

Chương I

Ngôn ngữ và môi trường lập trình

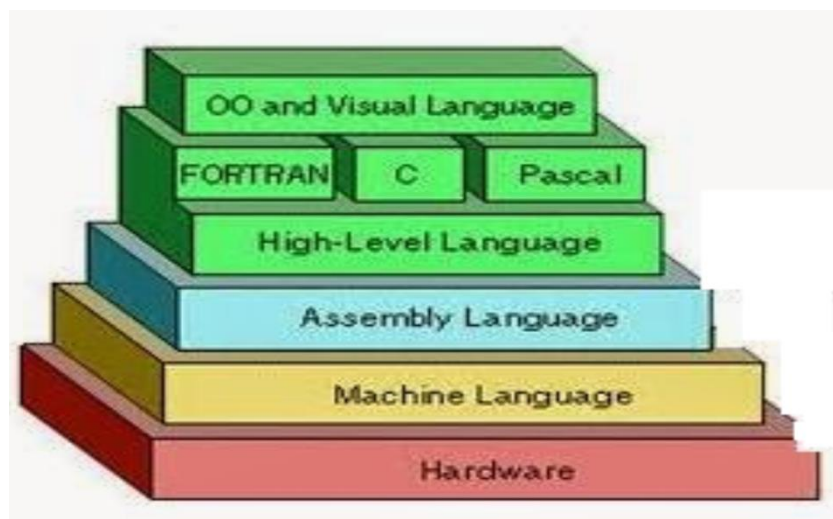
Chương trình và cách máy tính chạy chương trình

Một chương trình là một tập hợp các dòng lệnh nhằm thực hiện một nhiệm vụ nào đó.

Các dòng lệnh có thể được viết bằng ngôn ngữ lập trình như Assembly, C, Visual Basic, Java,...

Ngôn ngữ của máy tính còn gọi là ngôn ngữ máy (machine language) chỉ là tập hợp các chuỗi cấu tạo bởi hai bit 0 và 1. Ví dụ: 000111100.

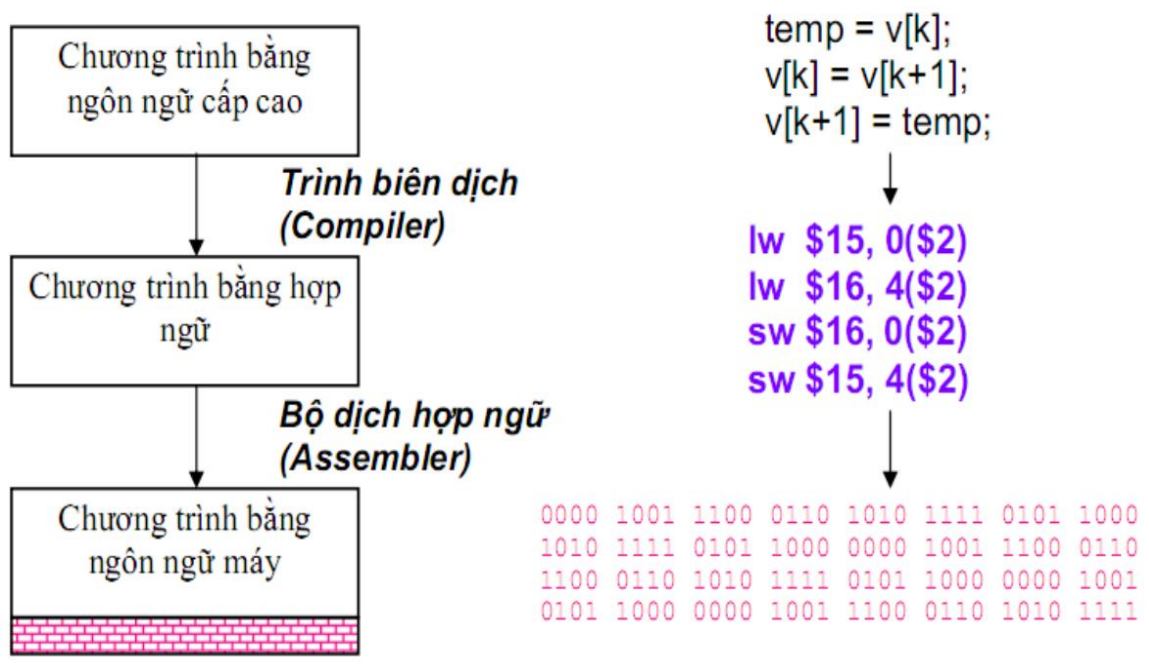
Ngôn ngữ máy khó sử dụng với người lập trình; vì vậy, các ngôn ngữ lập trình ra đời giúp người lập trình dễ dàng hơn. Sự tiến hoá của ngôn ngữ lập trình có thể hình dung qua hình sau:



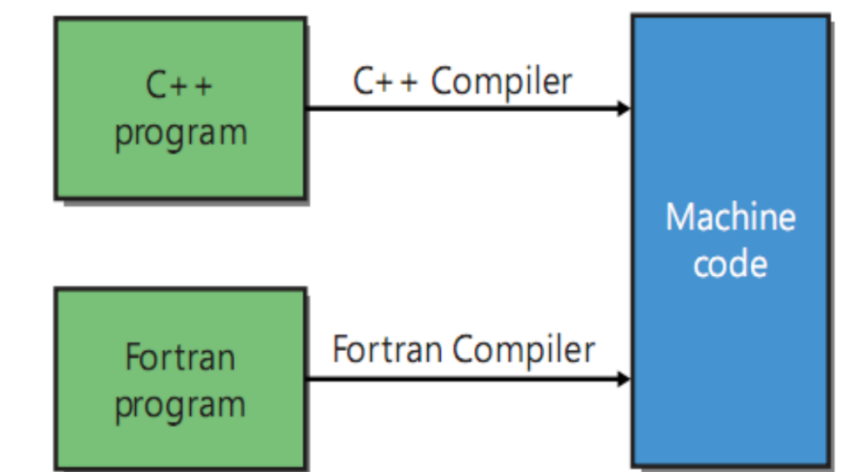
Nhìn vào hình trên chúng ta có thể phân chia ngôn ngữ lập trình thành các ngôn ngữ cấp cao như C, Pascal, C#,... và các ngôn ngữ cấp thấp như ngôn ngữ Assembly. Cao hay thấp ở đây là tham chiếu đến ngôn ngữ máy. Ngôn ngữ cấp càng cao thì càng dễ sử dụng bởi người lập trình nhưng chương trình được viết bằng các ngôn ngữ này sẽ mất nhiều thời gian để chuyển sang ngôn ngữ máy.

Chương trình được chuyển thành ngôn ngữ máy

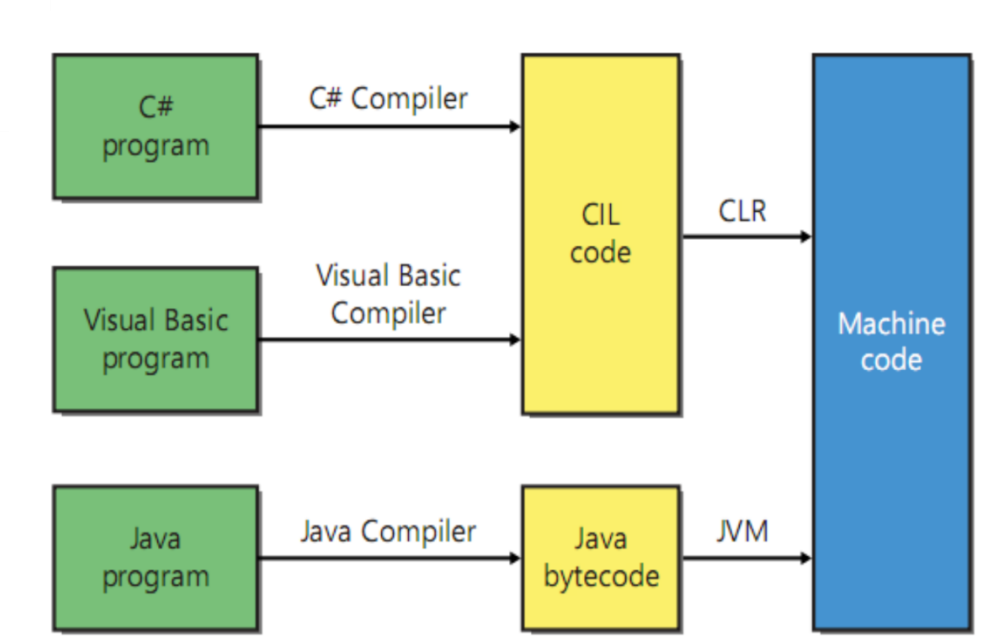
Chương trình viết bằng một ngôn ngữ lập trình khác ngôn ngữ máy. Do đó, để máy tính hiểu được, cần một trình biên dịch (compiler) chuyển sang ngôn ngữ máy. Xem ví dụ sau:



Tùy theo ngôn ngữ lập trình mà cơ chế chuyển sang ngôn ngữ máy sẽ khác nhau. Ví dụ chương trình viết bằng ngôn ngữ C++ và Fortran sẽ được chuyển trực tiếp sang ngôn ngữ máy bởi trình biên dịch:



Với những ngôn ngữ cấp cao hơn như C#, Visual Basic hay Java, cơ chế chuyển sang ngôn ngữ máy sẽ phức tạp hơn như sau:



Các đoạn mã từ chương trình viết bằng C#, Visual Basic hay Java sẽ được trình biên dịch chuyển sang các đoạn mã trung gian – C#, Visual Basic gọi là CIL (Common Intermediate Language), Java gọi là bytecode. Các đoạn mã trung gian này sẽ được một chương trình – các ngôn ngữ .NET như C# hay Visual Basic gọi là CLR (Common Language Runtime), Java gọi là JVM (Java Virtual Machine).

Ưu điểm của việc chuyển sang các đoạn mã trung gian là tính **khả chuyển** (portable). Các chương trình viết bằng C# hay Java sẽ được biên dịch chỉ một lần từ một máy A và có thể thực thi trên các máy B, C, ... miễn là các máy này có cài các chương trình như CLR hay JVM.

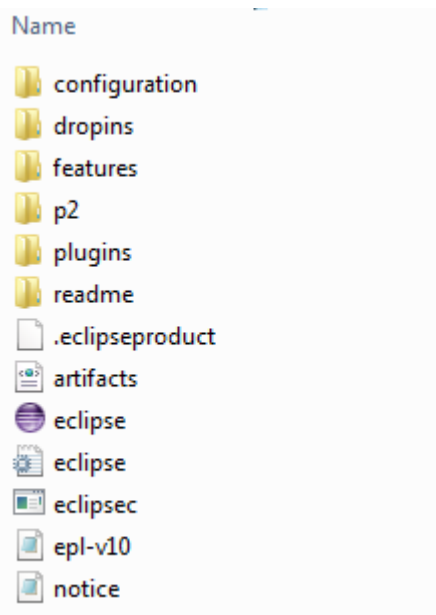
Một số đặc trưng của môi trường phát triển ứng dụng (IDE)

- Thiết kế chương trình dễ dàng
- Viết mã lệnh dễ dàng
- Kiểm thử chương trình dễ dàng
- Sửa lỗi dễ dàng

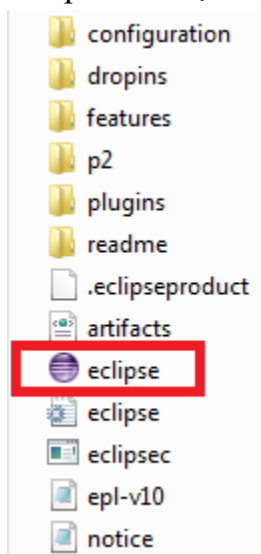
Cài đặt môi trường lập trình Java

- Tải Eclipse tại <https://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/keplersr1>, chọn bản Windows 32 bit hay 64 bit
- Tải Java tại <https://java.com/en/download/>
- Các bước cài đặt:
 - Cài đặt Java bằng cách nhấp đôi chuột vào tập tin exe vừa tải về

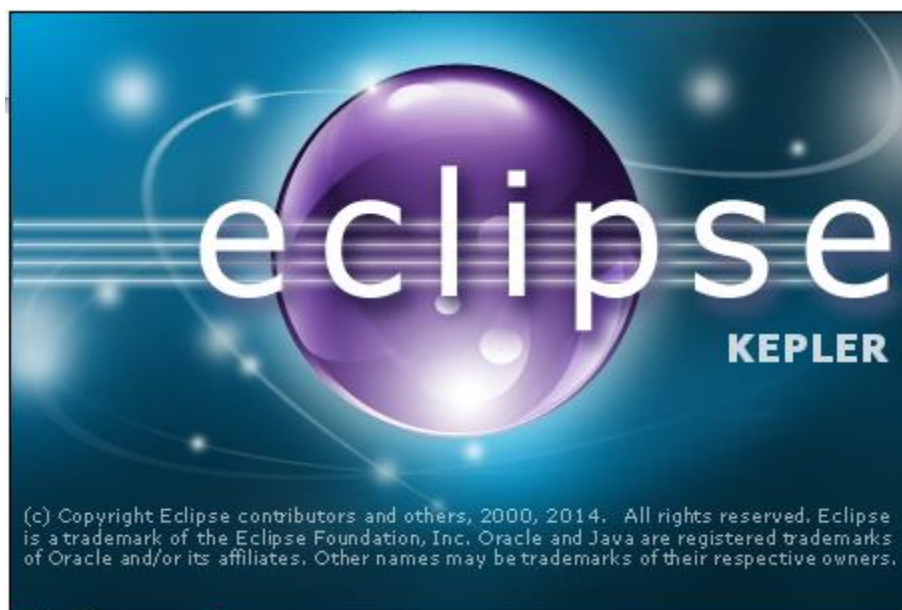
- Giải nén tập tin eclipse vừa tải sẽ được một thư mục **eclipse** chứa nội dung như sau:



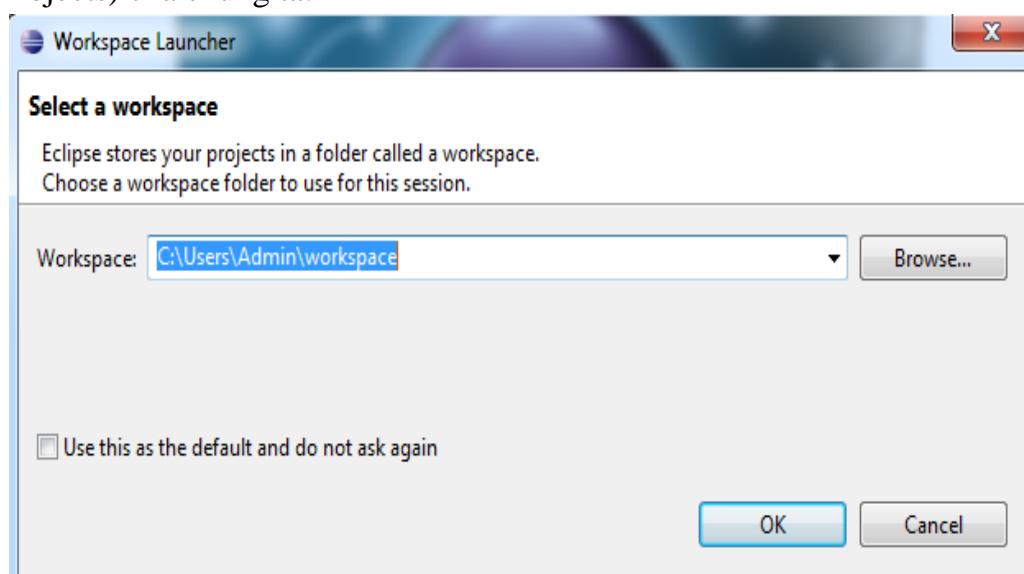
Nhấp đôi chuột vào tập tin **eclipse**:



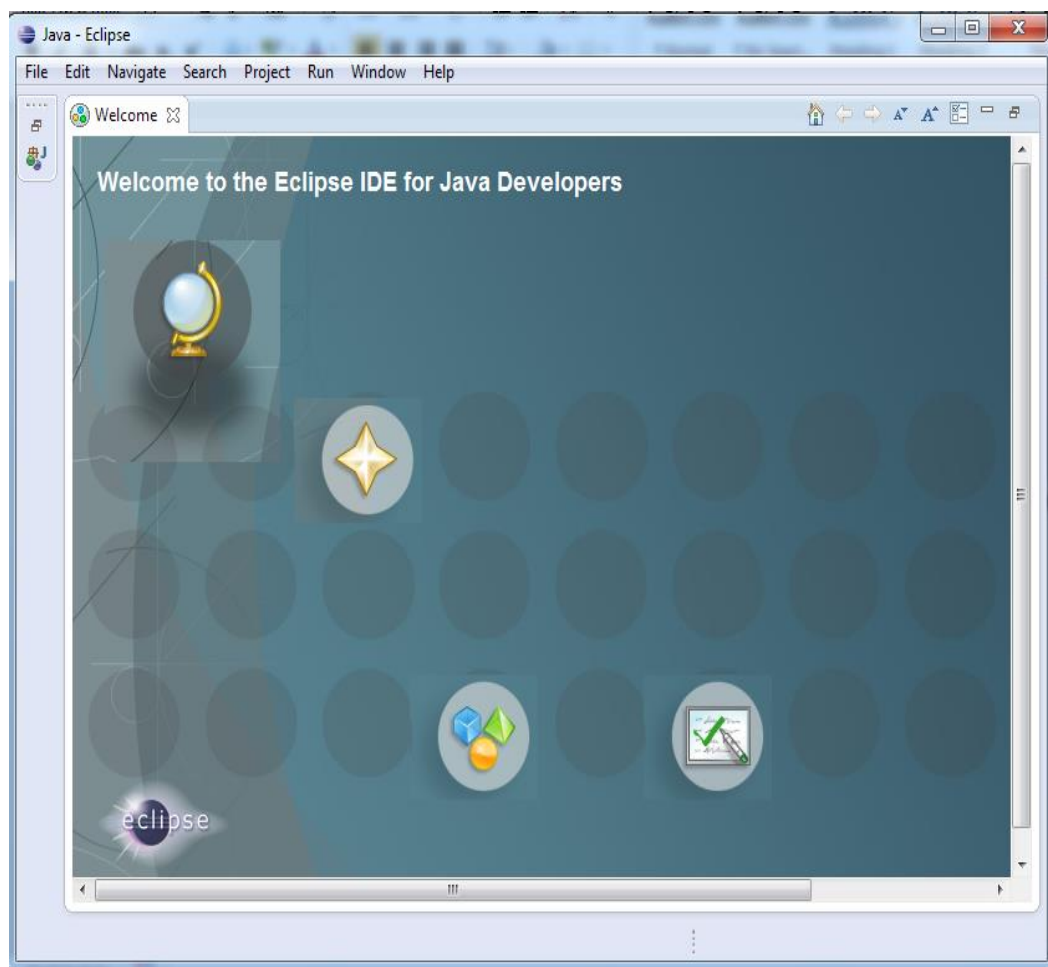
Sẽ xuất hiện:



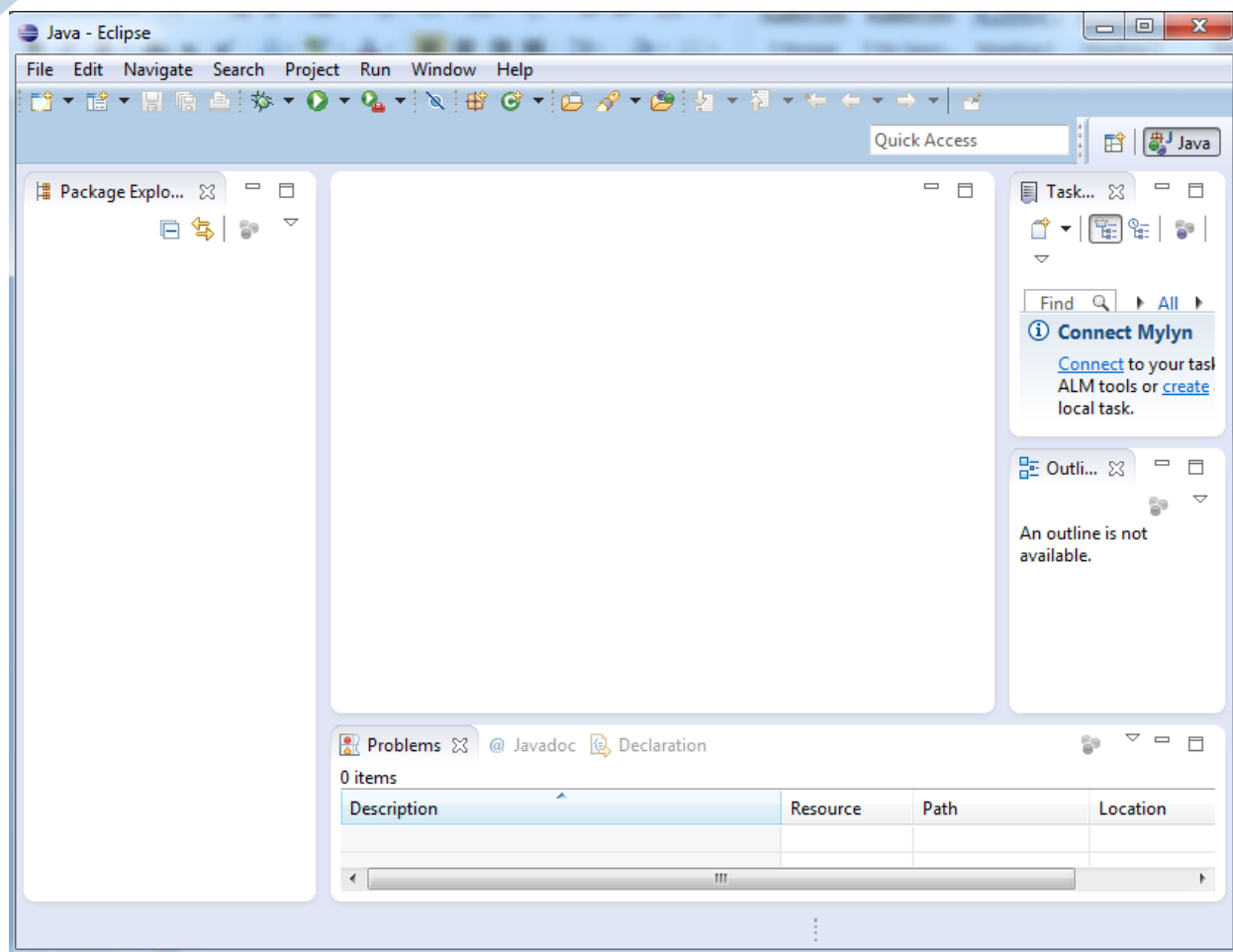
Và hộp thoại yêu cầu chọn Workspace, là vị trí chứa các dự án Java (Java Projects) của chúng ta:



Chọn **Workspace** bằng cách nhấn nút **Browse...**, chọn mục **Use this as the default and do not ask again** và nhấn **OK** sẽ xuất hiện môi trường làm việc Java như sau:

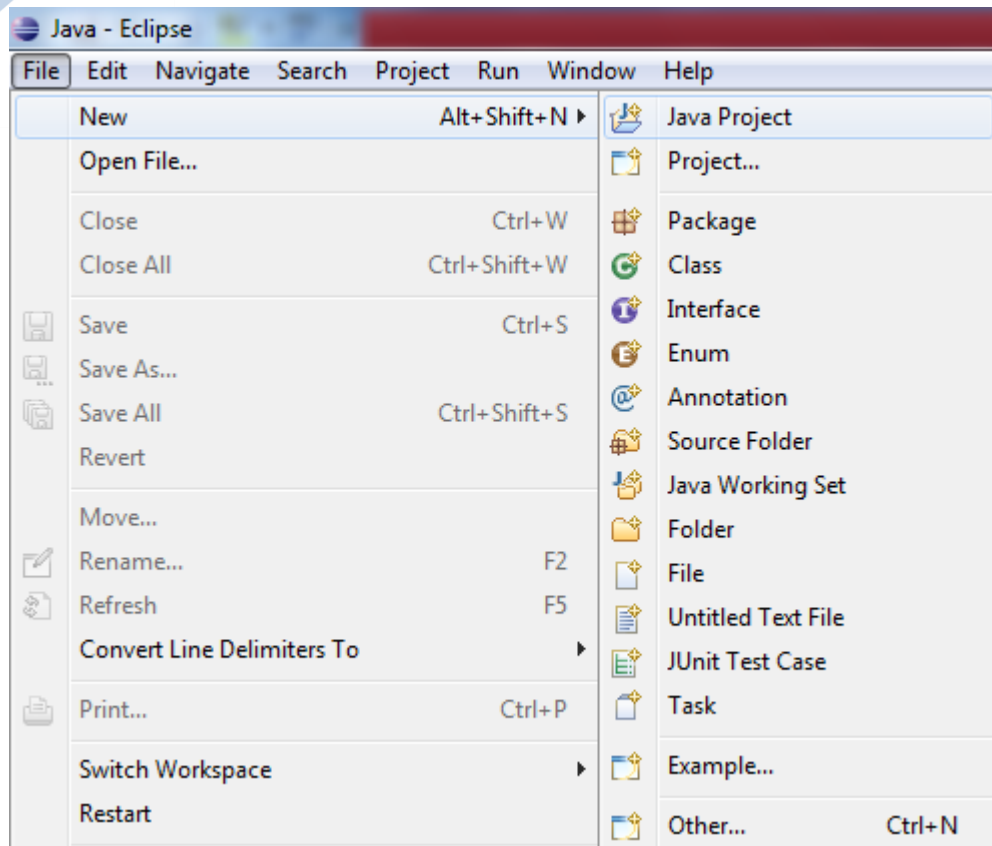


Đóng tab **Welcome** giao diện sau sẽ xuất hiện:

