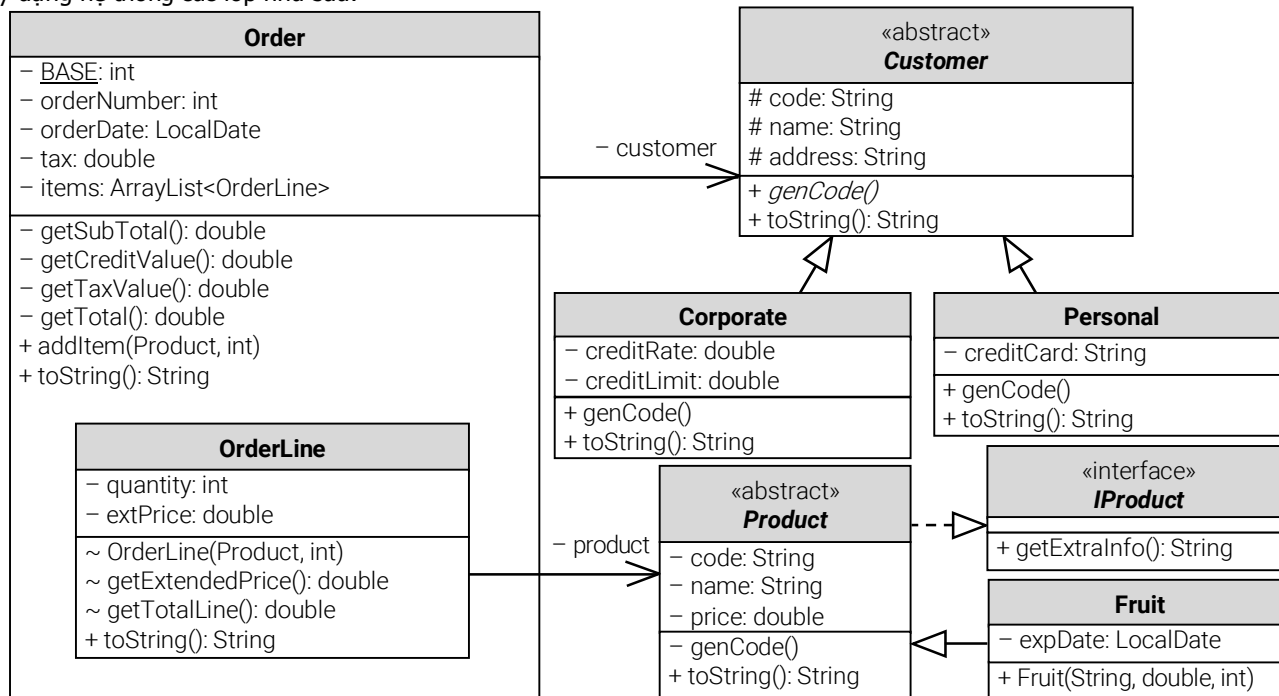
4. valueOf

String of 4 choices (0 / 1): 1 1 0 0

Your grade: B (8.3)

[Polymorphism]

Xây dựng hệ thống các lớp như sau:



Mỗi đơn hàng (Order) có số hiệu đơn hàng (orderNumber, là số tự sinh từ BASE) và ngày lập đơn hàng (orderDate, là ngày hiện tại). Mỗi đơn hàng được tính thuế tiêu dùng (tax) bằng % trên tổng giá trị đơn hàng đã trừ chiết khấu, chiết khấu cũng tính bằng % nếu có.

Lớp Order còn có các phương thức:

- getSubTotal() dùng tính tổng giá trị đơn hàng trước chiết khấu và trước thuế.
- getCreditValue() dùng tính giá trị chiết khấu cho đơn hàng từ % chiết khấu, trong trường hợp khách hàng là công ty.
- getTaxValue() dùng tính giá trị thuế tiêu dùng từ % thuế trên tổng giá trị đơn hàng đã trừ chiết khấu.
- getTotal() giá trị thanh toán đơn hàng, bằng tổng giá trị đơn hàng đã trừ chiết khấu cộng thuế.

Mỗi đơn hàng thuộc về một khách hàng, mỗi khách hàng (Customer) có thông tin: mã khách hàng (code), tên khách hàng (name) và địa chỉ khách hàng (address). Ngoài ra, khách hàng có thông tin riêng tùy theo các loại khách hàng sau:

- Khách hàng cá nhân (Personal) có thông tin về thẻ tín dụng (creditCard). Mã của khách hàng cá nhân có định dạng CUSXXXXX, trong đó XXXXX là chuỗi 5 số ngẫu nhiên.

- Khách hàng công ty (Corporate) có thông tin về chiết khấu hoa hồng (creditRate) tính bằng % trên tổng giá trị đơn hàng trước thuế. Phần trăm chiết khấu là số không âm và không được vượt quá giới hạn chiết khấu (creditLimit). Mã của khách hàng công ty có định dạng CORXXXXX, trong đó XXXXX là chuỗi 5 số ngẫu nhiên.

Mỗi đơn hàng chứa một danh sách các mục đặt hàng (OrderLine); để an toàn, lớp OrderLine là lớp nội của lớp Order.

Mỗi mục đặt hàng chứa thông tin sản phẩm (product), số lượng đặt mua (quantity) và giá sản phẩm (extPrice), giá sản phẩm lưu trong đơn hàng không bị ảnh hưởng khi cập nhật giá sản phẩm trong lớp Product. Tổng giá trị một mục hàng được tính từ phương thức getTotalLine().

Các mục đặt hàng (OrderLine) được thêm vào đơn hàng nhờ phương thức addItem() của lớp Order. Chú ý phương thức này tạo ra mục đặt hàng cho mỗi sản phẩm trước khi đưa vào mảng items thuộc lớp; nói cách khác, constructor lớp OrderLine chỉ được gọi từ phương thức addItem() của lớp Order.

Mỗi sản phẩm (Product) có thông tin: mã sản phẩm (code), tên sản phẩm (name) và giá sản phẩm (price). Mã của sản phẩm có định dạng ABCXXXXX, trong đó ABC là ba ký tự đầu (viết hoa) của tên sản phẩm, XXXXX là chuỗi 5 số ngẫu nhiên.

Lớp Product cài đặt interface IProduct, có phương thức getExtraInfo() trả về thông tin bổ sung cho từng loại sản phẩm.

Lớp Fruit, thừa kế lớp Product là một sản phẩm cụ thể, ngoài thông tin sản phẩm còn có thêm expDate là thời điểm sản phẩm hết hạn. Khi tạo đối tượng lớp Fruit, số ngày sử dụng (limit) kể từ hiện tại được nhập như đối số của constructor lớp Fruit, rồi từ đó tính ra ngày hết hạn expDate. Thông tin về ngày hết hạn hiển thị như thông tin bổ sung, sẽ được in thêm sau tên của sản phẩm.

Bạn thử dẫn xuất thêm một số lớp con của lớp Product với thông tin bổ sung phù hợp.

Phương thức toString() của lớp Order dùng xuất một đơn hàng cụ thể cho loại khách hàng cụ thể như kết quả chạy mẫu sau.

INVOICE #100 Date: [11 Mar 2016]

Customer : Karl Marx [CUS54444]

Address : 12, Albert Einstein, Berlin, German

Credit Card: 1234-5678-9012-3456

Code	Name	Price	Qty	Total Line
[USA54448]	USA Orange (Exp: 15/03/2016)	4.7	3	14.1
[CHI54450]	China Mango (Exp: 01/05/2016)	2.4	2	4.8
Subtotal: 18.9				

Tax : 0.567

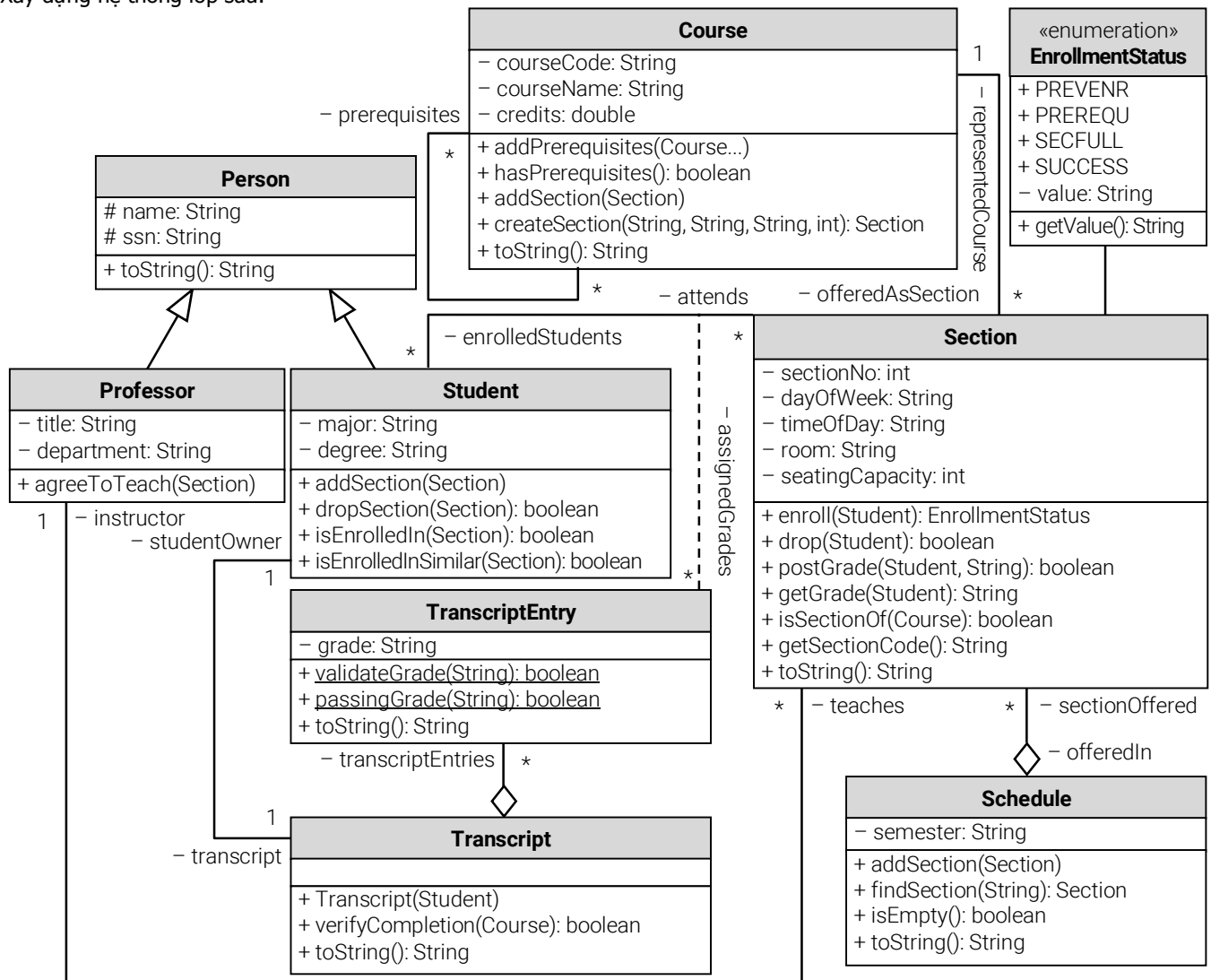
Total : 19.467

INVOICE #101 Date: [11 Mar 2016]
Customer : Apple Inc. [COR54445]
Address : Cupertino, CA 95014, USA
Credit Rate: 4.3%

