Lời nói đầu

Giáo trình này được soạn thảo 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 C, xin xem trước tài liệu Ngôn ngữ lập trình C, cùng người viết.
- 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 người viết.
- 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 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: 01/04/2015

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

Java – Ngôn ngữ

Kiểu dữ liệu

Trị được định nghĩa như một thứ có thể đặt vào một biến hoặc có thể truyền 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 định nghĩa 8 kiểu dữ liệu cơ bản, gồm 4 loại:

- Kiểu 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 bất kỳ, như on/off hoặc yes/no.

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

- Kiểu văn bản – `char`

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 Φ)

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 không kết thúc với `\0`.

- Kiểu nguyên – `byte`, `short`, `int` và `long`

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

`2`	Dạng thập phân của số nguyên 2.
`077`	Trị bát phân bắt đầu bằng `0`.
`0xBAAC`	Trị thập lục phân bắt đầu bằng `0x`.

Mọi kiểu số trong Java 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`

Chuỗi số nguyên cho phép dùng ký tự `_` để phân cách hàng ngàn. 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 !`

- Kiểu dấu chấm động – `float` và `double`.

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 (`float`), D hoặc d (`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 dấu +.

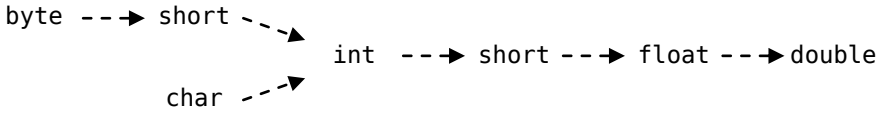
Chuỗi số có dấu chấm động ngầm định là `double`. Khi tính toán chính xác với số thực, bạn nên dùng lớp `BigDecimal` thay thế cho `float` và `double`.

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`.

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, 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 phải dùng ép kiểu (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.

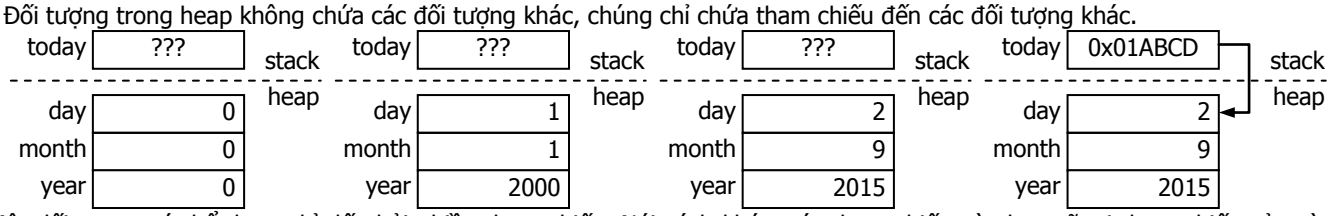
Java có các kiểu tham chiếu sau: annotation, array, class, enumeration và interface.

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

```
public 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() { ... }
}

public class Main {
  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, như sau:
- 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 bằng 0 hoặc null.
- Khởi tạo tường minh các thuộc tính của today nếu chúng khai báo với trị mặc định.
- Constructor được gọi sẽ thực hiện, các thuộc tính được khởi tạo từ tham số truyền cho constructor.
- Trị trả về từ toán tử new tham chiếu đến đối tượng mới trong heap, tham chiếu này được lưu trong biến tham chiếu.
- 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 alias (bí danh) của đối tượng, một đối tượng có thể được truy cập gián tiếp bởi một trong các alias của nó.
Chuyển kiểu mở rộng gọi là upcasting, thường thực hiện ngầm định, sẽ nâng cấp kiểu con (subtype) thành kiểu cha (supertype) của nó. Chuyển kiểu thu hẹp gọi là downcasting, thường phải dùng ép kiểu, tuy nhiên vẫn có khả năng ném ra runtime exception, thường phải dùng toán tử instanceof để kiểm tra tương thích kiểu trước khi ép kiểu.

```
Object strObj = "upcast me";      // Widening: Object <---- String
Object intObj = new Integer(10);
String s1 = (String) dtrObj;      // Narrowing: String <---- Object
String s2 = (String) intObj;      // ném java.lang.ClassCastException, int không ép kiểu được thành String
```

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 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 (thông qua tham chi iếu) nhưng không thay đổi được tham chiếu đến đối tượng đó (do tham chiếu truyền bằng trị).
Ví dụ sau cho thấy Java truyền bằng trị, phương thức gọi thay đổi tham chiếu bản sao, không ảnh hưởng đến tham chiếu truyền đến nó.

```
// tham số hình thức Circle là bản sao của tham số thực c, trong phương thức manipulate()
// thử cho tham chiếu bản sao này thay đổi bằng cách chỉ đến đối tượng Circle khác.
public void manipulate(Circle circle) {
  circle = new Circle(3);
}

Circle c = new Circle(2);
manipulate(c);
System.out.println("Radius: " + c.getRadius());  // Radius: 2. Tham chiếu c không thay đổi sau khi gọi hàm.
```

4. Annotation

Annotation cung cấp một cách liên kết metadata (dữ liệu về dữ liệu) với các thành phần của chương trình 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, như @Override, @Deprecate, @SuppressWarnings.

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

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Feedback { } // marker
public @interface Feedback {
  String reportName();
} // single value
public @interface Feedback {
  String reportName();
  String comment() default "None";
} // multi value
```

Dùng annotation 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);
```

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), 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 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.

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

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

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

```
int[] array1 = new int[5];                  // tạo mảng có 5 phần tử
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 cho các phần tử, rồi gán cho array3

Game[] gameList;                            // khai báo mảng và khởi tạo mặc định là null
gameList = new Game[10];                    // khởi tạo mảng có 10 phần tử nhưng vẫn rỗng
gameList[0] = new Game();                   // khởi tạo 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 gọi là mảng nhiều chiều (multidimensional array).

Biến

1. Biến và tầm vực

Có hai cách khai báo một biến:

- Bên trong một phương thức, gọi là biến cục bộ (local); còn gọi là biến *automatic*, *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 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 thực thi đi vào phương thức và bị hủy khi phương thức kết thúc. Chúng 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 không được khuyến khích.

- 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: biến `static` được tạo khi lớp được nạp vào JVM trong thời gian chạy, mặc dù lớp chưa sinh thể hiện nào, và tiếp tục tồn tại khi lớp còn tồn tại trong JVM.

  + 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.

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 (tiền tệ như $, ¢, ¥, hoặc £) là hợp lệ nhưng nên tránh dùng. Thực tế, không hạn chế số ký tự của định danh.

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` sau 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%8s%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 C/C++, tuy nhiên có một số khác biệt bạn cần chú ý.
- 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.
Các toán tử điều kiện, bao gồm các toán tử `&&` và `||`, cũng ngắn mạch tương đương với toán tử `&` và `|`.
Ngắn mạch (short-circuit) biểu thức logic là đặc điểm thừa kế C/C++. Khi toán tử `&&` được dùng, nếu biểu thức bên trái định trị `false` thì không cần định trị biểu thức tiếp. 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++, 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 hỗ trợ bao gồm `~`, `&`, `^` và `|`; tương đương các tác vụ bitwise NOT (bù), bitwise AND, bitwise XOR, 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 k bit tương đương với phép chia toán hạng thứ nhất cho 2^k. Khi dịch số âm, dấu (bít 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 `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
```

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`. Ngoài ra còn hỗ trợ kiểu `enum` và `String` (Java 7).
- Bạn dùng các phát biểu sau để điều khiển phát biểu lặp, chú ý `break` và `continue` không bắt buộc 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_expresion1);
    statement4;
  } while (test_expresion2);
```

+ <*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

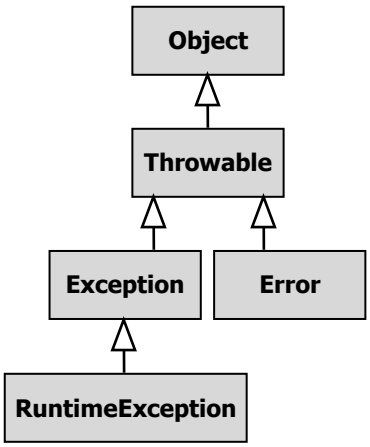
Exception (ngoại lệ) là cơ chế được dùng bởi nhiều ngôn ngữ lập trình hướng đối tượng để mô tả điều phải làm khi một sự cố không mong đợi xuất hiện. Sự cố này có thể là một lỗi, ví dụ phương thức triệu gọi với tham số không hợp lệ, kết nối mạng thất bại hoặc người dùng yêu cầu mở một tập tin không tồn 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. Chúng xuất hiện từ các điều kiện ngoài, dễ xảy ra trong khi chương trình làm việc. Ví dụ, tập tin yêu cầu không tìm thấy hoặc kết nối mạng 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 thể hiện lỗi gọi là exception thời gian chạy (runtime exception), ví dụ các exception ném ra khi thử truy xuất vượt khỏi phạm vi của mảng (out-of-bounds index), chia cho 0 (divide by zero) và null pointer.

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.



Lớp `Exception` là lớp cơ sở của cả hai loại exception trên. Thay vì kết thúc chương trình, tốt nhất bạn viết code xử lý exception và tiếp tục.

Lớp `Error` là lớp cơ sở được dùng cho các điều kiện lỗi nghiêm trọng, không kiểm soát được. Chương trình không thử phục hồi khi gặp các lỗi này, thường bạn cố gắng bảo lưu công việc người dùng đang làm và kết thúc chương trình.

Lớp `RuntimeException`, thừa kế từ lớp `Exception`, là lớp cơ sở được dùng cho các unchecked exception xuất hiện do lỗi (bug) trong chương trình. 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 có vấn đề trong thiết kế hoặc cài đặt. Ví dụ, exception `NullPointerException` ném ra khi bạn dùng một tham chiếu không liên kết với một đối tượng mới tạo hoặc có sẵn 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 xuất phát exception có thể tự xử lý exception hoặc ném (throw) exception trở lại phương thức gọi (caller) để báo rằng có vấn đề xuất hiện. Phương thức gọi cũng giải quyết tương 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:

```
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.TestProject.main(AddArguments.java:5)
```

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 truy cập mảng đã vượt khỏi giới hạn.
`ClassCastException`	Chỉ định việc thử ép kiểu xuống lớp con 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 truy cập mảng, chuỗi, vector, … đã vượt khỏi 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 uncheked exception.

a) Phát biểu `try-catch`

Java cung cấp cơ chế bắt các checked exception ném ra và thử phục hồi, bạn dùng khối `try-catch` bao lấy phát biểu có ném exception để bắt lấy exception nếu nó được ném ra.

```
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
```

Nếu thân khối `try-catch` 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 e3) {
    // 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` đầu tiên, vì `Exception` là lớp cơ sở của các exception nên khối này sẽ xử lý tất cả các exception bắt được, các khối `catch` tiếp sau không bao giờ được xử lý.

Java 7 cho phép kết hợp các exception trong một khối `catch`, tuy 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 `final` định nghĩa một khối code luôn luôn được thực hiện, cho dù có một exception đã ném ra.

```
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`.

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 không xử lý exception đó mà ném tiếp cho phương thức gọi nó. Bạn chỉ khai báo có ném exception và chuyển trách nhiệm xử lý exception cho phương thức gọi nó.

```
// 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 tiếp
public void delegate1() throws MyException {
  try {
    trouble();
  } catch (MyException e) {
    System.err.println(("Error caught and rethrown");
    throw e;
  }
}
// delegate2() không xử lý exception, ném tiếp
public void delegate2() throws MyException {
  delegate1();
}
// resolve() xử lý exception
public void resolve() {
  try {
    delegate2();
  } catch (MyException e) {
    System.err.println(e.getMesssage());
  }
}
```

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 ném ra `Exception` (lớp cha của `IOException`).

d) Xử lý stack-trace

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 này, và cứ tiếp tục như thế cho đến khi gặp hàm `main()`, do `main()` không xử lý exception, exception in ra ngõ xuất chuẩn và chương trình dừng thực thi.

```
public class UsingException {
  public static void main(String[] args) {
    try {
      trouble2();
    } catch (Exception exception) {
      System.err.println(exception.getMessage());
      exception.printStackTrace();
      // lấy thông tin stack-trace
      StackTraceElement[] traces = exception.getStackTrace();
      System.out.println("Class\t\tFile\t\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 từ 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.
- Thường chỉ khai báo các constructor sau: không tham số, `String` là tham số, `Throwable` là tham số, `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` vừa 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`, do 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`, 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 tự động đóng phải cài đặt interface `java.lang.AutoCloseable` hoặc interface `java.io.Closeable` (thừa kế `AutoCloseable`). Do đó, bạn có thể cài đặt interface `AutoCloseable` cho lớp tài nguyên riêng của bạn, hiện thực phương thức `close()` của interface; sau đó có thể sử dụng lớp tài nguyên của bạn trong phát biểu try-with-resources.
Có thể có nhiều dòng lệnh trong cặp `()` của `try`, mỗi dòng cho một loại tài nguyên `AutoCloseable`, dòng lệnh cuối không cần `;` 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, asserion 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 một biến 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`.
Một ưu điểm của assertion là chúng có thể loại bỏ hoàn toàn khỏi code, bạn có thể cho phép assertion trong lúc phát triển chương trình và bất hoạt nó trong khi biên dịch 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 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`:

```
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 variant).
Hậu điều kiện là giả định về trị hoặc mối quan hệ của các biến lúc kết thúc phương thức. Ví dụ, 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 luôn không âm.
Ưu điểm của assertion là có thể bất hoạt trong thời gian chạy. Mặc định assertion bất hoạt, khi phát triển code ta mới 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, vào Properties > Run, thêm tùy chọn `-ea` vào VM Options.

Biểu thức Lambda

1. Biểu thức Lambda
Biểu thức Lambda là một cấu trúc rất 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. Do khả năng suy kiểu (type inference) mạnh, biểu thức Lambda 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 kiểu dữ liệu), bên phải của `->` là thân của phương thức.
Kỹ thuật này cho phép viết ngắn gọn các kiểu lồng, thường sử dụng khi viết các 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
Runnable r = new Runnable() {
  @Override public void run() {
    System.out.println("Hello World");
  }
};

// (2) biểu thức Lambda tương đương
Runnable r = () -> System.out.println("Hello World");
```

2. Chuyển đổi biểu thức Lambda
Ví dụ trên cho thấy biểu thức Lambda được dùng như một trị, nó tự động chuyển thành đối tượng đúng kiểu, ở trên là kiểu interface `Runnable`.
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. Việc trả về này gọi là chuyển đổi (conversion) biểu thức Lambda thành thực thể cài đặt kiểu interface dự kiến.
- Kiểu interface dự kiến phải có *duy nhất một phương thức*, gọi là interface có phương thức trừu tượng đơn (SAM – single abstract method).
- Phương thức của interface dự kiến có signature phù hợp với các tham số của biểu thức Lambda.

Nói cách khác, biểu thức Lambda tạo một thực thể cài đặt interface dự kiến trước, thân của biểu thức Lambda dùng để cài đặt cho phương thức duy nhất của interface.

3. Tham chiếu chỉ đến phương thức (method reference)

Xem biểu thức Lambda sau:

```
(MyObject myObj) -> myObj.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 `MyObject` và trả về một chuỗi từ phương thức `toString()` của tham số nhận vào. Với biểu thức Lambda như vậy, Java 8 cung cấp một cú pháp dễ đọc/viết hơn:

```
MyObejct::toString()
```

Cách viết tắt của biểu thức Lambda như trên gọi là tham chiếu chỉ đến phương thức. Ví dụ:

```
List<String> list = Arrays.asList("dog", "cat", "fish", "iguana", "ferret");
// sắp xếp danh sách không phân biệt chữ hoa chữ thường bằng ba cách.
// (1) lớp nội vô danh
Collections.sort(list, new Comparator<String>() {
  @Override public int compare(String s1, String s2) {
    return s1.compareToIgnoreCase(s2);
  }
});

// (2) biểu thức Lambda
Collections.sort(list, (s1, s2) -> s1.compareToIgnoreCase(s2));
// viết cho rõ
Comparator<String> comp = (s1, s2) -> s1.compareToIgnoreCase(s2);
Collections.sort(list, comp);

// (3) tham chiếu chỉ đến phương thức
Collections.sort(list, String::compareToIgnoreCase);

System.out.println(list);
```

Encapsulation

Một trong những phương pháp cơ bản chúng ta dùng 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 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, dùng camel case (viết hoa các ký tự đầu các từ). Ví dụ, lớp `Fish`.
Thuộc tính (attribute) của đối tượng được định nghĩa thành các trường (field) trong lớp. Một trường của lớp là một biến dùng lưu trị 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 vẽ lớp một cách trực quan, tìm hiểu sử dụng công cụ tạo sơ đồ UML: UMLet tại http://www.umlet.com/
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, thể hiện (instance) còn được gọi là thực thể hoặc đối tượng.
Giao kết (contract) định nghĩa lớp cung cấp các dịch vụ nào, nghĩa là cho biết các phương thức có thể triệu gọi từ thể hiện của lớp. Cài đặt (implementation) 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 đối tượng đó, mà không cần biết dịch vụ 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.

```
public class CharStack {
  private char[] stackArray;      // cài đặt stack bằng mảng
  private int top;                // đỉnh của stack.
  // constructor
  public CharStack(int capacity) {
    stackArray = new char[capacity];
    top = -1;
  }
  // business methods
  public void push(char element) { stackArray[++top] = element; }
  public char pop() { return stackArray[top--]; }
  public char peek() { return stackArray[top]; }
  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 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();
  }
}
```

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 đến đối tượng hiện hành. Trong thân của phương thức thành viên, tham chiếu `this` được dùng như tham chiếu đến một đối tượng để truy cập thành viên của đối tượng đó.
Tham chiếu `this` là một tham chiếu `final`, không thể sửa đổi. Tham chiếu `this` thường được dùng:
- Phân biệt giữa biến thành viên và tham số truyền cho phương thức. Trong một lớp, thành viên được truy cập tường minh thông qua tham chiếu `this`, hoặc không tường minh không thông qua `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 `this` để phân biệt.

```
public class Point {
  int x, y;
  public Point(int x, int y) {
    x = x;                                  // không tham chiếu đến this.x
  }

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

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

```
    }
}
```

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

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 đã có để tạo lớp mới, đối tượng của các lớp đã 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 này có tham chiếu đến lớp khác 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 này thừa kế lớp khác khi xây dựng lớp.

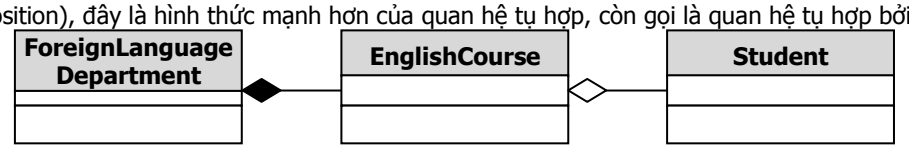
Thực tế, một lớp được tạo ra bằng cách phối hợp cả hai cách trên.

1. Composition

Một lớp có các dữ liệu thành viên là các đối tượng thuộc các lớp khác gọi là lớp phức hợp (composed class). Một lớp có các đối tượng là dữ liệu thành viên của lớp phức hợp được 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 toàn cục, biến cục bộ của hàm thành viên hoặc đối 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 chúng cùng mức, không có lớp nào quan trọng hơn. Quan hệ tụ hợp là quan hệ giữa toàn thể và bộ phận trong đó một lớp biểu diễn cái lớn hơn (tổng thể, lớp bao), còn lớp kia biểu diễn 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. 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.