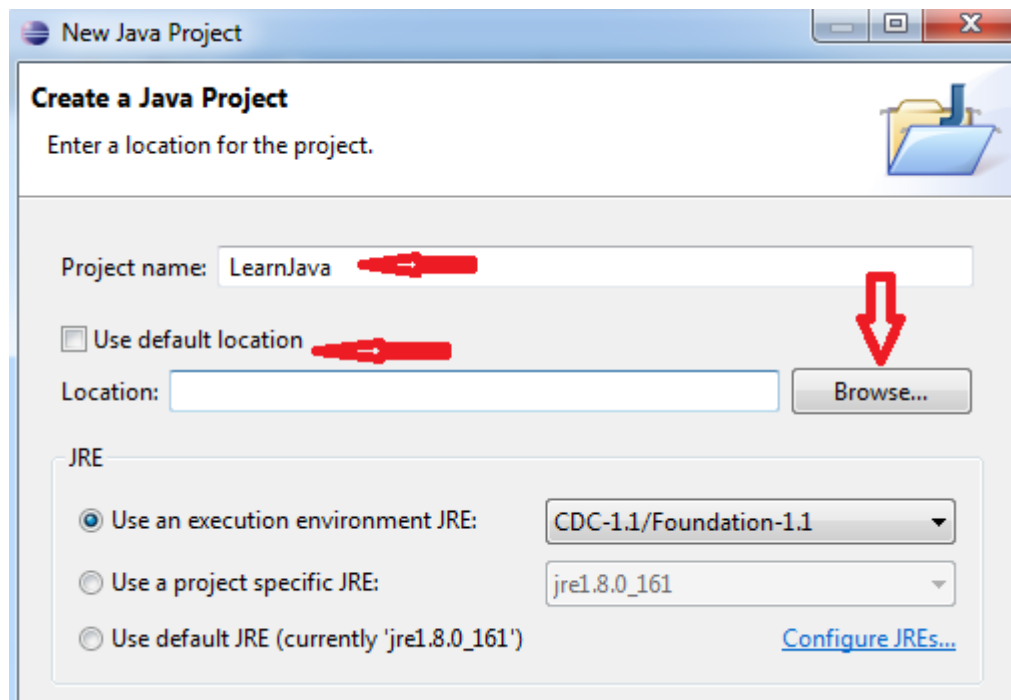


Tạo chương trình Java đầu tiên

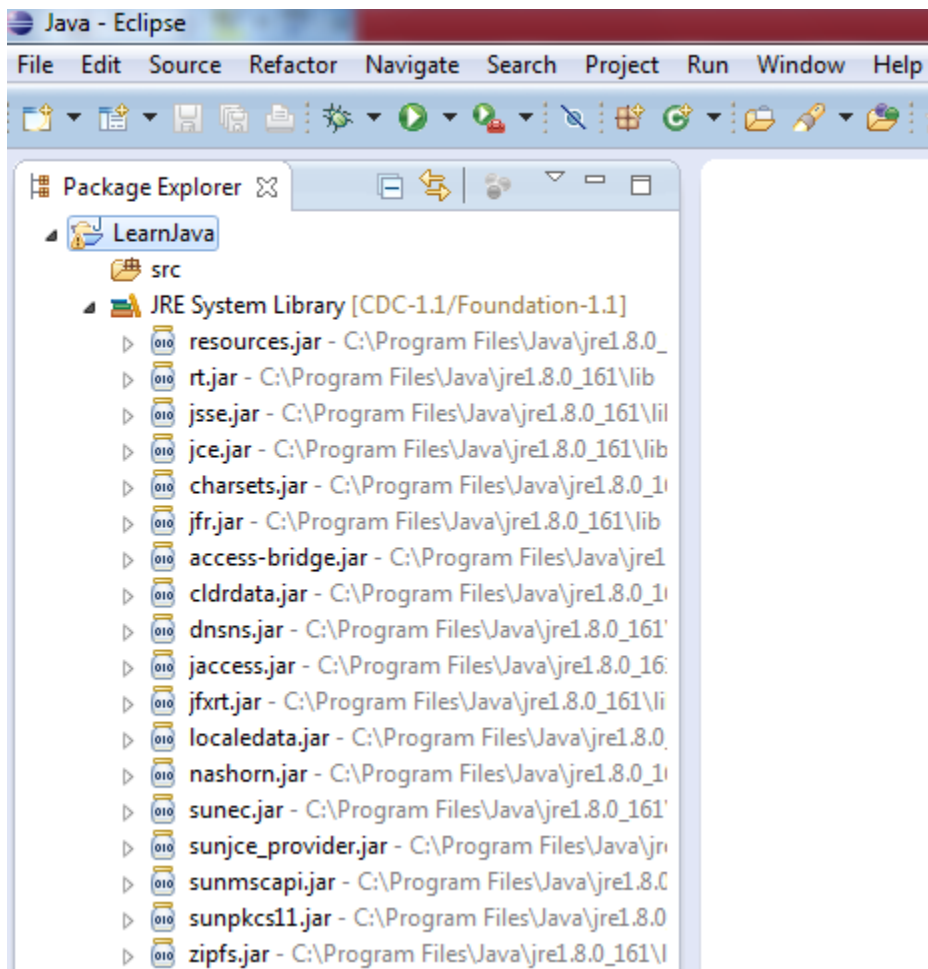
Bước đầu tiên để tạo chương trình **Java** là tạo một dự án **Java** trong **Workspace**. Trong **Workspace** có thể chứa rất nhiều dự án **Java** khác nhau. Để tạo dự án **Java** chúng ta vào **File > New > Java Project**



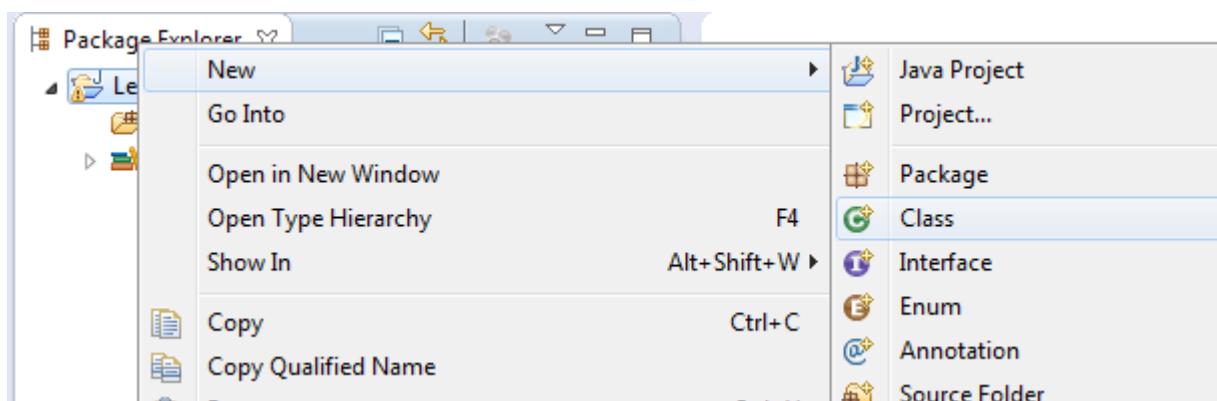
Nhập tên dự án trong **Project name** và có thể thay đổi vị trí lưu trữ dự án bằng cách bỏ dấu chọn tại **Use default location** và nhấn nút **Browse**:



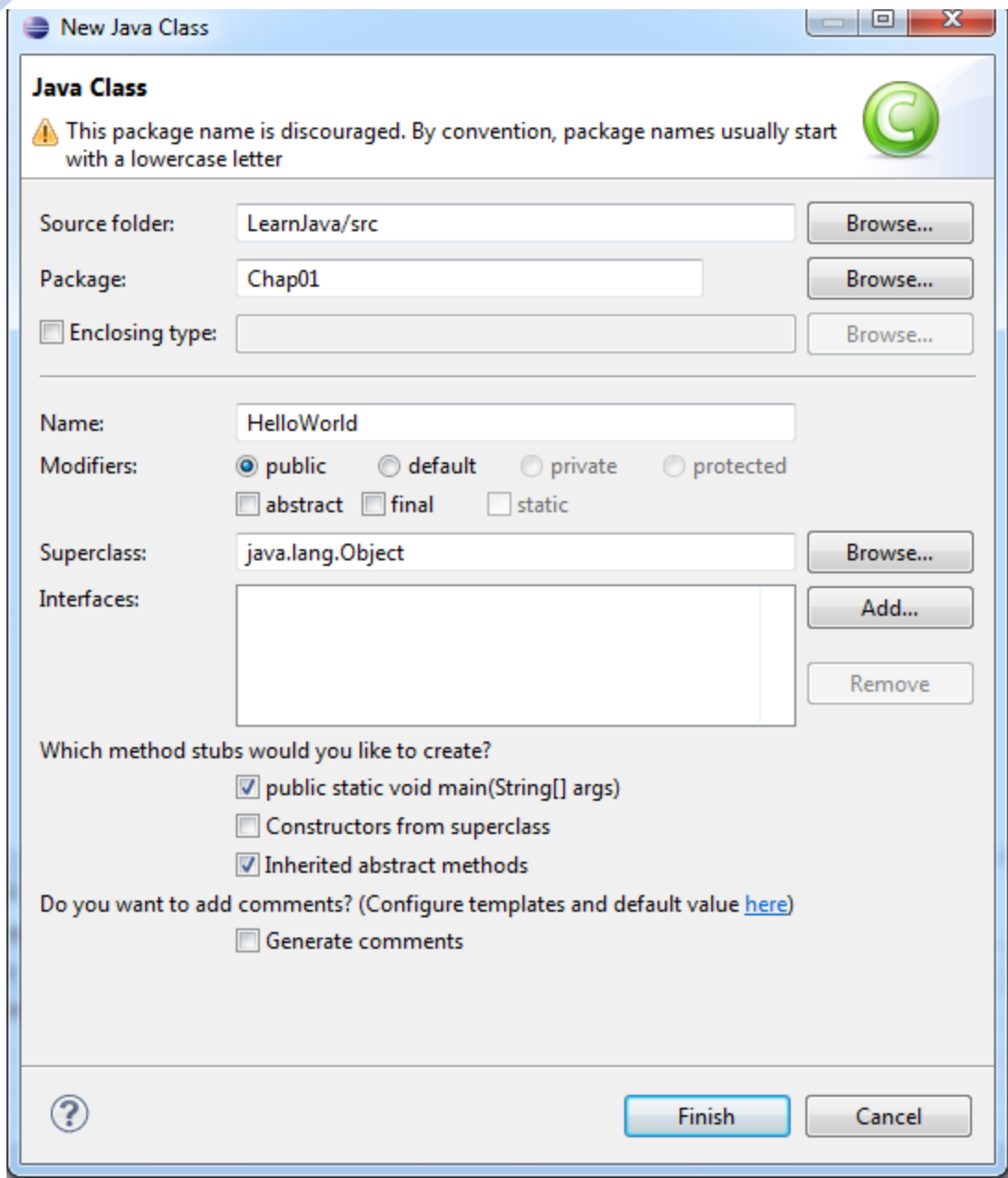
Vị trí mặc định là **Workspace** chúng ta đã chọn trong hộp thoại **Select a workspace**. Giữ nguyên các tùy chọn khác và nhấn **Finish**. Lúc này sẽ xuất hiện dự án (**LearnJava**) trong cửa sổ **Package Explorer** (bên trái) và một số thư viện sẵn có như sau:



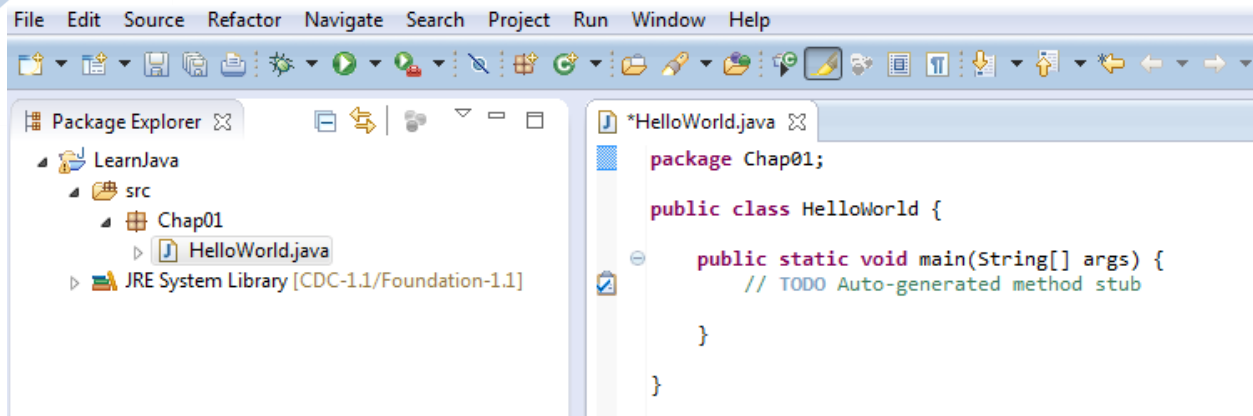
Kế tiếp nhấp chuột phải vào **LearnJava > New > Class**



Nhập các thông tin vào hộp thoại **New Java Class**



Source folder là thư mục chứa các tài nguyên của dự án, mặc định là thư mục **src**; **Package** là gói Java và một dự án sẽ chứa nhiều **Package** có sẵn và do chúng ta tạo ra như **Chap01**; **Name** là tên của lớp (class) và một **Package** có thể chứa nhiều lớp; **Modifier** là phạm vi của lớp; chọn **public static void main(String[] args)** để phát sinh phương thức **main** một cách tự động. Các tùy chọn khác giữ mặc định và nhấn **Finish**. Tập tin **HelloWorld.java** và lớp **HelloWorld** sẽ xuất hiện như sau:



Tên của tập tin **.java** sẽ trùng tên với tên lớp mà chúng ta tạo ra (ở đây là **HelloWorld**). Chi tiết về đoạn mã Java trong tập tin **HelloWorld.java** sẽ được thảo luận trong chương kế tiếp. Thêm dòng lệnh sau vào phương thức **main**:

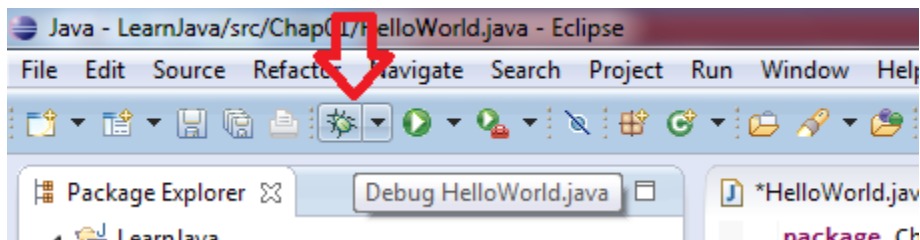
```
package Chap01;

public class HelloWorld {

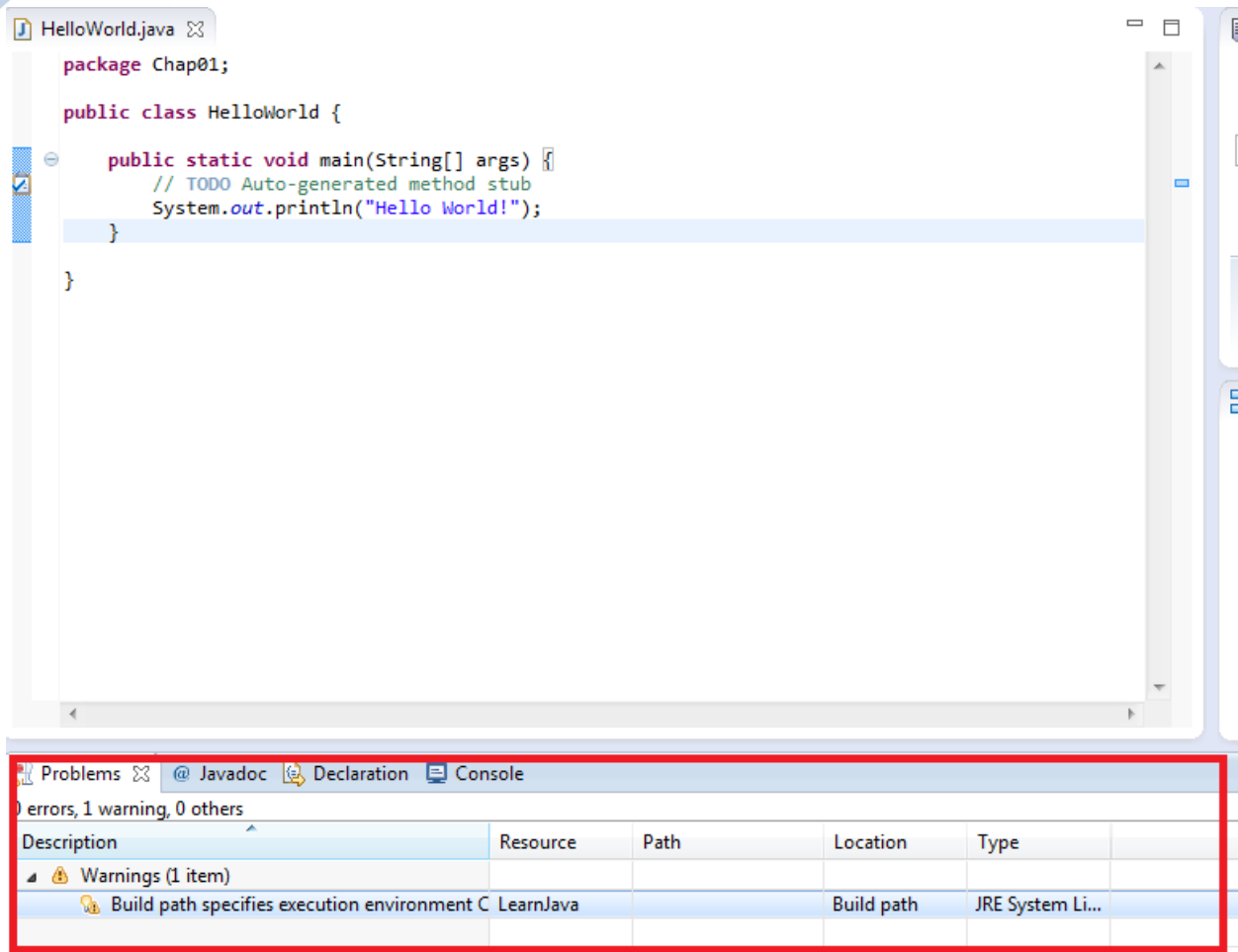
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Hello World!");
    }

}
```

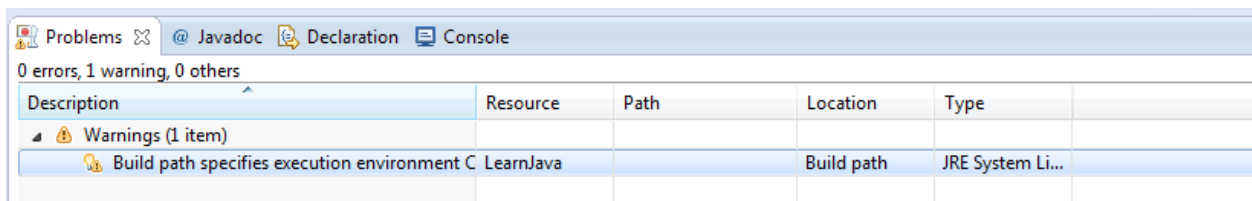
Kiểm tra chương trình bằng cách nhấn vào nút Debug (hay nhấn F11):



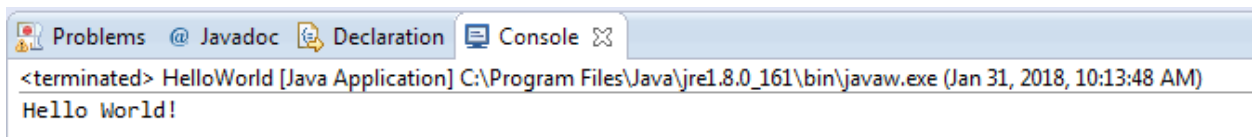
Hộp thoại **Save and Launch** xuất hiện. Nhấn **OK**. Thông tin về chương trình sẽ xuất hiện tại cửa sổ bên dưới:



Xem vấn đề của chương trình chúng ta có thể chọn tab **Problems**



Xem kết quả chọn **Console**

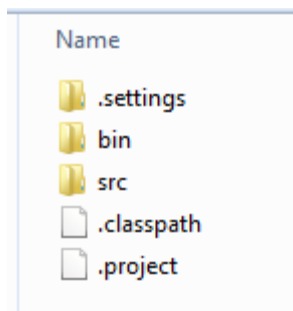


Chúng ta cũng có thể thực thi chương trình bằng nút **Run** (hay **Ctrl+F11**)



Run giống **Debug** nhưng khi chúng ta thực thi chương trình bằng **Debug** thì thông tin chúng ta nhận được về chương trình sẽ chi tiết, đầy đủ hơn **Run**. Điều này rất hữu ích khi chúng ta cần kiểm tra và sửa lỗi chương trình.

Khi chúng ta thực thi chương trình (bằng **Run** hay **Debug**), thư mục **Source Java** (Workspace) của chúng ta sẽ chứa thư mục **LearnJava** và trong **LearnJava** chứa nội dung sau:



Vào thư mục **src** chúng ta sẽ thấy thư mục **Chap01** và trong **Chap01** chứa tập tin **HelloWorld.java**.

Thư mục **setting** chứa các thông tin cấu hình của ứng dụng; thư mục **bin** chứa mã nhị phân của ứng dụng mà máy có thể hiểu được và các tập tin khác sinh ra từ quá trình biên dịch.

Chương II

Các thành phần cơ bản trong Java

Chương trình Java

Như đã đề cập ở chương I, một chương trình Java sẽ được chuyển sang mã trung gian bởi một chương trình gọi là trình biên dịch (compiler) trước khi được một chương trình khác gọi là **JVM (Java Virtual Machine)** chuyển sang mã máy.

Chương trình Java có một số đặc điểm:

- Chương trình Java gồm các dòng mã thực thi một nhiệm vụ nào đó.
- Chương trình Java phải đảm bảo các thành phần cơ bản của ngôn ngữ để chương trình có thể thực thi.
- Các thành phần cơ bản của Java sẽ được tìm hiểu trong các phần sau của bài giảng, nhưng để một chương trình Java thực thi cần đảm bảo một số thành phần cơ bản như trong ví dụ **HelloWorld** sau:

```
package Chap01;

public class HelloWorld {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Hello World!");
    }

}
```

Như đã đề cập qua trong chương I, một dự án Java sẽ chứa nhiều gói (package), mỗi gói sẽ chứa nhiều lớp và trong lớp sẽ chứa nhiều thành phần như phương thức, thuộc tính, dữ liệu, v.v.

Đoạn mã của tập tin **HelloWorld.java** gồm:

- Dòng lệnh đầu tiên khai báo gói **Chap01** bằng từ khoá **package**
- Dòng lệnh kế tiếp là khai báo lớp **HelloWorld** bằng từ khoá **class**. Từ khoá **public** chỉ phạm vi truy cập của lớp **HelloWorld** (sẽ tìm hiểu trong chương sau).
- Dòng lệnh khai báo phương thức **main** – là phương thức quan trọng và là đầu ra của chương trình.
- Dòng chú thích
- Dòng lệnh hiển thị chuỗi Hello World! ra màn hình bằng phương thức **println**.

Một số lưu ý:

- Dự án của chúng ta gồm một gói **Chap01**, gói **Chap01** chứa lớp **HelloWorld** và lớp **HelloWorld** chứa phương thức **main**. Phương thức **main** chứa một lệnh dùng để hiển thị chuỗi **Hello World!** ra màn hình.
- Chúng ta cùng xem lại lệnh gọi phương thức **println**:
`System.out.println("Hello World!");`
println là phương thức chứa trong lớp **System.out** (giống như lớp **HelloWorld** chứa **main**). Lớp **System.out** chứa trong gói **java.lang**.
- Java là ngôn ngữ case-sensitive tức là phân biệt hoa thường. Do đó, **Main** và **main** là khác nhau. Cần dùng chính xác các lệnh và từ khoá.

Các thành phần cơ bản**Chú thích (comment)**

Là những dòng văn bản do người lập trình viết ra để chú giải cho những gì mình viết trong chương trình.

Những dòng chú thích không được biên dịch trong quá trình thực thi chương trình.

Có thể phục vụ trong việc sửa lỗi (debug) chương trình.

Một dòng chú thích bắt đầu bằng dấu `//`; nhiều dòng bắt đầu bằng `/*` và kết thúc bằng `*/`.

Xem các ví dụ sau:

```
// Lệnh hiển thị dòng Hello World! ra màn hình
System.out.println("Hello World!");
```

Dòng đầu tiên là chú thích của người lập trình. Chú thích cũng có thể được viết thành nhiều dòng:

```
// Lệnh hiển thị
// dòng Hello World!
// ra màn hình
System.out.println("Hello World!");
```

Tương đương:

```
/* Lệnh hiển thị
   dòng Hello World!
   ra màn hình */
System.out.println("Hello World!");
```

Khoảng trắng (white space)

Khoảng trắng bao gồm việc xuống dòng (phím Enter), tab (phím Tab), hay khoảng trắng (phím Spacebar). Mặc định, trình biên dịch Java sẽ bỏ qua các khoảng trắng nên hai lệnh sau là tương đương:

```
System.out.println("Hello World!");
```

Và

```
System .
    .out
    .println("Hello World!");
```

Tuy nhiên có một số ngoại lệ, cụ thể là kiểu String, trình biên dịch sẽ hiểu khoảng trắng là kí tự hay chuỗi. Do đó **main** và **ma in** là khác nhau.

Từ khoá (keywords)

Là các từ được hỗ trợ bởi Java và người lập trình không thể thay đổi hay định nghĩa lại. Các từ khoá trong Java:

```
abstract else interface switch assert enum long synchronized boolean
extends native this break false new throw byte final null throws case
finally package transient catch float private true char for protected
try class go to public void const if return volatile continue
implements short while default import static do instanceof strictfp
double int super
```

Các định danh (identifiers)

Là tên biến, tên hằng hay tên các phương thức xuất hiện trong chương trình.

Định danh có thể được định nghĩa bởi Java như **main** hay **println** và có thể được định nghĩa bởi người sử dụng.

Định danh cấu tạo bởi các kí tự, kí số, dấu _, dấu \$. Không được bắt đầu bằng kí số.

Định danh có thể có chiều dài bất kỳ trong Java.

Một số định danh hợp lệ:

```
first
conversion
payRate
$Amount
```

Một số định danh không hợp lệ

```
First Name (định danh không có khoảng trắng)
Hello! (không dùng dấu ! trong định danh)
One+two (+ không được dùng trong định danh)
2nd (định danh không được bắt đầu bằng số)
```

Kiểu dữ liệu sơ cấp (primitive data types)

Các ngôn ngữ lập trình thường cung cấp một danh sách các kiểu dữ liệu sơ cấp (primitive data types) khác nhau dùng để biểu diễn các kiểu thông tin khác nhau như số (numbers), thời gian (date, time), văn bản (text), v.v.

Có 3 kiểu dữ liệu sơ cấp trong Java là kiểu số nguyên, kiểu thực (dấu chấm động) và kiểu luận lý.

Kiểu số nguyên gồm các kiểu sau:

Kiểu	Kích cỡ (Byte)	Giá trị
<i>char</i>	2 (16 bits)	0 -> 65535
<i>byte</i>	1 (8 bits)	-128 -> +127
<i>short</i>	2 (16 bits)	-32768 -> +32767
<i>int</i>	4 (32 bits)	-2147483648 -> +2147483647
<i>long</i>	8 (64 bits)	-2^{63} -> $2^{63}-1$

Kiểu *char* hay kiểu ký tự cũng được liệt vào kiểu số nguyên vì các ký tự hay ký hiệu sẽ tương ứng với một giá trị số nguyên trong bảng mã Unicode.