Code	Name	Price	Qty	Total Line
[USA54448]	USA Orange (Exp: 15/03/2016)	4.7	3	14.1
[CHI54450]	China Mango (Exp: 01/05/2016)	2.4	2	4.8
				Subtotal: 18.9
				Credit : 0.605
				Tax : 0.549
				Total : 18.844

[Case study]

Xây dựng hệ thống lớp sau:



Một môn học (Course) được dạy nhiều lần trong các khóa học (Section) khác nhau. Mỗi khóa học do một giáo viên (Professor) dạy và chiếm một mục trong thời khóa biểu (Schedule) của một học kỳ chỉ định, với thời gian và vị trí phòng học cụ thể. Sinh viên (Student) sẽ đăng ký tham dự các khóa học (Section). Điểm kết quả học một khóa học của sinh viên thể hiện như một mục trong học bạ (TranscriptEntry) và sẽ được lưu vào học bạ (Transcript).

Mô tả chi tiết:

- Lớp Person thể hiện một cá nhân có thông tin chung là name và ssn (Social Security Number, trị duy nhất).
- Lớp Professor thể hiện một giáo viên, ngoài những thông tin cá nhân còn có: title (chức danh), department (tên khoa quản lý giáo viên); và thông tin liên kết khác như teaches (danh sách các khóa học giáo viên đang dạy).

Phương thức agreeToTeach() dùng để thêm khóa học vào danh sách các khóa học đang dạy teaches, đồng thời cũng đăng ký người dạy (instructor) cho khóa học đó trong lớp Section.

- Lớp Student thể hiện một sinh viên, ngoài những thông tin cá nhân còn có: major (chuyên ngành), degree (học vị); và các thông tin liên kết khác như transcript (học bạ), attends (các khóa học đang tham dự).

Lớp Student có các phương thức:

- + addSection(): để thêm một khóa học vào attends khi đăng ký khóa học.
- + dropSection(): để loại một khóa học khỏi attends sau khi có điểm.
- + isEnrolledIn(): kiểm tra xem khóa học định đăng ký có trong attends chưa.
- + isEnrolledInSimilar(): kiểm tra xem đã hoàn thành môn học dạy trong khóa học này trước đây chưa. Nếu sinh viên học xong một môn, kết quả học được lưu vào học bạ và môn đó sẽ được loại khỏi attends.

Mỗi sinh viên được tạo ra sẽ có ngay một học bạ (Transcript); nói cách khác, constructor của lớp Transcript được gọi từ constructor của Student.

- Lớp Course thể hiện một môn học, một Course có courseCode (mã môn học), courseName (tên môn học), credits (số tín chỉ); và các thông tin liên kết như: prerequisites (danh sách các môn học tiên quyết, bắt buộc phải qua môn trước khi học môn này), offeredAsSection (danh sách các khóa học đang dạy môn học này).

Có thể thêm một hoặc nhiều môn học tiên quyết vào danh sách môn học tiên quyết bằng phương thức addPrerequisites(), thêm một khóa học vào danh sách các khóa học dạy môn này bằng phương thức addSection(). Phương thức createSection() sinh ra một khóa học với thông tin cụ thể và đưa nó vào danh sách offeredAsSection; nói cách khác, constructor của lớp Section được gọi thông qua phương thức createSection().

Lớp Course còn có phương thức hasPrerequisites() để kiểm tra xem môn học chỉ định có môn học tiên quyết nào hay không.

- Lớp Section thể hiện một khóa học, chứa các thông tin:

- + sectionNo là số lần dạy môn học này, được tính từ kích thước của offeredAsSection. sectionNo dùng để tạo mã khóa học với định dạng courseCode-sectionNo, trả về từ phương thức getSectionCode(), trong đó courseCode là mã môn học.
- + dayOfWeek (thứ trong tuần của ngày học, 3 ký tự) và timeOfDay (thời gian học trong ngày).
- + room (số hiệu phòng) và seatingCapacity (số lượng học viên tối đa).
- + các thông tin liên kết như: instructor (giáo viên đảm nhận khóa học), representedCourse (môn được dạy trong khóa học), offeredIn (thời khóa biểu chứa thông tin khóa học).

Lớp Section cũng có danh sách sinh viên đăng ký môn học (enrolledStudents, collection các Student) và danh sách điểm đã đạt của sinh viên (assignedGrades, collection các TranscriptEntry).

Lớp Section có các phương thức:

- + enroll(): đăng ký sinh viên tham gia khóa học. Phương thức enroll() sẽ kiểm tra xem sinh viên thuộc trạng thái đăng ký (enum EnrollmentStatus) nào sau đây:
 - PREVENR sinh viên đã đăng ký khóa học hoặc đã hoàn thành môn học, không được đăng ký học lần nữa.
 - PREREQU sinh viên chưa hoàn tất các môn tiên quyết của môn học chỉ định nên không được đăng ký môn học.
 - SECFULL khóa học đã đủ số lượng học viên nên không nhận đăng ký thêm.
 - SUCCESS sinh viên đủ điều kiện đăng ký môn học, tiến hành đăng ký cho sinh viên
- + drop(): loại bỏ sinh viên đăng ký. Khi loại bỏ thành công sinh viên khỏi danh sách enrolledStudents và loại khóa học này khỏi danh sách attends của sinh viên đó, phương thức trả về true.
- + postGrade(): ghi một dòng học bạ cho một sinh viên chỉ định, nội dung là môn học thuộc khóa học và điểm môn đó.
- + getGrade(): lấy điểm của khóa học cho một sinh viên chỉ định.
- + isSectionOf(): xem có phải khóa học đang dạy môn chỉ định.
- + getSectionCode: trả về mã khóa học có định dạng courseCode-sectionNo.

- Lớp Schedule thể hiện thời khóa biểu, có các thuộc tính semester (học kỳ), sectionOffered (danh sách các khóa học thuộc học kỳ) và có các tác vụ:

- + addSection(): thêm khóa học vào danh sách khóa học thuộc học kỳ.
- + findSection(): tìm một lớp trong danh sách các khóa học thuộc học kỳ.
- + isEmpty(): kiểm tra danh sách các khóa học thuộc học kỳ có rỗng không.

- Lớp TranscriptEntry mô tả một mục trong học bạ, là kết quả học một khóa học cụ thể của một sinh viên cụ thể. TranscriptEntry chứa các thông tin: student (sinh viên có tên trong học bạ), grade (điểm, lấy từ khóa học), section (khóa học) và transcript (học bạ chứa mục này); nó cũng chứa các phương thức static sau:

- + validateGrade(): dùng kiểm tra hợp lệ điểm nhập. Điểm nhập hợp lệ là A, B, C, D với điểm thêm bớt như B+, C-; và các điểm tốt là F (Failed), I (Ignored, bỏ môn).
- + passingGrade(): phương thức xác định sinh viên đã đạt điểm qua môn của khóa học đó chưa. Điểm qua môn tối thiểu phải đạt D.

- Lớp Transcript thể hiện học bạ của sinh viên, chứa transcriptEntries (các mục nhập, mỗi mục nhập tương ứng một khóa học sinh viên đã học), studentOwner (tên sinh viên có học bạ). Ngoài ra có các phương thức sau: addTranscriptEntry() (thêm một mục nhập vào học bạ) và verifyCompletion() (kiểm tra xem đã học một môn học nào đó chưa).

Một số lưu ý khi xây dựng các lớp:

- Viết đầy đủ các getter/setter cần thiết, viết thêm các phương thức công cụ, các constructor nạp chồng, các phương thức public khác nếu thấy cần thiết.
- Bảo đảm quan hệ giữa các lớp
- Viết phương thức toString() cho các lớp để dễ dàng hiển thị thông tin cho đối tượng mỗi lớp. Xem mẫu xuất bên dưới.