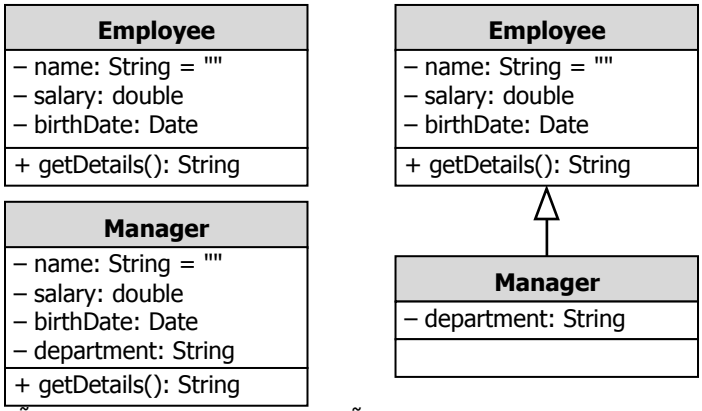


Quan hệ tụ hợp: quan hệ hợp thành (trái) và quan hệ kết tập (phải)

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

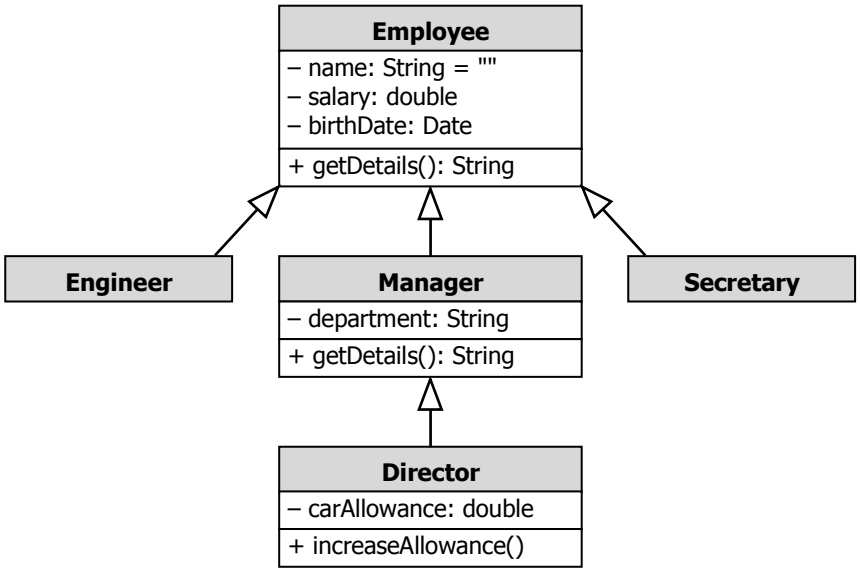
```
public class Employee {
  protected String name = "";
  protected double salary;
  protected Date birthDate;

  public String getDetails() { . . . }
}

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



Dẫn xuất lớp con từ một lớp có sẵn cho phép dùng lại lớp đã viết.

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.

Hình bên trình bày lớp cơ sở (base class, superclass, parent class) `Employee` và ba lớp con (derived class, subclass, child class) `Engineer`, `Manager` và `Secretary`. Lớp `Manager` cũng dẫn xuất ra lớp con `Director`. Lớp `Manager` thừa kế tất cả thuộc tính và phương thức của lớp `Employee` và có thêm thuộc tính mới `department`, viết lại phương thức `getDetails()` của lớp cha. Lớp `Director` thừa kế tất cả thành viên của `Employee` và `Manager`, có 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 hierachy). 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 tùy biến các hành vi thừa kế được và thêm vào các thuộc tính và 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. Điều khiển truy cập

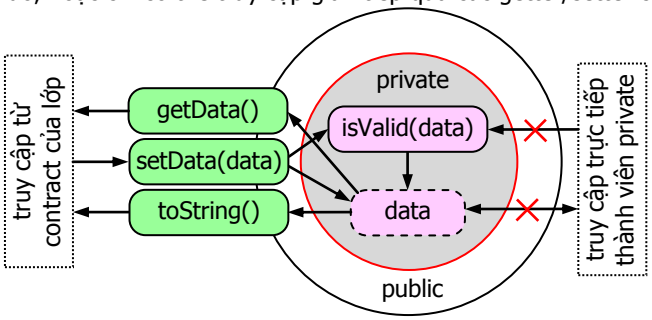
Biến và các phương thức của lớp có thể có một trong bốn cấp độ truy cập: `public`, `protected`, `default` và `private`. Các lớp cũng có cấp độ truy cập `public` hoặc `default` (còn gọi là truy cập cấp gói, package accessibility). Cấp độ truy cập được chỉ định bằng cách dùng các bổ từ truy xuất (access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

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

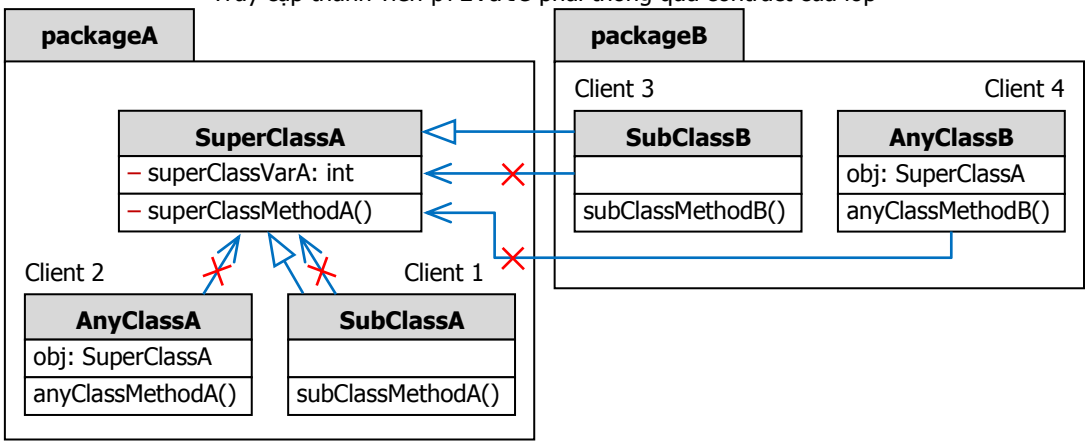
Bảng sau trình bày các cấp độ truy cập:

Access modifier	Trong cùng lớp	Trong cùng package	Trong lớp con	Khắp nơi
`private`	Yes			
`default`	Yes	Yes		
`protected`	Yes	Yes	Yes	
`public`	Yes	Yes	Yes	Yes

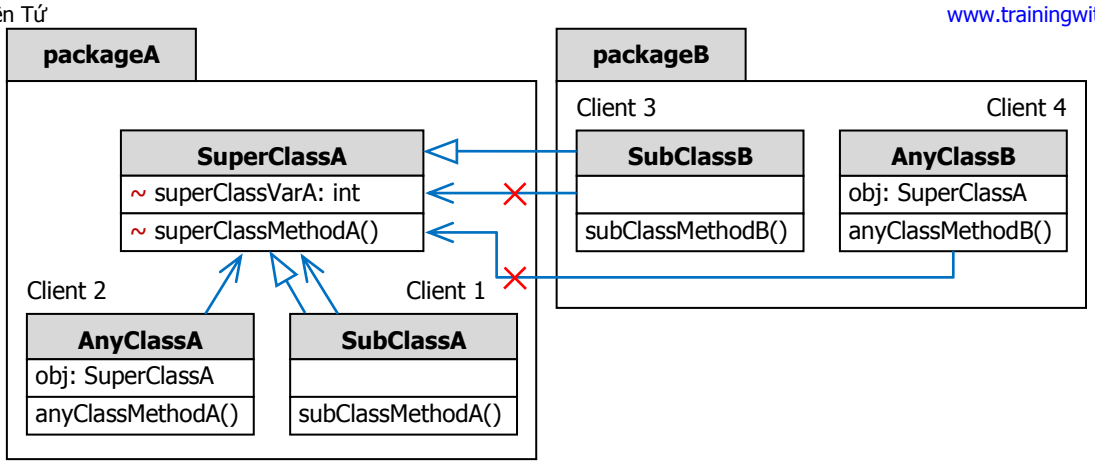
Một biến hoặc phương thức được đánh dấu `private` chỉ được truy cập bởi các phương thức và thành viên của cùng lớp với biến hoặc phương thức `private` đó, hoặc chỉ có thể truy cập gián tiếp qua các getter/setter của lớp.



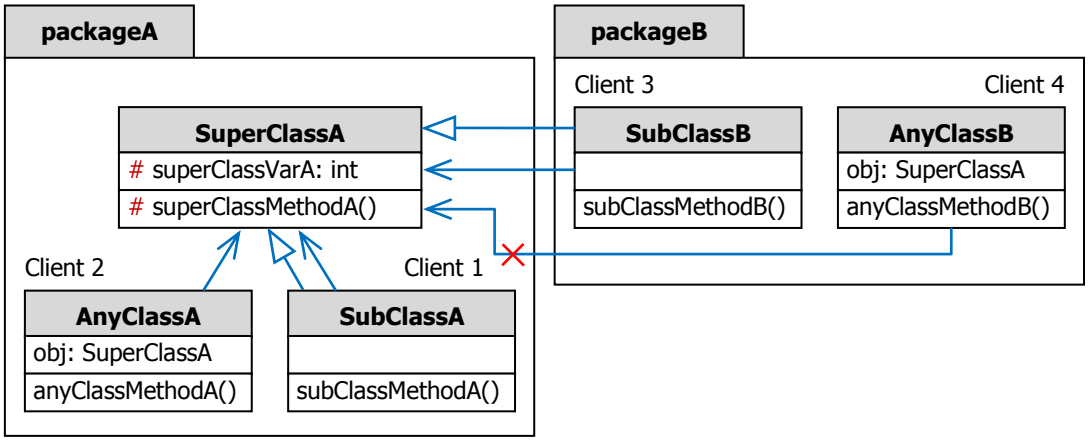
Truy cập thành viên `private` phải thông qua contract của lớp



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*.



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.



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`.

4. 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. Signature của phương thức bao gồm tên phương thức, kiểu và số lượng các tham số bao gồm 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.

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

  public String getDetails() {
    return String.format("Name: %s%nSalary: %f%n", name, salary);
  }
}

public class Manager extends Employee {
  private String department;

  @Override public String getDetails() {
    return String.format("%s%nManager of: %s%n", super.getDetails(), department);
  }
}
```

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 thể có cấp độ *truy cập thu hẹp* (narrow) so với phương thức gốc mà phải mở rộng (widen) hơn. Ví dụ, viết lại một phương thức `public` và đánh dấu chúng thành `private` là vi phạm.
Phương thức viết lại có thể: ném 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.
Annotation `@Override` 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 `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 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 với 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, 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
e.department = "Sales";
// do biến được khai báo như một Employee
```

1. Triệu gọi phương thức ảo

Giả sử bạn có:

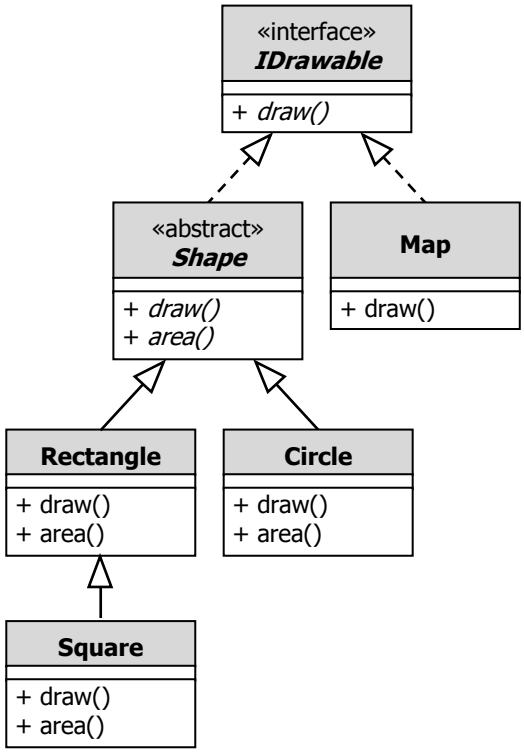
```
Employee e = new Employee();
Manager m = new Manager();
```

Nếu bạn 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`. 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 tham chiếu 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 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 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 dùng các collection không đồng nhất chứa các đối tượng được dẫn xuất từ một lớp cha chung gần hơn:

```
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 đối tượng thật 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 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*. 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 cung cấp phương thức có tham số thay đổi (variable-length arguments) hoặc *varargs*, cho phép chỉ viết một phương thức:

```
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 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) {
    // gọi ngầm định constructor mặc định tại đây
    this.name = name;
    this.salary = salary;
    this.birthDate = 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 bốn 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 và thứ ba gọi constructor thứ nhất, constructor thứ tư gọi constructor thứ hai.

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.

Contructor 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 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`. Như 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(this);
  }
  // private ngầm định là final
  private String genCode(String name) {
    return String.format("STU%05d", System.currentTimeMillis() % 100000);
  }
  // overiddable method
  public void setName(String name) {
    return (name == null || name.isEmpty()) ? "noname" : name;
  }

  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 nó không thừa kế các constructor. 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 chèn 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 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 của 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 Object {
  public Object() { }
}

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 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 của 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 của constructor: `name = "Joe Smith"`
    1.3.2 Gọi `this(name, null)`, tức `Employee(String, Date)`
      1.3.2.1 Kết nối các tham số của 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 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` được cung cấp để viết lại trong các lớp dẫn xuất từ nó. Nó tạo thành contract chung cho một đối tượng.

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)` 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 *tham chiếu đến cùng 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` 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: bắt 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 `obj` 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.
Đối tượng của một lớp viết lại phương thức `equals()` có thể được dùng như các phần tử trong một collection. Nếu chúng cũng viết lại phương thức `hashCode()`, 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<E>`, 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() {
    int hashValue = 11;
    hashValue = 31 * hashValue + day;
    hashValue = 31 * hashValue + month;
    hashValue = 31 * hashValue + year;
    return hashValue;
  }

  @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);
  }
}

// test
MyDate date1 = new MyDate(20, 11, 2014);
MyDate date2 = new MyDate(20, 11, 2014);
date1 == date2;            // false
date1.equals(date2);       // true
```

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()`.
Lớp `Object` định nghĩa một phương thức `toString()` mặc định, trả về tên lớp và địa chỉ tham chiếu của nó, không thường dùng. 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 dạng chuỗi trị mà chúng thể hiện.

Lớp bao

1. Lớp bao (wrapper class)
Java không xem các kiểu cơ bản (primitive) là đối tượng. Java cung cấp các lớp bao để thao tác các kiểu cơ bản như các đối tượng, 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 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.

Kiểu dữ liệu cơ bản	Lớp bao
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

Bạn có thể khởi tạo một đối tượng lớp bao, bằng cách gọi constructor tương ứng với trị 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
```

Lớp bao thường dùng khi cần lấy trị có kiểu cơ bản từ một chuỗi.

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

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 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.

```
// autoboxing
List list = new ArrayList();
list.add(100);       // list.add(new Integer(100));
Byte b = 100;        // chuyển int thành byte, rồi byte thành Byte
// autounboxing
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 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`, tăng từ 1; 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 một phương thức `static`, nhưng bạn có thể ẩn (hiding) chúng.

Phương thức viết lại (overriden) 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 c = new Count3();
Count2 c1 = c;
```

```
c.getTotalCount();  // triệu gọi getTotalCount() của Count3
c1.getTotalCount(); // triệu gọi getTotalCount() của Count2
```

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

3. Khối khởi tạo `static`

Một lớp có thể chứa một 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 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 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 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`. Dĩ nhiên, lớp `abstract` và `interface` 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*. 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");    // không cho phép
  price /= 2.0;                 // 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. Thử thay đổi trị của biến `final` sẽ gây lỗi biên dịch.

```
public class Bank {
  private static final double DEFAULT_INTERREST_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`.

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 constanst hoặc manifest constant), ví dụ `Integer.MAX_VALUE`.

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

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*.

```
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
}
```

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 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 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"),
  HEARTS   ("Hearts"),
  CLUBS    ("Clubs"),
  DIAMONDS ("Diamonds");
  // phần tùy chọn: constructor, field và method
  private final String name;

  private Suit(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }
}
```

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 construstor 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:

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

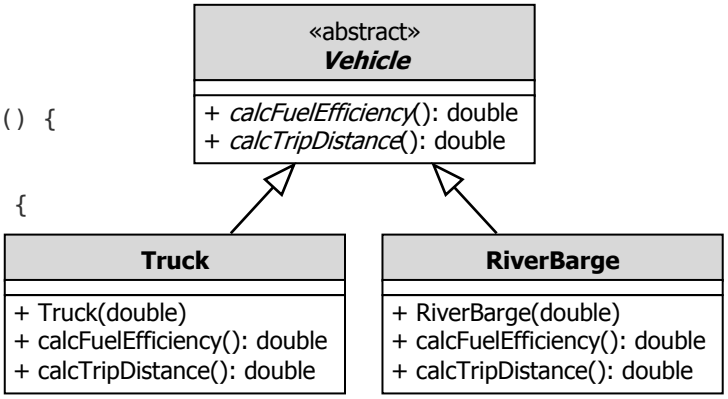
Do constructor không được thừa kế nên không khai báo `abstract` được. Constructor của lớp trừu tượng thường được đánh dấu là `protected` vì nó chỉ có ý nghĩa khi được triệu gọi từ lớp con cụ thể.

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

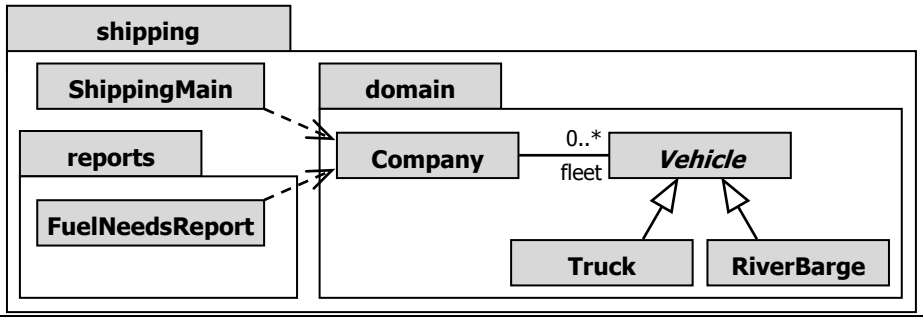
Do lớp trừu tượng chỉ mang tính khái niệm, trình biên dịch ngăn cản tạo thể hiện của lớp trừu tượng. Lớp trừu tượng có thể thừa kế từ một lớp cụ thể, nhưng không nên thực hiện điều đó.

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 là một giao ước (contract) các dịch vụ mà lớp sẽ cung cấp. Interface chỉ khai báo các phương thức mà không cung cấp bất kỳ cài đặt nào. Một lớp cụ thể cài đặt (`implements`) interface bằng cách định nghĩa tất cả các phương thức được khai báo trong interface. Lớp cài đặt interface nhưng không cung cấp cài đặt đầy đủ cho các phương thức của interface đó phải được khai báo là lớp `abstract`.

Tên interface là tính từ 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.

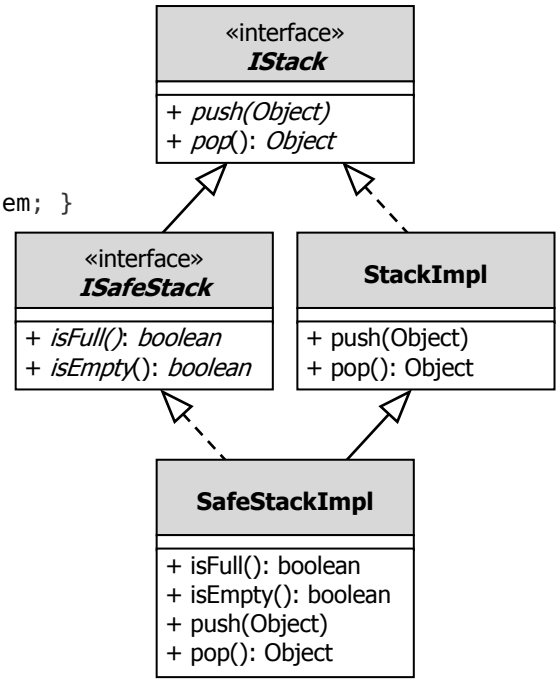
Cú pháp khai báo của interface:

```
<bổ từ truy cập> 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`, nhưng các bổ từ này có thể được bỏ qua. Trong Java 8, interface có thể chứa phương thức `static`.

```
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); }
  public boolean isEmpty() { return tos < 0; }
  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());       // Dollards
    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 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 { ... }
```

- 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.
- 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 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.

2. Interface đánh dấu
Interface với thân rỗng, không khai báo phương thức nào, được dùng như một interface đánh dấu (tagging/marker interface, hoặc ability interface) giúp trình biên dịch nhận biết và xử lý đặc biệt 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.
Java API cung cấp một số ví dụ về các interface đánh dấu như `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. Functional interface
Trong Java 8, interface chỉ có một phương thức trừu tượng (SAM – Single Abstract Method) gọi là functional interface. Bạn có thể đặt annotation `@FunctionalInterface` ngay trước khai báo của interface. Thể hiện của functional interface sẽ được tạo với biểu thức lambda.

```
@FunctionalInterface
public interface InterfaceName {
  // chỉ có một phương thức trừu tượng
  public void doAbstractTask();
  // cho phép nhiều phương thức default
  default public void performTask1() {
    System.out.println("Message from task 1.");
  }
  default public void performTask2(){
    System.out.println("Message from task 2.");
  }
}
```

4. Phương thức default
Java 8 cho phép interface có phương thức *đã được cài đặt mặc định*, gọi là phương thức `default`. Đôi khi ta cũng gọi là phương thức *optional*, do không bắt buộc lớp cài đặt interface phải cài đặt lại phương thức này. Nếu lớp cài đặt interface cài đặt lại cho phương thức `default`, code cài đặt trong lớp sẽ được dùng.
Ví dụ, interface `java.util.List` của JDK 8:

```
interface List<E> {
  // ...
  public default void sort(Comparator<? super E> c) {
    Collections.<E>sort(this, c);
  }
}
```

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

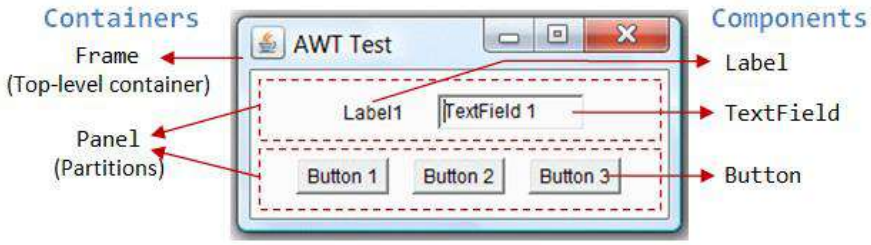
Lập trình giao diện đồ họa

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 (GWT).