Kiểu số thực gồm:

Kiểu	Kích cỡ (Byte)	Giá trị
<i>float</i>	4 (32 bits)	-3.4×10^{38} -> $+3.4 \times 10^{38}$
<i>double</i>	8 (64 bits)	-2.2×10^{308} -> $+2.2 \times 10^{308}$

Kiểu luận lý gồm:

Kiểu	Kích cỡ (Byte)	Giá trị
<i>boolean</i>	1 (8 bits)	false, true

Biến (variable)

Để làm việc với các dữ liệu, chúng ta cần lưu trữ nó trong một biến. Cú pháp khai báo một biến là:

```
Kiểu_Dữ_Liệu tên_biến;
```

tên_biến là tên của biến cần khai báo gồm các ký tự, số, dấu `_`, và không được bắt đầu bằng số. **Kiểu_Dữ_Liệu** là một trong các kiểu dữ liệu sơ cấp hay kiểu do người dùng định nghĩa.

Ví dụ khai báo một biến kiểu `int`, tên biến là `x` như sau:

```
int x;
```

Sau khi khai báo biến, chúng ta có thể gán giá trị cho chúng, ví dụ gán giá trị 5 đến biến `x`:

```
int x;  
x = 5;
```

Chúng ta cũng có thể kết hợp khai báo và gán giá trị cho biến như ví dụ:

```
int x = 5;
```

Một số chú ý khi đặt tên biến:

- Không bắt đầu bằng số
- Không trùng với các từ khoá (keywords), là các từ vựng có sẵn trong ngôn ngữ.
- Đặt theo mục đích, ví dụ **firstName**, **age**.
- Tuân theo chuẩn **camel case**, tức là nếu tên biến gồm nhiều từ như **firstName** (gồm **first** và **Name**) thì từ đầu tiên viết thường tất cả các ký tự, các từ kế tiếp sẽ viết hoa ký tự đầu tiên, như **firstName**.

Hằng (constant)

Hằng là vùng nhớ chứa nội dung không thay đổi (khác với biến chứa nội dung có thể thay đổi) trong quá trình chương trình thực thi.

Khai báo hằng trong Java:

```
[static] final Kiểu_dữ_liệu Tên_hằng = giá_trị_khởi_tạo;
```

Từ khoá **static** có thể dùng trong một số trường hợp đặc biệt, **final** bắt buộc; **Kiểu_dữ_liệu** là kiểu dữ liệu của hằng; **Tên_hằng** là tên của hằng; **giá_trị_khởi_tạo** là các giá trị tương ứng với kiểu dữ liệu **Kiểu_dữ_liệu** (hay còn gọi là **literal**).

Các ví dụ:

```
final double PI = 3.14;  
final int NO_OF_STUDENTS = 20;
```

Kiểu chuỗi (string)

Kiểu chuỗi (string) không phải là kiểu dữ liệu sơ cấp nhưng thường được sử dụng phổ biến với các kiểu dữ liệu sơ cấp khác.

Một chuỗi là một dãy các ký tự và có thể được dùng để thể hiện các văn bản (text), số (numbers), các ký hiệu Unicode. Chuỗi được chứa trong cặp nháy kép.

Trong Java, một biến kiểu chuỗi sẽ được khai báo kiểu **String**. Ví dụ biến *str*

```
String str = "This is a string variable!";
```

Các chuỗi có thể kết hợp với nhau bằng toán tử +, ví dụ:

```
String str = "Hello, " + "Minh!" ;// Hello, Minh!
```

Một chuỗi cũng có thể được định nghĩa trên nhiều dòng như ví dụ sau:

```
String description = "Strings can " +  
    "be defined on multiple " +  
    "lines with the + operator!";
```

Không thể gán trực tiếp một giá trị kiểu **String** vào một biến có kiểu dữ liệu sơ cấp dạng số như **int**, **float**. Ví dụ sau sẽ gây ra lỗi:

```
String j = "7";  
int i = 10 + j;
```

Giải pháp cho vấn đề này là dùng các phương thức chuyển kiểu chuỗi sang kiểu dữ liệu sơ cấp dạng số tương ứng như *Integer.parseInt* hay *Float.parseFloat*. Ví dụ trên được viết lại:

```
String j = "7";
int i = 10 + Integer.parseInt(j); // 17
```

Một số phương thức phổ biến:

Kiểu	Phương thức
<i>boolean</i>	<i>Boolean.parseBoolean</i>
<i>byte</i>	<i>Byte.parseByte</i>
<i>short</i>	<i>Short.parseShort</i>
<i>int</i>	<i>Int.parseInt</i>
<i>long</i>	<i>Long.parseLong</i>
<i>float</i>	<i>Float.parseFloat</i>
<i>double</i>	<i>Double.parseDouble</i>

Khi một giá trị kiểu số (**int**, **float**, v.v.) kết hợp với một giá trị kiểu **String** bằng toán tử +, kết quả trả về luôn là giá trị kiểu **String**. Ví dụ:

```
String greeting = "My age is " + 19;
System.out.println(greeting); // My age is 19
```

Ép kiểu (casting)

Ép kiểu trong Java là thay đổi giá trị của một biến hay một giá trị (literals) sang kiểu dữ liệu sơ cấp khác. Để ép kiểu, chúng ta chỉ cần đặt kiểu cần chuyển đến trong cặp dấu ngoặc đặt ngay trước giá trị cần ép kiểu. Các ví dụ sau sẽ giúp chúng ta dễ hình dung hơn:

```
int i = (int) 7.8; // chuyển giá trị 7.8 kiểu double sang kiểu integer
double d = (double) 89.7f; // chuyển giá trị 89.7f kiểu float sang kiểu double
short q = (short) (i*d); // chuyển giá trị i*d kiểu double sang kiểu short
System.out.println(i);
System.out.println(d);
System.out.println(q);
```

Kết quả:

```
7
89.69999694824219
7
```

Mỗi một kiểu dữ liệu sơ cấp như **int**, **float**, **double**, v.v. sẽ có một kiểu lớp tương ứng như **Integer**, **Float**, **Double**, v.v. quá trình chuyển kiểu từ kiểu sơ cấp sang kiểu lớp tương ứng, như chuyển **int** sang **Integer**, gọi là **boxing**. Quá trình ngược lại, như từ **Integer** sang **int**, gọi là **unboxing**. Xem thêm tại

<https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html>

Literal

Một literal là một giá trị chúng ta viết trong mã Java như 27, A, 5.85f, v.v. Chúng ta thường gán literal đến biến hay kết hợp các literals lại với nhau bằng toán tử và lưu kết quả nhận được trong biến. Các literals phổ biến có thể được liệt kê như bảng dưới đây:

Literal	Mô tả	Ví dụ
<i>Integer literal</i>	Là các số nguyên có dấu hay không dấu	3, -145
<i>Floating-point literal</i>	Là các số thực kiểu double hay kiểu float (có chữ f ở cuối)	3.14, 5.7f
<i>Boolean literal</i>	Chỉ có hai giá trị là true hay false	true, false
<i>Character literal</i>	Là các kí tự được đặt trong dấu nháy đơn	'h', 'a'
<i>String literal</i>	Là các chuỗi được đặt trong dấu nháy kép	"This is a string literal"

Chương III

Nhập/xuất dữ liệu

Làm việc với thiết bị nhập/xuất chuẩn (I/O devices)

Xuất dữ liệu

Một chương trình sẽ thực hiện 3 thao tác cơ bản: nhận dữ liệu, xử lý dữ liệu và xuất kết quả. Trong các chương trình Java đầu tiên, chúng ta đã làm quen với đối tượng xuất dữ liệu đến thiết bị chuẩn (cụ thể là màn hình) là **System.out** và các phương thức xuất dữ liệu cơ bản là **print** và **println**. Tuy nhiên, **print** và **println** có một số hạn chế trong vấn đề định dạng dữ liệu như ví dụ sau:

```
double PI = (double) 3.14f;
System.out.println("PI = " + PI);
```

Kết quả:

```
PI = 3.140000104904175
```

Giả sử chúng ta muốn hiển thị giá trị biến $PI = 3.14$, tức là chỉ cần hai chữ số phần thập phân thì **print** hay **println** sẽ không giải quyết được. May mắn thay, đối tượng **System.out** có hỗ trợ phương thức **printf** giúp chúng ta định dạng dữ liệu như ý muốn.

Phương thức **printf** có thể được sử dụng theo hai cách là:

```
System.out.printf(formatString);
```

Hay

```
System.out.printf(formatString, argumentList);
```

Ví dụ biến PI có thể viết lại như sau:

```
double PI = (double) 3.14f;
System.out.printf("PI = %.2f", PI);
```

Kết quả:

```
PI = 3.14
```

Học cách sử dụng **printf** tại

<https://docs.oracle.com/javase/tutorial/java/data/numberformat.html>

Bên cạnh **printf**, chúng ta cũng có thể dùng phương thức **format** của lớp **String**. Có thể học thêm về phương thức **format** tại <https://dzone.com/articles/java-string-format-examples>

Nhập dữ liệu

Để nhập dữ liệu từ thiết bị nhập chuẩn (như bàn phím,...), Java cung cấp lớp **Scanner**. Để dùng lớp này, đầu tiên chúng ta phải tạo một đối tượng nhập (ví dụ **console**) và kết nối nó với thiết bị nhập chuẩn như đoạn mã sau:

```
Scanner console = new Scanner(System.in);
```

Để sử dụng lớp *Scanner* chúng ta cần **import** lớp *Scanner* đến dự án:

```
import java.util.Scanner;
```

Sau khi tạo đối tượng từ lớp *Scanner*, chúng ta sẽ sử dụng các phương thức của lớp *Scanner* để nhận dữ liệu từ thiết bị nhập chuẩn. Một số phương thức phổ biến:

Phương thức	Mô tả	Ví dụ
<code>nextInt()</code>	Nhận số nguyên	<code>int a = console.nextInt();</code>
<code>nextDouble()</code>	Nhận số thực	<code>double b = console.nextDouble();</code>
<code>next()</code>	Nhận một chuỗi ký tự	<code>String str = console.next();</code>
<code>nextLine()</code>	Nhận một chuỗi ký tự trên dòng kế tiếp	<code>String str = console.nextLine();</code>
<code>next().charAt(0)</code>	Nhận một ký tự	<code>char ch = console.next().charAt(0);</code>

Ngoài các phương thức trên, lớp *Scanner* còn cung cấp nhiều phương thức khác có thể xem chi tiết tại https://www.tutorialspoint.com/java/util/java_util_scanner.htm.

Chương trình sau đây sẽ minh họa cách tính tổng hai số nguyên được nhập từ bàn phím và hiển thị tổng đó ra màn hình:

```
Scanner console = new Scanner(System.in);
System.out.print("Enter a = ");
int a = console.nextInt();
System.out.print("Enter b = ");
int b = console.nextInt();
System.out.println();
int c = a + b;
```

Khi hoàn thành có thể đóng đối tượng *Scanner* để tránh rò rỉ bộ nhớ:

```
if(console != null)
    console.close();
```

Làm việc với các hộp thoại (Dialog boxes)

Một cách khác để nhập/xuất dữ liệu trong Java là sử dụng lớp *JOptionPane* cho phép thực hiện nhập/xuất dữ liệu qua giao diện đồ họa hay các hộp thoại chuẩn. Lớp này được chứa trong gói *javax.swing*. Hai phương thức được dùng phổ biến trong lớp *JOptionPane* là *showInputDialog* cho phép người dùng nhập dữ liệu từ bàn phím và *showMessageDialog* cho phép hiển thị kết quả.

Cú pháp của phương thức *showInputDialog*

```
str = JOptionPane.showInputDialog(stringExpression);
```

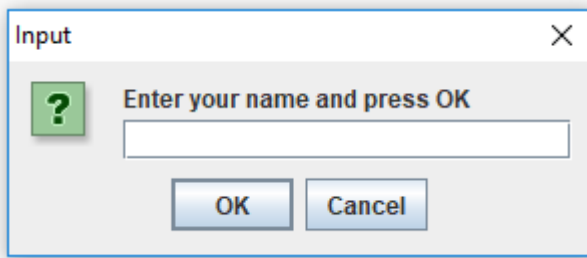
`stringExpression` là chuỗi thông điệp chúng ta muốn hiển thị trên hộp thoại, biến `str` kiểu `String` chứa giá trị được nhập từ bàn phím. Đoạn mã sau minh họa cách dùng phương thức *showInputDialog*:

```
String name = JOptionPane.showInputDialog("Enter your name and press OK");
System.out.printf("Hello %s", name);
```

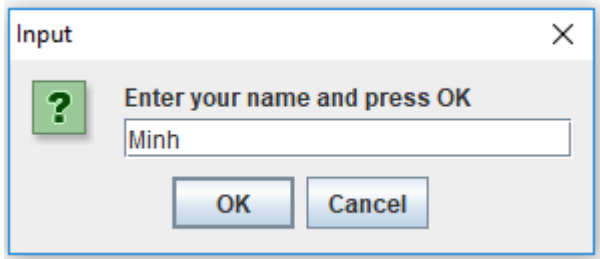
Lưu ý chúng ta cần import lớp *JOptionPane*:

```
import javax.swing.JOptionPane;
```

Kết quả:



Nếu nhập dữ liệu như sau:



Nhấp OK thì kết quả là:

Hello Minh

Cú pháp của phương thức `showMessageDialog`:

```
JOptionPane.showMessageDialog (parentComponent,
                                messageStringExpression,
                                boxTitleString, messageType);
```

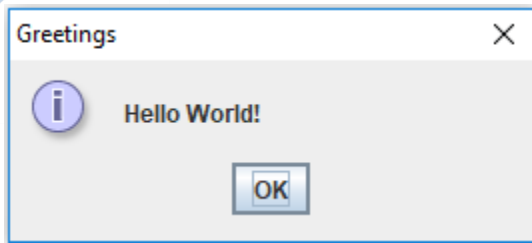
Các tham số của phương thức ***showMessageBox*** có thể được mô tả như sau:

Tham số	Mô tả
<i>parentComponent</i>	Đối tượng cha của đối tượng hộp thoại hiện tại. Mặc định là <i>null</i> sẽ xuất hiện hộp thoại giữa màn hình.
<i>messageStringExpression</i>	Là giá trị sẽ xuất hiện trong hộp thoại.
<i>boxTitleString</i>	Tiêu đề của hộp thoại
<i>messageType</i>	Sẽ nhận một trong các giá trị <code>JOptionPane.ERROR_MESSAGE</code> , <code>JOptionPane.INFORMATION_MESSAGE</code> , <code>JOptionPane.PLAIN_MESSAGE</code> , <code>JOptionPane.QUESTION_MESSAGE</code> , <code>JOptionPane.WARNING_MESSAGE</code> . Mỗi một giá trị tương ứng với một kiểu icon tương ứng.

Ví dụ 1:

```
JOptionPane.showMessageDialog(null, "Hello World!", "Greetings",
                                JOptionPane.INFORMATION_MESSAGE);
```

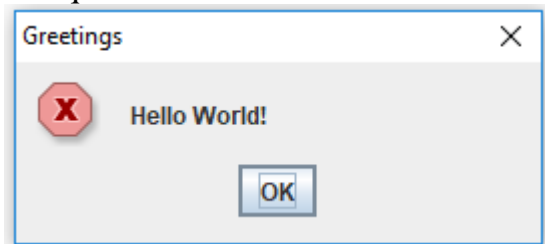
Kết quả



Ví dụ 2

```
JOptionPane.showMessageDialog(null, "Hello World!", "Greetings",  
JOptionPane.ERROR_MESSAGE);
```

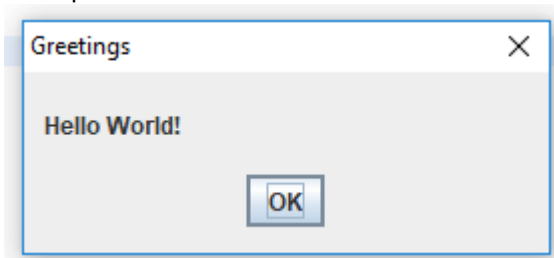
Kết quả:



Ví dụ 3:

```
JOptionPane.showMessageDialog(null, "Hello World!", "Greetings",  
JOptionPane.PLAIN_MESSAGE);
```

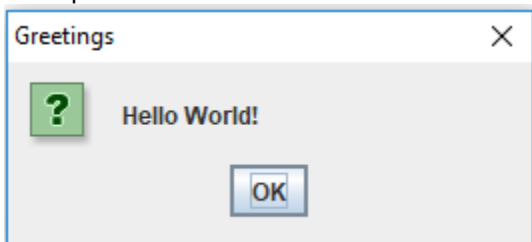
Kết quả



Ví dụ 4:

```
JOptionPane.showMessageDialog(null, "Hello World!", "Greetings",  
JOptionPane.QUESTION_MESSAGE);
```

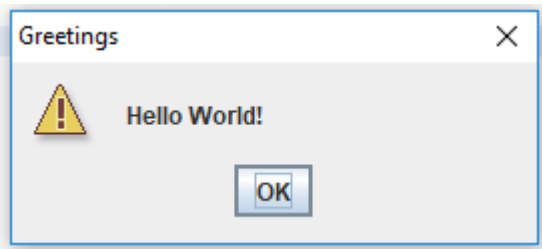
Kết quả



Ví dụ 5:

```
JOptionPane.showMessageDialog(null, "Hello World!", "Greetings",  
JOptionPane.WARNING_MESSAGE);
```

Kết quả



Làm việc với tập tin (file)

Chúng ta đã học cách nhập/xuất dữ liệu thông qua các thiết bị chuẩn như bàn phím/màn hình và hộp thoại. Nhưng những cách nhập/xuất này chỉ phù hợp cho số lượng dữ liệu nhỏ, và đối với các số liệu lớn, chúng ta sử dụng tập tin với hai lớp **FileReader** và **PrintWriter** trong gói **java.io**.

Đọc dữ liệu từ tập tin

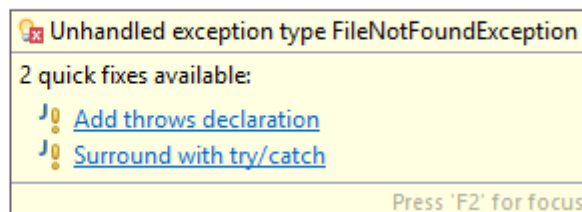
Để đọc dữ liệu từ tập tin, chúng ta dùng lớp **FileReader**. Giả sử chúng ta có tập tin **Data.txt** được lưu trữ tại **D:\LearnJava**. Để có thể đọc dữ liệu từ tập tin **Data.txt**, chúng ta thực hiện các bước sau:

- Import **java.io.*** và **java.util.***
- Tạo một đối tượng **Scanner** (giả sử tên là **inFile**) và khởi tạo nó đến tập tin **Data.txt** bằng đối tượng **FileReader** như sau:

```
Scanner inFile = new Scanner(new FileReader("D:\\LearnJava\\Data.txt"));
```

Lưu ý: có thể phát sinh lỗi như sau:

```
Scanner inFile = new Scanner(new FileReader("D:\\LearnJava\\Data.txt"));
```



Đây là thông báo yêu cầu chúng ta xử lý trong trường hợp tập tin không được tìm thấy. Chỉ cần kích chuột vào liên kết **Add throws declaration** và nếu chúng ta viết đoạn mã trên trong phương thức **main** thì lúc này phương thức **main** sẽ xuất hiện thêm lệnh **throws** như sau:

```
public static void main(String[] args) throws FileNotFoundException {
    // TODO Auto-generated method stub
    Scanner inFile = new Scanner(new FileReader("D:\\LearnJava\\Data.txt"));
}
```

- Đọc dữ liệu từ tập tin **Data.txt** qua đối tượng **inFile**. Lưu ý rằng, dữ liệu trong tập tin được phân biệt bởi khoảng trắng, ví dụ nội dung trong tập tin **Data.txt** là:
Hello Minh thì **Hello** và **Minh** phải được lưu trữ trong những biến khác nhau như đoạn mã sau:

```
String greeting = inFile.next();
String name = inFile.next();
System.out.print(greeting + " " + name);
```

- Đóng đối tượng đọc tập tin bằng phương thức *close()*:

```
inFile.close();
```

Đoạn mã hoàn chỉnh:

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

public class MyMainClass {

    public static void main(String[] args) throws FileNotFoundException {
        // TODO Auto-generated method stub

        Scanner inFile = new Scanner(new FileReader("D:\\LearnJava\\Data.txt"));
        String greeting = inFile.next();
        String name = inFile.next();
        System.out.print(greeting + " " + name); // Hello Minh
        inFile.close();

    }

}
```

Ghi dữ liệu đến tập tin

Để có thể lưu trữ dữ liệu đến tập tin, chúng ta dùng lớp *PrintWriter* theo hai bước sau:

- Tạo một đối tượng *PrintWriter* (ví dụ như *outFile*) và kết nối đến tập tin cần lưu trữ dữ liệu (ví dụ *Data.txt*) như đoạn mã sau:

```
PrintWriter outFile = new PrintWriter("D:\\\\LearnJava\\\\Data.txt");
```

- Sử dụng các phương thức như *print*, *println*, hay *printf* (giống *System.out*) của đối tượng *PrintWriter* để ghi dữ liệu đến tập tin như đoạn mã sau:

```
outFile.println("Welcome to Nha Trang!");
```

- Đóng đối tượng ghi tập tin bằng phương thức *close()*

```
outFile.close();
```

Ví dụ sau minh họa một chương trình lấy hai số thực được lưu trữ trong tập tin *Input.txt*, tính tổng của chúng và lưu trữ kết quả trong tập tin *Output.txt*. Hai tập tin *Input.txt* và *Output.txt* được chứa trong thư mục *D:\LearnJava*.

- Bước 1: vào thư mục *D:\LearnJava* và tạo hai tập tin *Input.txt* và *Output.txt*. Gõ hai số thực bất kỳ vào tập tin *Input.txt* ví dụ 3.5 6.0 (chú ý có khoảng trắng giữa 2 số), lưu và đóng tập tin *Input.txt*.
- Bước 2: viết đoạn mã Java như sau:

```
// reading data from Input.txt
Scanner inFile = new Scanner(new FileReader("D:\\LearnJava\\Input.txt"));
double val1 = inFile.nextDouble();
double val2 = inFile.nextDouble();
```

```
double result = val1 + val2;  
//storing result to Output.txt  
PrintWriter outFile = new PrintWriter("D:\\LearnJava\\Output.txt");  
outFile.printf(val1 + " + " + val2 + " = " + result);  
inFile.close();  
outFile.close();
```

Có thể xem chương trình hoàn chỉnh tại <https://github.com/TranNgocMinh/Learn-Java/blob/master/Chapter03/File.java>

Chương IV

Toán tử (operators) và biểu thức (expressions)

Toán tử (operator)

Toán tử là các thao tác để kết hợp các giá trị tạo ra một giá trị mới. Các giá trị được gọi là các toán hạng (operands). Ví dụ $2 + 3 = 5$ thì 2, 3 được gọi là các toán hạng, + là toán tử (số học), và 5 là kết quả.

Toán tử trong Java gồm các loại sau:

Toán tử gán (assignment operator)

Dùng để gán một giá trị đến một biến. Giá trị này có thể là hằng (như 6 hay "hello"), giá trị của biến khác, kết quả của một biểu thức hay hàm.

Kí hiệu: =

Ví dụ: gán giá trị 6 cho biến x

```
int x = 6;
```

Toán tử số học (arithmetic operator)

Cho phép thực hiện các thao tác tính toán giữa các giá trị số. Gồm:

Toán tử số học	Chức năng
+	Cộng hai giá trị
-	Trừ hai giá trị
*	Nhân hai giá trị
/	Chia hai giá trị
%	Lấy dư từ phép chia

* **Chú ý:** trong Java nếu chúng ta chia hai số nguyên, ví dụ lấy 7 chia 3, thì kết quả sẽ là một số nguyên, ví dụ 7 chia 3 là 3. Điều này chúng ta cần chú ý vì sẽ không cho kết quả như mong đợi.

Độ ưu tiên của các toán tử trong biểu thức theo thứ tự giảm dần từ trên xuống như sau:

Độ ưu tiên
*, /, %
+, -

Java cho phép kết hợp toán tử số học và toán tử gán. Xem bảng dưới đây:

Toán tử	Ví dụ	Ý nghĩa
+=	<code>x += 3</code>	<code>x = x + 3</code>
-=	<code>x -= 3</code>	<code>x = x - 3</code>
*=	<code>x *= 3</code>	<code>x = x * 3</code>
/=	<code>x /= 3</code>	<code>x = x / 3</code>
%=	<code>x %= 3</code>	Lấy dư từ phép chia x cho 3

Toán tử so sánh (comparison operator)

Cho phép so sánh hai giá trị, thường dùng trong các biểu thức điều kiện. Kết quả trả về là *true* hay *false*. Một số toán tử so sánh phổ biến:

Toán tử so sánh	Chức năng
==	Kiểm tra hai giá trị có bằng nhau không
!=	Kiểm tra hai giá trị khác nhau
<	Kiểm tra giá trị đầu có bé hơn giá trị sau không
>	Kiểm tra giá trị đầu có lớn hơn giá trị sau không
<=	Kiểm tra giá trị đầu có bé hơn hoặc bằng giá trị sau không
>=	Kiểm tra giá trị đầu có lớn hơn hoặc bằng giá trị sau không

Toán tử cộng chuỗi (concatention operator)

Trong Java, để kết hợp hai chuỗi (kiểu **String**) chúng ta dùng toán tử `+`. Có thể kết hợp với toán tử gán là `+=` Ví dụ: khai báo các biến sau:

```
String firstString = "Hello";
String secondString = "World";
String result;
```

Kết hợp hai chuỗi trong biến `firstString` và `secondString` và lưu kết quả trong biến `result` có thể viết như sau:

```
result = firstString + secondString;
```

hoặc

```
result = firstString;
result = result + secondString;
```

hoặc

```
result = firstString;
result += secondString;
```

Chú ý: do toán tử + được dùng cho kết hợp hai chuỗi đồng thời cũng là một toán tử số học nên cần thận trọng khi dùng toán tử này khi có liên quan đến chuỗi, ví dụ:

```
System.out.print("4" + "3"); // kết quả là "43" thay vì 7
```

Toán tử luận lý (logical operator)

Trả về giá trị **true** hay **false** từ việc kết hợp nhiều biểu thức. Một số toán tử

Toán tử luận lý	Chức năng
&	Trả về True nếu tất cả các biểu thức đều trả về True
	Trả về True nếu ít nhất một biểu thức trả về True
!	Trả về True nếu biểu thức là False và ngược lại
&&	Tương tự & nhưng sẽ không kiểm tra các biểu thức còn lại nếu biểu thức đầu tiên quyết định kết quả trả về.
	Tương tự nhưng sẽ không kiểm tra các biểu thức còn lại nếu biểu thức đầu tiên quyết định kết quả trả về.

Ví dụ:

```
int num1 = 3;
int num2 = 7;

num1 == 3 & num2 == 7    // true
num1 == 2 & num2 == 7    // false
num1 == 3 | num2 == 11   // true
!(num1 == 5)             // true
num1 == 2 && num2 == 7    //trả về false vì num1 khác 2 và không
                           //cần kiểm tra biểu thức num2 = 7
```

Biểu thức (expressions)

Biểu thức là một kết hợp giữa các toán hạng và các toán tử và có thể cho ra một kết quả nào đó. Biểu thức có thể đơn giản chỉ là một toán hạng nhưng cũng có thể rất phức tạp.

Ví dụ các biểu thức:

- Biểu thức đơn giản có thể chỉ là một số 3
- Biểu thức gồm toán hạng và toán tử: $3 + 2$
- Biểu thức phức tạp hơn: $((2 + 3) * 5) / 3$

Trong quá trình tính toán một biểu thức cần chú ý:

- Các toán tử có độ ưu tiên cao hơn sẽ được thực hiện trước
- Biểu thức trong dấu ngoặc đơn thực hiện trước
- Nếu trong một biểu thức xuất hiện các toán tử có cùng độ ưu tiên thì thực hiện tính toán từ trái sang phải

Ví dụ: $(2 - 3 + 7) / 3 * 2 = 4$

- Toán tử gán thực hiện kết hợp từ phải sang trái. Ví dụ:

$x = y = z = 1$; // 1 được gán cho biến z, biến y, và biến x

Tăng, giảm biến

Nếu muốn tăng giá trị biến count lên 1, chúng ta có thể dùng toán tử +:

```
count = count + 1;
```

Chúng ta cũng có thể dùng toán tử một ngôi (unary) (tức là toán tử chỉ có một toán hạng) ++ như sau:

```
count ++;
```

Tương tự, nếu muốn giảm 1 cho biến count, chúng ta có thể viết:

```
count = count - 1;
```

hay

```
count --;
```

Toán tử một ngôi ++ hay -- tùy theo vị trí đặt trước (prefix) hay sau (postfix) sẽ cho ra những kết quả khác nhau, ví dụ:

```
int x;  
x = 4;  
System.out.print (x++); // kết quả hiển thị là 4 vì x được hiển thị  
                        // trước khi tăng 1  
x = 4;  
System.out.print (++x); // kết quả hiển thị là 5 vì x hiển thị sau  
                        // khi tăng 1
```


Chương V

Cấu trúc điều khiển

Các câu lệnh quyết định

Khai báo các biến luận lý

Mệnh đề (hay biểu thức) luận lý là các mệnh đề chỉ nhận giá trị đúng (**true**) hay sai (**false**).

Java cung cấp kiểu dữ liệu **boolean** là kiểu dữ liệu luận lý. Một biến kiểu luận lý chỉ chứa một trong hai giá trị là **true** hay **false**.

Các toán tử luận lý

Trả về giá trị **true** hay **false** từ việc kết hợp nhiều biểu thức. Một số toán tử

Toán tử	Chức năng
&	Trả về true nếu tất cả các biểu thức đều trả về true
	Trả về true nếu ít nhất một biểu thức trả về true
!	Trả về true nếu biểu thức là false và ngược lại
&&	Tương tự & nhưng sẽ không kiểm tra các biểu thức còn lại nếu biểu thức đầu tiên quyết định kết quả trả về.
	Tương tự nhưng sẽ không kiểm tra các biểu thức còn lại nếu biểu thức đầu tiên quyết định kết quả trả về.

Ví dụ:

```
int num1 = 3;
int num2 = 7;

num1 == 3 & num2 == 7    // true
num1 == 2 & num2 == 7    // false
num1 == 3 | num2 == 11   // true
!(num1 == 5)             // true
num1 == 2 && num2 == 7    // trả về false vì num1 khác 2 và không
```

```
//cần kiểm tra biểu thức num2 = 7
```

Bên cạnh các toán tử luận lý còn có toán tử quan hệ hay bằng >, <, >=, <=, ==, ! dùng để tạo các biểu thức luận lý và độ ưu tiên của các toán tử. Có thể tham khảo lại chương IV.

Các lệnh điều kiện (conditional statement): *if* và *switch*

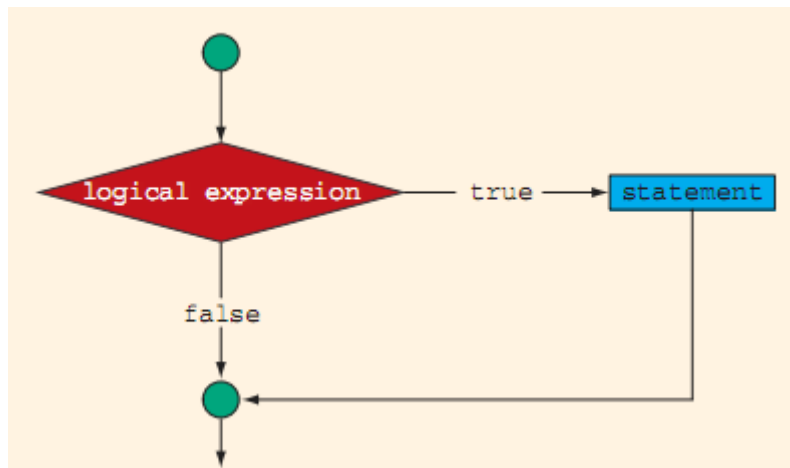
Khi viết code, chúng ta hay gặp những tình huống phải đưa ra những hành động khác nhau trong những điều kiện khác nhau. Để làm điều này, chúng ta sử dụng các lệnh điều kiện (conditional statements).

Lệnh *if*.

Cú pháp:

```
if (logical expression) {
    statement
}
```

Sơ đồ khối



Nếu biểu thức **logical expression** nhận giá trị **true** thì thực hiện lệnh (hay khối lệnh) **statement**, ngược lại không làm gì.

Ví dụ:

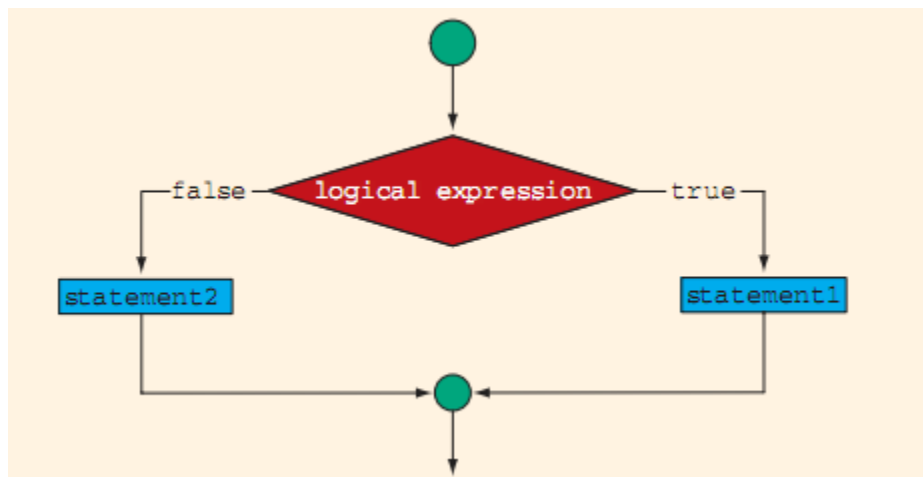
```
if (age < 18) {
    message = "You are under 18 years of age!";
}
```

Lệnh if...else

Cú pháp:

```
if (logical expression) {  
    statement1  
}  
else {  
    statement2  
}
```

Sơ đồ khối



Nếu biểu thức **logical expression** nhận giá trị **true** thì thực hiện lệnh (hay khối lệnh) **statement1**, ngược lại thực hiện lệnh (hay khối lệnh) **statement2**.

Ví dụ:

```
if (age < 18) {  
    message = "You are under 18 years of age!";  
}  
else {  
    message = "Welcome to our website!";  
}
```

Lệnh if...else if...else

Cú pháp:

```
if (logical expression 1) {  
    statement1  
}  
else if (logical expression 2) {  
    statement2  
}  
else {  
    statement3  
}
```

Nếu biểu thức **logical expression 1** nhận giá trị **true** thì thực hiện lệnh (hay khối lệnh) **statement1**, ngược lại (biểu thức **logical expression 1** nhận giá trị **false**) nếu biểu thức **logical expression 2** nhận giá trị **true** thực hiện lệnh (hay khối lệnh) **statement2** và nếu biểu thức **logical expression 2** nhận giá trị **false** thì thực hiện lệnh (hay khối lệnh) **statement3**.

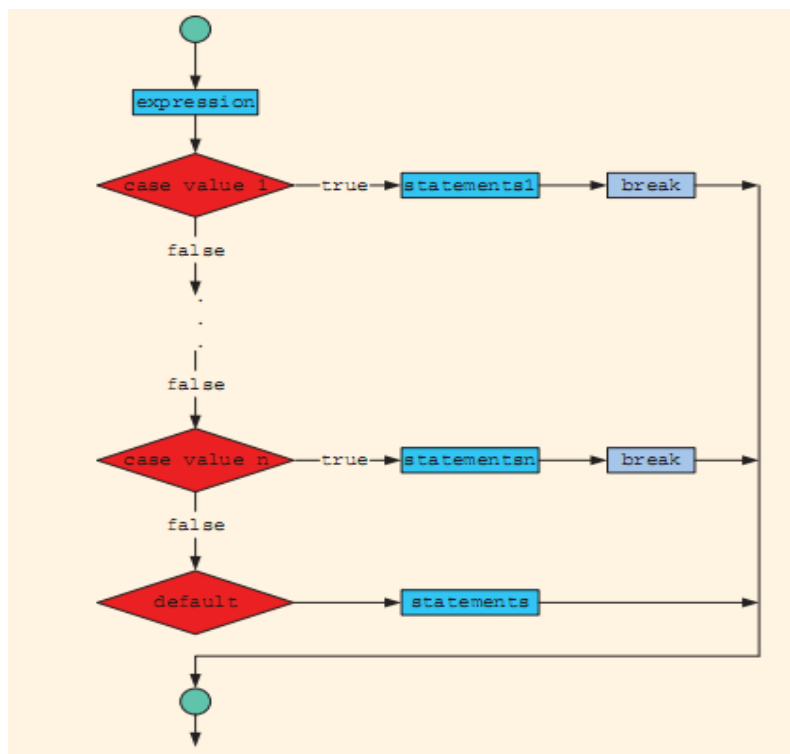
Ví dụ:

```
if (time < 10) {  
    greeting = "Good morning";  
}  
else if (time < 20) {  
    greeting = "Good day";  
}  
else {  
    greeting = "Good evening";  
}
```

Cũng là lệnh điều kiện giống *if* dùng để đưa ra những hành động khác nhau trong những điều kiện khác nhau, chúng ta có thể dùng lệnh *switch* trong Java. Cú pháp:

```
switch (expression) {
    case value 1:
        statement1
        break;
    .....
    case value n:
        statementn
        break;
    default:
        statements
        break;
}
```

Sơ đồ khối



Ví dụ:

```
char grade = 'C';
switch(grade) {
    case 'A':
        System.out.println("Excellent!");
        break;
    case 'B':
    case 'C':
        System.out.println("Well done");
        break;
    case 'D':
        System.out.println("You passed");
    case 'F':
        System.out.println("Better try again");
        break;
    default:
        System.out.println("Invalid grade");
}

System.out.println("Your grade is " + grade);
```

Một số lưu ý khi sử dụng lệnh **switch**:

- Chỉ dùng **switch** với các kiểu dữ liệu sơ cấp (primitive data types) như **int** hay **String**. Với các kiểu dữ liệu khác (bao gồm cả **float** và **double**), chúng ta phải dùng **if**.
- Nhãn cho mỗi **case** phải là hằng, như 42 hay "42", nếu là một biểu thức cần được xử lý trong thời điểm run time thì phải sử dụng **if**.
- Mỗi nhãn cho mỗi **case** phải là duy nhất.
- Nếu muốn thực hiện một (hay khối) lệnh cho nhiều giá trị nhãn **case**, chúng ta có thể cung cấp một danh sách các nhãn **case** liên tục (không bị ngắt bởi các lệnh hay **break**) và

các lệnh thực thi cho nhãn **case** cuối cùng cũng chính là cho tất cả các nhãn trong danh sách. Ví dụ: nếu biến `ch` nhận các giá trị “a”, “b”, “c” thì xuất ra màn hình “Hello”

```
char ch;
switch (ch)
{
    case "a":
    case "b":
    case "c":
        System.out.println ("Hello");
        break;
}
```

Nếu viết như sau sẽ bị lỗi:

```
char ch;
switch (ch)
{
    case "a":
    case "b":
        System.out.println ("Hello");
    case "c":
        System.out.println ("Hello");
        break;
}
```

Lệnh lặp

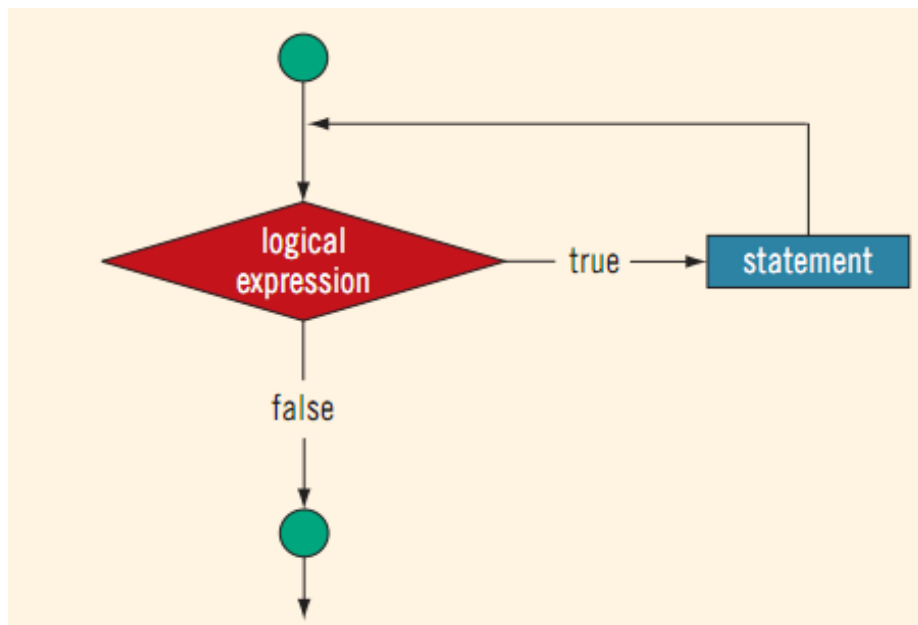
Trong lập trình, có những lệnh hay khối lệnh được lặp lại nhiều lần. Thay vì viết các lệnh hay khối lệnh này nhiều lần, chúng ta có thể dùng các lệnh lặp. **Java** hỗ trợ các lệnh lặp *while*, *do*, *for*.

Lệnh lặp *while*

Lệnh **while** sẽ lặp theo một điều kiện nào đó. Chừng nào biểu thức điều kiện **logical expression** còn đúng thì lệnh lặp sẽ vẫn thực thi lệnh (hay khối lệnh) **statement**. Cú pháp:

```
while (logical expression)
{
    statement
}
```

Sơ đồ khối



Ví dụ hiển thị các giá trị từ 0 đến 9:

```
int i = 0;
String numseq = "";
while (i < 10)
{
    numseq += " " + i;
    i = i + 1; // or i++;
}
System.out.println("The number sequence is "+ numseq);
```


Sai lầm phổ biến của những người bắt đầu dùng **while** là quên dòng lệnh $i = i + 1$. Biến i trong ví dụ trên đóng vai trò như lính canh (sentinel) để giúp vòng lặp hữu hạn. Biến i còn gọi là *sentinel variable*.

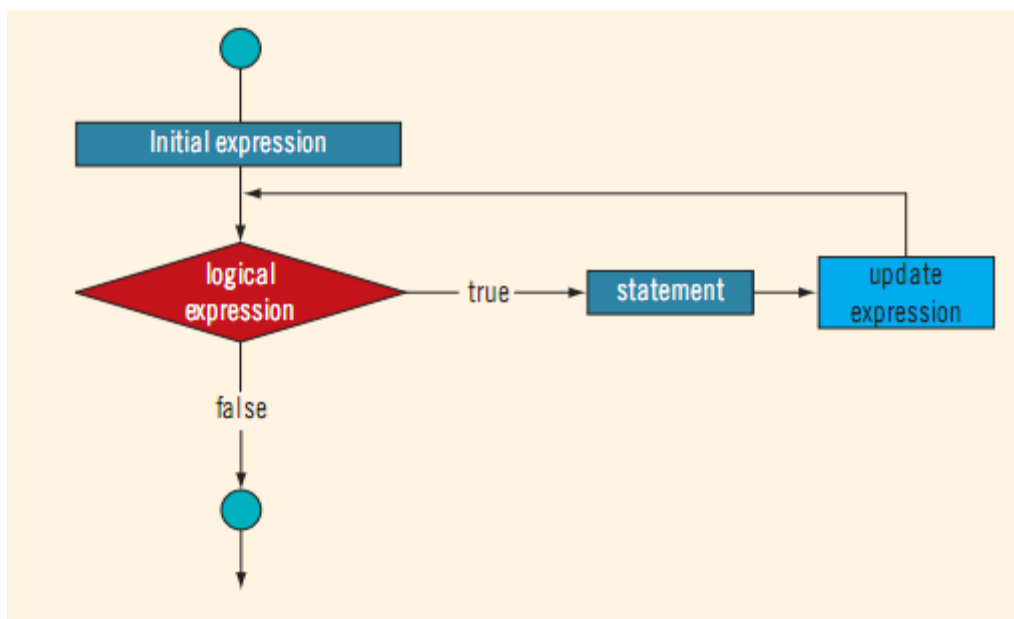
Lệnh lặp *for*

Dùng để lặp theo một số lần xác định cho trước, ví dụ thực hiện lệnh xuất dòng chữ **Hello** 5 lần. Cú pháp:

```
for (initial expression; logical expression; update expression)
{
    statement
}
```

initial expression, **logical expression**, **update expression** là còn gọi là các biểu thức điều khiển và chúng cách nhau bởi dấu chấm phẩy. **initial expression** là biểu thức khởi tạo giá trị để bắt đầu lặp, **logical expression** là biểu thức điều kiện để thực hiện lệnh (hay khối lệnh) **statement**, **update expression** là biểu thức cập nhật giá trị sau khi thực hiện **statement**, nếu **logical expression** vẫn còn **true** thì **for** sẽ tiếp tục thực thi **statement**.

Sơ đồ khối



Hiển thị các số từ 1 đến 10

```
int i;
```

```
String numseq = "";
for (i = 1; i <= 10; i++)
{
    numseq += " " + i;
}
System.out.println("The number sequence is "+ numseq);
```

Hiển thị các số chẵn trong dãy số nguyên từ 0 đến 10

```
int i;
String numseq = "";
for (i = 0; i <= 10; i = i + 2)
{
    numseq += " " + i;
}
System.out.println("The even sequence is "+ numseq);
```

Chúng ta có thể bỏ qua các thành phần **initial expression**, **logical expression**, **update expression** trong **for**, cụ thể, nếu chúng ta bỏ **logical expression** tức là chúng ta mặc định giá trị của nó là **true** thì vòng lặp **for** sẽ lặp vô hạn như ví dụ:

```
for (i = 0;;i++)
{
    System.out.println ("somebody stop me!");
}
```

Nếu chúng ta bỏ qua **initial expression** và **update expression** thì **for** sẽ trông như **while** như ví dụ sau:

```
int i = 0;
for (; i < 10;)
{
    System.out.println (i);
}
```

```
i++;
}
```

Có thể có nhiều biểu thức khởi tạo **initial expression** và nhiều biểu thức cập nhật **update expression** như ví dụ sau:

```
for (i = 0, j = 10; i <= j; i++, j--)
{
    ...
}
```

Chúng ta có thể khai báo một biến trong biểu thức **initial expression** với lưu ý:

- Không thể dùng biến này sau khi vòng lặp **for** kết thúc, ví dụ:

```
for (int i = 0; i < 10; i++)
{
    ...
}

System.out.println (i); // error
```

- Có thể khai báo trùng tên biến (trùng kiểu) cho các **for** khác nhau, ví dụ:

```
for (int i = 0; i < 10; i++)
{
    ...
}

for (int i = 0; i < 20; i += 2) // okay
{
    ...
}
```

Có thể dùng **for** để duyệt và nhận thông tin từ các phần tử của một đối tượng tập hợp (tương tự lệnh **foreach** trong C#)

Cú pháp:

```
for (element : collection)
{
    statement
}
```

Ví dụ:

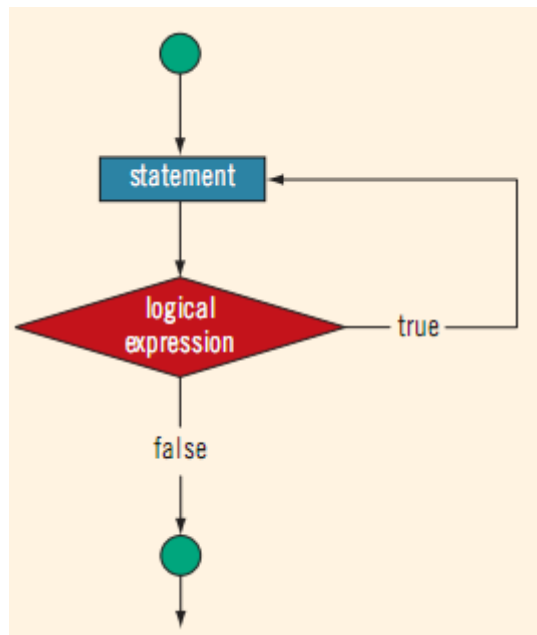
```
int[] fibarray = new int[] { 0, 1, 1, 2, 3, 5, 8, 13 };
for (int element : fibarray)
{
    System.out.print(" "+ element);
}
```

Lệnh lặp *do*

Lệnh ***while*** và ***for*** kiểm tra biểu thức điều kiện đầu tiên để bắt đầu lặp, nghĩa là, nếu biểu thức điều kiện là **false** thì các lệnh trong vòng lặp sẽ không được thực hiện. Lệnh lặp ***do*** thì khác; biểu thức điều kiện sẽ được kiểm tra sau mỗi lần lặp nên các lệnh trong lệnh lặp ***do*** sẽ thực hiện ít nhất một lần. Cú pháp:

```
do
{
    statement
} while (logical expression);
```

Sơ đồ khối



Xét ví dụ sau để phân biệt **while** và **do**:

Đoạn mã sau sẽ không thực thi vì $i = 11 > 10$

```
int i = 11;
String numseq = "";
while (i < 10)
{
    numseq += " " + i;
    i = i + 1;
}
System.out.println("The number is " + numseq);
```

Đoạn mã sau sẽ thực thi và kết quả là ***The number is 11***

```
int i = 11;
String numseq = "";
do
{
    numseq += " " + i;
```

```

        i = i + 1;
    } while (i < 10);

    System.out.println("The number is " + numseq);

```

Lệnh thoát vòng lặp

Có thể thoát khỏi vòng lặp **while**, **do** hay **for** bằng lệnh **break**. Ví dụ: vòng lặp sau sẽ thoát khi $i = 5$, tức là chỉ xuất các giá trị từ 1 đến 4

```

for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        break;
    }

    numseq += " " + i;
}

System.out.println(numseq);

```

Có thể bỏ qua một lần lặp để chuyển qua lần lặp kế tiếp bằng lệnh **continue**. Ví dụ xuất từ 1 đến 10 nhưng bỏ số 5:

```

for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        continue;
    }

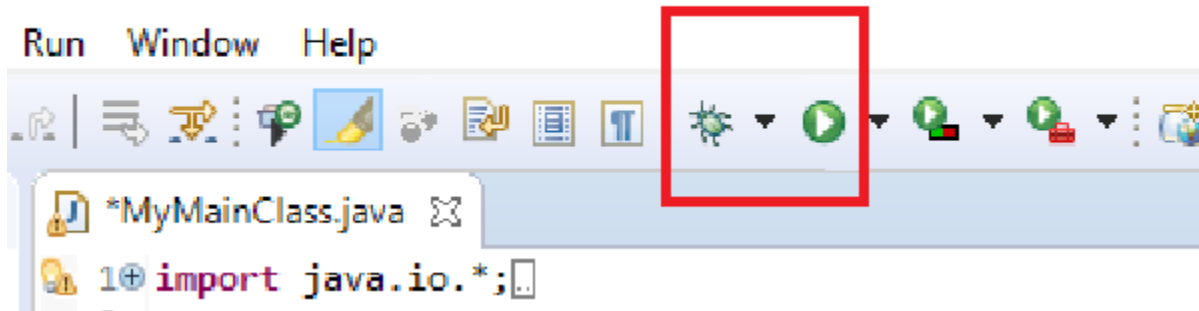
    numseq += " " + i;
}

System.out.println(numseq);

```

Debugging

Debugging là thuật ngữ dùng để chỉ khả năng kiểm tra, phát hiện lỗi, sửa lỗi và bảo trì mã chương trình. Đây là một trong những kỹ năng quan trọng nhất của người lập trình. Lời khuyên tốt nhất cho những người mới học lập trình Java trong môi trường **Eclipse** là, khi chạy một chương trình, nên sử dụng nút **Debug** thay vì sử dụng nút **Run**:



Giá trị của các biến

Hình thức đơn giản nhất để kiểm tra giá trị của các biến là hiển thị các giá trị này ra màn hình bằng phương thức, ví dụ **System.out.print()**. Tuy nhiên, môi trường **Eclipse** hỗ trợ chúng ta nhiều công cụ hữu ích giúp chúng ta debugging một cách hiệu quả. Giả sử chương trình **Java** của chúng ta như sau:

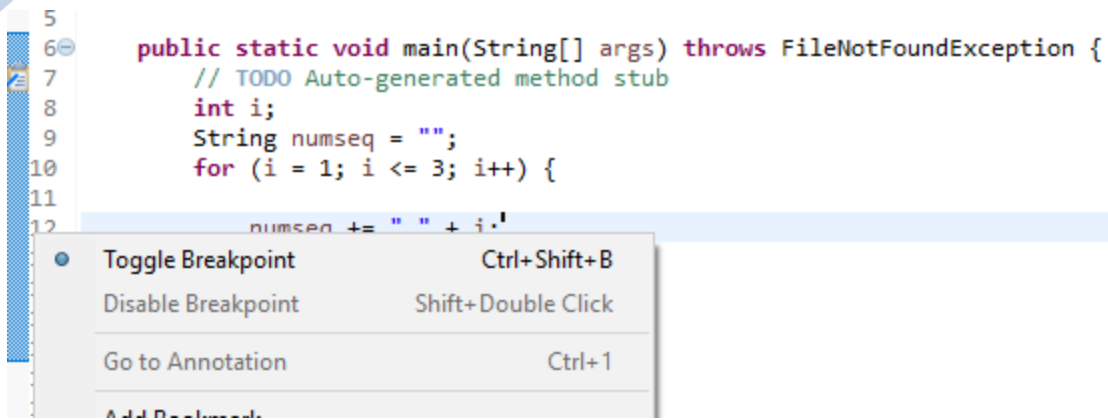
```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    int i;
    String numseq = "";
    for (i = 1; i <= 3; i++) {
        numseq += " " + i;
    }
    System.out.println(numseq);
}
```

Bây giờ chúng ta sẽ khám phá một số công cụ phục vụ quá trình debugging trong **Eclipse**.

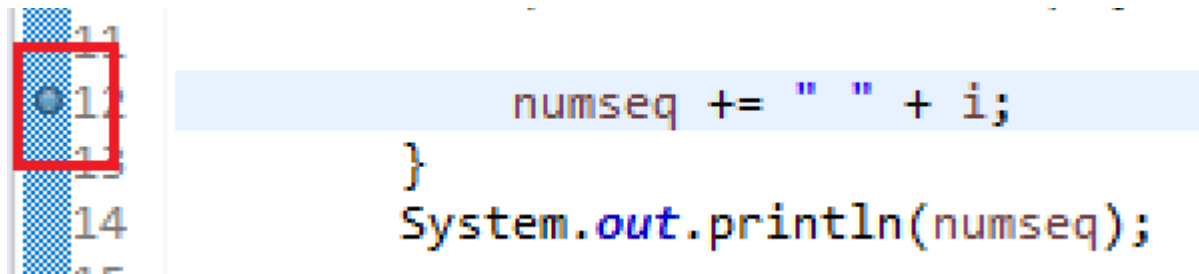
Thiết lập điểm dừng (breakpoint)

Điểm dừng là vị trí chương trình sẽ tạm dừng và cho phép chúng ta quan sát những gì đang diễn ra trong các đoạn mã chương trình. Ví dụ chúng ta muốn thiết lập điểm dừng tại lệnh gán đến biến *numseq* trong *for* như sau:

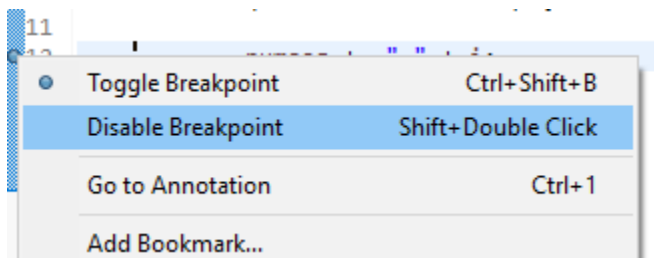
Kích chuột phải vào lề trái của **Eclipse** tại dòng lệnh và chọn **Toggle Breakpoint**



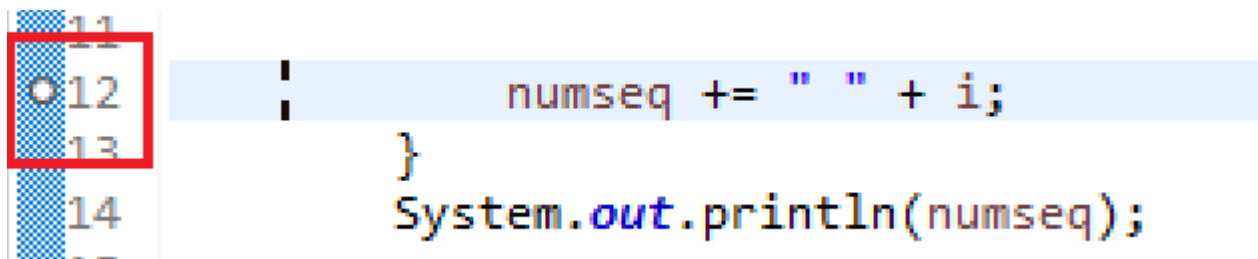
Một điểm dừng sẽ được thiết lập:



Điểm dừng có màu xanh tức là nó đang hoạt động (phân biệt với điểm dừng màu trắng) và chương trình sẽ tạm dừng khi đến điểm dừng. Chúng ta có thể cho điểm dừng ngừng hoạt động bằng cách nhấn chuột phải lên điểm dừng và chọn **Disable Breakpoint**:



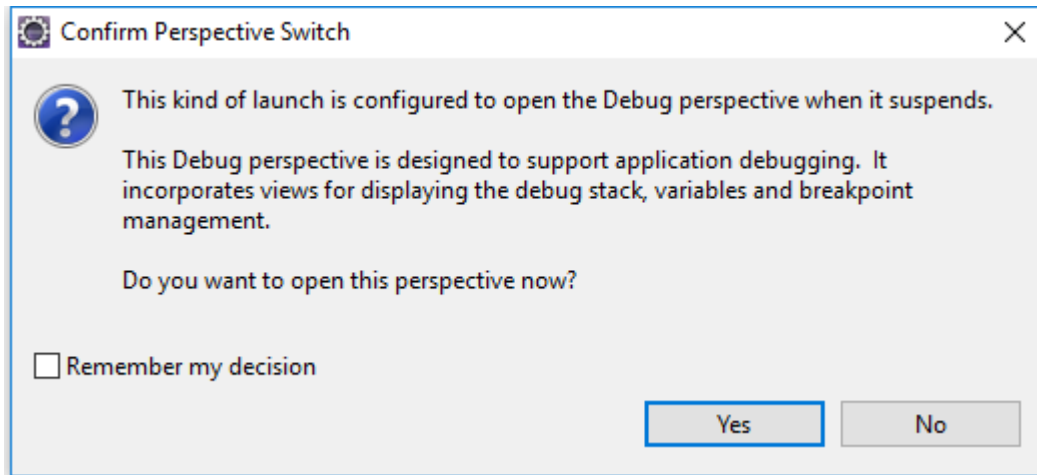
Lúc này điểm dừng sẽ chuyển sang màu trắng:



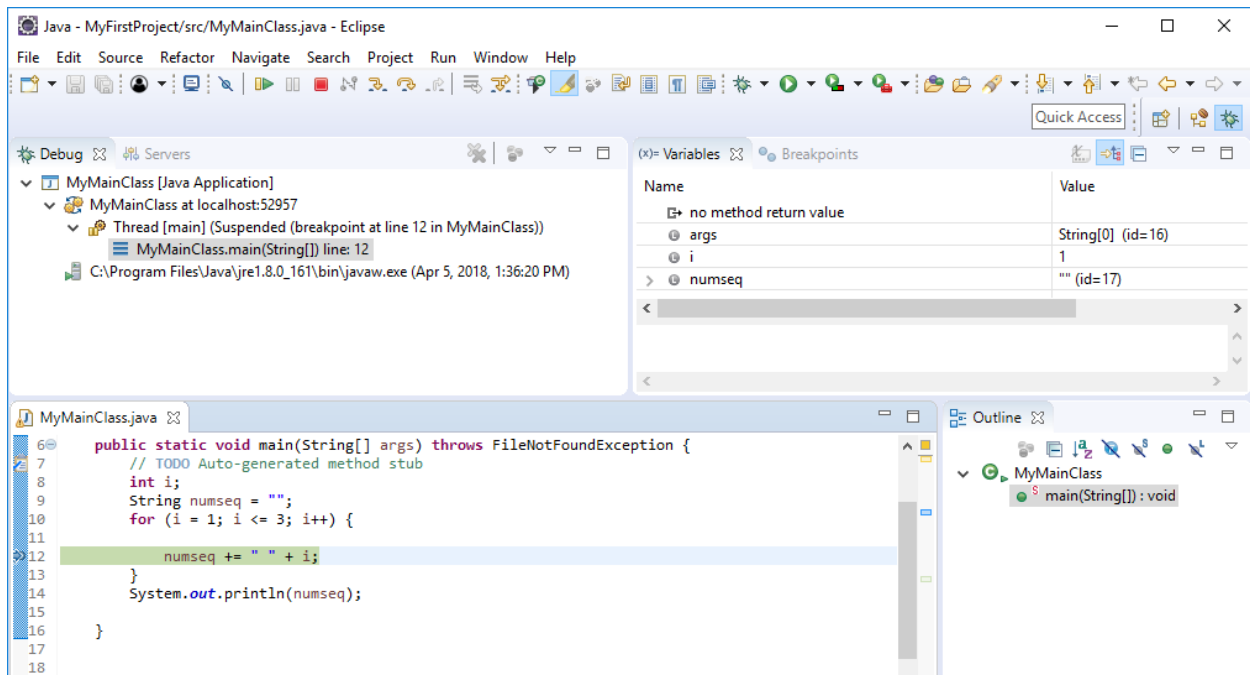
Và chương trình sẽ vẫn hoạt động bình thường khi đến điểm dừng.

Debug perspective trong Eclipse

Khi thực thi chương trình bằng nút **Debug**, nếu chương trình bắt gặp điểm dừng, một hộp thoại yêu cầu chúng ta chuyển sang chế độ **Debug** xuất hiện:



Chọn **Remember my decision** và nhấn **Yes**. Lúc này **Eclipse** sẽ chuyển chúng ta đến khung nhìn mới với nhiều công cụ hỗ trợ chúng ta debugging



Đề ý dòng lệnh của chúng ta lúc này có dấu mũi tên chòng lệnh điểm dừng:

```

5
6
7
8
9
10
11
12
13
14
15



```


```

public static void main(String[] args)
    // TODO Auto-generated method stub
    int i;
    String numseq = "";
    for (i = 1; i <= 3; i++) {
        numseq += " " + i;
    }
    System.out.println(numseq);

```

Nếu quan sát trên thanh công cụ của **Eclipse**, chúng ta sẽ thấy ba nút nhấn quan trọng là

Resume  - cho phép chương trình thực thi bình thường, **Suspend**  - tạm dừng

chương trình khi nó đang thực thi nhưng chưa gặp điểm dừng, và **Terminate**  - ngừng thực thi chương trình (các phiên bản **Eclipse** cũ hơn sẽ là **Play**, **Pause**, và **Stop**).

Chúng ta có thể chuyển đổi qua lại giữa chế độ bình thường và chế độ debugging bằng cách nhấn các nút ở góc phải trên trên thanh công cụ **Eclipse**:



Step Over và Step Into

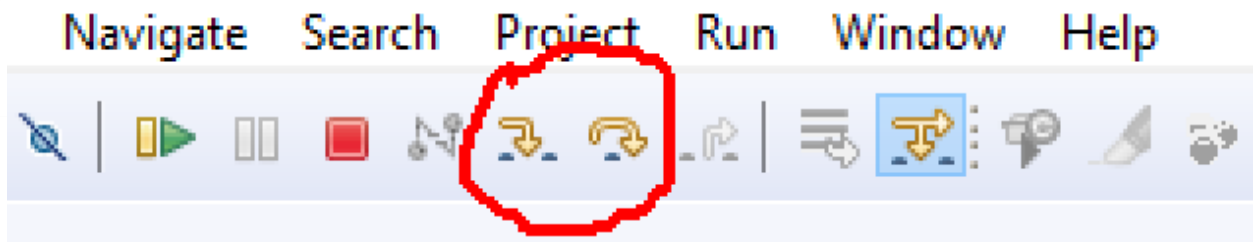
Khi chương trình tạm dừng, chúng ta có thể xem sự thay đổi các giá trị của tất cả các biến trong chương trình khi nó thực hiện một lần tại một thời điểm bằng chức năng **Step Over**



và **Step Into**



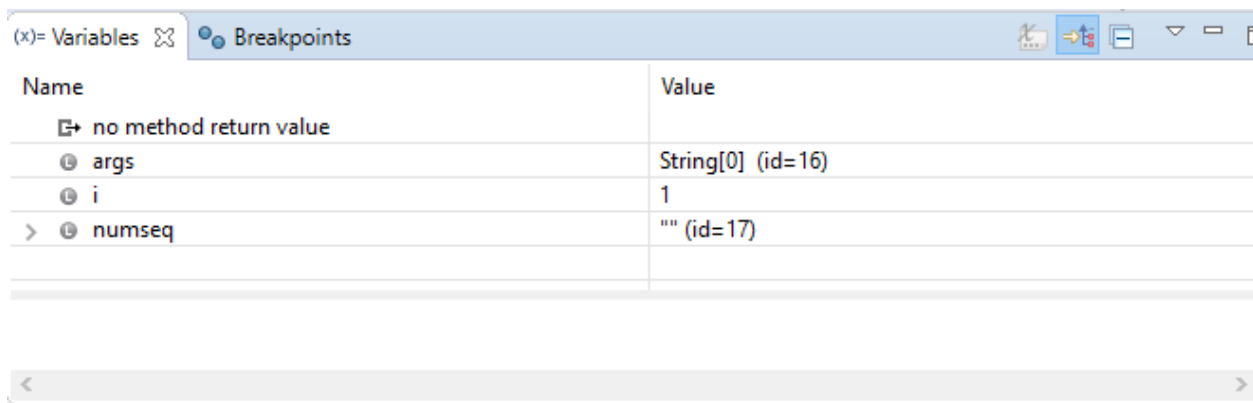
trên thanh công cụ của **Eclipse**:



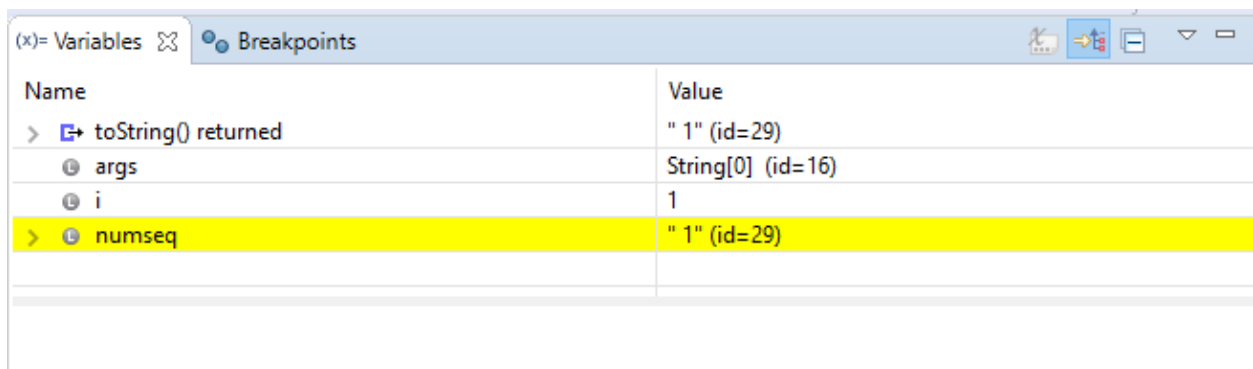
Với **Step Over**, chương trình sẽ thực thi dòng mã hiện tại và tạm ngừng trở lại. Nếu dòng mã hiện tại là lệnh gọi một phương thức thì **Step Over** xem lệnh này như một lệnh bình thường và sẽ không nhảy vào trong phần thân của phương thức. **Step Into** tương tự **Step Over** nhưng nếu lệnh hiện tại là lệnh gọi một phương thức thì chương trình sẽ nhảy vào phần thân phương thức và dừng.

Cửa sổ Variables

Với **Step Over** và **Step Into**, chúng ta có thể xem sự thay đổi giá trị của các biến qua cửa sổ **Variables**:



Lúc này, nếu nhấn **Step Over** thì kết quả:



Dòng lệnh lúc này:

```

MyMainClass.java
1 import java.io.*;
3
4 public class MyMainClass {
5
6     public static void main(String[] args) throws FileNotFoundException {
7         // TODO Auto-generated method stub
8         int i;
9         String numseq = "";
10        for (i = 1; i <= 3; i++) {
11
12            numseq += " " + i;
13        }
14        System.out.println(numseq);
15

```

Với những công cụ đơn giản mà hiệu quả trong **Eclipse**, chúng ta có thể dễ dàng debugging chương trình của mình.

Lệnh assert

Lệnh **assert** là lệnh hữu ích khi debugging hay testing. Lệnh **assert** dùng để đảm bảo một điều kiện luôn đúng, nếu điều kiện sai, chương trình sẽ dừng và cảnh báo lỗi. Mặc định, **Eclipse** bỏ qua lệnh **assert**. Để sử dụng tính năng của lệnh **assert**, chúng ta vào **Run > Run Configurations**, chọn dự án (Project) và lớp cần áp dụng lệnh **assert**. Tính năng dùng cho lệnh **assert** được gọi là **enable assertions** và viết tắt là **ea**.

Để hiểu hơn cách dùng lệnh **assert**, giả sử chúng ta có lớp **MainClass** trong dự án **MyFirstProject** như sau:

```

public class MainClass {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int so_bi_chia;// số bị chia
        int so_chia;// số chia
        Scanner scanner = new Scanner(System.in);
        System.out.println("Nhập số bị chia: ");
        so_bi_chia = Integer.parseInt(scanner.nextLine());
        System.out.println("Nhập số chia: ");
        so_chia = Integer.parseInt(scanner.nextLine());
        //Đảm bảo số chia phải khác 0
        assert(so_chia != 0);
        // In ra kết quả nếu số chia đảm bảo khác 0
        System.out.println(so_bi_chia + " / " + so_chia + " = " +
        (so_bi_chia / so_chia) + " So dư là: " + (so_bi_chia % so_chia));
    }

}

```

Khi thực thi chương trình:

Nhap so bi chia:

3

Nhap so chia:

0

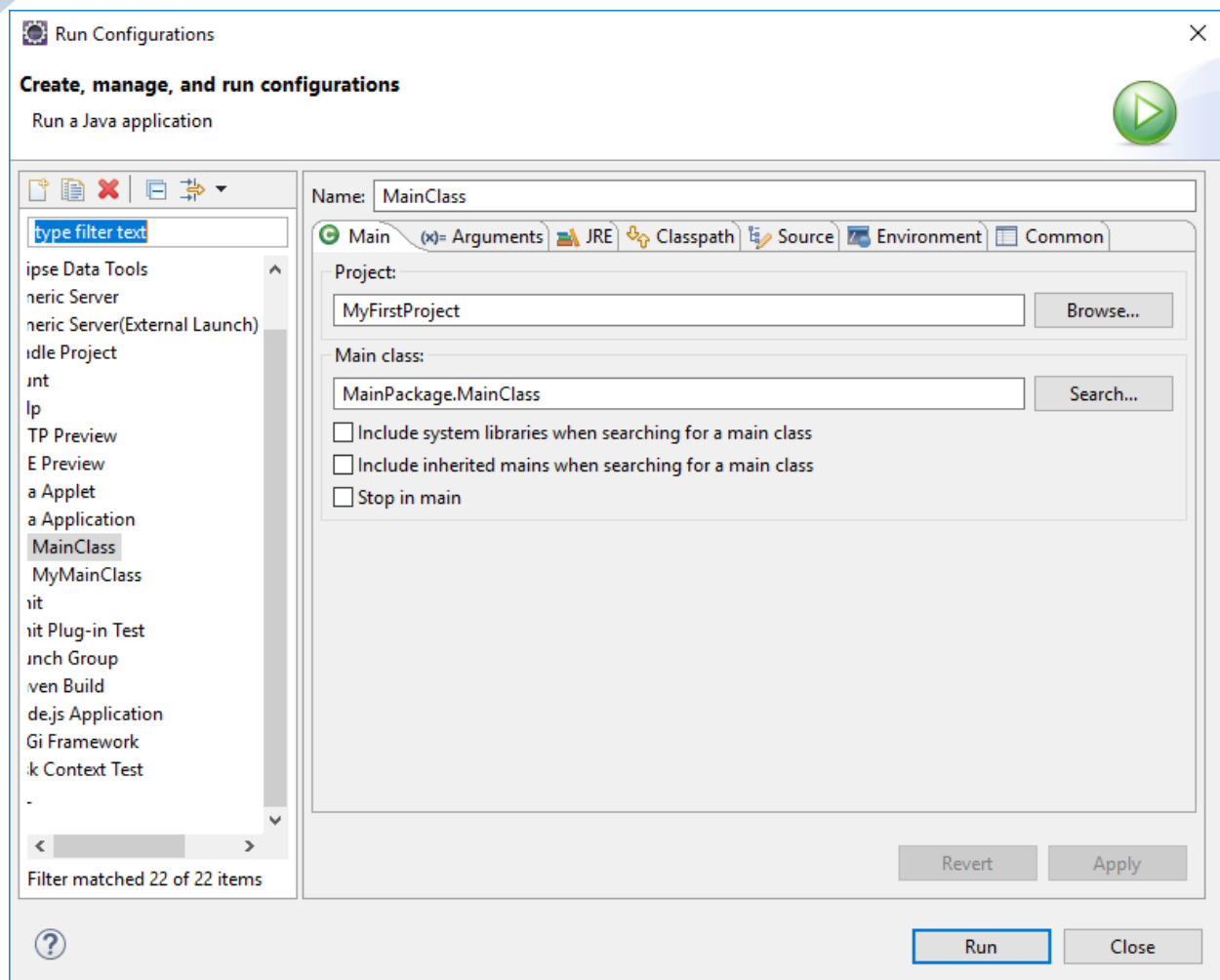
Kết quả sẽ phát sinh một ngoại lệ (vấn đề ngoại lệ sẽ tìm hiểu trong chương IX):

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at MainPackage.MainClass.main(MainClass.java:20)
```

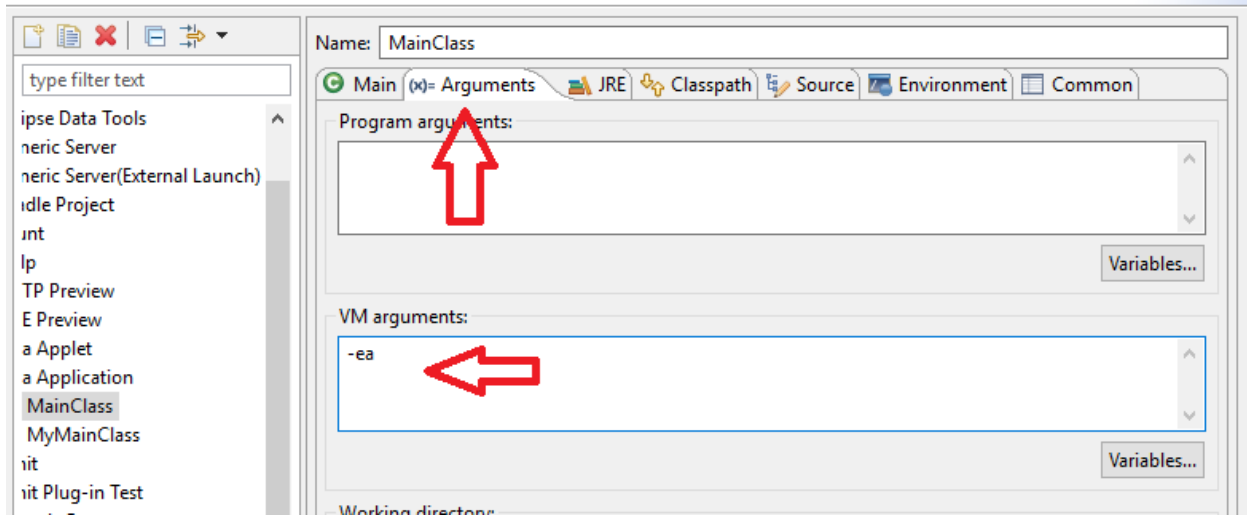
Và nếu chúng ta **Debug** chương trình:

```
12      System.out.println("Nhap so bi chia: ");
13      so_bi_chia = Integer.parseInt(scanner.nextLine());
14      System.out.println("Nhap so chia: ");
15      so_chia = Integer.parseInt(scanner.nextLine());
16      //Đảm bảo số chia phải khác 0
17      assert(so_chia != 0);
18      // In ra kết quả nếu số chia đảm bảo khác 0
19      System.out.println(so_bi_chia + " / " + so_chia + " = " +
20      (so_bi_chia / so_chia) + " So dư là: " + (so_bi_chia % so_chia));
21  }
```

Lúc này chương trình sẽ highlight dòng lệnh trực tiếp gây ra lỗi. Dùng lệnh **assert** bằng cách vào **Run > Run Configurations**, chọn **MainClass** trong dự án **MyFirstProject**:



Chọn tab **(x)= Arguments**. Nhập **-ea** trong **VM arguments**:



Nhấn **Apply** và **Close**. Nếu chúng ta thực thi lại chương trình (vẫn dùng các input là 3 và 0) bằng **Run** sẽ phát sinh ngoại lệ:

```
Exception in thread "main" java.lang.AssertionError
    at MainPackage.MainClass.main(MainClass.java:17)
```

Nếu dùng **Debug** chương trình sẽ highlight ngay tại lệnh **assert** để chúng ta có thể phát hiện và điều chỉnh lỗi:

```
12      System.out.println("Nhap so bi chia: ");
13      so_bi_chia = Integer.parseInt(scanner.nextLine());
14      System.out.println("Nhap so chia: ");
15      so_chia = Integer.parseInt(scanner.nextLine());
16      //Đảm bảo số chia phải khác 0
17      assert(so_chia != 0);
18      // In ra kết quả nếu số chia đảm bảo khác 0
19      System.out.println(so_bi_chia + " / " + so_chia + " = " +
20      (so_bi_chia / so_chia) + " So du là: " + (so_bi_chia % so_chia));
21  }
```

Lệnh **assert** cũng rất hữu ích trong testing (ví dụ Unit Testing).

Chương VI

Lập trình hướng đối tượng

Lớp (class) và đối tượng (object)

Java là ngôn ngữ lập trình hướng đối tượng (object oriented programming). Thành phần cơ bản nhất của các ngôn ngữ hướng đối tượng nói chung và **Java** nói riêng là lớp và đối tượng.

Lớp (class)

Lớp là một mô tả tổng quát về một kiểu đối tượng nào đó – những gì đối tượng có (các đặc trưng hay thuộc tính của đối tượng) và đối tượng có thể làm được gì (các chức năng của đối tượng). Mô tả trong **Java**, các đặc trưng hay thuộc tính là các biến thành viên của lớp, các tính năng là các hàm hay phương thức thành viên của lớp.

Ví dụ lớp **Circle** sở hữu thành phần quan trọng là *radius* và các phương thức *computePerimeter* – tính chu vi và phương thức *computeArea* – tính diện tích.