Test driver:

```
public class Main {
    public static void main(String[] args) {
        Schedule sc = new Schedule("Summer 2014");
        // mở Course và thêm các Course tiên quyết để đăng ký được Course đó
        Course c1 = new Course("I2C101", "Introduction to Computing", 1.5);
        Course c2 = new Course("OOP102", "Object-Oriented Programming", 3.5);
        c2.addPrerequisites(c1);
        Course c3 = new Course("CJV103", "Core Java Programming", 2.0);
        c3.addPrerequisites(c1, c2);
        // thêm các Professor tham gia giảng dạy các Section
        Professor p1 = new Professor("Greg Anderson", "123-45-7890", "Adjunct Professor", "Information Technology");
        Professor p2 = new Professor("Bjarne Stroustrup", "135-24-7908", "Full Professor", "Computer Science");
        Professor p3 = new Professor("James Gosling", "490-53-6281", "Full Professor", "Software Engineering");
        // thêm các Student đăng ký học các Section
        Student s1 = new Student("Barack Obama", "123-12-4321", "IT", "Ph.D.");
        Student s2 = new Student("Vladimir Putin", "246-13-7985", "Math", "M.S.");
        Student s3 = new Student("Francois Hollande", "135-78-2460", "Physics", "M.S.");
        // từ Course tạo Section dạy Course đó
        Section se1 = c1.createSection("Mon", "07:00-10:15 AM", "302", 24);
        Section se2 = c2.createSection("Tue", "02:15-05:30 PM", "405", 30);
        Section se3 = c3.createSection("Fri", "08:45-12:00 AM", "507", 22);
        // mời các Professor dạy các Section
        p1.agreeToTeach(se1);
        p2.agreeToTeach(se2);
        p3.agreeToTeach(se3);
    }
}
```



```

// đưa các Section đã thiết lập vào Schedule
sc.addSection(se1);
sc.addSection(se2);
sc.addSection(se3);
// đăng ký các Student vào các Section trong Schedule
se1.enroll(s1);
se1.enroll(s2);
se1.enroll(s3);
se1.postGrade(s2, "A"); // s1 và s2 đã học se1, s1 và s2 có thể đăng ký se2
se1.postGrade(s3, "B");
se2.enroll(s2);
se2.enroll(s3);
se2.postGrade(s3, "A+"); // s2 đã học se2, s2 có thể học se3
se3.enroll(s3);
// thông tin Course
System.out.println(c1);
System.out.println(c2);
System.out.println(c3);
// thông tin Professor
System.out.println(p1);
System.out.println(p2);
System.out.println(p3);
// thông tin Section
System.out.println(se1);
System.out.println(se2);
System.out.println(se3);
// thông tin Schedule
System.out.println(sc);
// thông tin Student
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}
}

```

Kết quả:

```

[Course Information]
Course No. : I2C101
Course Name: Introduction to Computing
Credits    : 1.5
Prerequisite Course: [none]
Offer As Section: I2C101-2

[Course Information]
Course No. : OOP102
Course Name: Object-Oriented Programming
Credits    : 3.5
Prerequisite Course:
  Introduction to Computing
Offer As Section: OOP102-2

[Course Information]
Course No. : CJV103
Course Name: Core Java Programming
Credits    : 2.0
Prerequisite Course:
  Introduction to Computing
  Object-Oriented Programming
Offer As Section: CJV103-2

[Person Information]
Name       : Greg Anderson
SS Number  : 123-45-7890
Professor-Specific Information:
  Title     : Adjunct Professor
  Department : Information Technology
Teaching Assignments for Greg Anderson:
  Introduction to Computing (Mon, 07:00-10:15 AM)

[Person Information]
Name       : Bjarne Stroustrup
SS Number  : 135-24-7908

```

Professor-Specific Information:

Title : Full Professor
Department : Computer Science

Teaching Assignments for Bjarne Stroustrup:

Object-Oriented Programming (Tue, 02:15-05:30 PM)

[Person Information]

Name : James Gosling
SS Number : 490-53-6281

Professor-Specific Information:

Title : Full Professor
Department : Software Engineering

Teaching Assignments for James Gosling:

Core Java Programming (Fri, 08:45-12:00 AM)

[Section Information]

Semester : Summer 2014
Course No. : I2C101
Section No.: 2
Offered : Mon, 07:00-10:15 AM
In Room : 302
Professor : Greg Anderson
Total of 3 students enrolled, as follow:
Francois Hollande
Barack Obama
Vladymir Putin

[Section Information]

Semester : Summer 2014
Course No. : OOP102
Section No.: 2
Offered : Tue, 02:15-05:30 PM
In Room : 405
Professor : Bjarne Stroustrup
Total of 2 students enrolled, as follow:
Francois Hollande
Vladymir Putin

[Section Information]

Semester : Summer 2014
Course No. : CJV103
Section No.: 2
Offered : Fri, 08:45-12:00 AM
In Room : 507
Professor : James Gosling
Total of 1 student enrolled, as follow:
Francois Hollande

[Schedule Information]

Schedule for Summer 2014:

Mon, 07:00-10:15 AM [Room: 302] Introduction to Computing (Greg Anderson)
Tue, 02:15-05:30 PM [Room: 405] Object-Oriented Programming (Bjarne Stroustrup)
Fri, 08:45-12:00 AM [Room: 507] Core Java Programming (James Gosling)

[Person Information]

Name : Barack Obama
SS Number : 123-12-4321

Student-Specific Information:

Major : IT
Degree : Ph.D.

Course Schedule for Barack Obama:

Mon, 07:00-10:15 AM [Room: 302] Introduction to Computing (Greg Anderson)

Transcript for Barack Obama: [none]

[Person Information]

Name : Vladymir Putin
SS Number : 246-13-7985

Student-Specific Information:

Major : Math
Degree : M.S.

Course Schedule for Vladymir Putin:

Tue, 02:15-05:30 PM [Room: 405] Object-Oriented Programming (Bjarne Stroustrup)
Transcript for Vladimir Putin:
Introduction to Computing (A)

[Person Information]

Name : Francois Hollande
SS Number : 135-78-2460

Student-Specific Information:

Major : Physics
Degree : M.S.

Course Schedule for Francois Hollande:

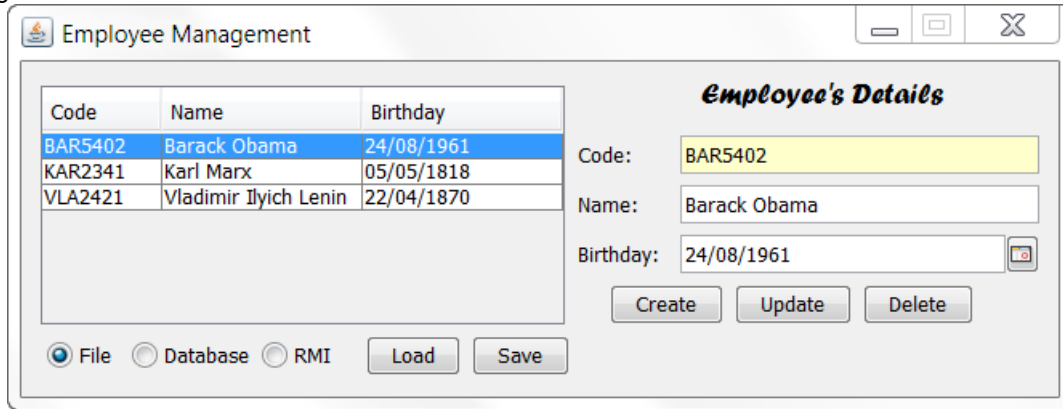
Fri, 08:45-12:00 AM [Room: 507] Core Java Programming (James Gosling)

Transcript for Francois Hollande:

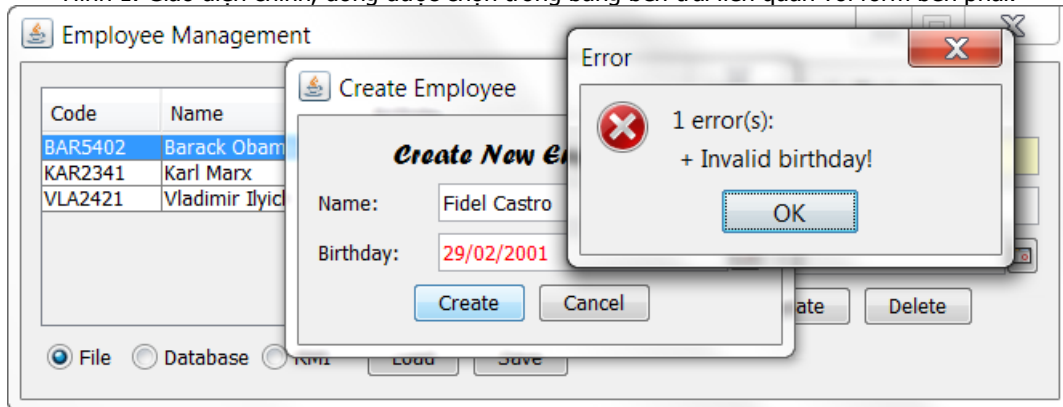
Introduction to Computing (B)
Object-Oriented Programming (A+)

[Swing]

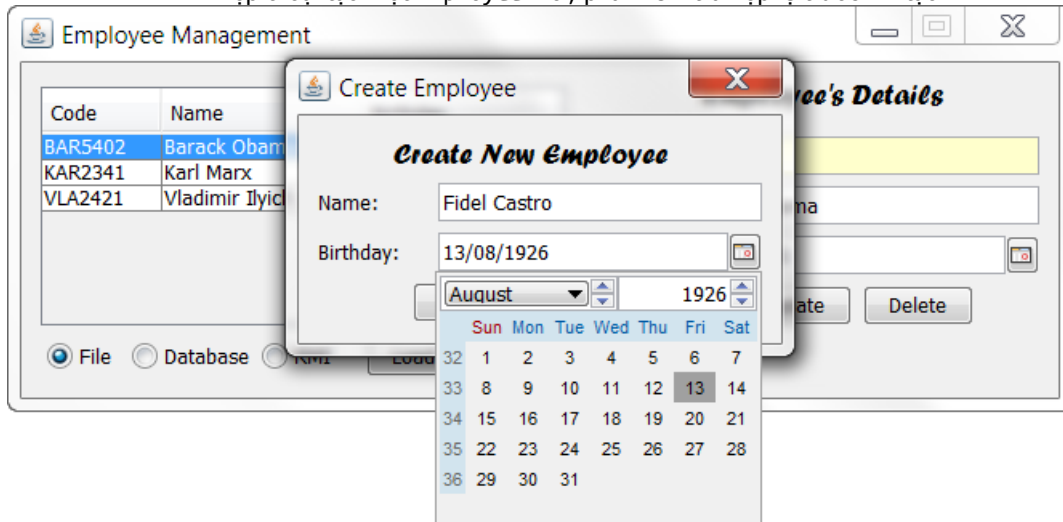
Viết ứng dụng nhỏ sau:



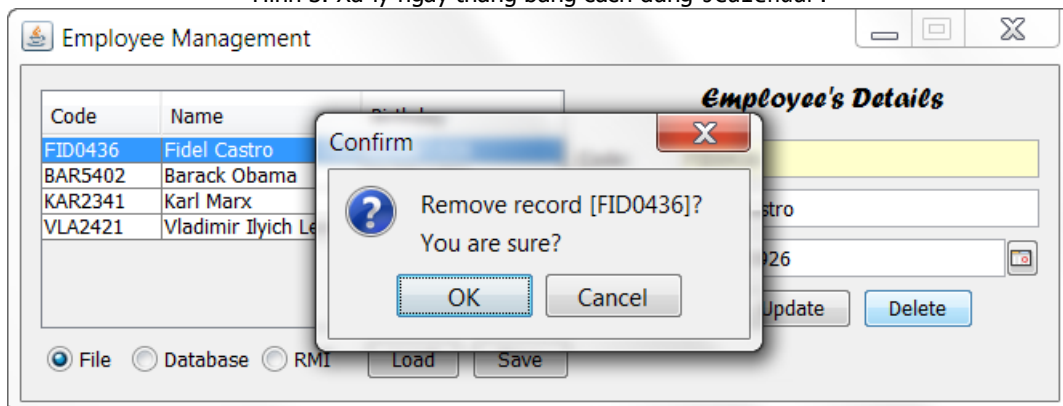
Hình 1. Giao diện chính, dòng được chọn trong bảng bên trái liên quan với form bên phải.