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.

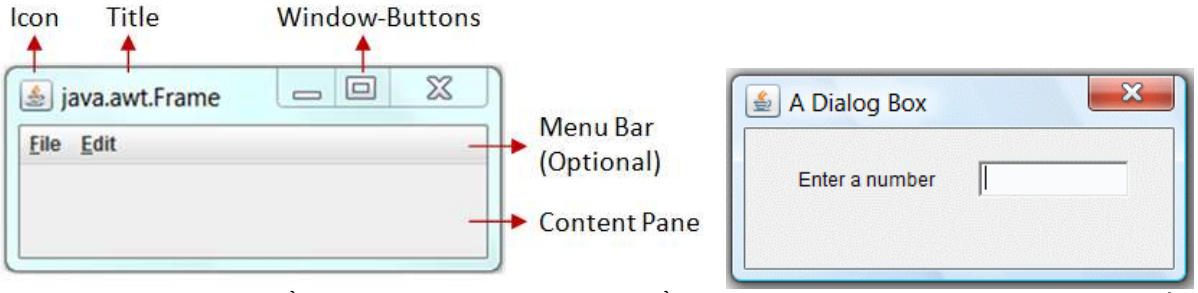


Trong hình trên, có ba container: một `Frame` và hai `Panel`. `Frame` là container cấp cao nhất (top-level), có: thanh tiêu đề (chứa icon, tiêu đề, và ba nút chuẩn Minimize/Maximize/Close), menu-bar tùy chọn và vùng hiển thị nội dung. `Panel` là một vùng hình chữ nhật được sử dụng để nhóm các component 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`:



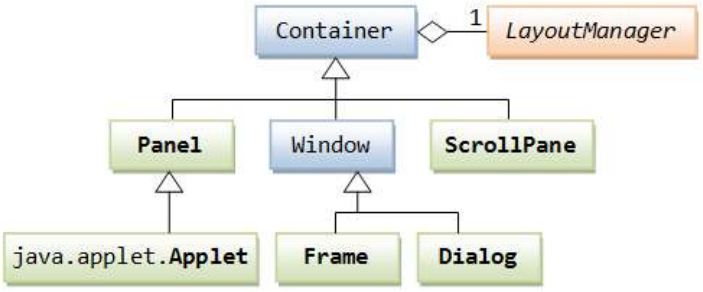
- Một `Frame` cung cấp một "cửa sổ chính" cho các ứng dụng GUI. Để viết một chương trình giao diện, ta thường bắt đầu với một lớp con dẫn xuất từ lớp `java.awt.Frame`.
- Một `Dialog` là một "cửa sổ pop-up" dùng tương tác với người sử dụng. Một `Dialog` có thanh tiêu đề (chứa icon, tiêu đề và nút Close 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 tự động cuộn ngang và/hoặc cuộn dọc cho component con.

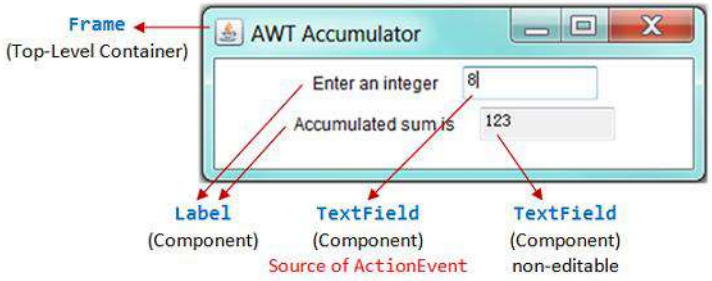
Hệ thống phân cấp các lớp container AWT như sau. Một container có một layout.



2. Các lớp component

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

3. Ví dụ minh họa



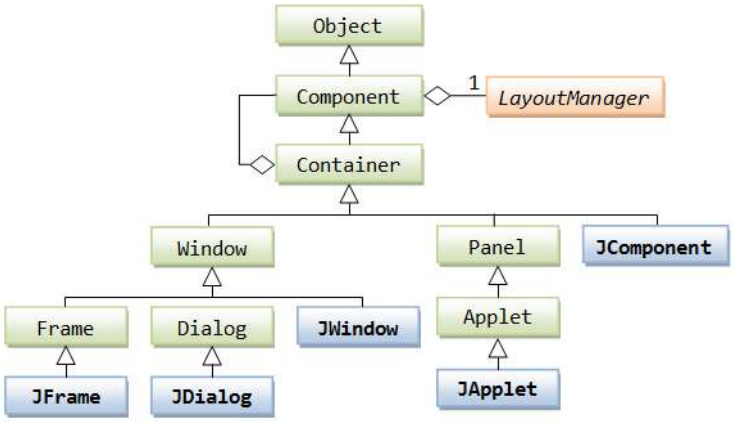

```
import java.awt.*;
import java.awt.event.*;
public class AWTAccumulator extends Frame implements ActionListener {
  private TextField txtInput;
  private TextField txtOutput;
  private int numberIn;
  private int sum = 0;

  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: "));
```

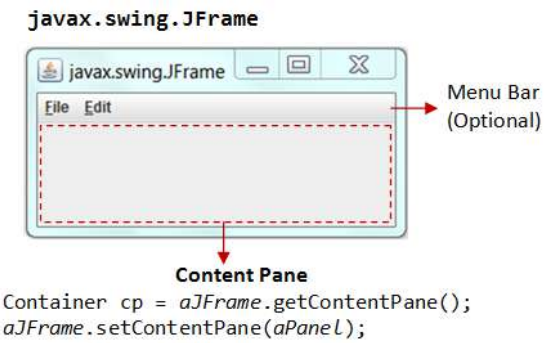
```
    // khởi gán từng component và thêm chúng vào container
    txtInput = new TextField(10);
    txtInput.addActionListener(this);
    add(txtInput);
    add(new Label("The Accumulated Sum is: "));
    txtOutput = new TextField(10);
    txtOutput.setEditable(false);
    add(txtOutput);
  }

  // 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) {
    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 cửa 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 (layout manager) của AWT (như `FlowLayout` và `BorderLayout` trong gói `java.awt`). 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:





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.

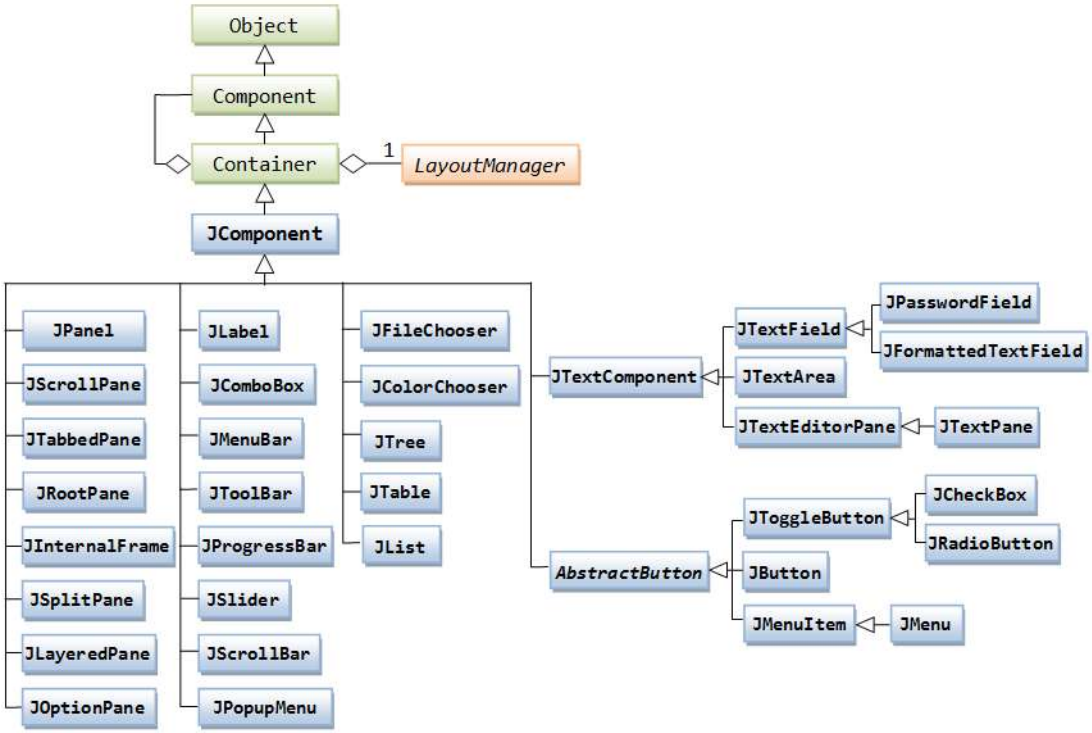
```
Container pane = getContentPane();
```

```
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 (gói `java.awt.event`). Swing giới thiệu một vài lớp xử lý sự kiện mới (gói `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.*;
import java.awt.event.*;
import javax.swing.*;
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(DISPOSE_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 TextField(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 TextField(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:

```
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 manage 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 một 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). 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) có trong listener sẽ xử lý sự kiện.

- 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 trong đối tượng 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

Trong mô hình ủy nhiệm sự kiện, JVM tạo ra các sự kiện 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 *listener*, sẽ nhận event tương ứng được chuyển tiếp từ đơn thể UI. Chúng chứa các bộ xử lý sự kiện nhận và xử lý sự kiện chuyển đến, nên cũng gói là lớp xử lý sự kiện. Theo cách này, đối tượng xử lý sự kiện tách biệt với đơn thể UI. Listener là các lớp cài đặt interface EventListener.

Các đối tượng event chỉ kích hoạt các listener đã đăng ký với đơn thể UI. Mỗi event có một interface listener tương ứng, interface này 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.

Lớp listener chứa các phương thức xử lý sự kiện:

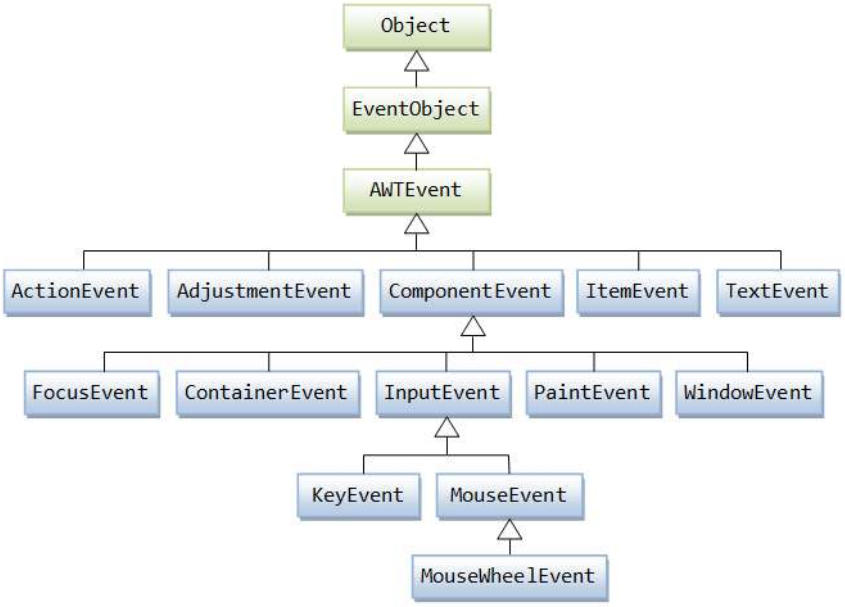
```
import java.awt.event.*;
import java.awt.Component;
import javax.swing.JOptionPane;
// lớp xử lý sự kiện
public class ButtonHandler implements ActionListener {
  // phương thức xử lý sự kiện, nhận tham số là đối tượng event
  @Override public void actionPerformed(ActionEvent e) {
    // tham số thứ nhất dùng xác định đơn thể cha của JOptionPane
    JOptionPane.showMessageDialog(((Component)e.getSource()).getParent(), e.getActionCommand());
  }
}
```

Đăng ký listener trong lớp giao diện người dùng:

```
JButton btnHello = new JButton();
btnHello.setText("Hello");
btnHello.setActionCommand("Button Hello is pressed!");
// 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`	`itemStateChange(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	`AdjustmentListener`	`adjustmentValueChanged(AdjustmentEvent)`

	Component	`ComponentListener`	`componentMoved(ComponenEvent)`
			`componentHidden(ComponenEvent)`
			`componentResized(ComponenEvent)`
			`componentShown(ComponenEvent)`

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`	`textValueChaged(TextEvent)`

Listener

1. GUI cũng là listener

Trong ví dụ trên, ta viết một *lớp listener riêng* để lắng nghe sự kiện cho giao diện người dùng. Tuy nhiên, xử lý sự kiện thường là cập nhật GUI, nên đôi khi lớp giao diện người dùng được cài đặt *kiêm cả nhiệm vụ listener*, bằng cách:

- Cài đặt các interface listener cho lớp giao diện người dùng. 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.

```
public class TwoListener extends JFrame implements MouseMotionListener, MouseListener {
  private JTextField txtInfo;

  public TwoListener() {
    // ...
    txtInfo = new JTextField;
    this.addMouseMotionListener(this);
    this.addMouseListener(this);
  }
```

```
  // các phương thức xử lý sự kiện của MouseMotionListener
  @Override public void mouseDragged(MouseEvent e) {
    txtInfo.setText("Mouse dragging ("+ e.getX() + ", " + e.getY() + ")");
  }
  // các phương thức xử lý sự kiện không dùng của MouseMotionListener
  @Override public void mouseMoved(MouseEvent e) { }

  // các phương thức xử lý sự kiện của MouseListener
  @Override public void mouseEntered(MouseEvent e) {
    txtInfo.setText("The mouse entered");
  }
  @Override public void mouseExited(MouseEvent e) {
    txtInfo.setText("The mouse has left");
  }
  // các phương thức xử lý sự kiện không dùng của MouseListener
  @Override public void mouseClicked(MouseEvent e) { }
  @Override public void mousePressed(MouseEvent e) { }
  @Override public void mouseReleased(MouseEvent e) { }
}
```

Ví dụ trên cũng cho thấy Java cho phép gán nhiều listener đến một đơn thể UI.

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 đã được cài đặt rỗng sẵn. Bạn có thể thừa kế các lớp tiếp hợp này và chỉ cần viết lại (override) đúng các phương thức mà bạn cần.

```
import java.awt.*;
import java.awt.event.*;

public class MouseClickHandler extends MouseAdapter {
  // chỉ cần cài đặt xử lý sự kiện click mouse trong 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 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 bao ngoài.

```
public class InnerFrame extends JFrame {
  private JTextField txtInfo;

  public InnerFrame() {
    // ...
    txtInfo = new JTextField;
    this.addMouseMotionListener(new MyMouseMotionListener());
    this.addMouseListener(new MouseClickHandler());   // lớp xử lý sự kiện nằm riêng bên ngoài
  }

  // lớp nội thừa kế lớp adapter MouseMotionAdapter
  class MyMouseMotionListener extends MouseMotionAdapter {
    @Override public void mouseDragged(MouseEvent e) {
      txtInfo.setText("Mouse dragging ("+ e.getX() + ", " + e.getY() + ")");
    }
  }
}
```

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 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.

```
public class AnonymousFrame extends JFrame {
  private JTextField txtInfo;
```

```
  public AnonymousFrame() {
    // ...
    txtInfo = new JTextField;
    this.addMouseMotionListener(new MouseMotionAdapter() {
      @Override public void mouseDragged(MouseEvent e) {
        txtInfo.setText("Mouse dragging ("+ e.getX() + ", " + e.getY() + ")");
      }
    });
    this.addMouseListener(new MouseClickHandler());   // lớp xử lý sự kiện nằm riêng bên ngoài
  }
}
```

Java cho phép khai báo lớp *bên trong một phương thức*, gọi là lớp nội cục bộ (local inner class). Chú ý là lớp nội vô danh thường là một lớp nội cục bộ. Xem ví dụ sau, trong phương thức `main()`:

```
public class Main {
  public static void main(String[] args) {
    // lớp nội vô danh
    new Thread(new Runnable() {
      @Override public void run() {
        System.out.println("Anonymous Inner Class: Running Thread");
      }
    }).start();

    // lớp nội cục bộ, khai báo trong phương thức main()
    class MyThread implements Runnable {
      @Override public void run() {
        System.out.println("Local Inner Class: Running Thread");
      }
    }
    new Thread(new MyThread()).start();
  }
}
```

Bạn có thể định nghĩa một interface nội, nhưng không thể định nghĩa một interface cục bộ.
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:

```
// 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. Gán listener cho lớp khác

Tình huống thường gặp trong lập trình GUI là chuyển thông tin giữa hai cửa sổ. 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, click nút `btnOk` trên hộp thoại. Hộp thoại `CreateDialog` đóng lại và thông tin `Student` mới tạo sẽ được hiển thị trong frame `StudentGUI`.

Có hai cách tiếp cận để giải quyết vấn đề này:
- `CreateDialog` phải giữ một tham chiếu `parent` chỉ đến form cha của nó.
  + Khi khởi tạo `CreateDialog`, `StudentGUI` truyền tham chiếu chỉ đến nó cho constructor của `CreateDialog`.

```
public class StudentGUI extends JFrame {
  public StudentGUI() {
    initComponents();
    setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    dialog.setLocationRelativeTo(null);
  }

  private void initComponents() {
    // tạo GUI
    // gán listener cho nút btnCreate để mở hộp thoại CreateDialog
    btnCreate.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        java.awt.EventQueue.invokeLater(new Runnable() {
```

```
          @Override public void run() {
           // 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);
          } // end run()
        });
      } // end actionPerformed()
    });
  }

  public void updateGUI(Student student) {
    // cập nhật UI của StudentGUI với student nhận được
  }

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

+ Sử dụng tham chiếu `parent` này, `CreateDialog` có thể gọi phương thức của `StudentGUI` để cập nhật UI.

```
public class CreateDialog extends JDialog {
  StudentGUI parent;

  public CreateDialog(Frame parent, boolean modal) {
    super(parent, modal);
    parent = (StudentGUI) parent;
    initComponents();
  }

  private void initComponent() {
    // ...
    // gán listener cho nút btnOk của CreateDialog
    btnOk.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        // tạo đối tượng student từ các input của form nhập rồi truyền
        parent.updateGUI(student);
        dispose();
      } // end actionPerformed
    });
  }
}
```

- Nếu phải xử lý nhiều nút phức tạp trong `CreateDialog`, bạn có thể dùng cách gán listener cho nó. Đối tượng `listener` trung gian sẽ được tạo trong `CreateDialog`.

+ `StudentGUI` sẽ cài đặt các phương thức cho `listener` trung gian và gán `listener` này cho `CreateDialog` khi khởi tạo nó, bằng cách truyền như tham số cho constructor hoặc bằng cách gọi setter.

```
public class StudentGUI extends JFrame implements CreateDialog.CreateDialogListener {
  public StudentGUI() {
    initComponents();
    setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    dialog.setLocationRelativeTo(null);
  }

  // cài đặt cho listener trung gian
  @Override public void onClickOK(Student student) {
    // cập nhật UI của StudentGUI với student nhận được
  }

  private void initComponents() {
    // tạo GUI
    // gán listener cho nút btnCreate để mở hộp thoại CreateDialog
    btnCreate.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        java.awt.EventQueue.invokeLater(new Runnable() {
          @Override public void run() {
```

```
            CreateDialog dialog = new CreateDialog();
            // gán listener trung gian cho CreateDialog
            dialog.setListener(StudentGUI.this);
            dialog.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
            dialog.setLocationRelativeTo(rootPane);
            dialog.setVisible(true);
          } // end run()
        });
      } // end actionPerformed()
    });
  }

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

+ Listener của `CreateDialog`, khi xử lý sự kiện các nút, sẽ gọi phương thức phù hợp trong `listener` trung gian này.

```
public class CreateDialog extends JDialog {
  private CreateDialogListener listener;
  // callback interface
  interface CreateDialogListener {
    void onClickOk(Student student);
  };

  public CreateDialog() {
    initComponents();
    setModal(true);
  }

  public void setListener(CreateDialogListener listener) {
    this.listener = listener;
  }

  private void initComponent() {
    // ...
    // gán listener cho nút btnOk của CreateDialog
    btnOk.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        // tạo đối tượng student từ các input của form nhập
        listener.onClickOk(student);
        dispose();
      } // end actionPerformed
    });
  }
}
```

Collection

Các kiểu collection

Một collection là một tập hợp các đối tượng thể hiện một nhóm đối tượng. Các đối tượng trong collection được gọi là các element. Thông thường, collection có thể chứa nhiều kiểu đối tượng, chúng thừa kế từ một kiểu cha chung.

- `Collection`. Nhóm các đối tượng gọi là element. Cho phép chỉ định thứ tự và cho phép trùng lặp.
- `Set`. Collection không thứ tự, không cho phép trùng lặp. Thử thêm các element trùng lặp đến `Set` sẽ không có kết quả.