Tạo một lớp trong **Eclipse**, ví dụ lớp **Circle**, tương tự cách tạo lớp **HelloWorld** trong chương I và II. Lưu ý rằng:

- Lớp phải được chứa trong một gói (package) nào đó
- Tên lớp trùng với tên tập tin Java, ví dụ *Circle.java*

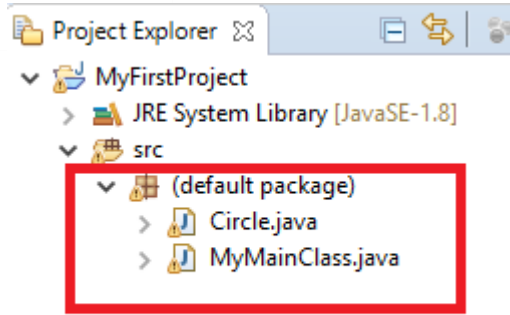
Cú pháp của một lớp, ví dụ lớp **Circle** như sau:

```
public class Circle {  
  
}
```

Để tiện theo dõi, chúng ta tạo thêm một lớp tên là **MyMainClass** chứa phương thức *main*. Tập tin **MyMainClass.java** có nội dung như sau:

```
public class MyMainClass {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
    }  
}
```


Lớp **Circle** và **MyMainClass** thuộc về cùng một package (default package):



Biến thành viên (member variables hay fields)

Biến thành viên của một lớp được khai báo bên trong lớp và không thuộc bất cứ phương thức nào của lớp. Ví dụ khai báo các biến thành viên cho lớp **Circle** như sau:

```
public class Circle {
    public float radius;
    private String name;
    protected float lineWidth;
    private static int circleCount;
}
```

Như khai báo trên, lớp **Circle** có 4 biến thành viên là **radius**, **name**, **linewidth**, và **circleCount**. Lưu ý rằng, bên cạnh kiểu dữ liệu, mỗi biến còn đi kèm với một từ khóa chỉ phạm vi truy cập là:

- **public**: các biến hay phương thức của lớp được đánh dấu là **public** có thể được truy cập (hay sử dụng) bởi tất cả các lớp.
- **private**: các biến (hay các phương thức nhưng trường hợp này ít khi xảy ra) được đánh dấu là **private** chỉ có thể được truy cập bởi chính lớp chứa biến đó.
- **protected**: các biến (hay các phương thức nhưng trường hợp này ít khi xảy ra) được đánh dấu là **protected** chỉ có thể được truy cập bởi các lớp hay đối tượng trong cùng một gói (package) với lớp chứa biến đó.
- **static**: nếu một biến thành viên được đánh dấu là **static**, chỉ có một bản sao của biến được chia sẻ cho tất cả các đối tượng của lớp đó. Các thành viên (biến hay phương thức) **static** được tham chiếu đến lớp thay vì đối tượng.

Đối tượng (object)

Nếu lớp là một mô tả tổng quát cho một kiểu đối tượng nào đó thì đối tượng là các thể hiện (instances) cụ thể của lớp đó. Ví dụ các đối tượng của lớp **Circle** sẽ là những **Circle** có các *radius* khác nhau.

Để tạo một đối tượng từ một lớp, chúng ta dùng từ khóa **new**. Ví dụ sau sẽ tạo hai đối tượng **c1** và **c2** của lớp **Circle** trong phương thức main của lớp **MyMainClass**:

```
public class MyMainClass {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Circle c1 = new Circle();
        Circle c2 = new Circle();
    }
}
```

Đề ý sau từ khóa **new** là **Circle()**, đây là phương thức khởi tạo mặc định của lớp **Circle**. Phương thức khởi tạo sẽ được đề cập kỹ hơn trong phần phương thức.

Chúng ta có thể truy cập đến các biến thành viên của lớp thông qua đối tượng bằng dấu chấm. Ví dụ truy cập và gán các giá trị đến biến *radius* của đối tượng **c1** và **c2** như sau:

```
c1.radius = 15.0f;
c2.radius = 30.0f;
```

vì *radius* có phạm vi truy cập là **public** nên chúng ta có thể truy cập từ lớp **MyMainClass**. Bây giờ chúng ta thử truy cập đến các biến thành viên còn lại:

```
c1.lineWidth = 3.5f; // ok
c2.lineWidth = 5.6f; // ok
c1.name = "firstobj"; // lỗi
c2.name = "secondobj"; // lỗi
```

Có thể truy cập đến *lineWidth* vì biến này có phạm vi là **protected** nên có thể được truy cập từ lớp **MyMainClass** – lớp cùng gói với lớp **Circle**.

Truy cập đến biến *name* sẽ bị lỗi vì phạm vi truy cập của biến này là **private**.

Phương thức (method)

Bên cạnh các biến thành viên, một thành phần quan trọng khác của một lớp là phương thức hay các hàm. Phương thức gồm hai kiểu: kiểu có giá trị trả về và kiểu không có giá trị trả về. Giống như biến, phương thức cũng có các phạm vi truy cập như **public**, **private**, hay **protected**. Phương thức cũng có thể là **static**.

Phương thức có giá trị trả về có cú pháp như sau:

```
[public|private|protected] [static] Datatype method_name (params list)
{
    ...
    return...;
}
```

Cần chú ý:

- Phương thức có kiểu dữ liệu trả về (**Datatype**) như kiểu *int*, *float*, *double*, v.v.
- Lệnh kết thúc trong phần thân phương thức là lệnh **return**. Theo sau **return** có thể là một giá trị, một biến, hay một biểu thức và tất cả phải có kiểu dữ liệu là **Datatype** – cùng kiểu với khai báo phương thức.
- **params list** là danh sách chứa các tham số. Danh sách có thể rỗng hay chứa nhiều tham số. Nếu chứa nhiều tham số thì các tham số cách nhau bởi dấu phẩy.

Ví dụ phương thức sau:

```
public int add (int a, int b) {
    return (a+b);
}
```

Phương thức trên có phạm vi truy cập là **public**; kiểu trả về là **int**; hai tham số trong danh sách tham số là **a** và **b**. Giá trị trả về của biểu thức **(a+b)** có kiểu **int**.

Nếu viết như sau trình biên dịch Java sẽ phát sinh lỗi:

```
public int add (double a, double b) {
    return (a+b);
}
```

a và **b** có kiểu dữ liệu là **double** nên **(a+b)** sẽ trả về giá trị có kiểu **double** khác kiểu **int** nên sẽ phát sinh lỗi. Cần thận trọng khi khai báo phương thức có giá trị trả về.

Phương thức không có giá trị trả về có cú pháp sau:

```
[public|private|protected] [static] void method_name (params list)
{
    ...
}
```

Có hai điểm khác biệt so với phương thức có giá trị trả về là kiểu dữ liệu được thay bằng từ khóa **void** và không có lệnh **return** tại cuối phần thân phương thức. Ví dụ phương thức **add** có thể được viết lại như sau:

```
public void add (double a, double b) {
    double c = a + b;
}
```

```

        System.out.println(c);
    }

```

Phương thức khởi tạo (constructor)

Phương thức mà mọi lớp đều có là phương thức khởi tạo (constructor). Phương thức khởi tạo là một phương thức đặc biệt:

- Tên phương thức trùng với tên lớp
- Không có kiểu trả về hay từ khóa **void**
- Phương thức khởi tạo mặc định là phương thức khởi tạo không có tham số trong danh sách tham số.

Để dễ hình dung, chúng ta sẽ khai báo các phương thức và phương thức khởi tạo cho lớp **Circle** như sau:

```

public class Circle {
    private float radius;
    private String name;
    private float lineWidth;
    private static int circleCount;

    // phương thức khởi tạo mặc định
    public Circle() {
        this.radius = 0.0f;
        this.name = "No name";
        this.lineWidth = 0.0f;
        circleCount++;
    }
    // Phương thức khởi tạo có tham số
    public Circle(String name, float radius) {
        this.radius = radius;
        this.name = name;
        this.lineWidth = 0.0f;
        circleCount++;
    }
    // các phương thức
    public void setRadius(float radius) {
        this.radius = radius;
    }
    public float getRadius() {
        return radius;
    }
    public void print(){
        System.out.println("Circle: " + name + " Rad: " + radius);
    }
}

```

```

protected void setLineWidth(float newWidth) {
    this.lineWidth = newWidth;
}
public static void zeroCount() {
    circleCount = 0;
}
public static int getCount() {
    return circleCount;
}
}

```

Có một vài điểm quan trọng cần chú ý ở đây:

- Các biến thành viên trong một lớp nên có phạm vi truy cập là **private** (hay có thể là **protected**) vì đây là thành phần quan trọng của lớp và chúng ta không muốn lớp khác truy cập trực tiếp. Các biến thành viên trong Java hay còn được gọi là các *fields* (tương đương với *fields* trong C#).
- Vì các biến thành viên có phạm vi là **private** nên các lớp khác muốn truy cập các biến này phải thông qua các phương thức. Các phương thức này được gọi là các *setter* hay *getter* trong C#, *property* trong Visual Basic.Net. Ví dụ để gán giá trị đến biến **radius** chúng ta dùng phương thức **setRadius**, để lấy giá trị của biến **radius** chúng ta dùng phương thức **getRadius**.
- Các tham số trong phương thức còn được gọi là các biến cục bộ (local variables). Các biến này chỉ có thể được sử dụng trong phạm vi phương thức. Biến cục bộ có thể trùng tên và kiểu với tên biến thành viên và phương thức sẽ ưu tiên sử dụng biến cục bộ. Do đó, trong một phương thức có biến cục bộ trùng tên và kiểu với biến thành viên, để phân biệt, chúng ta dùng từ khóa **this**. **this** dùng để chỉ đối tượng hiện tại tham chiếu đến biến thành viên và truy cập đến biến thành viên bằng dấu chấm. Ví dụ phương thức **setRadius** có biến cục bộ là **radius** (trong danh sách tham số) và giá trị của biến này được gán đến biến thành viên **radius** thông qua từ khóa **this**:

```
this.radius = radius;
```

- Trong một lớp có thể có nhiều phương thức khởi tạo. Các phương thức khởi tạo có thể được định nghĩa khác nhau bởi số tham số trong danh sách tham số hay có cùng số tham số nhưng có kiểu dữ liệu khác nhau. Quá trình này gọi là quá tải phương thức khởi tạo (constructor overloading). Trong lớp **Circle** có hai phương thức khởi tạo, một phương thức khởi tạo mặc định không có tham số và một phương thức khởi tạo chứa hai tham số là **name** và **radius**. Quá tải cũng có thể áp dụng cho các phương thức nói chung. Ví dụ chúng ta có thể tạo hai phương thức **setRadius** như sau:

```
public void setRadius(float radius) {
```

```
...
}

public void setRadius(int radius) {
...
}
```

- Biến **static** chỉ có thể được truy cập bởi các phương thức **static** và có thể truy cập trực tiếp thay vì dùng **this**. Ví dụ biến **circleCount** được sử dụng trong các phương thức **static** là **zeroCount** hay **getCount**.

Gọi phương thức

Các phương thức có thể được truy cập thông qua các đối tượng hay lớp (phương thức **static**) như đoạn mã sau:

```
public class MyMainClass {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        // tạo đối tượng c dùng constructor mặc định
        Circle c = new Circle();
        // tạo đối tượng b dùng constructor có hai tham số
        Circle b = new Circle("Dennis", 55.0f);
        // gọi phương thức static getCount
        System.out.println("Number of circles created: " +
            Circle.getCount());
        b.print(); // gọi phương thức print của b
        c.setRadius(27.0f); // gọi setRadius để gán c.radius đến 27.0f.
        c.setLineWidth(100.6f); // gán c.lineWidth đến 100.6f.
        System.out.println("Radius of C: " + c.getRadius());
        Circle.zeroCount(); // gọi phương thức static zeroCount
        System.out.println(
            "Number of circles created: " + Circle.getCount());
    }

}
```

Các thành phần static

Thành viên static

Các phương thức hay biến thành viên của một lớp có phạm vi truy cập là **public** và có kiểu **static** có thể được truy cập thông qua các thể hiện (đối tượng) hay tên lớp.

Ví dụ các phương thức **getCount()** hay **zeroCount()** là các phương thức **static**. Các phương thức này có thể được truy cập bằng tên lớp:

```
Circle.getCount();
```

hay thông qua một thể hiện:

```
c.getCount();
```

Biến **circleCount** là biến **static** của lớp **Circle**. Mỗi biến **static** được Java cấp phát bộ nhớ chỉ một lần và tất cả các đối tượng của một lớp sẽ tham chiếu đến cùng một không gian bộ nhớ của biến đó. Biến **static** tồn tại ngay cả khi không có đối tượng nào được khởi tạo và luôn được khởi tạo đến một giá trị mặc định.

Lớp static

Java cho phép chúng ta định nghĩa các lớp lồng nhau – tức là định nghĩa một lớp (inner class) trong một lớp khác (outer class). Một lớp được định nghĩa bên trong một lớp khác có thể là lớp **static** (static inner class). *Như vậy, khi chúng ta đề cập đến một lớp static tức là chúng ta đang đề cập đến một lớp được định nghĩa bên trong một lớp khác.*

Một lớp được định nghĩa bên trong một lớp khác có thể là lớp **static** (static inner class) hay một lớp bình thường (non-static inner class). Hai kiểu lớp này có một số điểm khác biệt:

- non-static inner class yêu cầu một tham chiếu của outer class trong khi static inner class không cần.
- non-static inner class có thể truy cập đến các thành viên (biến hay phương thức) static và không static của outer class. static inner class chỉ có thể truy cập đến các thành viên static của outer class.
- Thể hiện của non-static inner class được tạo ra thông qua thể hiện của outer class.

Một ví dụ về các lớp lồng nhau:

```
class OuterClass{
    private static String msg = "ngocminhtran";

    public static class StaticInnerClass{

        public void printMessage() {

            System.out.println("Message from nested static class: "
+ msg);
        }
    }

    public class NonStaticInnerClass{

        public void display(){
            System.out.println("Message from non-static nested
class: "+ msg);
```

```

    }
}
}

```

msg là biến thành viên của lớp **OuterClass**. Lớp **StaticInnerClass** có thể truy cập được *msg* vì *msg* là thành viên *static*. Lớp **NonStaticInnerClass** có thể truy cập được *msg* vì lớp này có thể truy cập được mọi thành viên static và non-static của lớp **OuterClass**. Các lớp này có thể được sử dụng như sau:

```

// tạo thể hiện của lớp StaticInnerClass
OuterClass.StaticInnerClass printer = new OuterClass.StaticInnerClass ();

printer.printMessage();

//tạo thể hiện của lớp NonStaticInnerClass thông qua thể hiện của OuterClass
// bằng hai dòng mã
OuterClass outer = new OuterClass();
OuterClass.NonStaticInnerClass inner = outer.new NonStaticInnerClass();

inner.display();

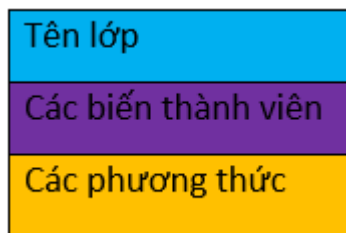
//tạo thể hiện của lớp NonStaticInnerClass thông qua thể hiện của OuterClass
// chỉ bằng một dòng mã
OuterClass.NonStaticInnerClass innerObject = new OuterClass().new
NonStaticInnerClass();

innerObject.display();

```

Lược đồ UML (Unified Model Language)

Một lớp có thể được biểu diễn bằng lược đồ UML. Một lược đồ UML biểu diễn cho một lớp có thể được chia thành 3 phần như sau:



Ví dụ lớp **Circle** có thể được biểu diễn bằng lược đồ UML như sau:

Circle
<ul style="list-style-type: none"> - radius: float - name: String - lineWidth: float - circleCount: int
<ul style="list-style-type: none"> + Circle() + Circle(String, float) + setRadius(float): void + getRadius(): float + print(): void + setLineWidth(float): void + zeroCount(): void + getCount():int

Màu sắc của mỗi phần là tùy ý. Tên lớp ở giữa phần đầu tiên, các biến bắt đầu bằng dấu “-”, các phương thức bắt đầu bằng dấu “+”.

Thừa kế (Inheritance)

Thừa kế là một trong những đặc điểm quan trọng nhất của lập trình hướng đối tượng. Thừa kế giúp chúng ta tiết kiệm thời gian, viết ít mã, và tránh lặp lại các đoạn mã. Thừa kế tạo ra một cấu trúc các lớp dựa theo mối quan hệ cha/con (parent/child) bằng cách kế thừa lại các đặc điểm từ các lớp (lớp cha) có sẵn và mở rộng thêm các đặc điểm trong các lớp mới (lớp con). Ví dụ chúng ta có lớp **Animal** như sau:

```
public class Animal {
    public boolean voluntaryMotion;
    public boolean requiresFood;
    public Animal(){
        voluntaryMotion = true;
        requiresFood = true;
    }
}
```

Lớp **Insect** thừa kế các thành viên có phạm vi truy cập là *public* hay *protected* từ lớp **Animal** và bổ sung thêm một số thành viên mới như sau:

```
public class Insect extends Animal {
    public boolean antennas;
    public String skeleton;
    public int numberOfLegs;
    public boolean wings;
    public Insect(){
        super();
    }
}
```

```

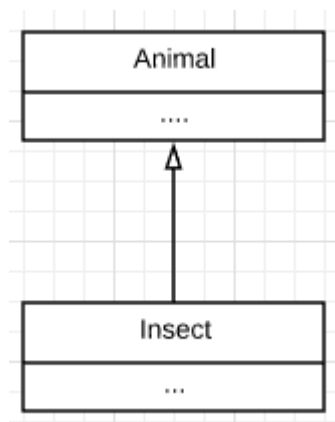
        antennas = true;
        skeleton = "Exoskeleton";
        numberOfLegs = 6;
        wings = true;
    }
    public void Print() {
        System.out.println("Antennas: " + antennas +
            " Skeleton: " + skeleton +
            " Number of Legs: " + numberOfLegs +
            " Wings: " + wings +
            " Voluntary Motion: " + voluntaryMotion +
            " Requires Food: " + requiresFood);
    }
}

```

Có một số chú ý trong định nghĩa lớp **Insect**:

- Lớp **Insect** thừa kế từ lớp **Animal** bằng cách dùng từ khóa *extends*
- Phương thức *super()* gọi phương thức constructor mặc định (không có tham số) từ lớp cha (Animal). *super()*, nếu được gọi, phải là dòng đầu tiên trong phương thức constructor của lớp con.
- **Insect** chỉ kế thừa các thành phần *public* hay *protected* từ lớp **Animal**.

Chúng ta có thể biểu diễn hai lớp có mối quan hệ cha/con bằng lược đồ UML sử dụng ký hiệu mũi tên:



Sử dụng lớp **Insect**:

```

public class MyMainClass {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        // Create an instance of Insect called i.
    }
}

```

```

    Insect i = new Insect();
    // Change the insect class's members:
    i.antennas = false;
    i.numberOfLegs = 100;
    // Call the Insect's Print method:
    i.Print();
    // Change the parent class's requiresFood member:
    i.requiresFood = false;
}
}

```

Đa hình (Polymorphism)

Bên cạnh tính năng thừa kế, đa hình là một trong những tính năng quan trọng nhất của lập trình hướng đối tượng. Với khả năng đa hình, các lớp con (child classes) có thể sử dụng lại các phương thức được thừa kế từ lớp cha (parent classes) theo những cách khác nhau. Chúng ta có thể sử dụng tính năng đa hình thông qua các lớp trừu tượng (abstract classes) hay các giao diện (interfaces).

Các lớp trừu tượng (abstract classes)

Giả sử chúng ta có lớp **Person** với phương thức thành viên là **Talk()**. Hai lớp **Student** và **Employee** thừa kế từ lớp **Person** với phương thức **Talk()** được định nghĩa lại phù hợp với từng đối tượng **Student** hay **Employee**. Để thực hiện được điều này, chúng ta có thể khai báo lớp cha **Person** là một lớp trừu tượng bằng cách dùng từ khóa **abstract**:

```

//Abstract parent class:
public abstract class Person {
    //Abstract method:
    public abstract void Talk();
}

```

Lưu ý rằng, từ khóa **abstract** được sử dụng trong khai báo lớp **Person** và khai báo phương thức **Talk()**. Không thể tạo thể hiện (hay đối tượng) từ một lớp trừu tượng. Các phương thức trong một lớp cha trừu tượng có thể được sử dụng lại trong các lớp con theo hai cách: phương thức trừu tượng và ghi đè phương thức.

Phương thức trừu tượng (abstract methods)

Phương thức **Talk()** được gọi là phương thức trừu tượng. Một phương thức trừu tượng không có phần thân phương thức.

Lớp **Student** thừa kế từ lớp **Person** có thể được định nghĩa như sau:

```
public class Student extends Person {
    public void Talk() {
        System.out.println("I am a Student!");
    }
}
```

Lớp **Employee**:

```
public class Employee extends Person {
    public void Talk() {
        System.out.println("I am an Employee!");
    }
}
```

Phương thức **Talk()** lúc này được định nghĩa lại phù hợp với từng đối tượng. Chúng ta có thể tạo các thể hiện **Student** hay **Employee** như sau:

```
Person student = new Student();
student.Talk();// I am a Student!
Person employee = new Employee();
employee.Talk();// I am an Employee!
```

Ghi đề phương thức (overriding methods)

Lớp trừu tượng, giống các lớp bình thường khác, cũng có thể chứa các biến hay phương thức thành viên. Lớp **Person** có thể khai báo lại như sau:

```
//Abstract parent class:
public abstract class Person {
    //Member variables:
    String name;
    int age;
    // Member method
    public void Intro() {
        System.out.println("My name is " + name + " and I am " + age + "
years old.");
    }
    //Abstract method:
    public abstract void Talk();
}
```

Các lớp **Student** và **Employee**, tất nhiên, cũng sẽ kế thừa các thành viên này:

```
Person student = new Student();
student.Talk();
student.Intro();// My name is...
```

Phương thức **Intro()** của lớp **Person** có thể được định nghĩa lại phù hợp cho các đối tượng **Student** và **Employee** bằng cách dùng kỹ thuật “ghi đè” phương thức. Đây là kỹ thuật đa hình khác bên cạnh dùng phương thức trừu tượng, ví dụ **Talk()**. Ghi đè phương thức **Intro()** trong lớp **Student**:

```
public class Student extends Person {
    public void Talk() {
        System.out.println("I am a Student!");
    }
    @Override
    public void Intro() {
        System.out.println("I love reading!");
    }
}
```

Lưu ý rằng, phía trên phương thức **Intro()** có dùng chú giải **@Override**. Chú giải này không bắt buộc nhưng nên sử dụng để làm cho đoạn mã trong sáng và dễ đọc hơn.

Lúc này, dùng phương thức **Intro()** thông qua thể hiện lớp **Student**:

```
Person student = new Student();
student.Talk();
student.Intro();// I love reading!
```

Phương thức khởi tạo (constructors)

Mặc dù chúng ta không được phép tạo các thể hiện (hay đối tượng) từ lớp trừu tượng, nhưng lớp trừu tượng cho phép chúng ta định nghĩa các phương thức khởi tạo. Phương thức khởi tạo mặc định trong lớp **Person**:

```
//Abstract parent class:
public abstract class Person {
    //Member variables:
    String name;
    int age;
    //Default constructor
    public Person() {
        name = "Minh";
        age = 18;
    }
    // Member method
    public void Intro() {
        System.out.println("My name is " + name + " and I am " + age
+ " years old.");
    }
    //Abstract method:
```

```

    public abstract void Talk();
}

```

Giống như ghi đè các phương thức, các phương thức khởi tạo cũng có thể được định nghĩa lại tại các lớp con. Phương thức khởi tạo của lớp **Student**:

```

public class Student extends Person {
    //constructor
    public Student() {
        name = "ngocminhtran";
        age = 30;
    }
    public void Talk() {
        System.out.println("I am a Student!");
    }

    @Override
    public void Intro() {
        System.out.println("I love reading!");
    }
}

```

Có thể sử dụng phương thức **super()** trong phương thức khởi tạo của lớp con nếu muốn gọi phương thức khởi tạo từ lớp cha. Chú ý rằng, lệnh gọi **super()** luôn được đặt đầu tiên trong phương thức khởi tạo của lớp con và có thể gọi theo nhiều phiên bản khác nhau của phương thức khởi tạo. Ví dụ lớp **Person** có hai phiên bản phương thức khởi tạo như sau:

```

//Abstract parent class:
public abstract class Person {
    //Member variables:
    String name;
    int age;
    //Default constructor
    public Person() {
        name = "Minh";
        age = 18;
    }
    //Another constructor
    public Person(String name, int age) {
        this.name = name ;
        this.age = age;
    }
    // Member method
    public void Intro() {
        System.out.println("My name is " + name + " and I am " + age
+ " years old.");
    }
}

```

```

    }
    //Abstract method:
    public abstract void Talk();
}

```

Gọi từ phương thức khởi tạo của lớp **Student**:

```

public class Student extends Person {
    //constructor
    public Student() {
        //super();
        //super("Minh",18);
    }
    ...
}

```

Chỉ được gọi một trong hai phiên bản *super()* vì lệnh *super()* luôn là lệnh đầu tiên.

Từ khóa instanceof

Từ khóa instanceof hữu ích khi chúng ta cần kiểm tra một thể hiện (hay đối tượng) thuộc kiểu lớp nào. Ví dụ:

```

Person obj = new Student();

if(obj instanceof Person)
    System.out.println("I am a Person.");
else
    System.out.println("I am not a Person.");

if(obj instanceof Employee)
    System.out.println("I am an Employee.");
else
    System.out.println("I am not an Employee.");

```

Kết quả:

```

I am a Person.
I am not an Employee.

```

Giao diện (interface)

Một giao diện tương tự một lớp trừu tượng nhưng chỉ chứa các phương thức trừu tượng hay các hằng (constants). Kể từ **Java 8**, giao diện chứa thêm các phương thức static (static methods) và các phương thức mặc định (default methods) (tìm hiểu thêm tại <https://dzone.com/articles/interface-enhancement-in-java8>). Khác với các phương thức trừu tượng, các phương thức static và phương thức mặc định có phần thân phương thức.

Phương thức mặc định có thể được ghi đè trong lớp thực thi còn phương thức static thì không thể.

Các lớp thực thi giao diện sẽ thừa kế các thành phần từ giao diện. Trừ khi là một lớp trừu tượng, tất cả các phương thức trong giao diện cần được định nghĩa trong lớp thực thi giao diện đó.