Hình 2. Hộp thoại tạo một Employee mới, phải kiểm tra hợp lệ trước khi tạo.



Hình 3. Xử lý ngày tháng bằng cách dùng JCalendar.



Hình 4. Đề nghị xác nhận trước khi xóa, kiểm tra hợp lệ trước khi cập nhật.

Chương trình có thể nạp hoặc lưu dữ liệu bằng một trong ba cách:

- Dùng tập tin văn bản, thông tin cá nhân được lưu thành các dòng có định dạng sau:
code:fullname:birthday

Các trường cách nhau bởi dấu ":".

code tự sinh có định dạng sau:

[3 ký tự đầu của fullname, viết hoa, space được thay thế bằng x][chuỗi số ngẫu nhiên 4 ký tự, lấy từ thời gian hiện tại]
birthday theo định dạng: dd/MM/yyyy

Chương trình cho phép hiển thị hộp thoại chọn tập tin để mở và hộp thoại chọn tập tin để lưu.

- Dùng JDBC truy cập cơ sở dữ liệu (MS SQLServer), với bảng dữ liệu sau:

Employee			
	Column Name	Data Type	Allow Nulls
🔑	code	varchar(15)	<input type="checkbox"/>
	name	varchar(30)	<input type="checkbox"/>
	birthday	datetime	<input type="checkbox"/>

- Dùng RMI, cũng lưu dữ liệu trong tập tin văn bản nhưng các thao tác được triệu gọi từ xa.

Có thể nạp dữ liệu bằng một cách và lưu dữ liệu bằng một cách khác, trong ba cách trên.

Các lớp được tổ chức theo đề nghị sau:

- Lớp POJO Employee, là đối tượng vận chuyển (transfer object) để chứa dữ liệu cần lưu chuyển.
- Lớp EmployeeList, thực chất là HashMap chứa các Employee, thuận tiện cho việc quản lý, tìm kiếm.
- Lớp Manager quản lý một EmployeeList, đây là đối tượng nghiệp vụ (business object) của ứng dụng.
- Lớp Validation, là lớp công cụ hỗ trợ kiểm tra hợp lệ dữ liệu nhập trước khi sử dụng.
- Các lớp giao diện dùng Swing: EmployeeFrame và EmployeeDialog.

Khi dùng RMI, lớp Manager được chuyển thành đối tượng triệu gọi từ xa (remoted business object), nằm phía RMI server. Interface ManagerInterface bộc lộ các phương thức của nó được chuyển cho chương trình phía client.

Trong giao diện chính (hình 1), dữ liệu lưu trữ dưới dạng bảng bên trái, dữ liệu của hàng được chọn từ bảng sẽ hiển thị tương ứng trong form bên phải.

Khi nhấn nút Create, hộp thoại nhập sẽ hiển thị, cho phép nhập thông tin để một Employee mới. Dữ liệu nhập sẽ được kiểm tra hợp lệ trước khi tạo một Employee mới, code của Employee này sẽ tự sinh (hình 2).

Khi nhấn nút Update, Employee có dữ liệu đang hiển thị trong form sẽ được cập nhật. Ngoài code không được thay đổi, các dữ liệu thay đổi khác sẽ được kiểm tra hợp lệ trước khi cập nhật.

- fullname không được rỗng.

- birthday là ngày hợp lệ.

Ngày tháng của dữ liệu nhập được chọn từ JCalendar 1.4: <http://toedter.com/jcalendar/> (hình 3).

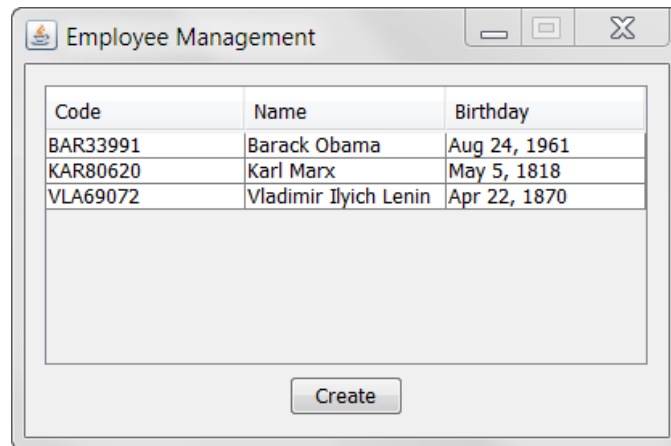
Khi nhấn nút Delete, một hộp thoại nhắc sẽ đề nghị xác nhận trước khi xóa thông tin của Employee có dữ liệu đang hiển thị trong form (hình 4).

Thêm, cập nhật và xóa chỉ ảnh hưởng đến EmployeeList do Manager đang quản lý. Chỉ khi nhấn nút Save, dữ liệu thay đổi mới lưu xuống tập tin hoặc cơ sở dữ liệu.

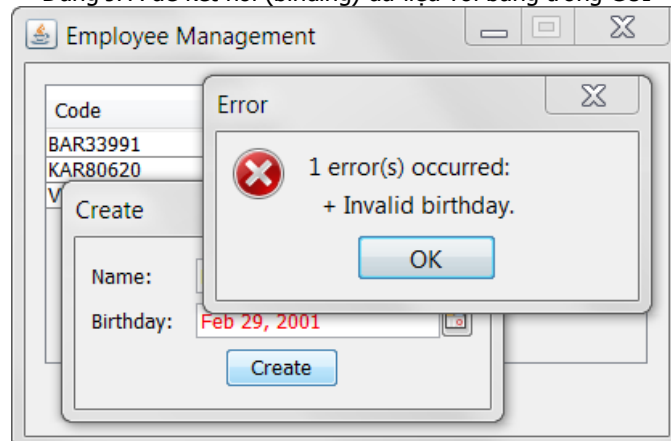
Hãy dùng JPA thay cho JDBC khi thực hiện bài tập trên.

[JDBC - JPA]

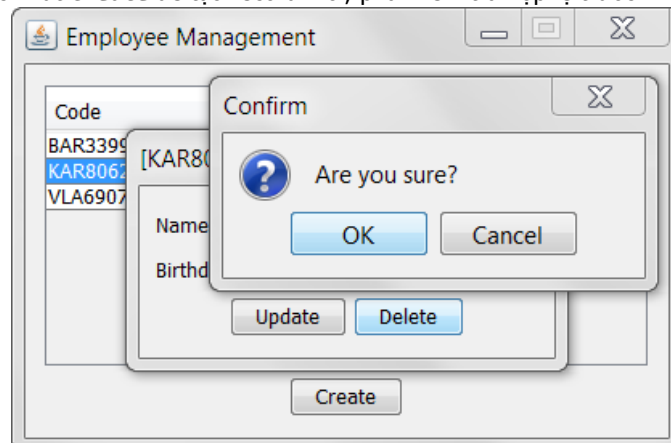
Viết ứng dụng nhỏ sau:



Dùng JPA để kết nối (binding) dữ liệu với bảng trong GUI



Click nút Create để tạo record mới, phải kiểm tra hợp lệ trước khi tạo.

Click dòng dữ liệu để xóa hoặc cập nhật record. Xác nhận trước khi xóa, kiểm tra hợp lệ trước khi cập nhật.
Bảng dữ liệu tương tự bài tập trước:

Employee			
	Column Name	Data Type	Allow Nulls
?	code	varchar(15)	<input type="checkbox"/>
	name	varchar(30)	<input type="checkbox"/>
	birthday	datetime	<input type="checkbox"/>

Các lớp được tổ chức theo đề nghị sau:

- Lớp POJO Employee là lớp Entity, ánh xạ với bảng cơ sở dữ liệu.
- Lớp DBManager cung cấp các tác vụ truy cập cơ sở dữ liệu bằng JPA.
- Lớp Validation, là lớp công cụ hỗ trợ kiểm tra hợp lệ dữ liệu nhập trước khi sử dụng.
- Các lớp giao diện dùng Swing: EmployeeFrame và EmployeeDialog.

Sinh code cho Employee mới và kiểm tra dữ liệu nhập hợp lệ tương tự bài tập trước.

Phụ lục 1 – NetBeans

NetBeans là một trong các IDE được nhiều lập trình viên Java lựa chọn. Để đạt tốc độ và hiệu quả lập trình cao nhất, lập trình viên cần tận dụng các khả năng của IDE. Tài liệu này giới thiệu 50 thủ thuật cần thiết khi lập trình với NetBeans.

Source Code Editing

1. Định dạng code thường xuyên trong khi code là một thói quen tốt. Trong khi lập trình, bạn có thể định dạng code tự động bằng **Alt+Shift+F**. Nếu bạn muốn một định dạng code riêng, hãy tùy biến auto-format trong Tools > Options > Editor, tab Formatting. Có thể tùy biến auto-format rất chi tiết, ví dụ bạn muốn tùy biến các dòng trống trong code; chọn Language là Java và Category là Blank Lines rồi thực hiện các thiết lập chi tiết.