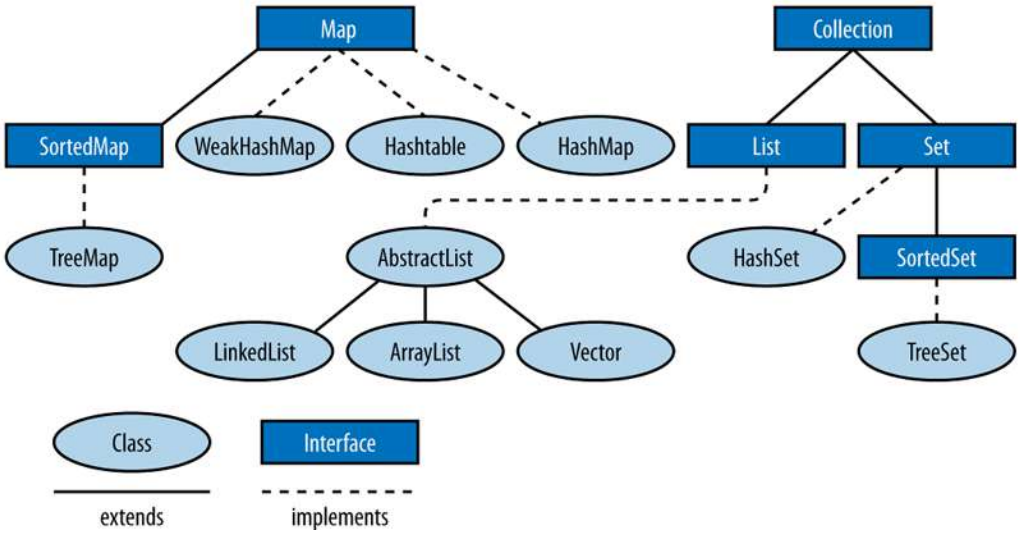
```
import java.util.*;
public class SetExample {
  public static void main(String[] args) {
    Set set = new HashSet();
    set.add("two");
    set.add("one");
    set.add("two");                // trùng lặp, không thêm vào Set
    System.out.println(set);       // [one, two]
  }
}
```

- `List`. Collection có thứ tự, cho phép trùng lặp.

```
import java.util.*;
public class ListExample {
  public static void main(String[] args) {
    List list = new ArrayList();
    list.add("two");
    list.add("one");
    list.add("two");
    System.out.println(list);     // [two, one, two]
  }
}
```

Collection duy trì các tham chiếu có kiểu `Object` chỉ đến các element. Nghĩa là bất kỳ đối tượng nào cũng có thể được lưu trữ trong collection, trước khi dùng các element buộc phải ép kiểu đúng element sau khi lấy nó từ collection.

Hình sau cho thấy các interface và các lớp cơ bản của Collections API. Lớp `HashSet` cung cấp một cài đặt cho interface `Set`. Lớp `ArrayList` và `LinkedList` cung cấp một cài đặt cho interface `List`.



Generics

Các lớp collection dùng kiểu `Object` để cho phép nhận và trả về các kiểu khác nhau. Bạn phải ép kiểu xuống một cách tường minh để lấy đúng đối tượng bạn cần, bạn cũng phải dùng lớp bao để đưa kiểu cơ bản vào collection. Điều này không an toàn kiểu.

```
ArrayList list = new ArrayList();
list.add(0, new Integer(42));
int total = ((Integer) list.get(0)).intValue();
```

Từ Java 5, giải pháp cho vấn đề này là generics. Generics cung cấp thông tin cho trình biên dịch về kiểu collection được dùng. Kiểu sẽ được xác định tự động trong thời gian chạy, tránh phải ép kiểu tường minh như với collection. Để chỉ định kiểu cho element của một collection đóng kiểu này trong trong cặp <>.

Dùng generics cho code trên, kết hợp với khả năng autoboxing, code trở nên đơn giản hơn:

```
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<String>` chúng ta gán kiểu `String` cụ thể đến `List` như một tham số.
Generic type có thể được dùng trong signature và thân của các phương thức như kiểu thật:

```
interface List<T> extends Collection<T> {
  boolean add(T t);
  T get(int index);
  // ...
}
```

Từ Java 7, khi tạo thực thể của một generic type, vế phải của phát biểu gán cho phép thiếu trị của tham số kiểu, gọi là cú pháp *diamond*.

```
List<Employee> salesList = new ArrayList<>();
```

Để chuyển collection cũ (gọi là 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;
```

Nói cách khác, `List` và `List<String>` trong ví dụ trên là tương thích, gọi là type erasure.
Các ràng buộc (bound) và 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 là subtype của kiểu P
`<T extends P & S>`	Kiểu ràng buộc lên; kiểu chỉ định T là subtype của kiểu P và cài đặt kiểu S
`<T super P>`	Kiểu ràng buộc xuống (lower bounded); kiểu chỉ định T là 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 là subtype của kiểu P
`<? extends P & S>`	Ký tự đại diện có ràng buộc; kiểu nào đó là subtype của kiểu P và cài đặt kiểu S
`<? super P>`	Ký tự đại diện ràng buộc xuống; kiểu nào đó là supertype của kiểu P

Khi dùng ký tự đại diện, bạn theo nguyên tắc get/put sau:
- Dùng `extends` khi chỉ lấy trị ra (get).

```
public 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.

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 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());
}
```

Phương thức `remove()` của `Iterator` cho phép loại bỏ element hiện hành do con trỏ của iterator chỉ đến. Nếu việc loại bỏ không được hỗ trợ bởi loại collection đang duyệt, `UnsupportedOperationException` sẽ được ném ra.
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.

Phương thức `set()` cho phép thay đổi element hiện hành do con trỏ của 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 `add()`, bạn sẽ nhận được element mới thêm vào.

2. Vòng lặp for tăng cường

Tuy vậy, iterator khó hiểu và phức tạp, phải gọi nhiều phương thức với cấu trúc thiếu trực quan.

```
public void deleteAll(Collection<NameList> c) {
  for (Iterator<NameList> i = c.iterator(); i = c.iterator(); i.hasNext()) {
    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, colllections API cung cấp tác vụ duyệt collection theo cách tiếp cận lập trình hàm, phương thức `forEach()`:

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

// (1) for tăng cường
for (String s : list) {
  System.out.println(s);
}

// (2) tác vụ hàm
list.stream().forEach((s) -> {
  System.out.println(s);
});

// có dùng tham chiếu phương thức
list.stream().forEach(System.out::println);
```

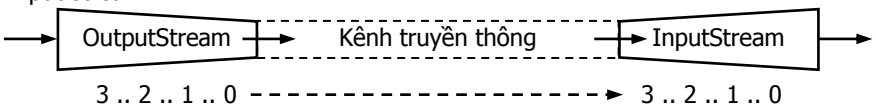
Stream

Khái niệm

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, … Stream có thể đọc từ kênh truyền thông hoặc ghi đến nó.

Java API cung cấp một số lớp stream cơ bản, có thể chia thành hai nhóm: input stream (stream nhập) và output stream (stream xuất).

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.



Output stream và Input stream

Tính chất của 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 xuất liên tục (sequential): chuỗi dữ liệu trong stream được truy xuất liên tục từng byte 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 stream thực hiện cả đọc lẫn ghi.
- Blocking: khi stream đang đọc/ghi, nó tạm thời bị khóa (blocking) cho đến khi xong tác vụ, không cho truy xuất từ tiến trình khác.

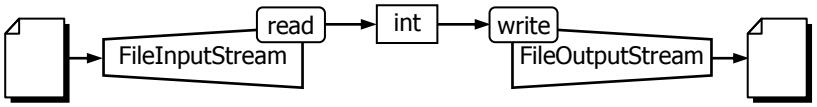
Stream API được thiết kế theo design pattern Decorator. Điều này cho phép "lồng stream": gọi constructor của stream có tính năng tăng cường với đối số là đối tượng stream có tính năng cơ bản, nhằm tăng tính năng của stream cơ bản. Thường các stream được lồng nhau cho đến khi nhận được stream có phương 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.*;

public class CopyBytes {
  public static void main(String[] args) throws IOException {
    FileInputStream in = null;
    FileOutputStream out = null;
    try {
      in = new FileInputStream("source.txt");
      out = new FileOutputStream("target.txt");
      int c;
      while ((c = in.read()) != -1)
        out.write(c);
    } finally {
      if (in != null) in.close();
      if (out != null) out.close();
    }
  }
}
```

Đôi khi ta cần đẩy dữ liệu ra khỏi stream, không cần đợi stream đầy. Điều này gọi là "súc" (flushing) stream. Một số stream hỗ trợ tự động "súc" (autoflush) khi có sự kiện nào đó, thiết lập bằng đối số trong constructor. Có thể "súc" stream bằng phương thức `flush()`, thường chỉ hiệu quả với stream có vùng đệm (buffer).

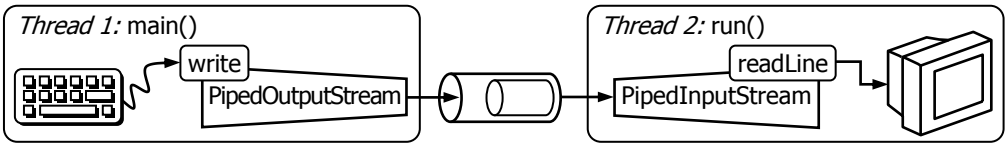
Khi sử dụng xong stream, cần đóng nó bằng phương thức `close()`, thường dùng trong khối `finally`. Đóng stream cũng là đóng kênh truyền thông liên kết với nó.

Hai loại byte stream thường được dùng và một số lớp của chúng:

- Các 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.

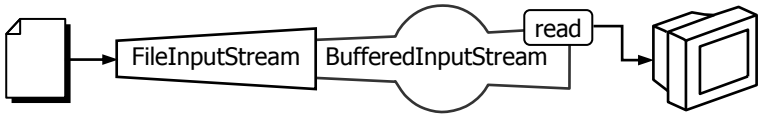
• `ByteArrayOutputStream` và `ByteArrayInputStream` dùng đọc/ghi dữ liệu từ một mảng byte. Thường dùng để chuyển dữ liệu vào một mảng, xử lý nó trước khi chuyển đi. Vì vậy từ lớp này có thể dùng để sinh ra các lớp stream phức tạp, như stream xử lý thông điệp, stream mã hóa dữ liệu, …

• `PipeOutputStream` và `PipeInputStream` thường tạo thành từng cặp tương ứng. Dùng để truyền thông tin giữa các thread trong một ứng dụng đơn bằng stream.

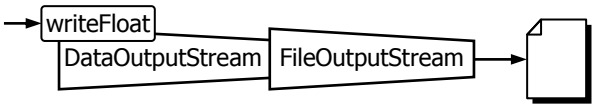


- Các stream lọc: cho phép thêm các chức năng cấp cao cho các stream của Java: `FilterInputStream` là stream lồng vào một `InputStream` có sẵn, cung cấp các tính năng mở rộng cho `InputStream`; tương tự, một `FilterOutputStream` lồng vào một `OutputStream` có sẵn để tăng cường các chức năng của nó.

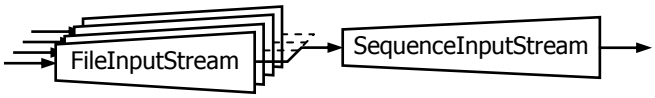
• `BufferedOutputStream` và `BufferedInputStream` cung cấp xuất/nhập có vùng đệm cho stream, tăng thêm hiệu quả cho xử lý xuất nhập. Constructor cần stream mà nó lồng vào và kích thước buffer mong muốn.



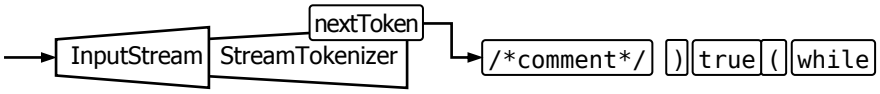
• `DataOutputStream` và `DataInputStream` tăng cường khả năng cho các `OutputStream` và `InputStream` bằng cách cho phép xử lý nhiều kiểu dữ liệu hơn như chuỗi, integer, số thực, …



• `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.

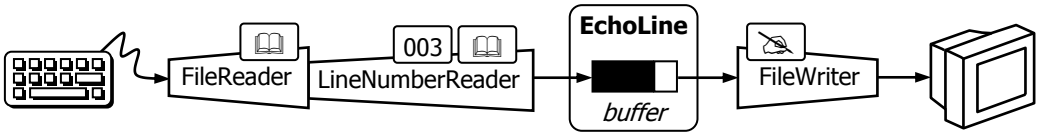


2. Character Stream

Java lưu trị 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 chúng quen thuộc 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
`OutputStream`	`Writer`	`InputStream`	`Reader`
`FileOutputStream`	`FileWriter`	`FileInputStream`	`FileReader`
`ByteArrayOutputStream`	`CharArrayWriter`	`ByteArrayInputStream`	`CharArrayReader`
–	`StringWriter`	*`StringBufferInputStream`*	`StringReader`
`PipedOutputStream`	`PipedWriter`	`PipedInputStream`	`PipedReader`
`FilterOutputStream`	`FilterWriter`	`FilterInputStream`	`FilterReader`
`BufferedOutputStream`	`BufferedWriter`	`BufferedInputStream`	`BufferedReader`
`PrintStream`	`PrintWriter`	`PushbackInputStream`	`PushbackReader`
		`LineNumberInputStream`	`LineNumberReader`
		`SequenceInputStream`	–
`DataOutputStream`	–	`DataInputStream`	–
`ObjectOutputStream`	–	`ObjectInputStream`	–
–	`OutputStreamWriter`	–	`InputStreamReader`

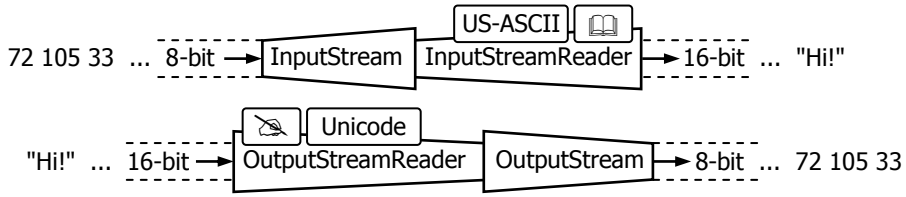
Tất cả các character stream thừa kế `Reader` và `Writer`, cũng giống như byte stream (thừa kế `InputStream` và `OutputStream`). Các tác vụ chính của các lớp có thể tham khảo từ `FileReader` và `FileWriter`. Xem ví dụ sau:



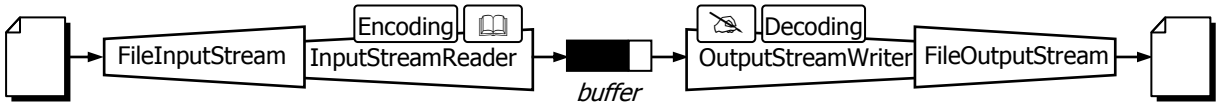

```
import java.io.*;

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.

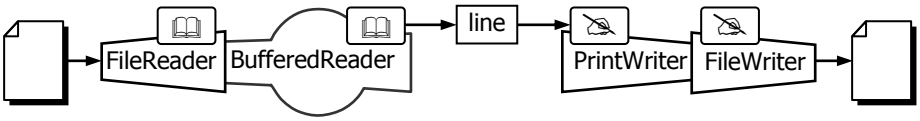


Khi cầu nối byte-character được thiết lập, cần chỉ định tập ký tự được dùng bởi byte stream. Các tập ký tự hỗ trợ: latin1, latin2, latin3, latin4, cirillic, arabic, greek, hebrew, latin5, ASCII, Unicode, UnicodeBig, UnicodeBigUnmarked, UnicodeLittle, UnicodeLittleUnmarked, UTF8. Tập ký tự mặc định tùy theo hệ thống.
Ví dụ chuyển đổi tập tin từ tập ký tự này sang tập ký tự khác:




```
import java.io.*;
// 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>");
    FileInputStream fileIn = new FileInputStream(args[1]);
    FileOutputStream fileOut = new FileOutputStream(args[3]);
    InputStreamReader in = new InputStreamReader(fileIn, args[0]);
    OutputStreamWriter out = new OutputStreamWriter(fileOut, args[2]);
    char[] buffer = new char[16];
    int numberRead;
    while ((numberRead = in.read(buffer)) > -1)
      out.write(buffer, 0, numberRead);
    out.close();
    in.close();
  }
}
```

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.*;

public class CopyLines {
  public static void main(String[] args) throws IOException {
    BufferredReader in = null;
    PrintWriter out = null;
    try {
      in = new BufferedReader(new FileReader("source.txt"));
      out = new PrintWriter(new FileWriter("target.txt"));
      String line;
      while ((line = in.readLine()) != null)
```

```
        out.println(line);
    } finally {
      if (int != null) in.close();
      if (out != null) out.close();
    }
  }
}
```

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 stream gắn với nó được gọi là các stream chuẩn:
- Standard Input truy xuất thông qua `System.in`, thông thường là bàn phím.
- Standard Output truy xuất thông qua `System.out`, thông thường là màn hình.
- Standard Error truy xuất thông qua `System.err`, thông thường là màn hình.

Các 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.io.*;
import java.util.*;

public class DayOfWeek {
  public static void main(String[] args) throws Exception {
    Scanner scanner = new Scanner(System.in);
    Calendar calendar = Calendar.getInstance();
    calendar.set(Calendar.DAY_OF_MONTH, scanner.nextInt());
    calendar.set(Calendar.MONTH, scanner.nextInt());
    calendar.set(Calendar.YEAR, scanner.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` cũng được cung cấp các stream xuất nhập (character stream), thường dùng cho việc kiểm tra password.

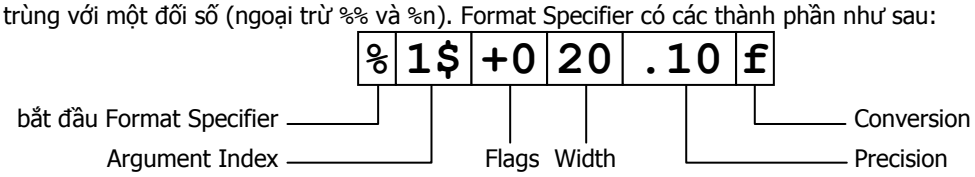
2. Quét (scanning) và định dạng (formatting)

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.

```
Scanner scanner = null;
try {
  scanner = new Scanner(new BufferedReader(new FileReader("data.txt")));
  scanner.useDelimiter(",\\s*");
  while (scanner.hasNext())
    System.out.println(scanner.next());
} finally {
  if (scanner != null) scanner.close();
}
```

• `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.

Một số 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.*;

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 {
    DataOutputStream out = null;
    try {
      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]);
      }
    } finally {
      out.close();
    }
    // đọc lại tập tin đã lưu
    DataInputStream in = null;
    double total = 0.0;
    try {
      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);
    } finally {
      in.close();
    }
  }
}
```

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 `Class1` 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 `readObject` và `writeObject` của `Class1` hỗ trợ thêm khi dùng object stream đọc hoặc ghi đối tượng lớp `Class1`.

```
import java.io.*;

class Class1 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();
  }
```

```
}

import java.io.*;
import java.util.Date;

public class Main {
  public static void main(String[] args) {
    try {
      // lưu đối tượng xuống tập tin
      ObjectOutput out = new ObjectOutputStream(new FileOutputStream("class1.ser"));
      Class1 c1 = new Class1();
      c1.date = new Date();
      out.writeObject(c1);
      out.flush();
      out.close();
      // đọc lại đối tượng (deserialization), ép kiểu đối tượng thành kiểu ban đầu
      ObjectInput in = new ObjectInputStream(new FileInputStream("class1.ser"));
      Class1 cc1 = (Class1)in.readObject();
      in.close();
      // kiểm tra đối tượng vừa phục hồi
      System.out.println("cc1.date " + cc1.date);
      System.out.println("cc1.restored " + cc1.restored);
    } catch ( Exception e ) {
      e.printStackTrace();
    }
  }
}
```

Thread

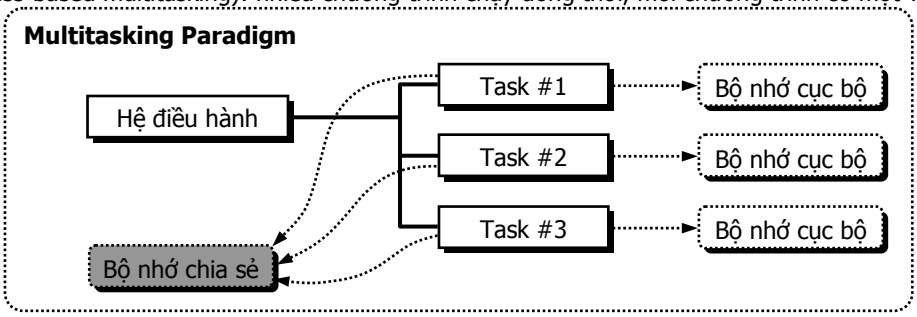
Khái niệm

1. Multitasking và Multithreading

Đa nhiệm (multitasking) là kỹ thuật cho phép nhiều công việc được thực hiện cùng một lúc trên máy tính. Nếu có nhiều CPU, các công việc có thể được thực hiện song song trên từng CPU. Trong trường hợp nhiều công việc cùng chia sẻ một CPU, từng phần của mỗi công việc sẽ được CPU thực hiện xen kẽ (điều phối CPU).