Giao diện được định nghĩa bằng từ khóa ***interface***. Ví dụ giao diện **iAnimal** như sau:

```
public interface iAnimal {
    public void eat();
    public void move();
}
```

Một lớp sẽ thực thi giao diện với từ khóa ***implements***. Ví dụ lớp **MammalInt** thực thi giao diện **iAnimal** như sau:

```
public class MammalInt implements iAnimal {

    public void eat() {
        System.out.println("Mammal eats");
    }

    public void move() {
        System.out.println("Mammal moves");
    }
}
```

Giống như lớp, giao diện có thể được mở rộng (hay thừa kế) sang các lớp khác bằng từ khóa ***extends***. Ví dụ:

```
//Filename: Sports.java
public interface Sports {
    public void setHomeTeam(String name);
    public void setVisitingTeam(String name);
}

//Filename: Football.java
public interface Football extends Sports {
    public void homeTeamScored(int points);
    public void visitingTeamScored(int points);
    public void endOfQuarter(int quarter);
}

//Filename: Hockey.java
public interface Hockey extends Sports {
```



```

public void homeGoalScored();
public void visitingGoalScored();
public void endOfPeriod(int period);
public void overtimePeriod(int ot);
}

```

Java chỉ cho phép thừa kế đơn với các lớp, nghĩa là một lớp chỉ được thừa kế từ một lớp cha. Tuy nhiên, với giao diện chúng ta có thể thực hiện đa thừa kế như sau:

```

public interface Hockey extends Sports, Event

```

Các lớp nặc danh (anonymous classes)

Lớp nặc danh là lớp không có tên và thường được sử dụng khi cần khai báo một lần tại một vị trí nào đó trong đoạn mã. Để dùng lớp nặc danh, chúng ta cần thực thi một giao diện hay mở rộng một lớp có sẵn. Ví dụ sau định nghĩa một lớp nặc danh:

```

public class MainClass {
    // Lớp cha
    static class OutputLyrics {
        public void output() {
            System.out.println("No lyrics supplied...");
        }
    }

    public static void main(String[] args) {
        //tạo một thể hiện từ lớp OutputLyrics.
        OutputLyrics regularInstance = new OutputLyrics();
        // tạo một lớp nặc danh thừa kế từ lớp OutputLyrics
        OutputLyrics anonymousClass = new OutputLyrics() {
            public void output() {
                System.out.println(
                    "Desmond has a barrow in the market place.");
            }
        };
        // gọi output dùng thể hiện của OutputLyrics
        regularInstance.output();
        // gọi output dùng thể hiện lớp nặc danh
        anonymousClass.output();
    }
}

```

Lớp nặc danh có thể được dùng như một tham số của một phương thức. Ví dụ sau minh họa một lớp nặc danh thực thi một giao diện (*MathOperation*) đóng vai trò như một tham số của phương thức (*performOperation*):

```

public class MainClass {
    // interface
    interface MathOperation {

```

```

        public int operation(int a, int b);
    }
    // Phương thức nhận một đối tượng như một tham số
    static int performOperation(int a, int b, MathOperation op) {
        return op.operation(a, b);
    }
    public static void main(String[] args) {

        int x = 100;
        int y = 97;
        // Gọi hàm PerformOperation với phép cộng:
        int resultOfAddition = performOperation(x, y,
            // lớp nặc danh như là một tham số
            new MathOperation() {
                public int operation(int a, int b) {
                    return a + b;
                }
            });
        // Gọi hàm PerformOperation phép trừ:
        int resultOfSubtraction = performOperation(x, y,
            // lớp nặc danh như là một tham số
            new MathOperation() {
                public int operation(int a, int b) {
                    return a - b;
                }
            });
        // Output Addition: 197
        System.out.println("Addition: " + resultOfAddition);
        // Output Subtraction: 3
        System.out.println("Subtraction: " + resultOfSubtraction);
    }
}

```

Chương VII

Mảng (array)

Kiểu tham trị và kiểu tham chiếu

Khi chúng ta khai báo một biến kiểu tham trị (hầu hết các kiểu dữ liệu sơ cấp như kiểu **int**, **double**, **char**, v.v. đều là kiểu tham trị ngoại trừ kiểu **String**), trình biên dịch sẽ phát sinh mã để cấp phát một vùng nhớ đủ lớn để chứa giá trị tương ứng. Ví dụ khai báo biến **x** kiểu **int**:

```
int x;
```

Trình biên dịch sẽ phát sinh mã cấp phát biến **x** một vùng nhớ kích cỡ 4 byte (32 bit). Nếu gán **x** một giá trị:

```
x = 5;
```

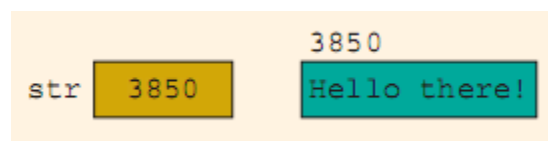
giá trị 5 sẽ được đặt trong vùng nhớ vừa được cấp phát:



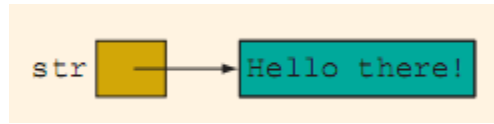
Khi chúng ta khai báo một biến kiểu tham chiếu (như kiểu **String**, **array**, **class**,...) thì trình biên dịch phát sinh mã để cấp phát một vùng nhớ đủ lớn để chứa địa chỉ của vùng nhớ chứa giá trị thực của biến. Ví dụ chúng ta khai báo biến kiểu **string**:

```
String str;  
str = "Hello there!";
```

lúc này sẽ có hai vùng nhớ, vùng nhớ chứa địa chỉ của vùng nhớ chứa giá trị **Hello there!** (ví dụ **3850**) và vùng nhớ chứa giá trị **Hello there!**:



Nếu hình dung khái niệm này tương tự khái niệm con trỏ trong C, chúng ta có thể trực quan như sau:



Bây giờ chúng ta sẽ khai báo thêm biến **y** kiểu **int** và gán **x** đến **y**:

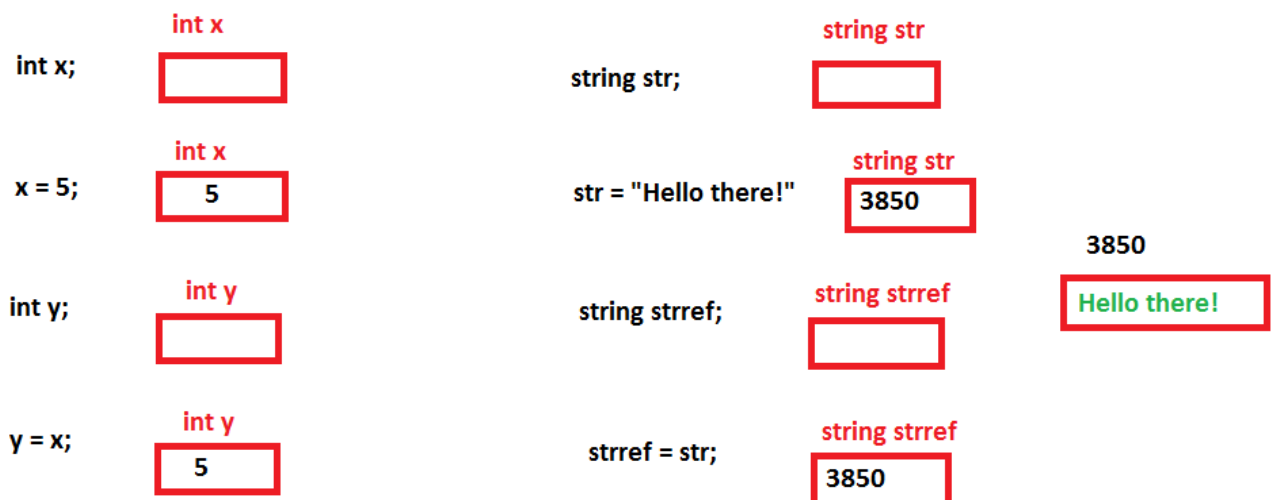
```
int x;
x = 5;
int y;
y = x;
x++;
```

lúc này biến **y** chứa một bản sao giá trị của biến **x**, những thay đổi của biến **x** (**x++**) sẽ không ảnh hưởng đến biến **y**.

Chúng ta cũng khai báo thêm biến **strref** kiểu **String** như sau:

```
String str;
str = "Hello there!";
String strref;
strref = str;
```

Lúc này hai vùng nhớ chứa địa chỉ của **strref** và **str** sẽ chứa cùng địa chỉ - là địa chỉ của vùng nhớ chứa **Hello there!**. Có thể hình dung về kiểu tham chiếu và tham trị từ các ví dụ trên một cách trực quan như sau:



Kiểu mảng

Mảng là một danh sách chứa các phần tử có cùng kiểu dữ liệu. Kiểu mảng là kiểu tham chiếu. Mảng có thể có một chiều hay nhiều chiều.

Mảng một chiều

Mảng một chiều là một danh sách chứa các phần tử cùng kiểu như các số nguyên, chuỗi,... Mỗi phần tử trong mảng được xác định tại một vị trí được gán chỉ số (index). Phần tử ở vị trí đầu tiên sẽ có chỉ số là 0, phần tử cuối cùng trong mảng có n phần tử có chỉ số là $n - 1$. Khai báo, truy cập (gán giá trị, hiển thị,...) các phần tử mảng dùng dấu ngoặc vuông.

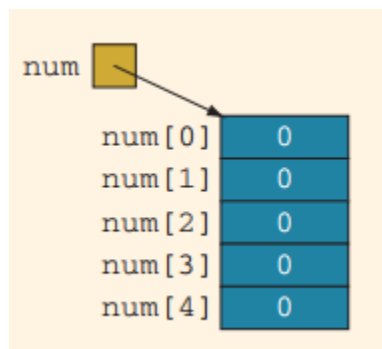
Cú pháp

```
kiểu_dữ_liệu[] tên_mảng = new kiểu_dữ_liệu[kích_cỡ_hay_số_phần_tử_trong_mảng];
```

Ví dụ khai báo mảng num chứa 5 số nguyên:

```
int[] num = new int [5];
```

Có thể hình dung mảng num một cách trực quan như sau:



Gán giá trị cho các phần tử trong mảng ta dùng cú pháp

```
tên_mảng [chỉ_số_của_phần_tử] = giá_trị;
```

Ví dụ gán các giá trị số nguyên cho hai phần tử đầu tiên trong mảng num như sau:

```
num [0] = 7; // Gán giá trị 7 cho phần tử thứ nhất trong mảng
num [1] = 19; // Gán 19 cho phần tử thứ hai
```

Có thể gán các phần tử khi khai báo mảng, ví dụ:

```
int[] num = {7,9};
```

Để truy cập phần tử trong mảng ta dùng cú pháp

```
tên_mảng [chỉ_số_phần_tử];
```

Để trả về kích cỡ của mảng, chúng ta dùng thuộc tính **length**

```
int size = arrInt.length; // size = 5
```

Duyệt mảng

Chúng ta có thể duyệt qua các phần tử của một mảng nhờ các lệnh lặp. Ví dụ dùng lệnh **for** để hiển thị tất cả các phần tử của mảng *arrInt* như sau:

```
int[] arrInt = {2,4,8,9};
for (int i = 0; i < arrInt.length; i++) {
    int val = arrInt[i];
    System.out.print(val + " ");
}
```

Cũng có thể dùng lệnh **foreach**:

```
int[] arrInt = {2,4,8,9};
for (int val: arrInt)
    System.out.print(val + " ");
```

Kết quả: 2 4 8 9

Mảng như là tham số của phương thức

Mảng có thể được sử dụng như tham số của phương thức, ví dụ:

```
public class MyMainClass {

    public static void main(String[] args) {

        int[] arrInt = {2,4,8,9};
        printArray(arrInt);

    }

    public static void printArray(int[] array) {
        for (int i = 0; i < array.length; i++) {
            System.out.print(array[i] + " ");
        }
    }

}
```

Mảng nhiều chiều

Cho đến thời điểm này chúng ta mới chỉ dừng lại ở mảng một chiều nhưng chúng ta có thể tạo ra mảng hơn một chiều. Có thể hình dung mảng một chiều chỉ là một dãy các phần tử, mảng hai chiều là một bảng các phần tử gồm các hàng và cột, mảng 3 chiều là một hình hộp, v.v.

Nếu khai báo mảng hai chiều, chúng ta có thể dùng cú pháp:

kiểu dữ liệu `[][]` biến mảng = `new kiểu dữ liệu[số hàng][số cột];`

Ví dụ sau khai báo và khởi tạo các giá trị cho mảng số thực (kiểu *float*) hai chiều **arr2D** có kích thước 7x5 như sau:

```
float[][] arr2D = new float[7][5];
arr2D[0][0] = 50.5f;
arr2D[4][4] = 60.5f;
arr2D[3][1] = 10.3f;
```

Có thể hình dung trực quan mảng **arr2D** như sau:

	0	1	2	3	4
0	50.5	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	0	10.3	0	0	0
4	0	0	0	0	60.5
5	0	0	0	0	0
6	0	0	0	0	0

Đoạn mã sau sẽ hiển thị mảng **arr2D** ra màn hình:

```
for (int i = 0; i < 7; i++) {
    for (int j = 0; j < 5; j++)
        System.out.print(array2d[i][j] + " ");
    System.out.println();
}
```

Kết quả:

```
50.5 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
0.0 10.3 0.0 0.0 0.0
```

```
0.0 0.0 0.0 0.0 60.5  
0.0 0.0 0.0 0.0 0.0  
0.0 0.0 0.0 0.0 0.0
```

Bộ não chúng ta chỉ có thể hình dung được mảng có số chiều tối đa là 3 (vì chúng ta đang sống trong một thế giới 3 chiều). Với các mảng có số chiều lớn hơn 3, thật khó để hình dung một cách trực quan nhưng chúng ta có thể thao tác với các mảng này như đoạn mã minh họa mảng 4 chiều sau:

```
// khai báo mảng 4 chiều  
int[][][][] arr4D = new int[3][4][5][6];  
// gán tất cả các phần tử của mảng đến giá trị 1729  
for(int a = 0; a < 3; a++) {  
    for(int b = 0; b < 4; b++) {  
        for(int c = 0; c < 5; c++) {  
            for(int d = 0; d < 6; d++) {  
                arr4D[a][b][c][d] = 1729;  
            }  
        }  
    }  
}
```


Chương VIII

Các gói (packages)

Các gói (packages)

Như trong các ví dụ ở chương I và II chúng ta đã làm quen với khái niệm gói (package) trong **Java**. Gói là một cách tổ chức các lớp thành các nhóm. Khái niệm gói tương tự khái niệm thư mục (folder) trong **Windows**. Một thư mục có thể chứa nhiều tập tin tương tự một gói chứa nhiều lớp. Hai tập tin có thể có cùng tên nhưng phải thuộc về hai thư mục khác nhau; tương tự, hai lớp có thể cùng tên nhưng phải thuộc về những gói khác nhau.

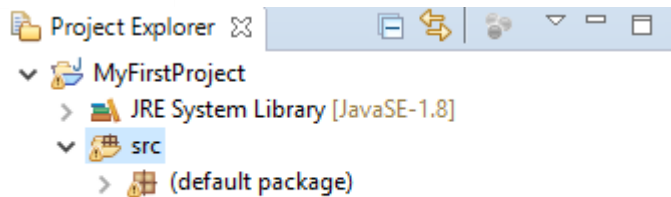
Gói mặc định (default package)

Khi một dự án được tạo, một gói mặc định (default package) cũng được tạo ra. Nếu người dùng thêm một lớp đến dự án, sẽ có một yêu cầu nhập tên gói chứa lớp đó:

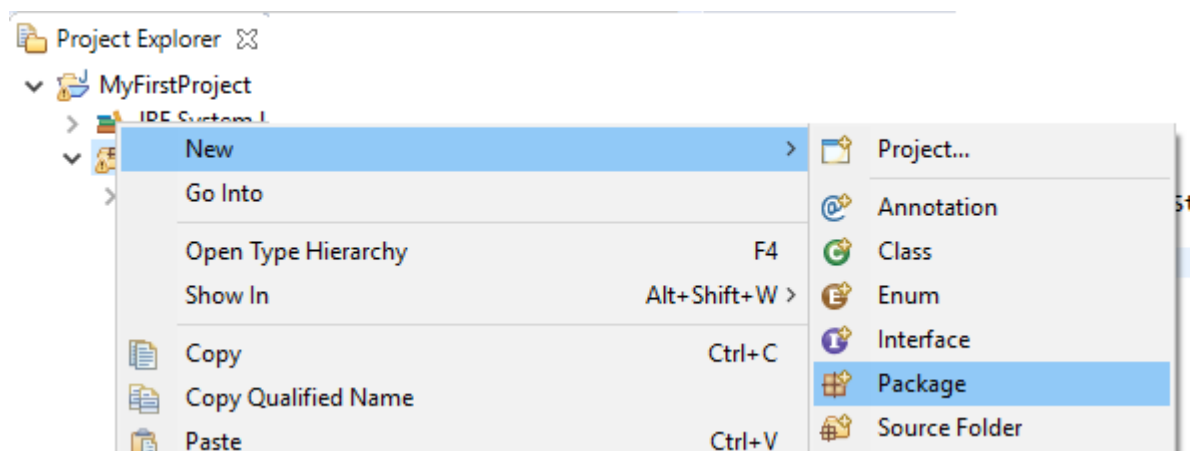
The screenshot shows the 'New Java Class' dialog box. The 'Package' field is highlighted with a red rectangle and contains the text '(default)'. The 'Source folder' field contains 'MyFirstProject/src'. The 'Name' field is empty. The 'Modifiers' section has radio buttons for 'public' (selected), 'package', 'private', and 'protected', and checkboxes for 'abstract', 'final', and 'static'. The 'Superclass' field contains 'java.lang.Object'. The 'Interfaces' field is empty. There are 'Browse...', 'Add...', and 'Remove' buttons on the right side of the dialog.

Thêm các gói đến dự án

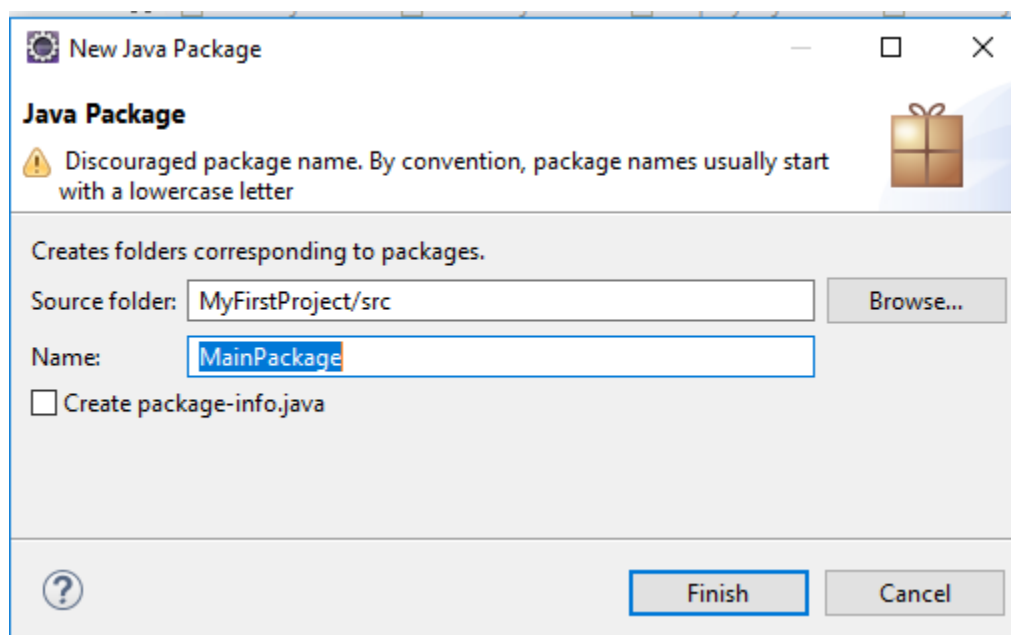
Như đã đề cập ở trên, khi tạo một dự án mới, một gói mặc định sẽ được thêm vào dự án, cụ thể là thêm vào thư mục **src** của dự án:



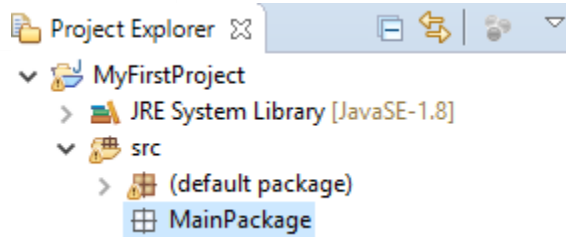
Để thêm một gói đến dự án **Java**, chúng ta nhấp chuột phải vào thư mục **src** trong cửa sổ **Project Explorer** chọn **New > Package**



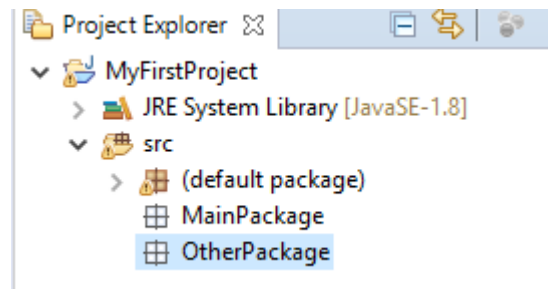
Nhập tên gói, ví dụ **MainPackage**, trong hộp thoại **New Java Package**:



Nhấn **Finish** và thư mục *src* lúc này:



Tương tự thêm gói **OtherPackage**:



Thêm các lớp đến một gói

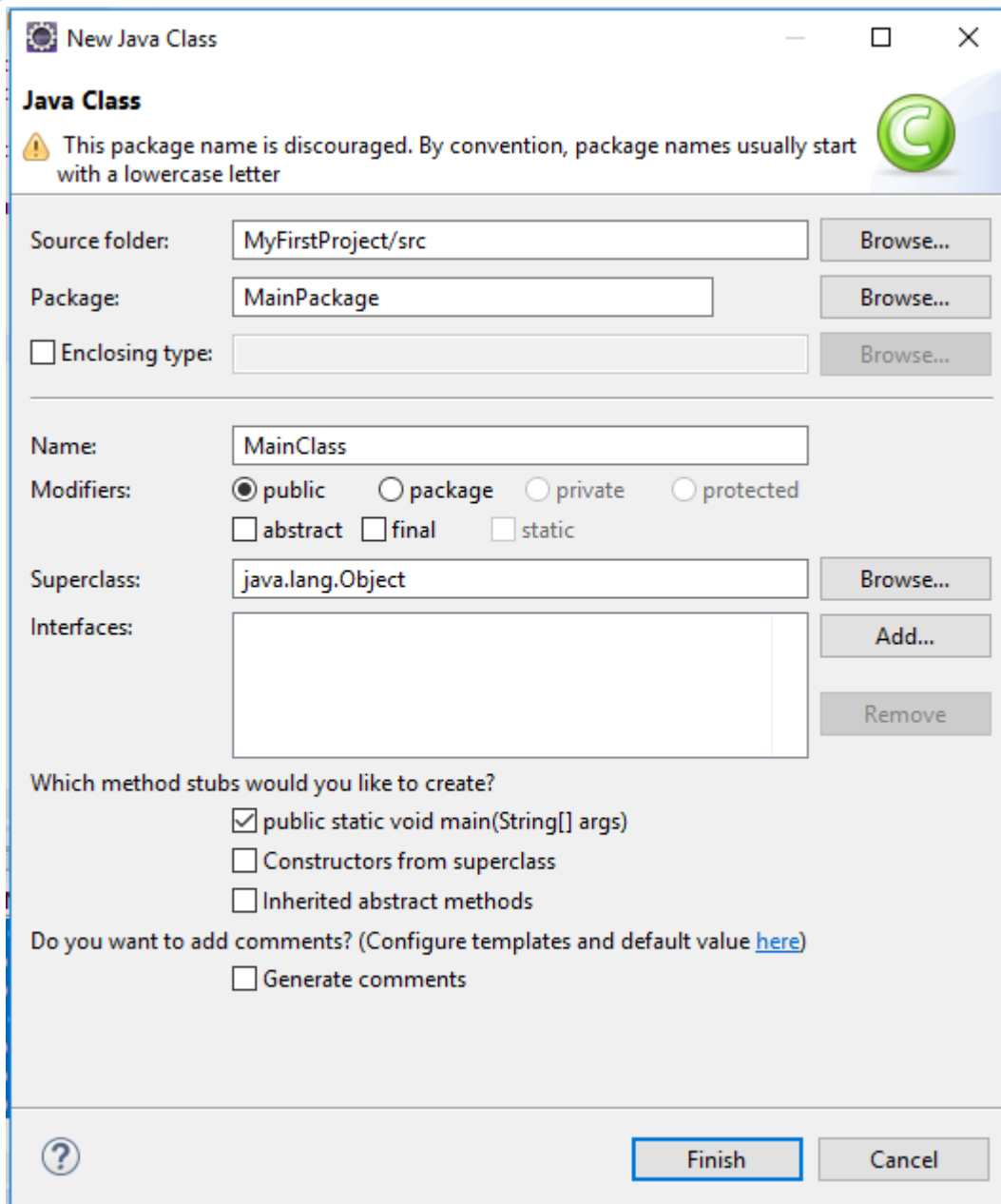
Một lớp có thể được thêm vào một gói bất kỳ theo ba cách:

- Nhập tên gói vào hộp thoại **New Java Class**
- Dùng từ khóa **package**.
- Thêm trực tiếp đến gói

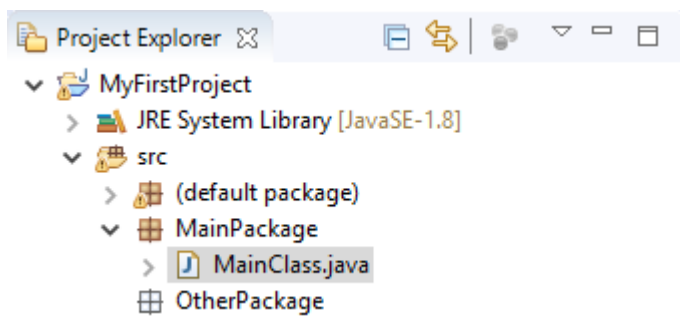
Hai cách đầu hữu ích trong trường hợp chúng ta chưa tạo gói trước đó. Cách thứ 3 hữu ích khi chúng ta đã tạo sẵn các gói trong thư mục *src*.

Ví dụ thêm lớp **MainClass.java** đến gói **MainPackage** theo cách thứ nhất:

- Nhấp chuột phải vào thư mục *src* trong cửa sổ **Project Explorer** chọn **New > Class**:
- Nhập các thông tin vào hộp thoại **New Java Class**:



Nhấn **Finish**. Lúc này trong gói **MainPackage** sẽ xuất hiện lớp **MainClass**:



Đoạn mã của lớp **MainClass.java**:

```
package MainPackage;


public class MainClass {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }


}
```

Ví dụ thêm lớp **OtherClass.java** đến gói **OtherPackage** theo cách thứ hai:

- Nhấp chuột phải vào thư mục *src* trong cửa sổ **Project Explorer** chọn **New > Class**
- Nhập tên lớp nhưng không chọn gói:

 New Java Class

Java Class

 The use of the default package is discouraged.

Source folder:

Package:

☐ Enclosing type:

Name:

Modifiers: ☒ public ☐ package ☐ private ☐ protected
☐ abstract ☐ final ☐ static


Superclass:

Interfaces:

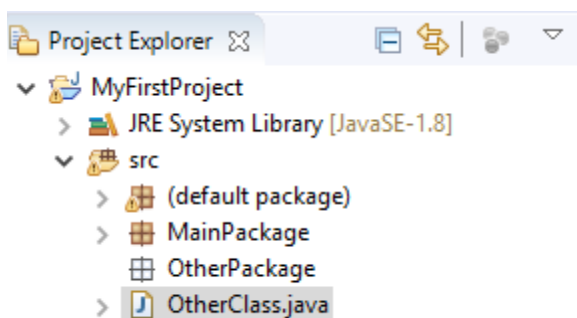
Which method stubs would you like to create?