2. Bạn thường dùng comment (chú thích) để "khóa" tạm một đoạn code khi cần. Hãy thêm/xóa comment bằng cách dùng: **Ctrl+Shift+C**, hoặc nhanh hơn, dùng **Ctrl+/.**

3. Bạn không nên để các dòng trống không cần thiết trong code, xóa nhanh dòng trống bằng **Ctrl+E**, hoặc **Shift+Delete**.


4. Bạn không cần phải nhập các dòng import. Thay vào đó, dùng **Ctrl+Shift+I** để import một cách tự động. Tuy nhiên, cần chú ý xem qua trước để chắc rằng import được đề nghị là chính xác, nhất là khi xây dựng ứng dụng dùng nhiều thư viện. Dùng **Alt+Shift+I** để chọn import cho tên lớp ngay tại con trỏ.

5. Thao tác sao chép một dòng hoặc một khối thường được sử dụng khi code. Bạn không cần copy-paste như thường lệ, hãy đặt con trỏ tại dòng cần sao chép hoặc chọn khối cần sao chép, dùng **Ctrl+Shift+↓** để sao chép dòng/khối xuống phía dưới (hoặc **Ctrl+Shift+↑** để sao chép lên phía trên).

6. Để di chuyển một dòng hoặc một khối, đặt con trỏ tại dòng cần di chuyển hoặc chọn khối cần di chuyển, rồi dùng **Alt+Shift+↓** để di chuyển dòng/khối xuống phía dưới (hoặc **Alt+Shift+↑** để di chuyển lên phía trên).

Như vậy, thay vì thực hiện copy rồi paste sang vị trí khác, ta sao chép khối (**Ctrl+Shift+↓**) rồi di chuyển khối vừa sao chép đến chỗ mới (**Alt+Shift+↓**).




Các phím tắt **Alt+Shift+←** hoặc **Alt+Shift+→** cũng di chuyển dòng code sang trái hoặc phải, chủ yếu để giống hàng code.

7. Khi cần sao chép (copy & paste), di chuyển một khối code lớn, bạn hãy tận dụng khả năng *code folding*. Click dấu  để thu khối code lại, như vậy khối code chỉ nằm trên một dòng, dễ dàng được chọn để sao chép.

8. Bạn gọi một phương thức không tham số. Bạn nhập tên phương thức, theo sau là dấu **(**, NetBeans sẽ tự thêm dấu **)**. Lúc này bạn nhập luôn **;** mà không phải về cuối dòng.

9. Nếu phương thức gọi yêu cầu phải nhập tham số, sau khi nhập tham số, con trỏ lúc này ở vị trí nào đó trong dòng code. Dùng **Ctrl+;** sẽ chèn một dấu **;** vào cuối dòng code. Hơn nữa, nếu bạn dùng **Ctrl+Shift+;** dấu **;** sẽ được chèn vào cuối dòng và con trỏ bắt đầu trên một dòng mới chèn thêm ngay dưới dòng hiện hành.

10. Bạn gọi một phương thức, nhập tên phương thức, theo sau là dấu **(**, NetBeans sẽ tự thêm dấu **)**. Nếu bạn muốn xem danh sách các tham số có thể có của phương thức đó, dùng **Ctrl+P**.

11. Lỗi, cảnh báo hoặc hint thể hiện bằng icon  hoặc icon  trên cột biên trái, click vào sẽ giúp bạn hiểu nguyên nhân lỗi, cảnh báo và cách xử lý. Từ đây ta có một thủ thuật hữu dụng: thay vì click vào icon , dùng **Alt+Enter** khi con trỏ tại dòng này. Trong cửa sổ xuất hiện bạn có thể làm nhiều thứ, ví dụ tự động khai báo các biến cục bộ hoặc các trường của lớp, tự động ép kiểu, ...


12. Để xem nhanh các định danh cần theo dõi, bạn đặt con trỏ lên định danh đó để làm sáng (highlight) tất cả các vị trí của chúng trong phương thức. Hữu ích nhất là đặt con trỏ lên định danh tại điểm trả về của phương thức để xác định những thay đổi của định danh đó trước khi trả về. Những định danh khai báo mà không sử dụng sẽ có đường lượn sóng màu xám phía dưới, giúp bạn kiểm soát được code của mình.

13. Chọn nhanh một khối code, đặt con trỏ tại vị trí bất kỳ trong vùng chọn, dùng **Alt+Shift+period** (dấu chấm) một số lần, vùng chọn sẽ mở rộng dần. **Alt+Shift+comma** (dấu phẩy) có tác dụng ngược lại, sẽ thu nhỏ vùng chọn. Nếu muốn chọn khối bao bởi dấu **{}** hoặc **()**, đặt con trỏ sau dấu **{** hoặc **(**, nhấn **Ctrl+Shift+[**; đặt con trỏ trước **}** hoặc **)** để bao luôn **{}** hoặc **()**.

Code Completion

14. Một phím tắt sử dụng rất nhiều, đặc biệt hữu dụng, là **Ctrl+Space** có khả năng đề nghị tên từ các từ viết tắt. Ví dụ bạn nhập **ISR** [**Ctrl+Space**] sẽ đề nghị `InputStreamReader`.

Phím tắt **Ctrl+Space** còn đề nghị các code template (có icon ) cho phép viết nhanh các khối code như `for`, `while`, `if`. Ví dụ: **for** [**Ctrl+Space**], chọn template thích hợp, rồi tiến hành sửa template theo nhu cầu. Bạn hãy tập viết `while`, `if`, ... theo cách này.

15. Chọn một khối code (từ dòng cuối lên dòng đầu) rồi click vào icon  nằm bên trái dòng thứ nhất. NetBeans sẽ đề nghị đưa các dòng code vừa chọn vào khối (comment, `if`, `for`, `while`, `try`, ...) một cách thuận tiện, gọi là chức năng Surround with.

16. Code completion thường thêm code theo chế độ chèn (insert) mặc định. Ngoài ra, còn chế độ viết đè (overwrite), cần khi chọn nhầm đề nghị và muốn chọn lại. Nhấn giữ **Ctrl** khi chọn đề nghị mới để thêm code vào theo chế độ viết đè.

17. Khi đặt tên cho định danh, code completion đề nghị một tên định danh, suy từ kiểu dữ liệu. Ví dụ: `StudentList` [Ctrl+Space], code completion đề nghị tên định danh `sl`.

Thậm chí, nhập một ký tự đầu, code completion cũng đề nghị tên có thể. Ví dụ: `Student a` [Ctrl+Space], code completion đề nghị tên định danh `aStudent`.

18. Code completion bằng **Ctrl+Space** cũng cho phép sinh (generate) constructors và getters/setters. **Ctrl+Space** rồi chọn constructor muốn sinh trong danh sách xuất hiện (có chữ **generate** bên phải). Riêng getters/setters dùng chức năng Insert code (**Alt+Insert**) thì tốt hơn.

Alt+Insert là phím tắt đặc biệt hữu dụng, có thể gọi là phím tắt "sinh code".

19. Code template là chức năng đặc biệt thuận tiện trong NetBeans. Viết từ tắt và [tab] để hoàn chỉnh code. Ví dụ: `psvm`[tab] cho public static void main, `psfi`[tab] cho private static final int, `sout`[tab] cho System.out.println(), `fore`[tab] cho vòng lặp foreach, `fori`[tab] cho vòng lặp có biến đếm, `newo`[tab] tạo đối tượng (rất thuận tiện).

Bạn luôn nên dùng code template để viết nhanh các từ khóa nếu có thể: **protected**, **private**, **return**, **implements**, **extends**.

Cần học từ từ danh sách code template, xem danh sách này trong Tools > Options > Editor, tab Code Templates.

20. Để viết khối try-catch có hai cách. Dùng code template sau: `trycatch`[tab]. Hoặc chọn khối cần bao bởi try-catch, **Alt+Enter** rồi chọn Surround with try {...}. Viết if-else cũng dùng code template: `ifelse`[tab].

Refactoring

21. Đôi khi bạn muốn gom một đoạn code vào một phương thức, để có thể gọi chúng lại từ phương thức khác. Điều này thực hiện dễ dàng bằng chức năng Introduce Method. Chọn đoạn code cần gom vào phương thức, nhấn **Alt+Shift+M**.

22. Bạn không cần phải khai báo trước các hằng số. Viết hằng số trước rồi tổ chức lại code bằng cách đặt con trỏ tại hằng số và nhấn **Alt+Shift+C**.

23. Bạn có thể đổi tên định danh hàng loạt bằng chức năng Replace (**Ctrl+H**), hoặc Refactoring > Rename. Tuy nhiên, nếu muốn đổi tên định danh hàng loạt và nhanh trong một phương thức, thuận tiện nhất là đặt con trỏ vào định danh cần đổi, nhấn **Ctrl+R** và nhập tên định danh mới.

Nếu bạn dùng hộp thoại Refactoring > Rename, click nút Preview để xem các khác biệt, đảm bảo rằng nó giống như mong muốn của bạn.

Navigation


24. Mỗi khi lưu code, History chứa một bản lưu của code, bạn có thể so sánh các bản lưu này với bản hiện hành bằng cách chọn bản lưu trong History. Nút History có trên toolbar (cạnh nút Source) của cửa sổ chứa tập tin. NetBeans dùng công cụ Diff để so sánh chúng.



25. Nếu bạn cần mở một tập tin bất kỳ, dùng chức năng Go to File, phím tắt **Alt+Shift+O**.

26. Chuyển đến nơi khai báo một tên định danh/phương thức, nhấn giữ **Ctrl** để tên định danh/phương thức chuyển thành link. Click lên link đó.

Ngoài ra bạn có thể dùng **Ctrl+B** (xem khai báo) hoặc **Ctrl+Shift+B** (xem source). Bạn cũng có thể quay trở lại bằng phím tắt **Alt+←** hoặc icon .

27. Bạn muốn chuyển nhanh qua lại giữa các tập tin đang mở, dùng **Ctrl+PageUp/PageDown**. Thuận tiện hơn, bạn dùng **Ctrl+Tab** và lựa chọn tập tin cần xem trong cửa sổ xuất hiện.