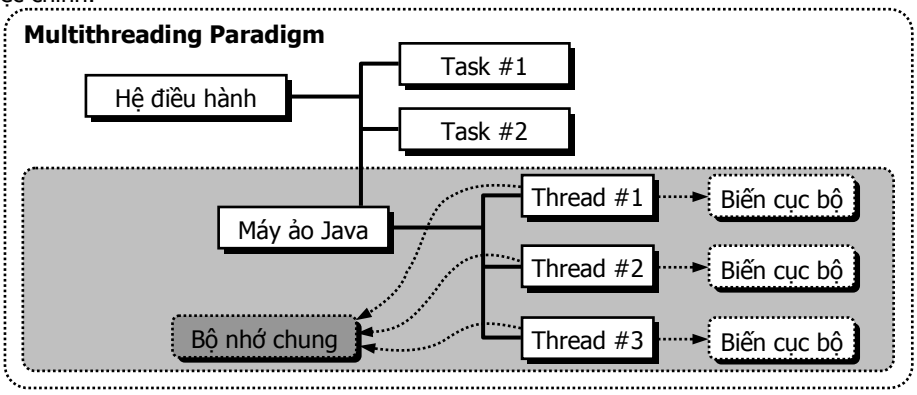
Có hai kỹ thuật đa nhiệm cơ bản:

- Đa tiến trình (Process-based multitasking): nhiều chương trình chạy đồng thời, mỗi chương trình có một vùng dữ liệu độc lập.



Process-based multitasking

- Đa tuyến trình (Thread-based multitasking): một chương trình có nhiều tuyến trình cùng chạy đồng thời. Các tuyến trình dùng chung vùng dữ liệu của chương trình. Đôi khi ta chỉ cần tạo ra một tuyến trình để thực hiện song song một công việc đơn giản cùng với một công việc chính.



Thread-based multitasking

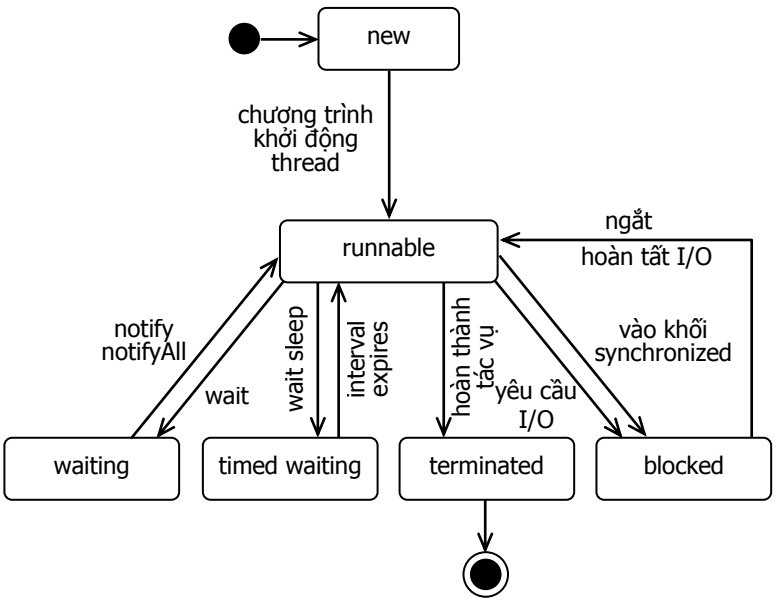
Tuyến trình (thread) là luồng thi hành độc lập của một tác vụ trong chương trình. Một chương trình có nhiều tuyến trình thực hiện cùng lúc gọi là chương trình đa tuyến trình (multithreaded).

2. Vòng đời của thread

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 theard 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 hệ điều hành để chờ đợi truy xuất 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`.

Java SE 7 giới thiệu thêm framework fork/join hỗ trợ cho lập trình song song.

1. Thừa kế lớp Thread

Trong lớp dẫn xuất lớp Thread, những tác vụ trong phương thức run() được xem như chạy trên một thread riêng. Phương thức này không thể gọi trực tiếp. Khi phương thức start() của đối tượng (thuộc lớp thừa kế lớp Thread) được gọi, nó sẽ tự động gọi run().

```
import java.util.Random;

public class PrintTask extends Thread {

  @Override public void run() {
    for (int i = 0; i < 10; ++i) {
      try {
        System.out.println(getName() + " being executed.");
        sleep(new Random().nextInt(3000));    // dừng ngẫu nhiên 0 - 3 giây
      } catch (InterruptedException exception) {
        System.out.println(exception.getMessage());
      }
    }
  }
}
```

Chạy hai thread:

```
public class RunnableTester {
  public static void main(String[] args) {
    new PrintTask().start();
    new PrintTask().start();
  }
}
```

Thread sẽ kết thúc khi thực thi hết các lệnh trong run() hoặc khi stop() được gọi.

2. Cài đặt 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 công việc cần thực hiện song hành.

```
import java.util.Random;

public class PrintTask implements Runnable {
  private String threadName;              // tên thread

  public PrintTask(String name) {
    threadName = name;
  }

  // phương thức run chứa code thực hiện bởi thread mới
  @Override public void run() {
    try {
      System.out.printf("%s going to sleep for %d milliseconds.\n", threadName, sleepTime);
      Thread.sleep(new Random().nextInt(5000));       // cho thread ngủ
    } catch (InterruptedException exception) {
      System.out.println(exception.getMessage());
    }
    System.out.printf("%s done sleeping\n", threadName);
  }
}
```

Mỗi đối tượng PrintTask do cài đặt interface Runnable sẽ thực hiện song hành các công việc mô tả trong phương thức run().

Cách đơn giản, có thể tạo một đối tượng thuộc lớp cài đặt Runnable và bọc trong đối tượng thuộc lớp Thread:

```
public class RunnableTester {
  public static void main(String[] args) {
    new Thread(new PrintTask("thread1")).start();
    new Thread(new PrintTask("thread2")).start();
  }
}
```

3. Framework Executor

Các thread cài đặt interface Runnable còn được thực thi bởi một đối tượng thuộc một lớp cài đặt interface java.util.concurrent.Executor.

Một đối tượng Executor thường tạo và quản lý một nhóm các thread có sẵn. Các thread cài đặt interface Runnable được truyền như tham số đến phương thức execute(). Đối tượng Executor gán mỗi Runable 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.

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class RunnableTester {
  public static void main(String[] args) {
    // tạo và đặt tên một số thread
    PrintTask task1 = new PrintTask("thread1");
    PrintTask task2 = new PrintTask("thread2");

    System.out.println("Starting Executor");

    // tạo ExecutorService từ Executors để quản lý các thread
    ExecutorService threadExecutor = Executors.newCachedThreadPool();

    // khởi động các thread và đưa chúng vào trạng thái chạy
    threadExecutor.execute(task1);              // khởi động task1
    threadExecutor.execute(task2);              // khởi động task2

    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\n");
  }
}
```

4. Cài đặt 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 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 run() của Runnable.

```
import java.util.concurrent.Callable
import java.util.Random;

public class PrintTask implements Callable {
  private String threadName;            // tên thread

  public PrintTask(String name) {
    threadName = name;
  }
  // phương thức call chứa code thực hiện bởi thread mới
  @Override public Object call() {
    try {
      System.out.printf("%s going to sleep for %d milliseconds.\n", threadName, sleepTime);
      Thread.sleep(new Random().nextInt(5000));      // cho thread ngủ
    } catch (InterruptedException exception) {
      exception.printStackTrace();
    }
    System.out.printf("%s done sleeping\n", threadName);
    return null;
  }
}
```

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.

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class RunnableTester {
  public static void main(String[] args) {
    PrintTask task1 = new PrintTask("thread1");
    PrintTask task2 = new PrintTask("thread2");

    System.out.println( "Starting Executor" );
    ExecutorService threadExecutor = Executors.newCachedThreadPool();

    // ví dụ này không xử lý đối tượng Future trả về
```

```
    threadExecutor.submit(task1);
    threadExecutor.submit(task2);
    threadExecutor.shutdown();
    System.out.println("Threads started, main ends\n");
  }
}
```

Một số vấn đề liên quan đến thread

1. 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.

Một thread mới sẽ thừa kế độ ưu tiên từ thread tạo ra nó. Các phương thức sau dùng để lấy và gán độ ưu tiên cho thread:

```
final int getPriority();
final void setPriority( int priority );
```

2. Một số phương thức của thread

`void sleep( long t );`	thread tạm dừng (ngủ) trong thời gian *t* miligiây.
`void yield();`	nhường điều khiển cho thread khác.
`void interrupt();`	ngắt hoạt động của thread.
`void join();`	yêu cầu chờ kết thúc

```
Thread t = new Thread(MyThread);
t.start();
// thực hiện một vài công việc khác
t.join();
// tiếp tục sau khi thread t hoàn thành
```

Một số phương thức: `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;
  }
}
```

3. Thread dịch vụ (demon thread)

Deamon thread thường là các thread hỗ trợ môi trường thực thi của các thread khác. Ví dụ: garbage collector của Java là một deamon thread. Chương trình kết thúc khi tất cả các thread không phải deamon thread kết thúc.

Các phương thức dùng cho deamon thread:

`void setDaemon( boolean isDaemon );`	đặt một thread trở thành deamon thread
`boolean isDaemon();`	kiểm tra một thread có phải là deamon thread không

4. 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"));
}
```

5. 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.

```
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 lên tài nguyên dùng chung. Vấn đề này đưa đến lỗi do truy xuất cùng lúc tài nguyên dùng chung, không kiểm soát được từ nhiều thread.

Vấn đề này có thể được giải quyết nếu cho mỗi thread một thời gian truy xuất độc quyền đến đoạn code chia sẻ tài nguyên. Lúc này, các thread khác muốn truy xuất tài nguyên chung đó sẽ bị giữ lại chờ đợi đến lượt mình. Điều này gọi là độc quyền truy xuất (*mutual exclusion*), cho phép lập trình viên thực hiện đồng bộ thread (*thread synchronization*), phối hợp nhiều thread song hành cùng truy xuất tài nguyên dùng chung.

Điển hình là bài toán Producer – Consumer: Producer đặt trị vào buffer nếu buffer còn trống, Consummer lấy trị từ buffer nếu buffer không rỗng. Tại một thời điểm chỉ có một đối tượng được truy xuất buffer. Nếu không sẽ gây đụng độ.

```
// interface Buffer chỉ định các phương thức được gọi bởi Producer và Consumer
public interface Buffer {
  public void set(int value);     // đặt trị vào Buffer
  public int get();               // lấy trị ra khỏi Buffer
}
```

Java dùng phương pháp *lock* (khóa) để thực hiện đồng bộ thread. 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 xuất vào tài nguyên, khi đó đối tượng `Lock` sẽ không cho phép thread khác có đượckhóa, cho đến khi thread đang chạy giải phóng khóa (bằng cách gọi phương thức `unlock()`). Lớp `java.util.concurrent.locks.ReentrantLock` là cài đặt cơ bản của interface `Lock`.

Khi thread đã dùng đối tượng `Lock` khóa tài nguyên dùng chung để truy xuất, nó có thể chờ một điều kiện nào đó, 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, 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 xuất đến buffer dùng chung
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;

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 có dữ liệu (đầy, không có chỗ trống)

  // đặt một trị vào buffer
  @Override public void set(int value) {
    accessLock.lock();                      // lock buffer để ghi
    try {
      while (occupied) {                    // trong lúc buffer không có chỗ trống, đưa thread 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;                       // buffer có chỗ trống rồi, đặt một trị mới vào
      occupied = true;
      System.out.println(" Producer writes %d", buffer);
      canRead.signal();                     // báo cho thread đang chờ là có thể đọc buffer (buffer có dữ liệu)
    } catch (InterruptedException exception) {
      exception.printStackTrace();
    } finally {
      accessLock.unlock();                  // unlock buffer, cho phép thread khác truy xuất 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();                      // lock buffer để đọc
    try {
      while (!occupied) {                   // trong lúc chưa có dữ liệu đọc, đưa thread 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;                   // buffer có dữ liệu rồi, lấy trị từ nó
      System.out.println("Consumer reads %d", readValue);
      canWrite.signal();                    // báo cho thread đang chờ là có thể ghi vào buffer
    } catch (InterruptedException exception) {
      exception.printStackTrace();
    } finally {
      accessLock.unlock();                  // unlock buffer, cho phép thread khác truy xuất buffer
    }
    return readValue;
  }
}
```

Producer: (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 trị 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 exception) {
        exception.printStackTrace();
      }
```

```
    }
    System.out.printf("\n%s\n%s\n", "Producer done producing.", "Terminating Producer.");
  }
}
```

Consumer: (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.out.printf("\n%s %d.\n%s\n",
                      "Consumer read values totaling", sum, "Terminating Consumer.");
  }
}
```

Kiểm tra truy xuất 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();
    }
    application.shutdown();
  }
}
```

IV. Monitor và khối 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 cho phép 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 sẽ bị khóa. Khi thread đó thực hiện xong phương thức, đối tượng 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) {
```

```
    System.out.println("synchronized");
  }
}
```

là tương đương với:

```
public synchronized void foo() {
  System.out.println("synchronized");
}
```

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.

```
public class SynchronizedBuffer implements Buffer {
  private int buffer = -1;              // tài nguyên dùng chung cho thread Producer và Consumer
  private boolean occupied = false;     // báo buffer bận

  // 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();
      }
    }
    buffer = value;
    occupied = true;
    System.out.println(" Producer writes %d", 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();
      }
    }
    occupied = false;
    int readValue = buffer;
    System.out.println("Consumer reads %d", readValue);
    notify();                           // báo cho thread chờ có thể vào trạng thái chạy
    return readValue;
  }                                     // giải phóng lock trên SynchronizedBuffer
}
```

Networking

TCP

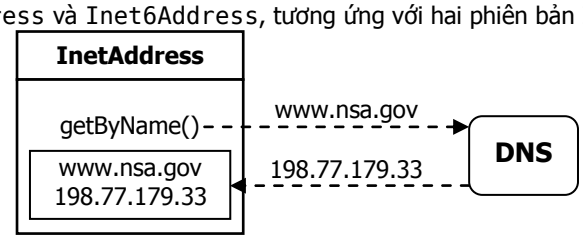
1. Địa chỉ Socket

Khi client bắt đầu truyền thông 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 có thể được chỉ định trong Java bằng cách dùng chuỗi chứa:

- 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 *phân giải* thành địa chỉ dạng số trước khi được dùng.

Lớp trừu tượng `InetAddress` thể hiện một đích 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 IP đang dùng.



Để lấy các địa chỉ của một host cục bộ, ta dùng lớp trừu tượng `NetworkInterface`. Lớp này cung cấp truy xuất đến thông tin của tất cả các 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 đối 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.getHostName() + "/" + address.getHostAddress());
        }
      } catch (UnknownHostException e) {
        System.out.println("\tUnable to find address for " + host);
      }
    }
  }
}
```

`InetAddress`: Khởi tạo, truy xuất

```
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 một định dạng nhất 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 phương thức.

`NetworkInterface`: Khởi tạo, lấy thông tin

```
static Enumeration<NetworkInterface> getNetworkInterfaces()
static NetworkInterface getByInetAddress(InetAddress address)
static NetworkInterface getByName(String name)
Enumeration<InetAddress> getInetAddresses()
String getName()
String getDisplayName()
```

Thường dùng `getNetworkInterfaces` để lấy danh sách các interface của host (kể cả lopback, 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

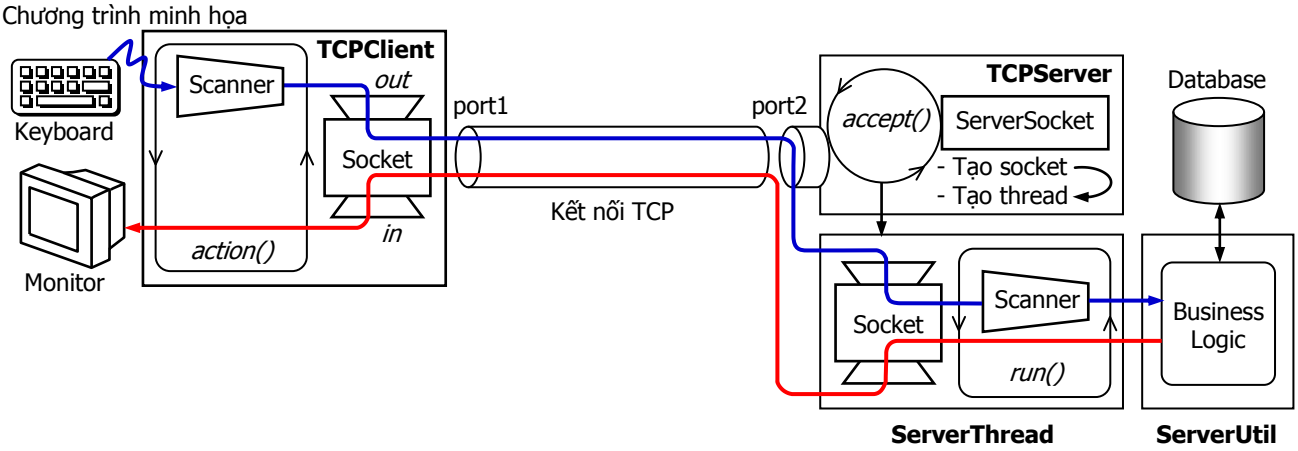
Java cung cấp hai lớp cho TCP: `Socket` và `ServerSocket`.

Một kết nối TCP là một kênh truyền thông hai chiều ảo 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 và 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

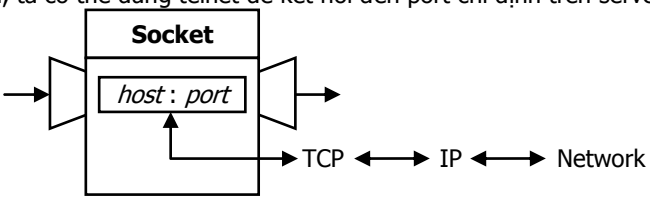


a) TCP Client
Server thụ động lắng nghe tại một port chỉ định (gọi là port dịch vụ), client kết nối đến server tại port dịch vụ đó. Chương trình client điển hình có ba bước:
- Sinh ra một đối tượng `Socket`, kết nối đến host (tên host hoặc địa chỉ IP) và port chỉ định.
- Trao đổi thông tin bằng cách 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 stream với các phương thức 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`.

```
import java.io.*;
import java.util.*;
import java.net.*;

public class TCPClient {
  static final int PORT = 1234;
  static final String HOST = "127.0.0.1";
  private PrintWriter out = null;
  private Scanner in = null;
  private Socket socket = null;

  public TCPClient() {
    try {
      // tạo Socket kết nối với server tại host (IP hoặc hostname) và port chỉ định
      socket = new Socket(HOST, PORT);
      // lấy stream cơ bản từ Socket và lồng thêm stream để có các stream hữu dụng hơn
      out = new PrintWriter(socket.getOutputStream(), true);
      in = new Scanner(socket.getInputStream());
    } catch (Exception e) {
      try { socket.close(); } catch (IOException e1) { }
      e.printStackTrace();
    }
  }

  public void action() {
    Scanner keyboard = new Scanner( System.in );
    String s = "";
    try {
      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 (Exception e) {
      System.out.println("Server has departed");
    } finally {
      in.close();
      out.close();
      try { socket.close(); } catch (IOException e) { }
    }
  }
```

```
  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.



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 cient); 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 host, 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 xuất 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` 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.
+ Server tạo một thread và trao `Socket` vừa tạo ra cho thread để tiến hành giao tiếp với client trong một thread riêng.