28. Trong một tập tin, bạn có thể đến ngay vị trí thay đổi lần cuối bằng **Ctrl+Q**, hoặc icon .

29. Bookmark trong code giúp bạn chuyển nhanh đến dòng code cần quan tâm. Bạn thêm/xóa bookmark bằng **Ctrl+Shift+M**, di chuyển đến bookmark bằng các icon  , hoặc dùng **Ctrl+Shift+period** (dấu chấm)/**Ctrl+Shift+comma** (dấu phẩy).

30. Các phím tắt sau, dùng nhảy nhanh đến các tab quan trọng: **Ctrl+number** với number 1: Projects, 2: Files, 4: Output, 5: Services, 7: Navigator. Một cửa sổ cũng khá quan trọng, **Ctrl+Shift+8**: Palette.

31. Để xem xét nhanh các lỗi, cảnh báo, bookmark trong code hiện hành và nhảy đến ngay điểm đó, bạn click vào các vạch màu (color stripe) trên cột cạnh thanh trượt bên phải. Chú ý chỉ thị màu của vạch: đỏ (lỗi), vàng (cảnh báo), tím (breakpoint), xám (bookmark).

32. Bạn cần tìm kiếm nhanh các tên định danh/phương thức, trong code hiện hành. Dùng **Ctrl+7** để đến cửa sổ Navigator, nhập tên phương thức/trường cần tìm. Nếu cần xem nhánh dẫn xuất của lớp hiện tại, dùng **Alt+F12**.

Workspace

33. Bạn muốn mở rộng tối đa NetBeans, dùng: **Alt+Shift+Enter**. Mở rộng thêm chút nữa: View > Toolbars > Small Toolbar Icons. Nếu cần loại luôn toolbar của cửa sổ: click phải vào biên trái của cửa sổ, bỏ chọn Show Editor Toolbar trong menu tắt.

34. Bạn đang code trong một cửa sổ và cần mở rộng tối đa kích thước cửa sổ để thêm không gian làm việc. Bạn có thể double-click lên tiêu đề của cửa sổ hiện hành. Tuy nhiên, nhanh nhất là dùng **Shift+Esc**.

35. Để chia (split) không gian làm việc theo chiều ngang hoặc dọc, bạn kéo cửa sổ sát trên hoặc sát bên trái, khung đỏ xuất hiện để nghị bạn chia không gian làm việc theo chiều ngang hoặc dọc, thả mouse sau khi chọn.

36. Bạn đã mở khá nhiều cửa sổ, bây giờ cần đóng nhanh chúng để không gian làm việc thoáng. Dùng **Ctrl+Shift+W** hoặc nhanh hơn: giữ phím **Shift** và click lên nút [x] của một cửa sổ bất kỳ trong chúng. Nếu bạn cần thao tác chi tiết hơn, dùng Window > Documents (**Shift+F4**).

37. Đôi khi bạn muốn thay đổi nhanh kích thước Font của vùng làm việc, để trình bày hoặc để dễ xem code, dùng: **Alt+Scroll**.

Time Savers

38. Chức năng auto-scanning của NetBeans làm mất nhiều thời gian chờ đợi khi khởi động. Bạn có thể bất hoạt chức năng này: vào Tools > Options > Miscellaneous, tab Files. Bỏ chọn Enable auto-scanning of sources.

39. Các phím tắt không thể thuộc ngay được. Bạn nên lên kế hoạch để thuộc dần các phím tắt, hãy cố gắng thay thế dần thói quen cũ, ép mình sử dụng các phím tắt này cho đến lúc trở thành thói quen. Sau một thời gian áp dụng phím tắt, tốc độ code của bạn sẽ tăng lên đáng kể.

Danh sách các phím tắt lấy từ: Help > Keyboard Shortcuts Card.

40. Bạn cần xem nhanh Javadoc của một phương thức, đặt con trỏ ngay tên phương thức rồi nhấn **Ctrl+Shift+Space**. Nếu việc xem Javadoc là thường xuyên, bạn hãy mở thường trực cửa sổ Javadoc: Window > Other > Javadoc.

41. Để thêm các Javadoc comment tự động cho lớp, phương thức, nhập **/**[Enter]**.

42. Bạn có thể nhân bản project một cách dễ dàng bằng chức năng Copy trong menu tắt của project, không cần dùng phím tắt và không cần Paste. Các lớp cũng được nhân bản bằng Copy (**Ctrl+C**) và Paste (**Ctrl+V**).

Miscellaneous

43. Nếu bạn dùng thường xuyên một thao tác, hãy tạo phím tắt cho nó trong Tools > Options > Keymap. Ví dụ: gán phím tắt **Ctrl+CLOSE_BRACKET** cho Project Properties trong Keymap, dùng phím tắt này khi cần mở Properties của project hiện hành.

44. NetBeans của bạn gặp sự cố? Bạn không nhất thiết phải cài đặt lại NetBeans, chỉ cần xóa thư mục **UserDir**. Với Windows 7: C:\Users\<username>\AppData\Roaming\ và với Windows XP: C:\Documents and Settings\<username>\Application Data\NetBeans\8.1\). Thậm chí, đôi khi bạn chỉ cần xóa /var/cache. Bạn có thể mất một số plug-in nhưng khôi phục lại chúng rất dễ dàng.

45. Bạn lỡ tay xóa mất tập tin trong project? Hãy vào menu tắt của project, chọn History > Revert Deleted để khôi phục tập tin vừa bị xóa.

46. NetBeans dùng chức năng Tools > Diff để so sánh hai tập tin. Khi so sánh, bạn chú ý màu chỉ thị: Red (**Delete**), Green (**Add**), Blue (**Modified**). Hãy cấu hình Diff bỏ qua khoảng trắng để so sánh hiệu quả hơn: vào Tools > Options > Miscellaneous, tab Diff và bỏ chọn: Ignore Changes In Inner Whitespace.

47. NetBeans dùng Ant làm build-system, muốn tùy biến quá trình build của NetBeans bạn cần biết dùng Ant. Ví dụ, target Ant sau chỉnh LF trong code thành CRLF, cho phép xem code bằng Notepad: <target name="fixCRLF"><fixcrlf srcdir="src" eol="crlf"/></target> vào build-impl.xml, chạy target này riêng rẽ với Ant bằng cách vào menu tắt của build.xml > Run Target và chọn target fixCRLF của ta.

Bạn cũng có thể tạo một phím tắt cho target của Ant, click để gắn cây tại build.xml, chọn target rồi chọn Create Shortcut... trong menu tắt.

48. Bạn có thể loại các ký tự space thừa trong code khi lưu. Vào Tools > Options > Editor, tab On Save, phần Remove Trailing Whitespace From, chọn All Lines hoặc Modified Lines Only (chỉ áp dụng cho những dòng có thay đổi).


49. Nếu bạn viết các ứng dụng Web, kích thước mặc định của tab (tính bằng Space, 4) có thể làm bạn khó theo dõi code. Vào Tools > Options > Editor > Formatting và đặt Number of Spaces per Indent lại bằng 2.

50. Nếu bạn chạy server (Tomcat, JBoss) trong NetBeans, nên xóa log (**Ctrl+L**) trước khi triển khai ứng dụng để có thể quan sát lỗi mới nhất. Có thể xóa các tập tin log cũ của Tomcat tại thư mục <Catalina Base>/logs, xem đường dẫn đến <Catalina Base> tại Tools > Servers.

Phụ lục 2 – Eclipse

Eclipse là IDE được các lập trình viên Java sử dụng nhiều nhất trong môi trường chuyên nghiệp. Để đạt tốc độ và hiệu quả lập trình cao nhất, lập trình viên cần tận dụng các khả năng của IDE. Tài liệu này giới thiệu 50 thủ thuật cần thiết nhất khi lập trình với Eclipse.

Source Code Editing

1. Định dạng code thường xuyên trong khi code là một thói quen tốt. Trong khi lập trình, bạn có thể định dạng code tự động bằng **Ctrl+Shift+F**. Nếu bạn muốn một định dạng code riêng, hãy tùy biến auto-format trong Windows > Preferences > Java > Code Style, tab Formatter. Tại đây bạn có thể chọn một profile [built-in] hoặc bấm vào nút Edit... để sửa profile đang chọn.
2. Bạn thường dùng comment (chú thích) để "khóa" tạm một đoạn code hoặc uncomment (mở chú thích) để cho "chạy" đoạn code đã comment. Hãy thêm/xóa comment bằng cách dùng: **Ctrl+/** hoặc **Ctrl+Shift+C**.
3. Bạn có thể xóa nhanh một dòng bằng tổ hợp phím **Ctrl+D**.
4. Bạn không cần phải nhập các dòng import. Thay vào đó dùng **Ctrl+Shift+O** để import một cách tự động. Tuy nhiên, khi xây dựng ứng dụng dùng nhiều thư viện, Eclipse sẽ hiển thị hộp thoại liệt kê danh sách các thư viện liên quan để import cho tên lớp ngay tại con trỏ.
5. Thao tác sao chép một dòng thường được sử dụng khi code. Bạn không cần copy-paste như thường lệ, hãy đặt con trỏ tại dòng cần sao chép, dùng **Ctrl+Alt+↓** để sao chép dòng hiện tại xuống phía dưới (chức năng Copy Lines) hoặc **Ctrl+Alt+↑** để sao chép dòng hiện tại lên phía trên (chức năng Duplicate Lines).
Có thể nối dòng (liên tục) bằng cách dùng **Ctrl+Alt+J**.
6. Tương tự như thao tác sao chép, để di chuyển một dòng hoặc một khối, đặt con trỏ tại dòng cần di chuyển hoặc chọn khối cần di chuyển, rồi dùng **Alt+↓** để di chuyển dòng/khối xuống phía dưới (hoặc **Alt+↑** để di chuyển dòng/khối lên phía trên). Như vậy, thay vì thực hiện copy rồi paste sang vị trí khác, ta sao chép khối (**Ctrl+Alt+↓/↑**) rồi di chuyển khối vừa sao chép đến chỗ mới (**Alt+↓/↑**).
7. Khi cần sao chép (copy & paste), di chuyển một khối code lớn, bạn hãy tận dụng khả năng *code folding*. Click dấu [-] để thu khối code lại, như vậy khối code chỉ nằm trên một dòng, dễ dàng được chọn để sao chép.
Dùng **Ctrl+-** (Numpad) để thu (collapse) thành viên của lớp hiện tại hoặc **Ctrl+/** (Numpad) để thu tất cả (collapse all). **Ctrl++** (Numpad) để mở rộng, và **Ctrl+*** (Numpad) để mở rộng tất cả.
Dùng **Ctrl+Shift+↓/↑** để di chuyển giữa các thành viên của lớp (property và method).
8. Bạn gọi một phương thức không tham số. Bạn nhập tên phương thức, theo sau là dấu (, Eclipse sẽ tự thêm dấu). Lúc này bạn bấm phím tab để về cuối dòng rồi nhập ;. Nếu phương thức gọi yêu cầu phải nhập tham số, dùng phím tab để di chuyển giữa các vị trí tham số với nhau.
9. Vào Window > Preferences > Java > Editor > Typing, rồi chọn Semicolons trong phần Automatically insert at correct position. Bây giờ, nhập một dấu ; tại vị trí bất kỳ trong dòng code, Eclipse sẽ chuyển dấu ; đến cuối dòng code và đặt con trỏ bên phải dấu ; đó để bạn có thể bắt đầu một dòng mới.
Có thể dùng **Ctrl+Enter** để chèn một dòng phía dưới dòng hiện tại hoặc **Ctrl+Shift+Enter** để chèn một dòng phía trên dòng hiện tại.
10. Bạn gọi một phương thức. Bạn nhập tên phương thức, theo sau là dấu (, Eclipse sẽ tự thêm dấu), nếu bạn muốn xem danh sách các tham số có thể có của phương thức đó, dùng **Ctrl+Shift+Space**.
11. Để xem lỗi, cảnh báo hoặc hint, di chuyển dấu nháy đến vị trí được báo lỗi hoặc cảnh báo, dùng **Ctrl+1** (quick fix) để hiển thị menu tắt Quick Fix có nhiều gợi ý hiệu chỉnh lỗi hoặc cảnh báo. Ngoài ra, có thể đặt con trỏ mouse lên vị trí lỗi/cảnh báo hoặc click vào icon cảnh báo bên trái để hiển thị tooltip nêu nguyên nhân của lỗi/cảnh báo.
12. Để xem nhanh các định danh cần theo dõi, bạn đặt con trỏ lên định danh đó để làm sáng (highlight) tất cả các vị trí của chúng trong phương thức. Hữu ích nhất là đặt con trỏ lên định danh tại điểm trả về của phương thức để xác định những thay đổi của định danh đó trước khi trả về. Tính năng này chỉ hoạt động khi ta chọn icon Toggle Mark Occurrences  (**Alt+Shift+O**) trên toolbar.
13. Chọn khối code liên quan gần nhất bằng **Alt+Shift+↑** (Select Closing Element). Thu nhỏ vùng chọn bằng **Alt+Shift+↓**. Một cách hay để chọn khối code là **double-click** ngay trước dấu ngoặc mở hoặc ngay sau dấu ngoặc đóng của khối code.
14. Source > Clean Up... giúp hiệu chỉnh nhiều vấn đề cùng một lúc và giúp thiết lập một phong cách code phù hợp. Ví dụ:
 - Chuyển các vòng lặp thành vòng lặp tốt hơn nếu có thể.
 - Đánh dấu tất cả các phương thức được dẫn xuất và viết lại với annotation @Override.
 - Tổ chức lại phần import.
 - Định dạng lại code.
 - Loại bỏ code không cần thiết.

Tuy nhiên, Clean Up thật sự hữu ích nếu thiết lập như một hành động được thực hiện khi lưu code. Vào Window > Preferences, chọn nhánh Java > Editor > Save Actions, chọn Additional actions. Nhấn nút Configure... cấu hình những hành động sẽ được thực hiện khi lưu code.

Code Assist

15. Phím tắt sử dụng rất nhiều, đặc biệt hữu dụng, là **Ctrl+Space** có khả năng đề nghị tên đầy đủ từ các ký tự ban đầu hoặc gần đúng, ví dụ **array[Ctrl+Space]** sẽ đề nghị Array, ArrayList, ... **ISR[Ctrl+Space]** sẽ đề nghị InputStreamReader (chức năng nhập các ký tự tắt này gọi là *camel case matches*, được thiết lập mặc định).

Phím tắt **Ctrl+Space** còn đề nghị các code template/snippet cho phép viết nhanh các khối code như for, while, if. Ví dụ: **for[Ctrl+Space]**, chọn template thích hợp, rồi tiến hành sửa template theo nhu cầu. Bạn hãy tập viết while, if, ... theo cách này.

16. Code assist, còn gọi là content assist, thường thêm code theo chế độ chèn (insert) mặc định. Ngoài ra, còn chế độ viết đè (overwrite), cần khi chọn nhầm đề nghị và muốn chọn lại. Nhấn giữ **Ctrl** khi chọn đề nghị mới để thêm code vào theo chế độ viết đè.

17. Khi đặt tên cho định danh, code assist đề nghị một tên định danh, suy từ kiểu dữ liệu. Ví dụ: **Student [Ctrl+Space]**, code assist đề nghị tên định danh **list** hoặc **studentList**.

Thậm chí, nhập một ký tự đầu, code assist cũng đề nghị tên có thể. Ví dụ: **StudentList a[Ctrl+Space]**, code assist đề nghị tên định danh **aList** hoặc **aStudentList**.

18. Sau khi lựa chọn trong danh sách mà code assist đề nghị, thay vì dùng phím Enter để chọn, hãy dùng phím **;** để chèn dấu ; ngay sau phương thức được chọn. Chú ý là code assist hỗ trợ import luôn cho kiểu mới sử dụng.

19. Code assist bằng **Ctrl+Space** cũng cho phép sinh (generate) constructor hoặc getter/setter cho từng thuộc tính của lớp. Nhưng để thêm đầy đủ, phải vào Source (**Alt+Shift+S**), chọn Generate Constructor using Fields... và Generate Getters and Setters...

20. Một cách dùng code assist khác, gọi là chế độ *insert common prefixes*, nhập một phần từ khóa rồi nhấn nhiều lần **Alt+/** để chọn các đề nghị xuất hiện xoay vòng.

21. Code template là chức năng đặc biệt thuận tiện trong Eclipse, cung cấp cho từng loại ngôn ngữ sử dụng (Java, HTML). Viết từ tắt và **[Ctrl+Space]**, chọn rồi **[Enter]** để hoàn chỉnh code. Ví dụ: **main[Ctrl+Space Enter]** cho public static void main, **sysout[Ctrl+Space]** cho System.out.println(), **for[Ctrl+Space Enter]** cho vòng lặp có biến đếm, **fore[Ctrl+Space Enter]** cho vòng lặp foreach, **new[Ctrl+Space Enter]** tạo đối tượng (rất thuận tiện), **private [Ctrl+Space Enter]** tạo phương thức private. Bạn luôn nên dùng chọn code template trực tiếp từ View **Templates** nếu mở sẵn. Thêm vào danh sách code template này bằng cách vào Window > Preferences > Java > Editor > Templates, nhấn nút New...

22. Phím tắt **Alt+Shift+Z** cũng rất hữu dụng, gọi là phím tắt "bọc khối", cung cấp nhiều hỗ trợ khi làm việc với khối code (block editing) như đưa khối code vào thân phát biểu if, for, do-while, try-catch, ... Chọn khối code, nhấn **Alt+Shift+Z** rồi chọn số tương ứng với loại phát biểu sẽ bọc khối code.

23. Ngược với tác dụng của phím tắt trên, để loại bỏ phát biểu bọc bên ngoài khối code, chọn khối code kể cả phát biểu bọc ngoài, nhấn **Ctrl+1** rồi chọn Remove surrounding statement.

24. Để viết khối try-catch có vài cách.

- Dùng code template sau: **try[Ctrl+Space Enter]**.
- Chọn khối cần bao bởi try-catch, **Alt+Shift+Z** rồi chọn Try/catch Block... (thường là số 6).
- Đặt con trỏ lên dòng code đang báo lỗi cần try-catch, **Ctrl+1** rồi chọn Surround with try/catch.

Refactoring

25. Phím tắt **Ctrl+1** đặc biệt hữu ích khi sửa lỗi, nhưng cũng rất hữu ích khi cần kiến trúc lại code. Nó cung cấp một danh sách rất nhiều hiệu chỉnh cho code, tùy theo ngữ cảnh: Invert equals, Invert 'if' statement, Replace with getter/setter, Convert local variable to field, Extract to local variable, Convert 'switch' to 'if-else', Add cast to, ...

26. Đôi khi bạn muốn gom một đoạn code vào một phương thức, để có thể gọi chúng lại từ phương thức khác. Điều này thực hiện dễ dàng bằng chức năng Extract Method. Chọn đoạn code cần gom vào phương thức, nhấn **Alt+Shift+M**.

Cách khác, chọn đoạn code, nhấn **Ctrl+1** và chọn Extract to method.

27. Bạn có thể đổi tên định danh hàng loạt bằng chức năng Find/Replace (**Ctrl+F**), hoặc Refactoring > Rename. Tuy nhiên, nếu muốn đổi tên định danh nhanh trong một phương thức, thuận tiện nhất bạn nên dùng tính năng rename (**Alt+Shift+R** hoặc **Ctrl+2 R**) để đổi tên tại tất cả các vị trí tham chiếu đến, thậm chí trong một class (file) khác.

Navigation

28. Trong Eclipse, các khái niệm quan trọng nhất cần biết là *Perspective*, *View* và *Editor*. Một *Perspective* chứa nhiều *View* và *Editor*.

- View: thường cung cấp tính năng xem và duyệt các thông tin của project như cấu trúc project (Package Explorer), cấu trúc lớp (Outline), ...

Mở các View bằng Window > Show View (**Alt+Shift+Q**).

- Editor: cửa sổ biên soạn một thành phần nào đó được liệt kê trong View. Trong cửa sổ biên soạn các tập tin được mở thành từng tab.

- Perspective: phối cảnh làm việc, trong đó bố trí các View và Editor cho phù hợp với loại công việc (code, debug, ...). Mỗi lần thay đổi loại công việc, các View và Editor bố trí lại theo phối cảnh mới.

Mở các Perspective bằng Window > Open Perspective, hoặc từ các icon góc phải toolbar.