```
import java.net.*;
import java.util.concurrent.*;

public class TCPServer {
  static final int PORT = 1234;
  private ServerSocket server = null;

  public TCPServer() {
    try {
      server = new ServerSocket(PORT);  // tạo ServerSocket với port lắng nghe chỉ định
    } catch (Exception e) {
      System.out.println("[LOG] Server start error!");
    }
  }

  public void action() {
    Socket socket = null;
    ExecutorService executor = Executors.newCachedThreadPool();
    System.out.println("[LOG] Server listening...");
    try {
      // khi có kết nối, accept trả về một Socket, kết nối Socket tương ứng phía client
      while ((socket = server.accept()) != null) {
        // trao Socket cho ServerThread để xử lý riêng kết nối trong một thread, rồi đặt thread vào thread pool
        executor.execute(new ServerThread(socket));
        System.out.println("[LOG] Thread generating...");
      }
    } catch (Exception e) {
      System.out.println("[LOG] " + e.getMessage());
    }
  }

  public static void main(String[] args) {
    new TCPServer().action();
  }
}
```

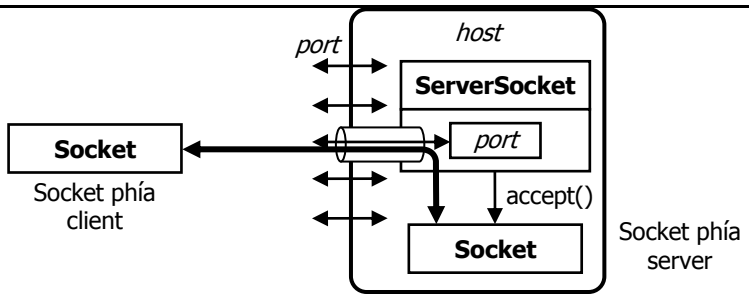
b.2. Server Thread

Để tránh giao tiếp bị ngưng (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.*;
```

```java
import java.util.*;
import java.net.*;

public class ServerThread implements Runnable {
  private Scanner in = null;
  private PrintWriter out = null;
  private Socket socket;

  public ServerThread(Socket socket) throws IOException {
    this.socket = socket;
    // từ Socket (phía Server) lấy stream cơ bản và lồng thêm stream để có các stream hữu dụng hơn
    this.in = new Scanner(this.socket.getInputStream());
    this.out = new PrintWriter(this.socket.getOutputStream(), true);
  }

  @Override public void run() {
    boolean done = false;
    try {
      while (!done) {
        String command = in.nextLine().trim();
        if (isValid(command)) {
          // xử lý dữ liệu nhập nhờ lớp công cụ, trả kết quả về client bằng stream xuất
          int kw = Integer.parseInt(command.substring( 2, command.length() ).trim());
          out.println(ServerUtil.getMoney(kw));
        } else {
          out.println("Bad Command. Example: KW 120");
        }
      }
    } catch (Exception e) {
      System.out.println("[LOG] Client has departed!");
      done = true;
    } finally {
      try { socket.close(); } catch (IOException e) { }
    }
  }

  // phương thức hỗ trợ, dùng kiểm tra cú pháp lệnh do client gửi đến
  private boolean isValid(String s) {
    String pattern = "^KW \\d+$";
    return s.matches( pattern );
  }
}
```



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.

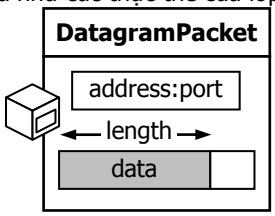
c) Server Util
Lớp `ServerUtil` tách phần thao tác nghiệp vụ (business logic) ra khỏi phần giao tiếp mạng. Vì lớp này được xem như lớp công cụ nên các phương thức đều `static`. Ví dụ về một phương thức nghiệp vụ:

```
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 mảnh, giao thức UDP không bảo đảm các phân mảnh được nhận đầy đủ và lắp ráp đúng thứ tự.

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 đóng gói, gọi là datagram, được 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 chứa 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 tạo `DatagramPacket` 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 tạo `DatagramPacket` 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)
```

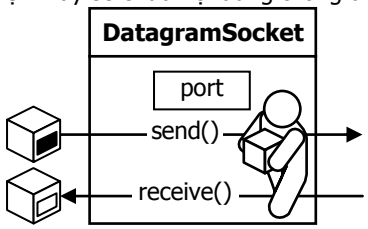
Bổ sung cho constructor, các phương thức này cung cấp một cách khác để truy xuất và thay đổi address của một `DatagramPacket`. Chú ý là phương thức `receive` của `DatagramSocket` thiết lập địa chỉ và port của gói nhận thành địa chỉ và port của người gửi.

`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 chung 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)
```

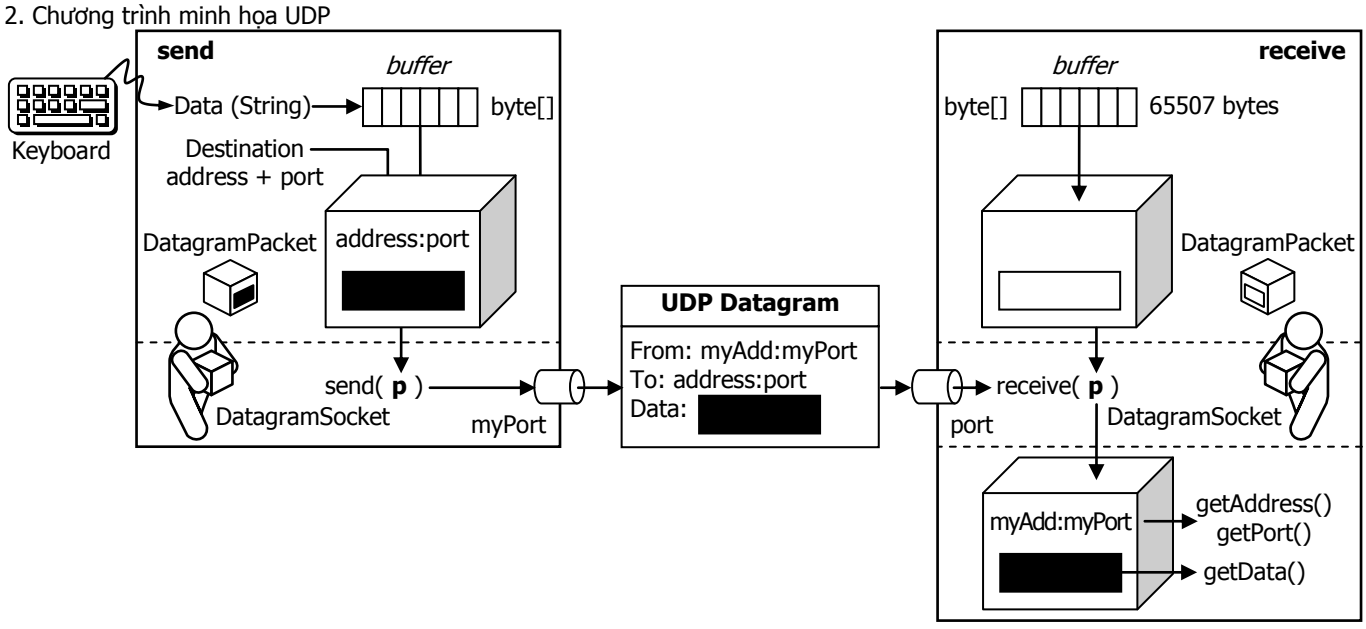
Phương thức `send` dùng gửi `DatagramPacket`. Packet được gửi tới địa chỉ được chỉ định bởi `DatagramPacket`.
Phương thức `receive` khóa (block) cho đến khi nhận được một datagram, dữ liệu datagram nhận được sẽ được sao chép vào `DatagramPacket` cho trước (là đối số của `receive`).

`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 UDP



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).
- Trao đổi thông tin bằng cách gửi và nhận các thực thể của `DatagramPacket` bằng cách dùng các phương thức `send` và `receive` của `DatagramPacket`.
- Khi chấm dứt, giải phóng socket bằng cách dùng phương thức `close` của `DatagramSocket`.

```
import java.io.*;
import java.util.*;
import java.net.*;

public class UDPClient {
  static final int PORT = 1234;
  static final String HOST = "127.0.0.1";
  private DatagramSocket socket = null;
  private InetAddress host = null;

  public UDPClient() {
    try {
      socket = new DatagramSocket();
      host = InetAddress.getByName(HOST);       // địa chỉ đích, thực thể lớp InetAddress
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void action() {
    Scanner keyboard = new Scanner(System.in);
    String s = "";
    try {
      while (true) {
        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();
    } 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 UDPClient().action();
  }
}
```

UDP client và UDP server đều dùng các phương thức `send` và `receive`, mô tả cách chuẩn bị, gửi và nhận datagram:

- `send`: chuyển dữ liệu thành mảng byte, tạo `DatagramPacket` gửi đi (với địa chỉ và port đích), gửi bằng phương thức `send` của `DatagramSocket`.
- `receive`: chuẩn bị `DatagramPacket` nhận với vùng đệm (rỗng) có kích thước tối đa (65507 byte), gọi phương thức `receive` của `DatagramSocket` với datagram là đối số. Trả `DatagramPacket` nhận được trở về.

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 (đối số của phương thức receive) chứa địa chỉ của client cần để gửi trả lời.
- Gửi datagram trả lời bằng cách dùng phương thức send của DatagramSocket.

```
import java.io.*;
import java.net.*;

public class UDPServer {
  static final int PORT = 1234;
  private DatagramSocket socket = null;

  public UDPServer() {
    try {
      socket = new DatagramSocket(PORT);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void action() {
    InetAddress host = null;
    int port;
    String s = "";
    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
        host = packet.getAddress();
        port = packet.getPort();
        s = new String(packet.getData()).trim();
        System.out.println("[log] " + s);
        // xử lý dữ liệu nhập nhờ business object, đóng gói kết quả vào datagram và gửi về bằng phương thức send
        if (isValid(s)) {
          int kw = Integer.parseInt(s.substring(2, s.length()).trim());
          send(ServerUtil.getMoney(kw), host, port);
        } else {
          send("Bad Command. Example: KW 120", host, port);
        }
      }
    } catch (Exception e) {
      e.printStackTrace();
    } 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;
  }

  private boolean isValid(String s) {
    String pattern = "^KW \\d+$";
    return s.matches(pattern);
  }

  public static void main(String[] args) {
    new UDPServer().action();
  }
}
```

3. Broadcast và Multicast
Có hai kiểu dịch vụ one-to-many (một gửi nhiều): broadcast và multicast. Với broadcast, mọi host trên mạng (cục bộ) sẽ nhận một bản sao của thông điệp. Với multicast, thông điệp được gửi đến một địa chỉ multicast (lớp D), và mạng sẽ phân phối nó chỉ cho những host đăng ký nhân các thông điệp được chuyển đến địa chỉ này. Nói chung, 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 thay vì một địa chỉ (unicast) thông thường. IPv6 không cung cấp địa chỉ broadcast, thay vào đó là địa chỉ link-local multicast (FFO2::1). Địa chỉ IPv4 broadcast cục bộ (255.255.255.255) cho phép gửi thông điệp đến mọi host trên cùng một mạng, thông điệp gửi đến địa chỉ này không bao giờ được router chuyển tiếp ra ngoài mạng.
Trong Java, viết code cho unicast và broadcast là như nhau, ta chỉ cần dùng địa chỉ broadcast cục bộ 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 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` dùng quản lý các thành viên của nhóm multicast. Kết nối đến một nhóm bằng `MulticastSocket` 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

```
import java.io.*;
import java.net.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MulticastChat implements Runnable, WindowListener, 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();
    this.start();
  }

  protected void initNet() 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 void init() {
    frame = new JFrame("MulticastChat [" + group.getHostAddress () + ":" + port + "] - " + nick);
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    frame.setSize(360, 200);
    frame.setResizable(false);
    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);
    frame.setLayout(new BorderLayout());
    frame.add(scroll, "Center");
    frame.add(input, "South");
  }

  public synchronized void start() throws IOException {
    if (listener == null) {
      initNet();
      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();
      }
    }
  }

  // WindowListener
```

```
  public void windowOpened(WindowEvent event) { input.requestFocus(); }
  public void windowClosing(WindowEvent event) {
    try { stop(); } catch (IOException e) { }
  }
  public void windowClosed(WindowEvent event) { }
  public void windowIconified(WindowEvent event) { }
  public void windowDeiconified(WindowEvent event) { }
  public void windowActivated(WindowEvent event) { }
  public void windowDeactivated(WindowEvent event) { }

  // ActionListener
  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) { }
      socket.close();
    }
  }

  // Runnable
  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].indexOf (":") < 0))
      throw new IllegalArgumentException("Syntax: MulticastChat <group>:<port> <nick>");
    int idx = args[0].indexOf(":");
    InetAddress group = InetAddress.getByName(args[0].substring(0, idx));
    int port = Integer.parseInt(args[0].substring( idx + 1 ));
    new MulticastChat(group, port, args[1]);
  }
}
```

RMI

RMI (Remote Method Invocation) cung cấp khả năng triệu gọi các phương thức của một đối tượng từ xa 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 này RMI được cài đặt đơn giản hơn rất nhiều, bài này trình bày các bước cài đặt chủ yếu của RMI. Để đơn giản, chúng tôi không bàn đến quản lý bảo mật cho ứng dụng 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 phía RMIServer.
Cài đặt đối tượng remote như sau:
- Định nghĩa interface remote.
- Định nghĩa lớp thực thi của đối tượng remote.
Lưu ý vai trò của interface remote là bộc lộ các phương thức được triệu gọi của đối tượng remote. Interface này sẽ được chuyển cho phía client.
Vì vậy, nếu chúng ta đã có sẵn đối tượng nghiệp vụ, để chuyển đối tượng này thành đối tượng triệu gọi từ xa, tạo interface remote chứa các phương thức được triệu gọi và điều chỉnh cho đối tượng đó cài đặt interface remote này.

a) Định nghĩa 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` khi gặp các lỗi liên quan đến kết nối mạng hoặc lỗi server.

```
public interface ServerUtilInterface extends Remote {
  public String getMoney(int kw) throws RemoteException;
}
```

b) Định nghĩa lớp thực thi của đối tượng remote
Phía RMIServer, lớp thực thi của đối tượng remote sẽ cài đặt interface remote 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 có dùng một POJO để truyền qua mạng cần 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
RMIServer thực hiện các công việc sau:

a) Khởi động dịch vụ Naming
Đối tượng remote có bản chất phân tán, vị trí của chúng phải trong suốt đối với client. Vì vậy, RMIClient phải định vị được đối tượng remote để có thể triệu gọi từ xa các phương thức. Điều này được thực hiện dưới sự hỗ trợ của dịch vụ Remote Object Registry (RMI Registry). RMI Registry là 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 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`.
Trước đây phải khởi động RMI Registry 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 (instatiate) 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 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.
- RMI server 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`. Nhiều cài đặt RMI trước đây không thành công là do lớp Stub này 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 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 iServer = ServerUtilInterface)registry.lookup("rmi://127.0.0.1/kw_service");
```

b) Triệu gọi các phương thức từ xa

RMIClient triệu gọi phương thức từ xa của đối tượng remote như triệu gọi phương thức của một đối tượng bình thường. Ngoại trừ việc ném ra exception `RemoteException` nếu triệu gọi không thành công.

Việc triệu gọi được thực hiện thông qua interface remote, còn thực thi triệu gọi được thực hiện ẩn bên dưới do lớp Stub đảm nhiệm.

```
int kw = 123;
System.out.println(iServer.getMoney(kw));
```

JDBC

SQL

Dữ liệu trong cơ sở dữ liệu quan hệ được lưu trữ trên nhiều bảng, bao gồm các hàng và cột. Mỗi bảng thường mô tả một thực thể hoặc mối quan hệ giữa hai thực thể. Mỗi hàng trong cơ sở dữ liệu là bản ghi được xác định duy nhất với một định danh gọi là khóa chính.

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:

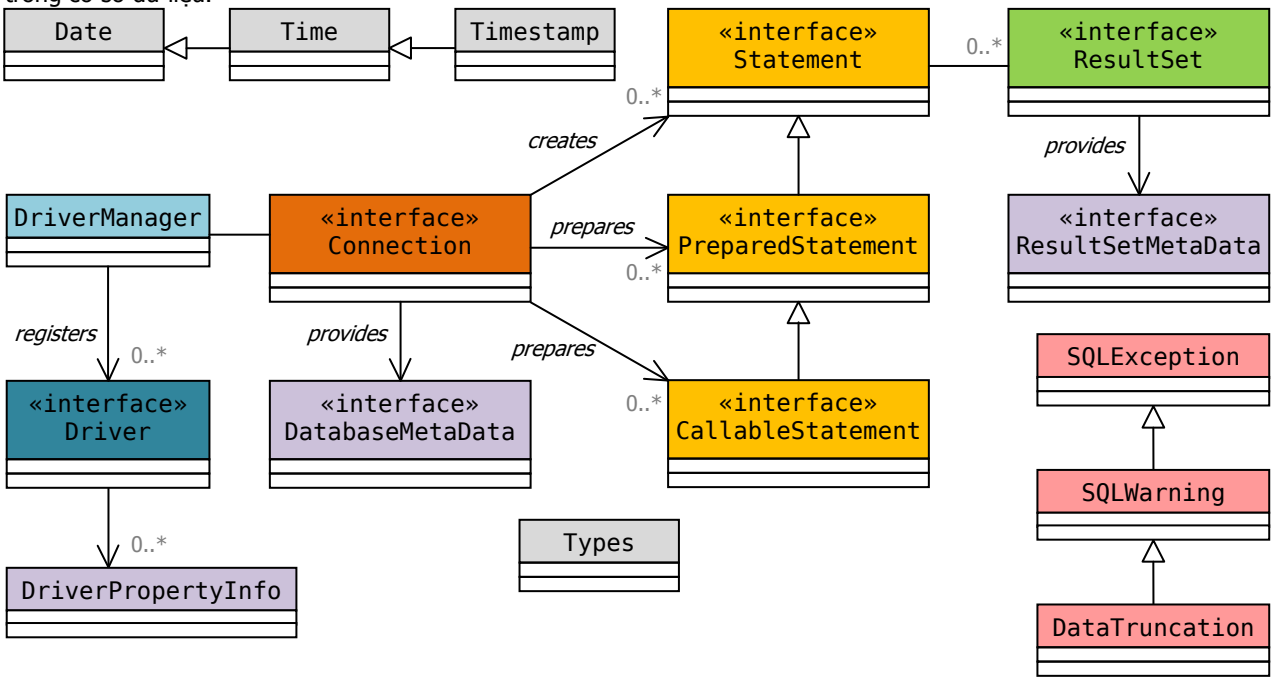
- Ngôn ngữ thao tác dữ liệu (DML – Data Manipulation Language) sử dụng các từ khóa SELECT, INSERT, UPDATE, DELETE để truy vấn và thao tác lên dữ liệu trong bảng.
- Ngôn ngữ định nghĩa dữ liệu (DDL – Data Definition Language) được dùng để tạo và thao tác lên cấu trúc của bảng: CREATE TABLE, ALTER TABLE, ADD COLUMN, DROP COLUMN, ADD FOREIGN KEY, …

Stored procedure được định nghĩa như một tập các khai báo SQL được lưu trữ ngay trong cơ sở dữ liệu và sau đó, được triệu gọi bởi một chương trình, một trigger hay một stored procedure khác.

Bạn có thể tham khảo SQL và stored procedure từ tài liệu khác.

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.



1. 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 cơ sở dữ liệu, hầu hết JDBC API được định nghĩa bằng cách dùng interface và cho 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 cơ sở dữ liệu khác nhau (hoặc cho cùng một cơ sở dữ liệu).

Bạn có thể dùng 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 để 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. 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 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 discovery) vào thư mục META-INF/services của gói chứa driver.
JVM nạp lớp `DriverManager` của JDBC API. Lớp `DriverManager` nạp tất 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`.

2. `Connection`
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:

MySQL	`jdbc:mysql://hostname:portNumber/databaseName`
ORACLE	`jdbc:oracle:thin:@hostname:portNumber:databaseName`
DB2	`jdbc:db2:hostname:portNumber/databaseName`
PostgreSQL	`jdbc:postgresql://hostname:portNumber/databaseName`
Java DB/Apache Derby	`jdbc:derby:dataBaseName(embedded)`
	`jdbc:derby://hostname:portNumber/databaseName(network)`
Microsoft SQL Server	`jdbc:sqlserver://hostname:portNumber;databaseName=dataBaseName`
Sybase	`jdbc:sybase:Tds:hostname:portNumber/databaseName`

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());
```

3. `Statement`
a) `Statement`
Từ thực thể `Connection`, bạn tạo 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 hoặc trả về một số `int` cho biết số dòng trong bảng cơ sở dữ liệu bị ảnh hưởng.

```
Connection conn = DriverManager.getConnection(url, user, pwd);
Statement stmt = conn.createStatement();
ResultSet rs = stmt.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 do 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 trị `boolean` được trả về, truy vấn đã được thực hiện. Thông thường, phương thức này dùng để gọi một stored procedure.

```
ResultSet rs;
int numRows;
boolean status = stmt.execute("");    // true nếu trả về ResultSet
if (status) {
  rs = stmt.getResultSet();           // lấy ResultSet
  // xử lý ResultSet ...
} else {
  numRows = stmt.getUpdateCount();    // lấy số dòng bị ảnh hưởng
  if (numRows == -1) {      // nếu -1, không có kết quả
    System.out.println("No results");
  } else {
    System.out.println(numRows + " rows affected.");
  }
}
```

- `Statement` dùng lại được, nếu bạn thực hiện một chuỗi các truy vấn, bạn chỉ cần một thực thể `Statement`.

b) `PreparedStatement`
Truy vấn SQL có thể nhận các tham số, việc chèn trực tiếp tham số vào câu truy vấn phức tạp (phân biệt chuỗi và số), đồng thời dễ dẫn đến 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 thiết lập bằng cách gọi các phương thức setXxx(index, value) với index là số thứ tự của ký tự (?) tính từ 1, và value là trị của tham số có kiểu dữ liệu Xxx.

```
PreparedStatement psmt = connection.prepareStatement(
    "SELECT AVG(GPA) FROM Student WHERE age>=? AND age<?");
// thiết lập các tham số thực:
pstmt.setInt(1, age);
pstmt.setInt(2, age+10);
ResultSet rs = pstmt.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 trị tham số khác nhau:

```
PreparedStatement pstmt = connection.prepareStatement(
    "INSERT INTO OrderItems VALUES(?,?,?,?)")
for (Product product : listProduct) {
  // thiết lập các tham số thực cho một truy vấn trong lô:
  pstmt.setInt(1, orderId);
  pstmt.setInt(2, product.getId());
  pstmt.setInt(3, product.getQuantity());
  pstmt.setBigDecimal(4, product.getPrice());
  pstmt.addBatch();
}
pstmt.executeBatch();
```

c) CallableStatement
Bạn có thể gọi stored procedure lưu trong cơ sở dữ liệu, có tham số hoặc không, bằng cách dùng CallableStatement.

```
CallableStatement cstmt = connection.prepareCall("{CALL findByPrice (?, ?)}");
// thiết lập các tham số thực:
cstmt.setInt(1, age);
cstmt.setInt(2, age+10);
ResultSet rs = cstmt.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().

4. 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.

a) Di chuyển tới 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 getXxx() nạp chồng 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";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
  String title = rs.getString("Title");
  java.util.Date pubDate = new java.util.Date(rs.getDate("PubDate").getTime());
  float price = rs.getFloat("Price");
  books.add(new Book(title, pubDate, price));
}
```

b) 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";
ResultSet rs = stmt.executeQuery(query);
ResultSetMetaData rsmd = rs.getMetaData();
rs.next();
int colCount = rsmd.getColumnCount(); // số cột trong ResultSet
System.out.println("Column Count: " + colCount);
for (int i = 1; i <= colCount; i++) {
  System.out.println("Table Name  : " + rsmd.getTableName(i));
  System.out.println("Column Name : " + rsmd.getColumnName(i));
  System.out.println("Column Size : " + rsmd.getColumnDisplaySize(i));
}
```

c) 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 ánh 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 ánh 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.

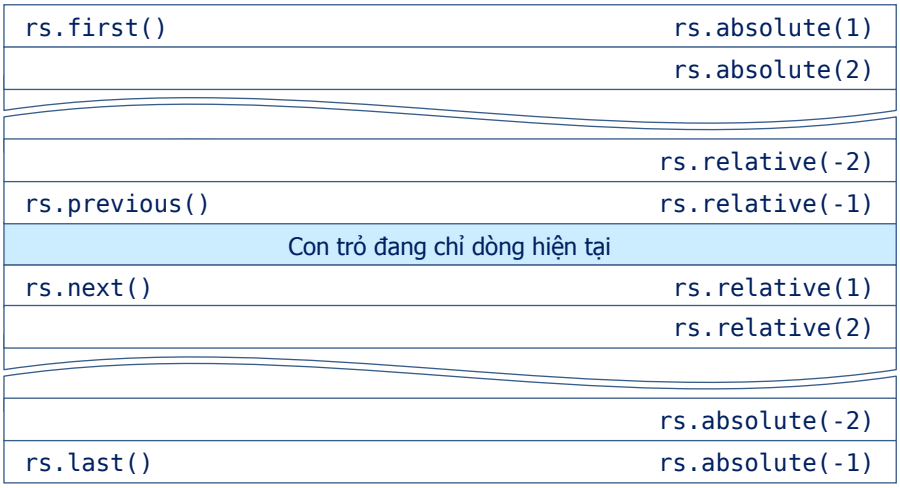
Kiểm tra cơ sở dữ liệu và JDBC driver có hỗ trợ các tùy chọn trên bằng DatabaseMetaData.

```
Connection conn = DriverManager.getConnection(...);
DatabaseMetaData dbmd = conn.getMetaData();
if (dbmd.supportsResultSetType(ResultSet.TYPE_FORWARD_ONLY)) {
  System.out.print("Supports TYPE_FORWARD_ONLY");
  if (dbmd.supportsResultSetConcurrency(ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE)) {
    out.println(" and supports CONCUR_UPDATABLE");
  }
}

if (dbmd.supportsResultSetType(ResultSet.TYPE_SCROLL_INSENSITIVE)) {
  System.out.print("Supports TYPE_SCROLL_INSENSITIVE");
  if (dbmd.supportsResultSetConcurrency(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE)) {
    System.out.println(" and supports CONCUR_UPDATABLE");
  }
}

if (dbmd.supportsResultSetType(ResultSet.TYPE_SCROLL_SENSITIVE)) {
  out.print("Supports TYPE_SCROLL_SENSITIVE");
  if (dbmd.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 conn = DriverManager.getConnection(...);
Statement stmt = conn.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:



d) 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 conn = DriverManager.getConnection(...);
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
...
// thử cập nhật
rs.absolute(2);           // chuyển đến dòng thứ 2 của ResultSet
rs.updateFloat("balance", 42.55f);
rs.updateRow();
rs.cancelRowUpdates();    // bỏ qua các cập nhật trong ResultSet
// thử chèn
rs.moveToInsertRow();     // chuyển đến dòng đặc biệt trong ResultSet dùng insert
rs.updateInt("acctNum", 999999);
rs.updateString("surname", "Harrison");
rs.updateString("firstName", "Christine Dawn");
rs.updateFloat("balance", 25.00f);
rs.insertRow();
// thử xóa
rs.moveToCurrentRow();    // chuyển đến dòng vừa chèn
rs.deleteRow();
```

5. 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 trở lại 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`.

JDBC thiết lập mặc định transaction cho từng phát biểu SQL, gọi là chế độ autocommit, bạn cần bất hoạt chế độ này để nhóm các phát biểu SQL thành 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_COMITTED và TRANSACTION_READ_UNCOMMITTED.

Bạn cũng có thể thiết lập transaction là đọc ghi (mặc định, false) hoặc chỉ đọc (true):

```
connection.setReadOnly(true);
```

6. 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 `getNextExcption()`.

```
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 rs = stmt.executeQuery(someQuery);
SQLWarning warning = stmt.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 (rs.next()) {
  int value = rs.getInt(1);
  SQLWarning warning = rs.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();
  }
}
```

Build system

Ant là một build-system dùng xây dựng (build) nhanh ứng dụng. Ant được sử dụng phổ biến, xem như là nền tảng của build-system cho các IDE như NetBeans. 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 J2EE, đóng gói sản phẩm, làm việc nhóm qua mạng, các project open source, …

Ant

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`.

Vào hộp thoại System Properties (⊞+pause), tab Advanced và chọn nút Environment Variables. Trong phần User variables, nhấn nút New để thêm biến môi trường: Variable name=ANT_HOME, Variable value=`C:/java/ant`

Ngoài ra, nên thêm đường dẫn chỉ đến thư mục `bin` của Ant (`C:/java/ant/bin`) vào đường dẫn hệ thống (hiệu chỉnh trị của biến môi trường PATH). Logoff để các biến môi trường có hiệu lực.

Ant xem 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*, 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.

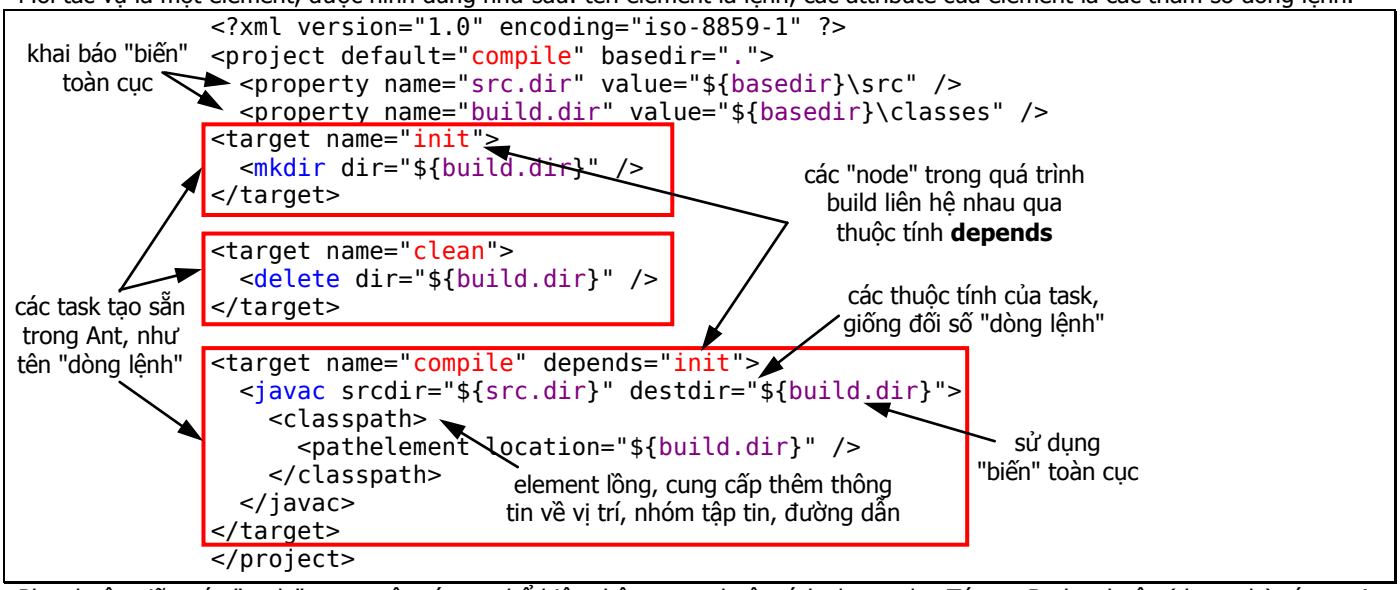
Trong NetBeans, build.xml chứa tập tin build thật sự là nbproject/build-impl.xml.

2. Tập tin build

Tập tin build.xml điển hình bao gồm:

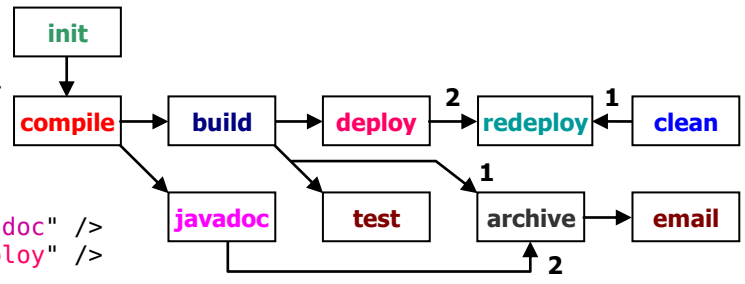
- Phần khai báo các "biến" toàn cục bằng element `property`, biến toàn cục chẳng hạn `src.dir`, sau khi khai báo sẽ dùng như `${src.dir}`.
- Phần khai báo các tác vụ (task) bằng element `target`. Mỗi tác vụ là một "node" trong cây tác vụ dùng để xây dựng ứng dụng. Mỗi tác vụ gọi một hoặc nhiều tác vụ trong hàng trăm tác vụ cài đặt sẵn của Ant.

Mỗi tác vụ là một element, được hình dung như sau: tên element là lệnh, các attribute của element là các tham số dòng lệnh.



Phụ thuộc giữa các "node" trong cây tác vụ thể hiện 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, để thực hiện tác vụ `redeploy`, trước hết cần thực hiện tác vụ `clean`, sau đó thực hiện chuỗi tác vụ `init – compile – build – deploy`.

Khi một tác vụ trong chuỗi tác vụ được gọi, các tác vụ trước nó sẽ được thực hiện trước, kể từ tác vụ đầu chuỗi.

Có thể gọi một tác vụ 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 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 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 (thường có ví dụ tổng quát kèm theo), hoặc tra cứu trên http://ant.apache.org/index.html

3. Thực hành

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:

- tự động biên dịch `HelloWorld.java`.
- đóng gói thành `project.jar`.
- 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`

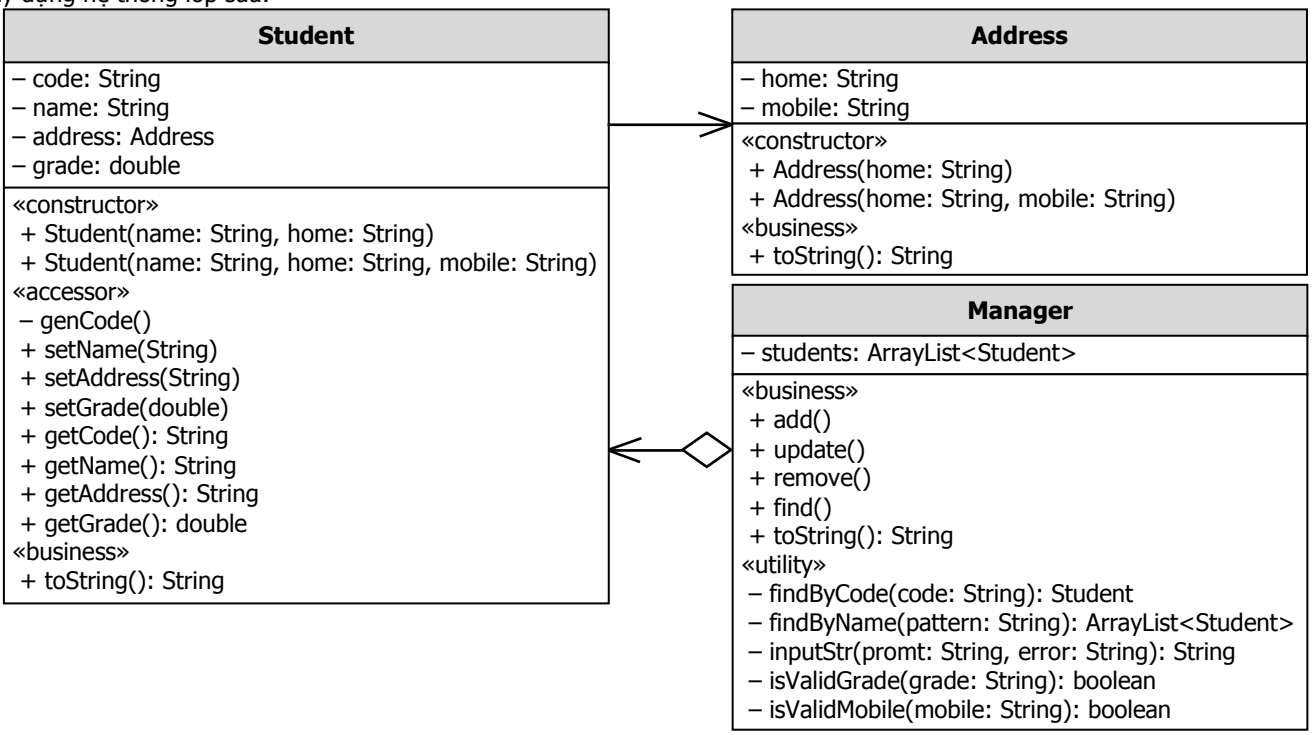
Maven

(under construction)

Bài tập

[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 4 ký tự số lấy từ 4 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 `ArrayList` chứa các đối tượng lớp `Student`. Lớp `Manager` có thể:

- Thêm một đối tượng mới thuộc lớp `Student` vào `ArrayList` do nó quản lý bằng phương thức `add()`, đối tượng mới chỉ cần các thông tin (`name`, `address`), điểm (`grade`) sẽ cập nhật sau và `code` sẽ sinh tự động bằng `genCode()`.
- Loại một đối tượng `Student` chỉ định bằng `code` khỏi `ArrayList` do nó quản lý bằng phương thức `remove()`.
- Cập nhật một đối tượng `Student` chỉ định bằng `code` trong `ArrayList` do nó quản lý 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` trong danh sách sinh viên 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 sẽ được chọn.
- Xuất danh sách sinh viên bằng phương thức `toString()` của lớp `Manager`.

Lớp `Manager` có một số phương thức công cụ hỗ trợ xây dựng các phương thức nghiệp vụ trên:

- `findByCode()`: tìm một `Student` theo mã.
- `findByName()`: trả về `ArrayList` các `Student` có tên chứa chuỗi `pattern` nhập vào. Phương thức `find()` gọi phương thức công cụ này.
- `inputStr()`: dùng khi nhập liệu cho một đối tượng `Student` mới, cảnh báo khi chuỗi nhập không hợp lệ và yêu cầu nhập lại cho đến khi chuỗi nhập hợp lệ.
- `isValidGrade()`: khi cập nhật điểm, yêu cầu kiểm tra điểm nhập phải là số thực và hợp lệ (0 ≤ `grade` ≤ 10).
- `isValidMobile()`: khi nhập số điện thoại, yêu cầu kiểm tra số điện thoại phải là chuỗi số 10 ký tự.

Test driver dùng một menu hiển thị các chức năng của chương trình.

Nếu người dùng nhập lựa chọn cho menu khác 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.

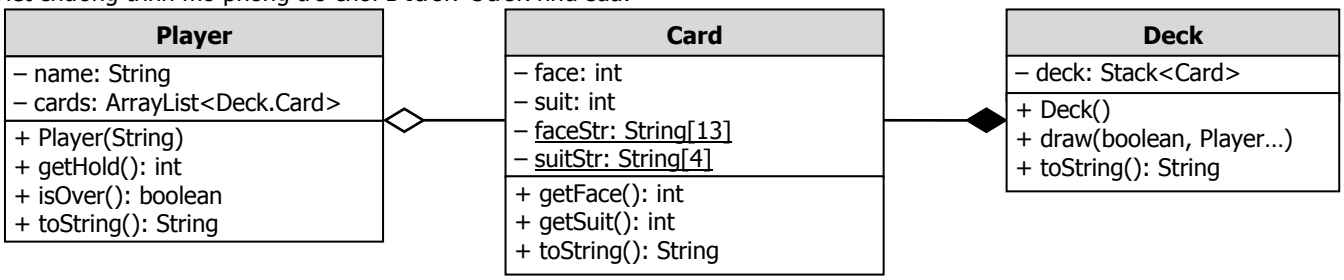
Kết quả chạy mẫu như sau:

```
---------- 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 [LIO4430]!
```

```
---------- 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? LIO4430
Hint: use 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(s):
[LIO4934] 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
[LIO4934] 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? LIO4430
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. Phương thức `draw()` có thể chia bài cho nhiều người chơi trong một lần gọi phương thức.

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 dừng 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: delear 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.

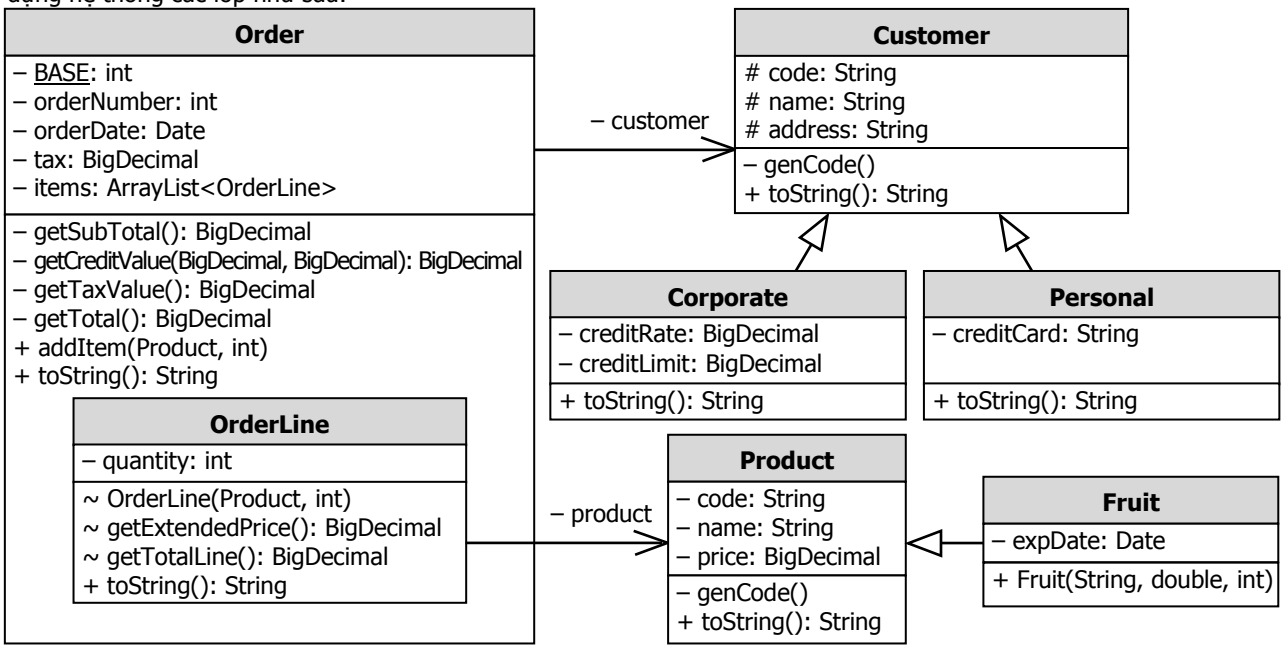
Khi được hỏi, nếu người chơi nhập khác "y" hoặc "n", chương trình báo lỗi và yêu cầu nhập lại.

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 hearts, 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
```

[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 đánh thuế tiêu dùng (`tax`) tính bằng % trên tổng giá trị đơn hàng đã trừ chiết khấu cũng tính bằng % nếu có. Giá trị thanh toán đơn hàng được tính từ phương thức `getTotal()`, bằng tổng giá trị đơn hàng cộng thuế.