Nếu đóng một số View của Perspective, có thể phục hồi phối cảnh bằng Window > Restore Perspective...

Duyệt qua các View/Editor đang mở và chuyển đến View/Editor mong muốn bằng **Ctrl+F7**.

29. Nếu bạn cần mở một tập tin bất kỳ, dùng chức năng Go to File, phím tắt **Ctrl+Shift+R**, nhập một phần tên tập tin, chọn theo danh sách gợi ý bên dưới và nhấn **Enter**.

30. Chuyển đến nơi khai báo một tên định danh, phương thức, nhấn giữ **Ctrl** để tên định danh, phương thức chuyển thành link. Click lên link đó. Cách khác, khi con trỏ đang nằm ngay tên định danh, phương thức, nhấn **F3**.

Khi con trỏ mouse nằm ngay trên lời gọi phương thức, nhấn giữ phím **Shift** để xem mã nguồn của phương thức.

31. Khi làm việc với khối code, việc xác định cặp `{}` bao khối code là cần thiết, đặt con trỏ sau dấu `{` hoặc `}`, dấu cùng cặp sẽ được đánh dấu. Nếu con trỏ ngay giữa khối code, nhấn **Ctrl+Shift+P**, con trỏ sẽ chuyển lên ngay sau dấu `{`, nhấn **Ctrl+Shift+P** lần nữa, con trỏ sẽ chuyển xuống ngay sau dấu `}` cùng cặp.

32. Bạn có thể nhanh chóng truy cập các View hoặc Editor đang mở, các Perspective, Preferences và rất nhiều thứ khác. Đơn giản nhấn **Ctrl+3** và nhập một phần từ khóa, Eclipse sẽ cố gắng tìm thứ gì đó phù hợp với từ khóa gợi ý.

Cách khác, nhập từ khóa vào ô Quick Access nằm cuối toolbar.


33. Bạn muốn chuyển nhanh qua lại giữa các tập tin đang mở trong Editor, dùng **Ctrl+PageUp/PageDown**. Hoặc bạn dùng **Ctrl+E** rồi lựa chọn tập tin cần xem trong cửa sổ xuất hiện. Thuận tiện hơn, duyệt nhanh qua các tập tin đang mở trong Editor bằng **Ctrl+F6**.

34. Bạn có thể đến ngay vị trí thay đổi lần cuối trong tập tin thay đổi lần cuối bằng **Ctrl+Q**.


35. Bookmark trong code giúp bạn chuyển nhanh đến dòng code cần quan tâm. Bạn thêm/xóa bookmark bằng cách click lên cột xám bên trái dòng code, chọn Add Bookmark... hoặc Remove Bookmark. Truy cập bookmark cụ thể trong danh sách các bookmark, hiển thị trong View Bookmark (Window > Show View).

36. Để xem xét nhanh các lỗi, cảnh báo, bookmark trong code hiện hành và nhảy đến ngay điểm đó, bạn click vào các vạch màu (error stripe) trên cột cạnh thanh trượt bên phải. Chú ý chỉ thị màu: đỏ (lỗi), vàng (cảnh báo), xanh (bookmark).

Cách khác, duyệt nhanh qua các lỗi/cảnh báo (warning) bằng **Ctrl+period** (dấu chấm). Khi con trỏ nhảy đến lỗi/cảnh báo, dùng **Ctrl+1** để hiệu chỉnh lỗi.

37. Khi tạo một chú thích (`/**[Enter]`) với **TODO**, trình biên dịch tạo ra một task tương ứng gắn (icon  tại cột xám bên trái) để nhắc. Thẻ TODO này và các thẻ tương tự được tạo và cấu hình tại Window > Preferences, chọn nhánh Java > Compiler > Task Tags. Các thẻ này có độ ưu tiên khác nhau, thể hiện mức độ quan trọng khác nhau của task. Truy cập task cụ thể trong danh sách các task, hiển thị trong View Tasks (Window > Show View).

38. Bạn cần xem khung (outline) của mã nguồn, có thể mở View Outline hoặc **F4**. Nhưng nhanh nhất là dùng **Ctrl+F3** hoặc nhanh hơn, dùng **Ctrl+O**. Nhấn **Ctrl+O** một lần nữa để xem các thành viên thừa kế được.

39. Tính năng breadcrumb của Eclipse là một cách nhanh hơn để có thông tin về "đường dẫn" từ project → package → class → method. Bật tính năng này bằng **Alt+Shift+B** hoặc click icon Toggle Breadcrumb  trên toolbar. Click lên từng dấu ▶ sau mỗi thành phần trên "đường dẫn" này để lấy thêm thông tin chi tiết.

40. Eclipse lưu trữ danh sách các trang đã đi qua, các chỉnh sửa trên một trang. Dùng **Alt+ ←/→** để duyệt lui/tới trong danh sách này.

Workspace

41. Bạn đang code trong một Editor và cần mở rộng tối đa kích thước cửa sổ để thêm không gian làm việc. Bạn có thể double-click lên tiêu đề của cửa sổ hiện hành.

42. Để chia (split) không gian làm việc theo chiều ngang hoặc dọc, bạn kéo cửa sổ sát trên hoặc sát bên trái, khung màu xanh lá xuất hiện đề nghị bạn chia không gian làm việc theo chiều ngang hoặc dọc, thả mouse sau khi chọn.

43. Bạn đã mở khá nhiều cửa sổ, bây giờ cần đóng nhanh chúng để không gian làm việc thoáng. Nhấp phải lên cửa sổ cần giữ lại, chọn Close Others hoặc Close All (**Ctrl+Shift+W**) nếu cần đóng tất cả lại. Bạn có thể Bấm **Ctrl+W** để chỉ đóng cửa sổ làm việc hiện hành.

44. Editor quá nhiều tab hoặc Project bạn quá nhiều thứ hiển thị trong khi bạn chỉ cần tập trung vào một trong số đó. Click phải vào Project (hoặc Package, Folder) và chọn Go Into.

Time Savers

45. Bạn nên lên kế hoạch để thuộc dần các phím tắt, hãy cố gắng thay thế dần thói quen cũ, ép mình sử dụng các phím tắt này cho đến lúc trở thành thói quen. Sau một thời gian áp dụng phím tắt, tốc độ code của bạn sẽ tăng lên đáng kể.

Danh sách các phím tắt có thể xem nhanh bằng **Ctrl+Shift+L**. Dùng tổ hợp phím này hai lần sẽ mở hộp thoại dùng thiết lập các phím tắt.

46. Bạn cần xem nhanh Javadoc của một phương thức, đặt con trỏ ngay tên phương thức để hiển thị tooltip. Nhấn **F2** để xem toàn bộ văn bản. Nếu việc xem Javadoc là thường xuyên, bạn hãy mở thường trực View Javadoc: Window > Show View > Javadoc. Nếu dùng **Shift+F2**, Javadoc sẽ mở trong tab Browser.

47. Để thêm các Javadoc comment tự động cho lớp, phương thức, nhập **/**[Enter]**, hoặc dùng **Alt+Shift+J**.

48. Bạn có thể nhân bản project một cách dễ dàng bằng chức năng Copy (**Ctrl+C**) và Paste (**Ctrl+V**) trong menu tắt của project. Các lớp cũng được nhân bản bằng cách tương tự.

Nếu copy một đoạn code và paste (**Ctrl+V**) vào một project, một lớp mới sẽ được thêm vào project để chứa đoạn code đó.


Miscellaneous

49. Khi biên soạn một tập tin, Eclipse lưu phiên bản trước vào Local History, bạn có thể so sánh các bản lưu này với bản hiện hành bằng cách chọn bản lưu trong Refactor > History... Truy cập Local History từ menu tắt của tập tin hoặc từ View History.

Dùng công cụ Compare With... để so sánh các bản lưu với nhau theo thời gian.

Dùng công cụ Replace With... để quay trở về phiên bản trước.

Nếu bạn lỡ xóa mất tập tin trong project. Hãy vào menu tắt của project, chọn Restore from Local... để khôi phục tập tin vừa bị xóa.

50. Một số ô nhập (có icon  nhỏ, bên trái) trong các hộp thoại của Eclipse cũng hỗ trợ code assist, ví dụ ô nhập Superclass của hộp thoại New Java class. Nhập một phần từ khóa rồi nhấn **Ctrl+Space** để được hỗ trợ.

Các ô nhập cũng hỗ trợ con trỏ nhắc thông minh (smart caret), với chuỗi nhập theo quy ước Camel (viết hoa các ký tự đầu mỗi từ), **Ctrl+→/←** để nhảy đến tới/lui từng từ, **Ctrl+Shift+→/←** để chọn từ sau/trước (để nhập từ khác thay thế) và **Ctrl+Del** để xóa từ ngay sau con trỏ nhắc.

Tài liệu tham khảo

(Theo năm xuất bản)

- [1] Khalid A. Mughal, Rolf W. Rasmussen – **A Programmer's Guide to Java® SE 8 Oracle Certified Associate (OCA). A Comprehensive Primer** – Addison - Wesley, 2016. ISBN: 978-0-13-293021-5
- [2] Kathy Sierra, Bert Bates – **OCA OCP Java SE 7 Programmer I & II Study Guide** – McGraw-Hill Education, 2015. ISBN: 978-0-07-177199-3
- [3] Richard Warburton – **Java 8 Lambdas** – O'Reilly Media, Inc., 2014. ISBN: 978-1-449-37077-0
- [4] Robert Liguori, Patricia Liguori – **Java 8 Pocket Guide** – O'Reilly Media, Inc., 2014. ISBN: 978-1-491-90086-4
- [5] Paul Deitel, Harvey Deitel – **Java How To Program, Early Objects, 10th Edition** – Prentice Hall, 2014. ISBN: 978-0-1338078-0-6
- [6] Jan Graba – **An Introduction to Network Programming with Java, Thrid Edition** – Springer, 2013. ISBN: 978-1-4471-5254-5
- [7] **Poornachandra Sarang – Java Programming – McGraw-Hill, 2012. ISBN 978-0-07-163361-1**
- [8] Jacquie Barker – **Beginning Java Objects: From Concepts to Code, Second Edition** – Apress, 2005. ISBN: 1-59059-457-6