Lớp Order còn có các phương thức:

- `getSubTotal()` dùng tính giá trị đơn hàng trước chiết khấu và trước thuế.
- `getCreditValue()` dùng tính giá trị chiết khấu cho đơn hàng trong trường hợp khách hàng là công ty.
- `getTaxValue()` dùng tính giá trị thuế tiêu dùng.

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 loại khách hàng sau:

- Khách hàng cá nhân (`Personal`) có thông tin về thẻ tín dụng (`creditCard`).
- 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 không được vượt quá giới hạn chiết khấu (`creditLimit`).

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`) và số lượng đặt mua (`quantity`). Tổng giá trị một mục hàng được tính từ phương thức `getTotalLine()`. Các mục đặt hàng 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` 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`).

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`.

Khách hàng và sản phẩm được tạo mã tự động bằng phương thức `genCode()`, mã tự động gồm: ba ký tự đầu của tên (`name`) 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).

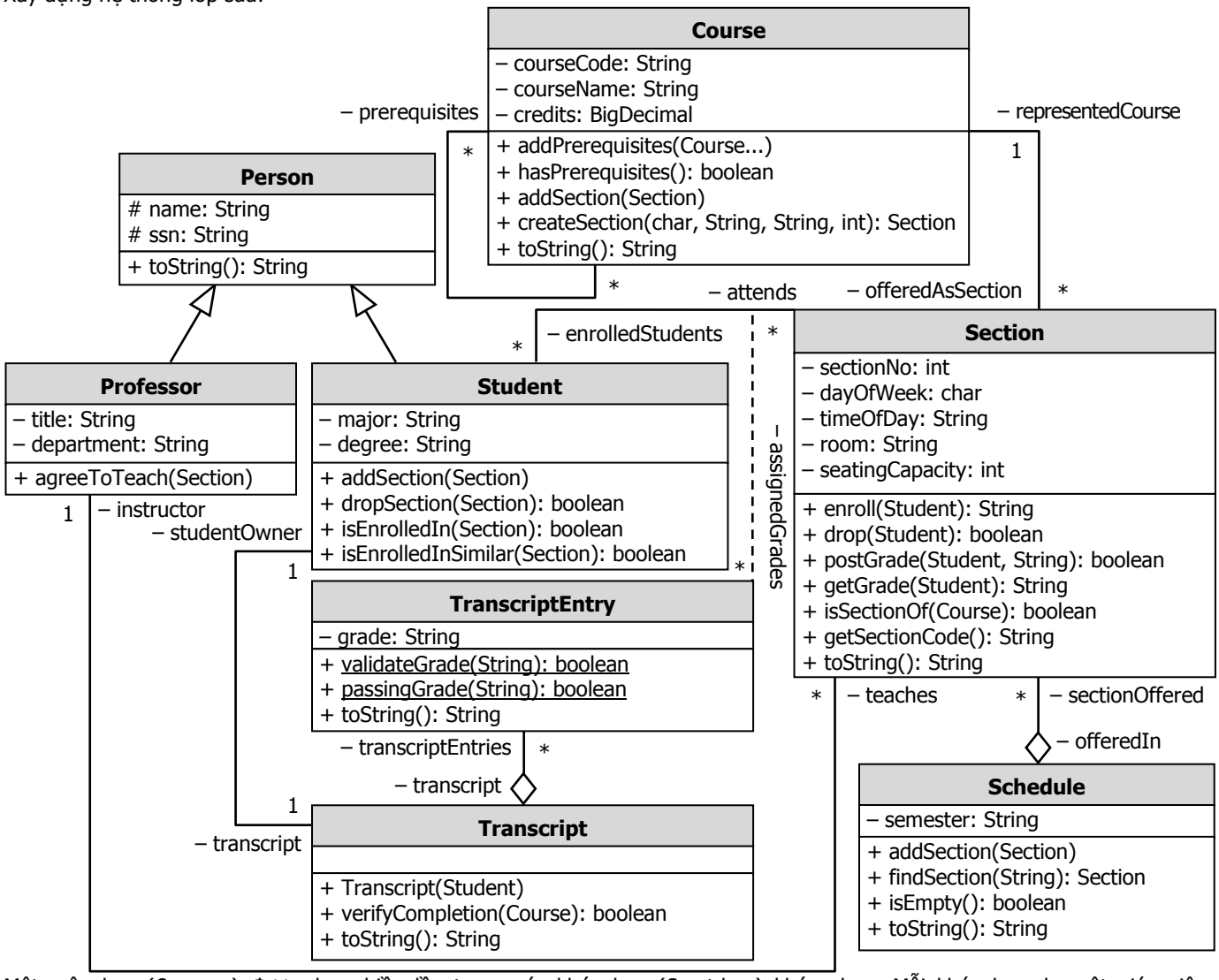
Phương thức `toString()` của lớp `Order` dùng để xuất một đơn hàng đơn hàng cụ thể cho loại khách hàng cụ thể như kết quả chạy mẫu dưới đây:

```
INVOICE #100   Date: [11 Mar 2014]
Customer   : Karl Marx [KAR54444]
Address    : 12, Albert Einstein, Berlin, German
Credit Card: 1234-5678-9012-3456
-------------------------------------------------
Code       Name         Price    Qty   Total Line
-------------------------------------------------
[USA54448] USA Orange   4.7      3     14.1
[CHI54450] China Mango  2.4      2     4.8
-------------------------------------------------
                            Subtotal: 18.9
                            Tax     : 0.567
                            Total   : 19.467

INVOICE #101   Date: [11 Mar 2014]
Customer   : Apple Inc. [APP54445]
Address    : Cupertino, CA 95014, USA
Credit Rate: 4.3%
-------------------------------------------------
```

```
Code       Name         Price    Qty   Total Line
-------------------------------------------------
[USA54448] USA Orange   4.7      3     14.1
[CHI54450] China Mango  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).
- 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 đang dạy).

Phương thức `agreeToTeach()` dùng để thêm khóa học vào danh sách các khóa học nhận 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 đã đăng ký khóa học này trong `attends` chưa.
+ `isEnrolledInSimilar()`: kiểm tra xem đã học 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: `prerequistes` (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 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) 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 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. Phương thức này thành công, trả về true, khi loại bỏ thành công sinh viên khỏi danh sách enrolledStudents và loại khóa học này khỏi danh sách attends của sinh viên đó.
  + 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 rớt 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("Vladymir 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('M', "07:00-10:15 AM", "302", 24);
    Section se2 = c2.createSection('T', "02:15-05:30 PM", "405", 30);
    Section se3 = c3.createSection('F', "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:

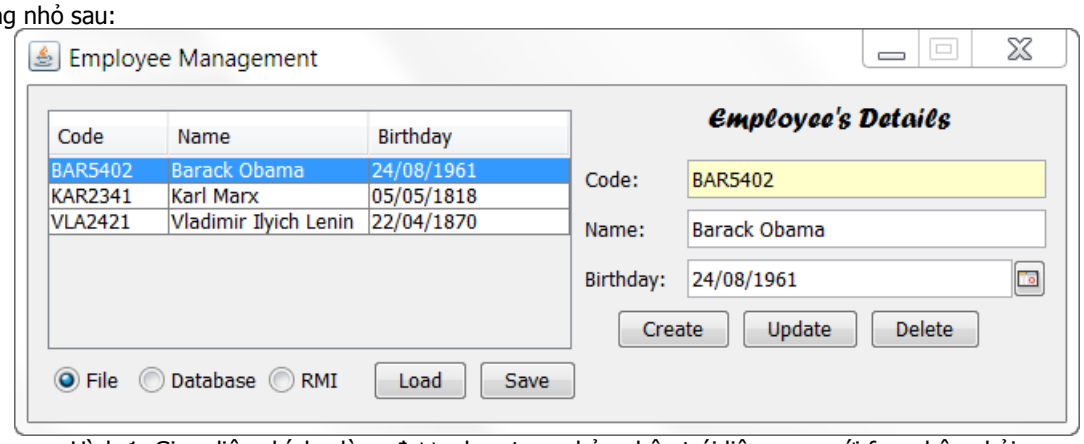
```
[Course Information]
  Course No. : I2C101
  Course Name: Introduction to Computing
  Credits    : 1.5
  Prerequisite Courses: [none]
  Offered As Section(s): I2C101-1

[Course Information]
  Course No. : OOP102
  Course Name: Object-Oriented Programming
  Credits    : 3.5
  Prerequisite Courses:
    Introduction to Computing
  Offered As Section(s): OOP102-1

[Course Information]
  Course No. : CJV103
  Course Name: Core Java Programming
  Credits    : 2
  Prerequisite Courses:
    Introduction to Computing
    Object-Oriented Programming
  Offered As Section(s): CJV103-1

[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 (M, 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 (T, 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 (F, 08:45-12:00 AM)

[Section Information]
  Semester   : Summer 2014
  Course No. : I2C101
  Section No.: 1
  Offered    : M, 07:00-10:15 AM
  In Room    : 302
  Professor  : Greg Anderson
  Total of 3 students enrolled, as follows:
    Vladymir Putin
    Francois Hollande
    Barack Obama

[Section Information]
  Semester   : Summer 2014
  Course No. : OOP102
  Section No.: 1
  Offered    : T, 02:15-05:30 PM
  In Room    : 405
  Professor  : Bjarne Stroustrup
  Total of 2 students enrolled, as follows:
    Vladymir Putin
    Francois Hollande

[Section Information]
  Semester   : Summer 2014
  Course No. : CJV103
  Section No.: 1
  Offered    : F, 08:45-12:00 AM
  In Room    : 507
  Professor  : James Gosling
  Total of 1 students enrolled, as follows:
    Francois Hollande

[Schedule Information]
Schedule for Summer 2014:
  T, 02:15-05:30 PM [Room: 405] Object-Oriented Programming (Bjarne Stroustrup)
  F, 08:45-12:00 AM [Room: 507] Core Java Programming (James Gosling)
  M, 07:00-10:15 AM [Room: 302] Introduction to Computing (Greg Anderson)

[Person Information]
  Name       : Barack Obama
  SS Number  : 123-12-4321
Student-Specific Information:
  Major      : IT
  Degree     : Ph.D.
Course Schedule for Barack Obama:
  M, 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
```

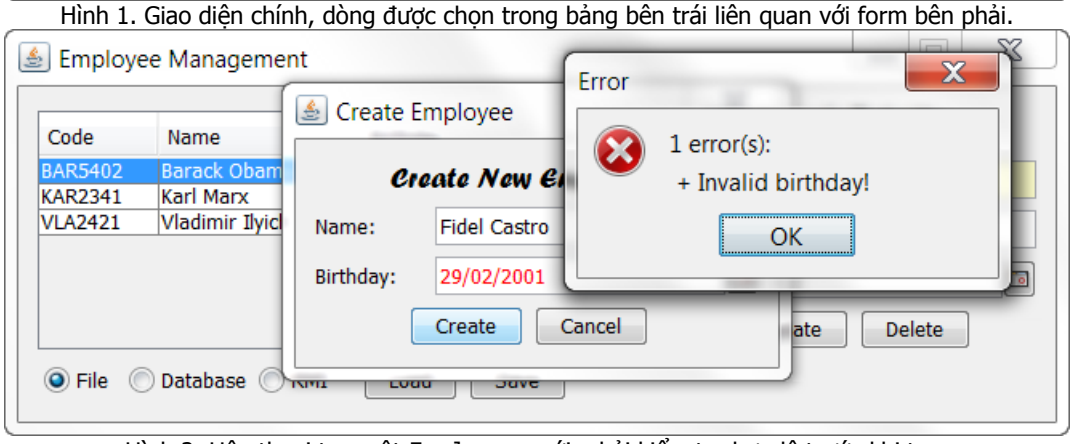
```
Student-Specific Information:
  Major      : Math
  Degree     : M.S.
Course Schedule for Vladymir Putin:
  T, 02:15-05:30 PM [Room: 405] Object-Oriented Programming (Bjarne Stroustrup)
  M, 07:00-10:15 AM [Room: 302] Introduction to Computing (Greg Anderson)
Transcript for Vladymir Putin:
  Introduction 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:
  T, 02:15-05:30 PM [Room: 405] Object-Oriented Programming (Bjarne Stroustrup)
  F, 08:45-12:00 AM [Room: 507] Core Java Programming (James Gosling)
  M, 07:00-10:15 AM [Room: 302] Introduction to Computing (Greg Anderson)
Transcript for Francois Hollande:
  Introduction to Computing (B)
  Object-Oriented Programming (A+)
```

[Case study]

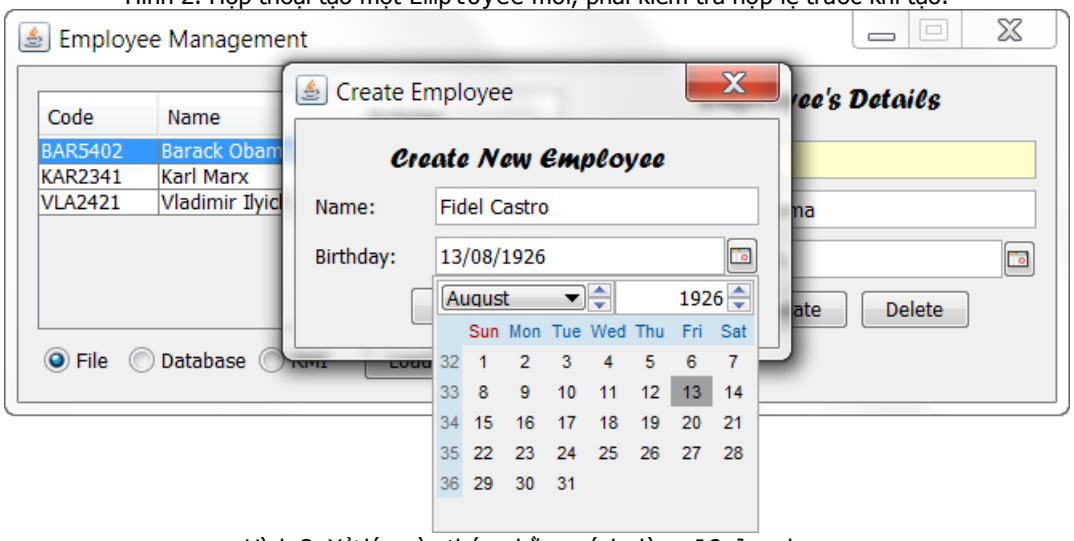
Viết ứng dụng nhỏ sau:



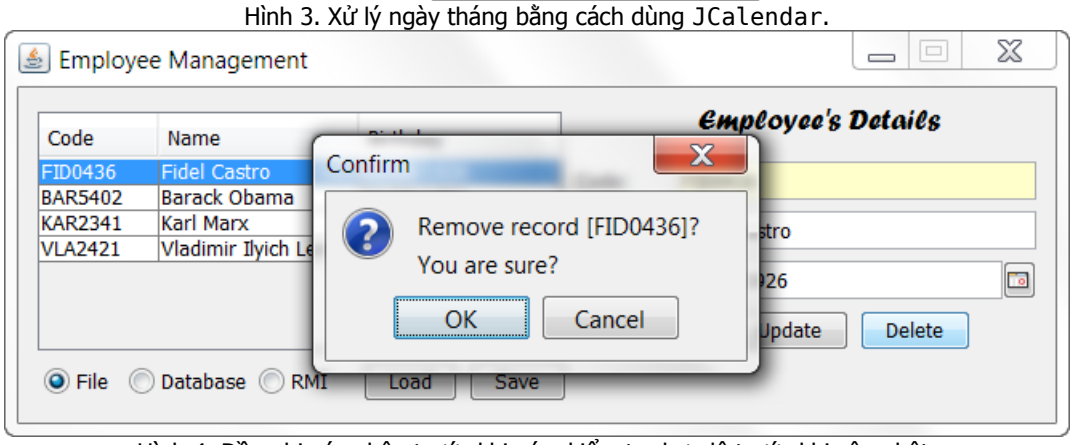
Hình 1. Giao diện chính, dòng được chọn trong bảng bên trái liên quan với form bên phải.



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:

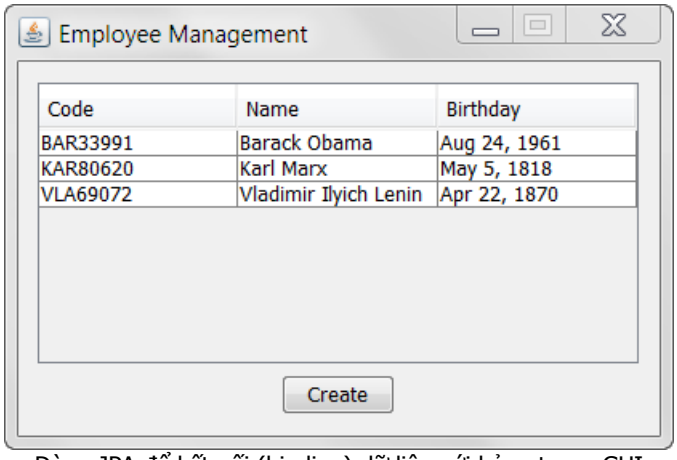
Employee

Column Name	Data Type	Allow Nulls
code	varchar(15)	☐
name	varchar(30)	☐
birthday	datetime	☐

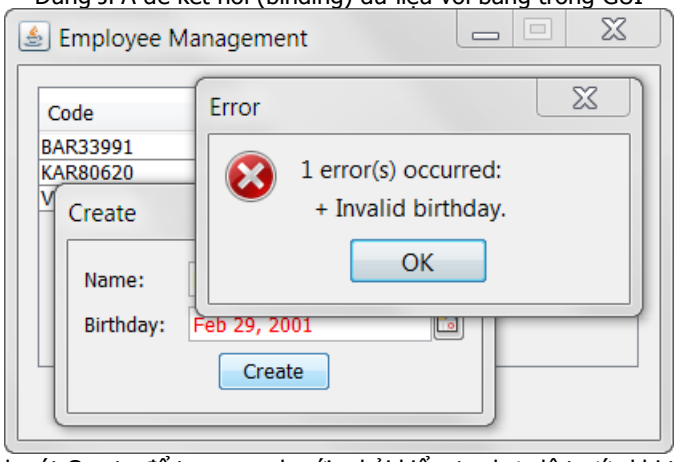
- Dùng RMI, cũng lưu dữ liệu trong tập tin văn bản nhưng các thao tác được triệu gọi từ xa.
Có thể nạp dữ liệu bằng một cách và lưu dữ liệu bằng một cách khác, trong ba cách trên.
Các lớp được tổ chức theo đề nghị sau:
- Lớp POJO `Employee`, là đối tượng vận chuyển (transfer object) để chứa dữ liệu cần 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.

[Case study]

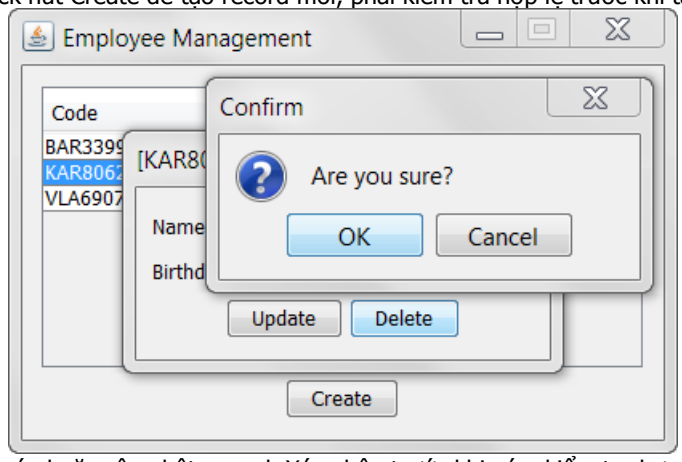
Viết ứng dụng nhỏ sau:



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)	☐
name	varchar(30)	☐
birthday	datetime	☐

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.

Tài liệu tham khảo

(Theo năm xuất bản)

[1] Kathy Sierra, Bert Bates – **OCA OCP Java SE 7 Programmer I & II Study Guide** – McGraw-Hill Education, 2015. ISBN: 978-0-07-177199-3

[2] Robert Liguori, Patricia Liguori – **Java 8 Pocket Guide** – O'Reilly Media, Inc., 2014. ISBN: 978-1-491-90086-4

[3] Paul Deitel, Harvey Deitel – **Java How To Program, Early Objects, 10th Edition** – Prentice Hall, 2014. ISBN: 978-0-1338078-0-6

[4] Jan Graba – **An Introduction to Network Programming with Java, Thrid Edition** – Springer, 2013. ISBN: 978-1-4471-5254-5

[5] Khalid A. Mughal, Rolf W. Rasmussen – **A Programmer's Guide to Java SCJP Certification A Comprehensive Primer, Third Edition** – Pearson Education, Inc., 2009. ISBN: 978-0-321-55605-9

[6] Jacquie Barker – **Beginning Java Objects: From Concepts to Code, Second Edition** – Apress, 2005. ISBN: 1-59059-457-6