☐ public static void main(String[] args)
☐ Constructors from superclass
☐ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))
☐ Generate comments



Nhấn **Finish**. Lúc này lớp **OtherClass.java** sẽ được thêm đến thư mục **src** nhưng không thuộc về gói nào:

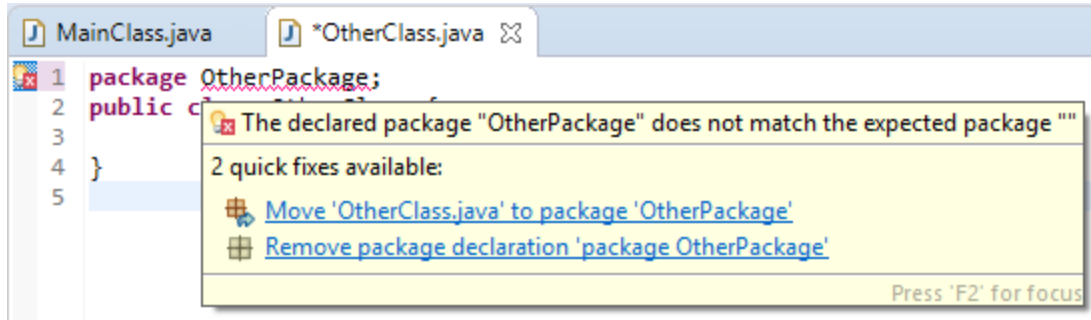


- Thêm lệnh *package* đến lớp **OtherClass**:

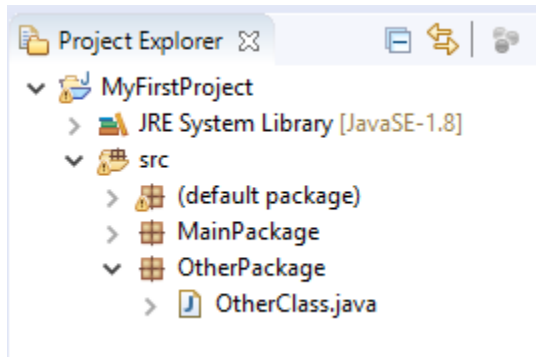
```
package OtherPackage;
public class OtherClass {
}

```

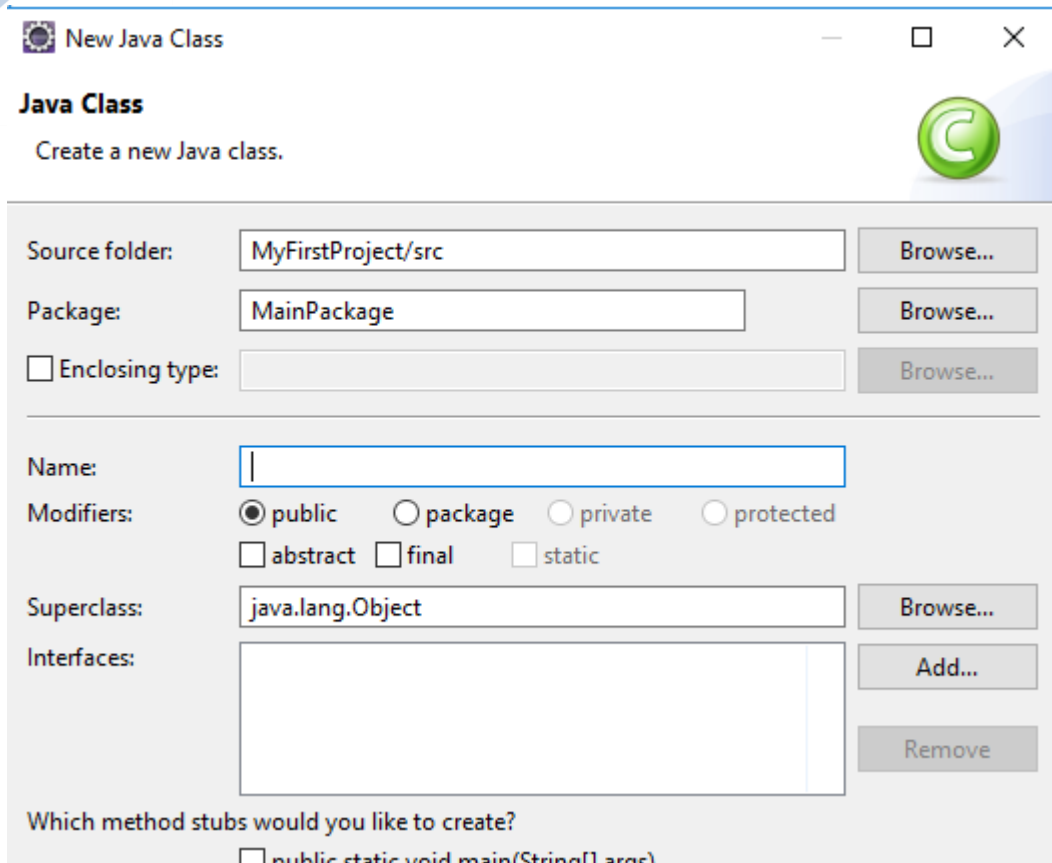
Lúc này sẽ xuất hiện một lỗi như sau:



Nhấn chuột trái vào liên kết *Move ‘OtherClass.java’ to package ‘OtherPackage’* để di chuyển lớp **MainClass** đến gói **OtherPackage**:



Trong trường hợp chúng ta đã tạo sẵn các gói trong thư mục *src*, cách đơn giản nhất để thêm một lớp đến một gói là chọn trực tiếp gói để thêm lớp bằng cách nhấp chuột phải lên gói đó, ví dụ gói **MainPackage**, và chọn **New > Class**. Gói được chọn sẽ xuất hiện trong mục **Package**.



Sử dụng các lớp

Để sử dụng các lớp từ các gói khác nhau chúng ta dùng lệnh ***import***. Ví dụ chúng ta muốn sử dụng lớp **OtherClass** (cùng các thành viên của nó) trong phương thức ***main*** của lớp **MainClass**, chúng ta thêm lệnh ***import*** đến lớp **MainClass**:

```
import OtherPackage.OtherClass;
```

Theo sau từ khóa ***import*** là tên gói cùng với tên lớp cần sử dụng sau dấu chấm. Nếu muốn sử dụng tất cả các lớp trong gói OtherPackage thì chúng ta dùng dấu * thay cho tên lớp:

```
import OtherPackage.*;
```

Tạo thể hiện lớp **OtherClass** trong phương thức ***main*** của lớp **MainClass**:

```
package MainPackage;
import OtherPackage.OtherClass;
public class MainClass {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        OtherClass o = new OtherClass();
    }
}
```


}

}

Tổ chức các lớp trong các gói có vai trò rất quan trọng trong các dự án lớn.

Chương IX

Xử lý ngoại lệ (handling exceptions)

Các ngoại lệ (exceptions)

Một ngoại lệ (exception) là một vấn đề xuất hiện trong quá trình thực thi chương trình. Khi một ngoại lệ xuất hiện, chương trình sẽ bị ngắt và dừng đột ngột mà không có bất kỳ gợi ý nào, vì vậy, các ngoại lệ này cần phải được xử lý.

Ngoại lệ xuất hiện bởi nhiều lý do như người dùng nhập dữ liệu không hợp lệ, tập tin không được tìm thấy hay kết nối mạng bị ngắt, v.v.

Một vài ngoại lệ do người dùng, một vài do lỗi người lập trình, và một vài do sự hạn chế của các tài nguyên vật lý. Căn cứ vào những điều này, ngoại lệ có thể chia thành 3 loại:

- Ngoại lệ xuất hiện trong thời gian biên dịch (the compile time) được gọi là **các ngoại lệ được kiểm tra (checked exceptions)**. Ví dụ cho kiểu ngoại lệ này là ngoại lệ ***FileNotFoundException*** được phát sinh nếu một tập tin không tồn tại khi sử dụng đối tượng ***FileReader*** như ví dụ sau:

```
File file = new File("E://file.txt");
FileReader fr = new FileReader(file);
```

Ngoại lệ có thể trông như sau:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
    Unhandled exception type FileNotFoundException

    at MainPackage.MainClass.main(MainClass.java:12)
```

- Ngoại lệ xuất hiện trong thời gian thực thi (the time of execution) được gọi là **các ngoại lệ không được kiểm tra (unchecked exceptions)**. Các ngoại lệ này thường do các lỗi khi lập trình như lỗi logic (liên quan đến thuật toán) hay lỗi cú pháp. Ví dụ cho kiểu ngoại lệ này là ***ArrayIndexOutOfBoundsException*** xuất hiện khi chúng ta truy cập đến phạm vi vượt quá giới hạn của một mảng như ví dụ sau:

```
int num[] = {1, 2, 3, 4};
System.out.println(num[5]);
```

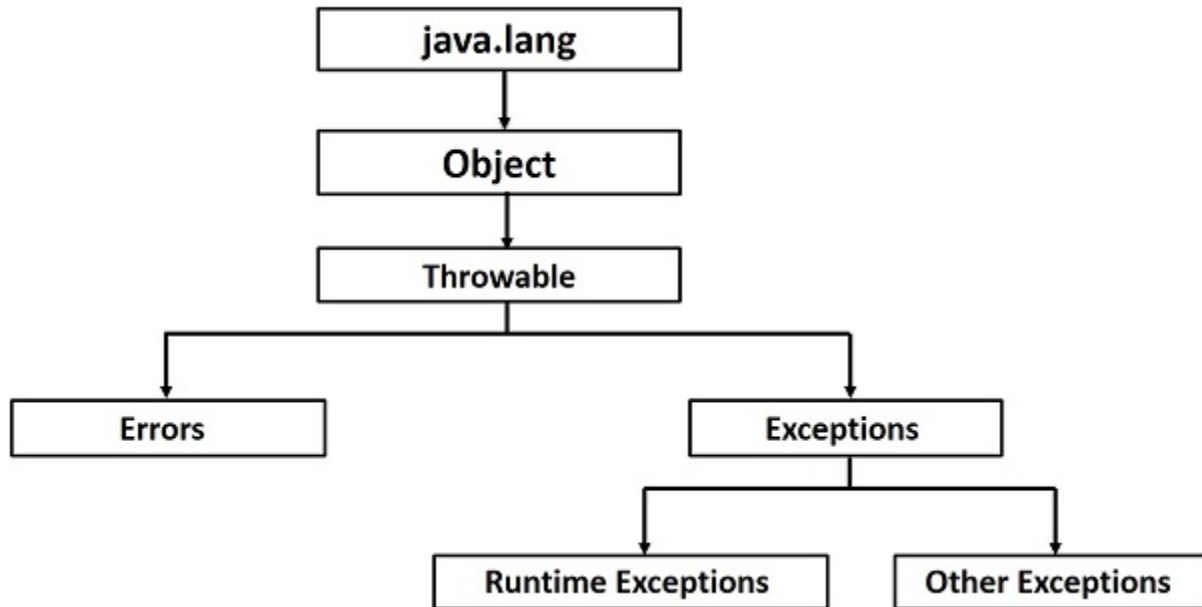
Ngoại lệ có thể trông như sau:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
    at MainPackage.MainClass.main(MainClass.java:12)
```

- Ngoại lệ phát sinh do các lỗi vượt qua sự kiểm soát của người dùng hay người lập trình. Một ví dụ là lỗi tràn bộ nhớ.

Cấu trúc ngoại lệ trong Java (Exception hierarchy)

Các ngoại lệ và các lỗi trong Java được tổ chức theo cấu trúc sau:



Như sơ đồ trên, tất cả các lớp ngoại lệ đều là các lớp con của lớp **Exceptions**. Các lỗi được tập hợp trong lớp **Errors**. Cả hai lớp **Exceptions** và **Errors** đều là con của lớp **Throwable**.

Xử lý ngoại lệ (handling exceptions)

Sử dụng khối lệnh try/catch/finally

Cách xử lý ngoại lệ phổ biến nhất và hiệu quả nhất là sử dụng khối lệnh **try/catch**:

```

try {
    // Khối lệnh thực thi
} catch (ExceptionName e1) {
    // Khối lệnh thực thi khi ngoại lệ xuất hiện
}
  
```

ExceptionName là tên của ngoại lệ mà chúng ta muốn xử lý khi nó xuất hiện ví dụ **FileNotFoundException** hay **ArrayIndexOutOfBoundsException**. Sau try là khối lệnh thực thi cần được bảo vệ từ các ngoại lệ; khối lệnh trong catch sẽ thực thi nếu ngoại lệ **ExceptionName** xuất hiện.

Cũng có thể có nhiều lệnh **catch** sau **try**:

```
try {
    // Khối lệnh thực thi
} catch (ExceptionType1 e1) {
    // Khối lệnh thực thi khi ngoại lệ ExceptionType1 xuất hiện
} catch (ExceptionType2 e2) {
    // Khối lệnh thực thi khi ngoại lệ ExceptionType2 xuất hiện
} catch (ExceptionType3 e3) {
    // Khối lệnh thực thi khi ngoại lệ ExceptionType3 xuất hiện
}
```

Lệnh **finally** cũng có thể được sử dụng theo sau **try** hay **catch** nhưng thông thường là theo thứ tự **try/catch/finally**:

```
try {
    // Khối lệnh thực thi
} catch (ExceptionName e1) {
    // Khối lệnh thực thi khi ngoại lệ xuất hiện
} finally {
    // Khối lệnh luôn luôn thực thi
}
```

Khối lệnh **finally** luôn luôn thực thi bất kể ngoại lệ có xuất hiện hay không.

Đoạn chương trình sau minh họa cách sử dụng **try/catch/finally**:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    int so_bi_chia; // số bị chia
    int so_chia; // số chia
    Scanner scanner = new Scanner(System.in);
    System.out.println("Nhập số bị chia: ");
    so_bi_chia = Integer.parseInt(scanner.nextLine());
    System.out.println("Nhập số chia: ");
    so_chia = Integer.parseInt(scanner.nextLine());
    try {
        System.out.println(so_bi_chia + " / " + so_chia +
            " = " + (so_bi_chia / so_chia) + " So dư là: " +
            (so_bi_chia % so_chia));
    } catch (ArithmeticException e) {
        System.out.println("Không thể chia cho 0!");
    } finally {
        System.out.println("Kết thúc chương trình.");
    }
}
```

Thực thi chương trình:

```
Nhap so bi chia: 3
Nhap so chia: 0
Khong the chia cho 0!
Ket thuc chuong trinh.
```

Sử dụng throw và throws

Nếu một phương thức không xử lý kiểu ngoại lệ được kiểm tra (checked exception) như **FileNotFoundException** thì phương thức phải được khai báo với từ khóa **throws**, ví dụ phương thức **main** sử dụng đối tượng **FileReader**:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    File file = new File("E://file.txt");
    FileReader fr = new FileReader(file);
}
```

Phương thức **main** trên sẽ có lỗi trong như sau:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    File file = new File("E://file.txt");
    FileReader fr = new FileReader(file);
}
```

Chỉ cần thêm khai báo **throws**:

```
public static void main(String[] args) throws FileNotFoundException {
    // TODO Auto-generated method stub
    File file = new File("E://file.txt");
    FileReader fr = new FileReader(file);
}
```

Khai báo với **throws** không dùng để xử lý ngoại lệ mà chỉ dùng để trì hoãn xử lý ngoại lệ. Nếu thực thi phương thức **main** trên vẫn sẽ phát sinh ngoại lệ.

Khác với **throws**, lệnh **throw** dùng để phát sinh một ngoại lệ cụ thể, ví dụ:

```
throw new FileNotFoundException();
```

Chương X

Lập trình giao diện người dùng (GUI programming)

Giao diện người dùng (Graphical User Interface - GUI)

Một giao diện người dùng (gọi tắt là GUI) là một cửa sổ (window) chứa các điều khiển (controls) như button, text box, combo box, v.v. Người dùng tương tác với giao diện bằng cách dùng chuột, con trỏ hay bàn phím.

Java cho phép chúng ta tạo giao diện người dùng bằng cách sử dụng một trong hai gói là **AWT** hay **Swing**. **AWT** (Abstract Window Toolkit) là tập con của **Swing** nên **Swing** được dùng chủ yếu nhưng trong chương trình **Java** phải *import* cả hai gói **Swing** và **AWT**:

```
import java.awt.*;
import javax.swing.*;
```

Các điều khiển trong giao diện đồ họa dùng gói **Swing** luôn bắt đầu bằng chữ “J” ví dụ *JLabel*, *JButton*, v.v. Muốn tạo giao diện người dùng, một lớp phải kế thừa lớp **JFrame** như sau:

```
import java.awt.*;
import javax.swing.*;

public class MainClass extends JFrame {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }

}
```

Đối tượng **JFrame** là một cửa sổ (window) cho phép chúng ta đóng, thay đổi kích thước và có thể đặt các điều khiển. Một cửa sổ có thể được khởi tạo trong phương thức khởi tạo của một lớp kế thừa lớp **JFrame** (ví dụ **MainClass**) như sau:

```
public MainClass() {
    // thiết lập kích thước cửa sổ
    this.setSize(640, 480);
    // thiết lập thao tác đóng ứng dụng khi cửa sổ đóng
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    // cho phép cửa sổ hiển thị
    this.setVisible(true);
}
```

Chúng ta dùng phương thức *setSize* để thiết lập kích thước cho cửa sổ. Phương thức *setDefaultCloseOperation* cho phép đóng ứng dụng khi cửa sổ đóng, nếu không dùng phương thức này, ứng dụng vẫn tiếp tục chạy khi cửa sổ đã đóng. Phương thức *setVisible* cho phép hiển thị cửa sổ.

Bây giờ chỉ việc tạo một đối tượng cửa sổ giao diện:

```
MainClass mainWindow = new MainClass();
```

Đoạn mã hoàn chỉnh của lớp **MainClass**:

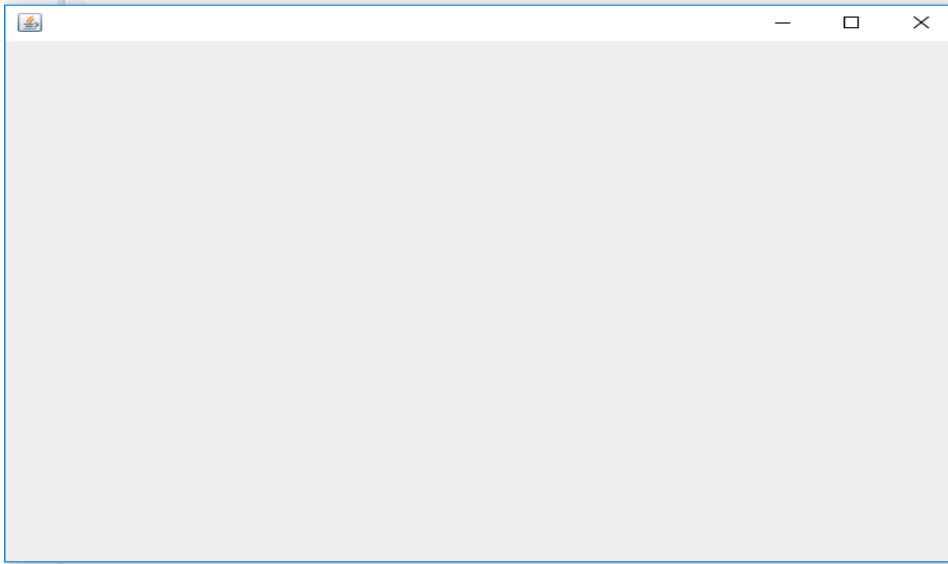
```
import java.awt.*;
import javax.swing.*;

public class MainClass extends JFrame {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        MainClass mainWindow = new MainClass();
    }

    public MainClass() {
        // thiết lập kích thước cửa sổ
        this.setSize(640, 480);
        // thiết lập thao tác đóng cửa sổ
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        // cho phép cửa sổ hiển thị
        this.setVisible(true);
    }
}
```

Kết quả khi thực thi:



Các điều khiển không thể đặt trực tiếp đến cửa sổ mà phải được đặt trong đối tượng **JPanel**. Như vậy, muốn thêm các điều khiển như button, text box, combo box, v.v. đến cửa sổ, đầu tiên chúng ta phải tạo một điều khiển **JPanel**:

```
JPanel panel = new JPanel(new FlowLayout());
```

Đối số cho phương thức khởi tạo của lớp **JPanel** là một đối tượng **FlowLayout**. Đây là một đối tượng tạo bố cục (layout) trên đối tượng **JPanel**. Một vài dạng bố cục phổ biến có thể được mô tả như bảng sau:

Kiểu bố cục	Mô tả
<i>BorderLayout</i>	<i>Panel được phân chia thành trên (top), dưới (bottom), trái (left), phải (right) và giữa (center)</i>
<i>BoxLayout</i>	<i>Đặt các điều khiển trong một cột đơn hay một hàng đơn</i>
<i>CardLayout</i>	<i>Cho phép chúng ta chuyển đổi giữa các tập điều khiển</i>
<i>FlowLayout</i>	<i>Sắp xếp các điều khiển trên một hàng đơn</i>
<i>GridBagLayout</i>	<i>Tạo một bố cục dạng lưới và các điều khiển có thể chiếm giữ nhiều ô</i>
<i>GridLayout</i>	<i>Tạo một bố cục dạng lưới</i>
<i>GroupLayout</i>	<i>Tạo các bố cục ngang (horizontal) và dọc (vertical) tách biệt</i>
<i>SpringLayout</i>	<i>Tạo bố cục cho phép các điều khiển có mối quan hệ theo vị trí của chúng.</i>

Sau khi tạo một đối tượng **JPanel** với một layout, bước kế tiếp chúng ta sẽ thêm các điều khiển đến đối tượng **JPanel** này. Đoạn mã sau sẽ thêm điều khiển **JLabel** và **JButton** đến đối tượng *panel* vừa tạo:

```
panel.add(new JLabel("Test Button: "));
```



```
panel.add(new JButton("Click me!"));
```

Bước cuối cùng là thiết lập nội dung cho *panel* với phương thức `setContentPanel`:

```
this.setContentPanel(panel);
```

Đoạn mã hoàn chỉnh của lớp **MainClass**:

```
import java.awt.*;
import javax.swing.*;

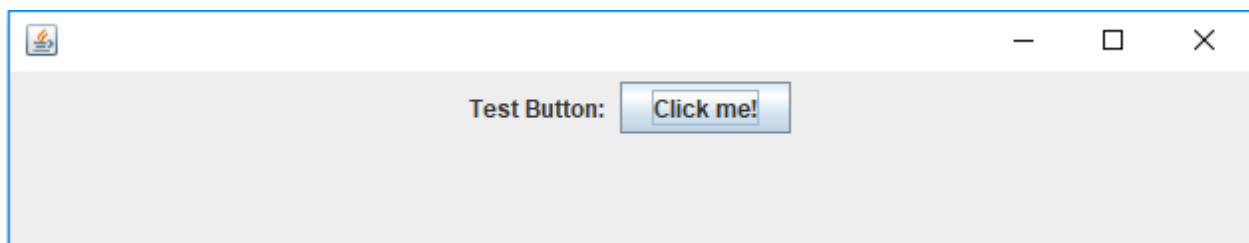
public class MainClass extends JFrame {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        MainClass mainWindow = new MainClass();
    }

    public MainClass() {

        // tạo một đối tượng JPanel
        JPanel panel = new JPanel(new FlowLayout());
        // Thêm hai điều khiển đến panel
        panel.add(new JLabel("Test Button: "));
        panel.add(new JButton("Click me!"));
        // Thiết lập nội dung hiện tại cho panel
        this.setContentPanel(panel);
        // thiết lập kích thước cửa sổ
        this.setSize(640, 480);
        // thiết lập thao tác đóng cửa sổ
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        // cho phép cửa sổ hiển thị
        this.setVisible(true);
    }
}
```

Kết quả:



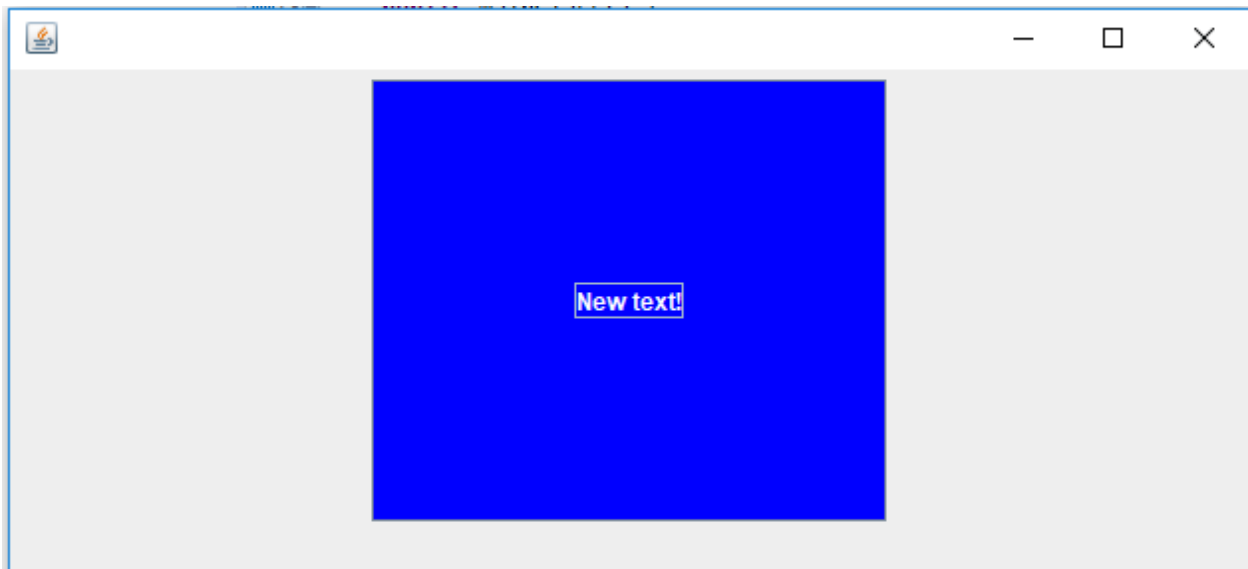
Các điều khiển là thành phần cơ bản của giao diện đồ họa người dùng và mỗi điều khiển sẽ có những mục đích khác nhau. Java cung cấp nhiều lớp với những phương thức cho những điều khiển riêng biệt và đồng thời cũng cung cấp những phương thức dành cho mọi điều khiển. Ví dụ một button có thể được định dạng qua các phương thức như sau:

```

JButton btn = new JButton("Initial Text");
btn.setText("New text!");
String text = btn.getText();
btn.setVisible(false);
btn.setVisible(true);
btn.setMargin(new Insets(100, 100, 100, 100));
Dimension dim = btn.getSize();
btn.setBackground(Color.BLUE);
btn.setForeground(Color.WHITE);
btn.setEnabled(false);
btn.setEnabled(true);
btn.setSize(new Dimension(10, 10));
btn.setBounds(new Rectangle(20, 20, 200, 60));
panel.add(btn);

```

Kết quả:



Sự kiện (events) và xử lý sự kiện (events handling)

Giao diện ActionListener

Sự kiện là các hành động của người dùng như nhấn phím, nhấp chuột, di chuyển chuột, v.v. hay các sự xuất hiện như hệ thống phát sinh một thông báo. Hệ thống cần đáp ứng

các sự kiện khi chúng xảy ra, ví dụ hiển thị một thông báo khi người dùng nhấn chuột vào một button.

Để xử lý sự kiện khi chúng xuất hiện, chúng ta cần lắng nghe các sự kiện đó để đưa ra các hành động, ví dụ thực thi đoạn mã hiển thị một thông điệp, phù hợp. Trong Java chúng ta làm điều này bằng cách thực thi các phương thức trong giao diện

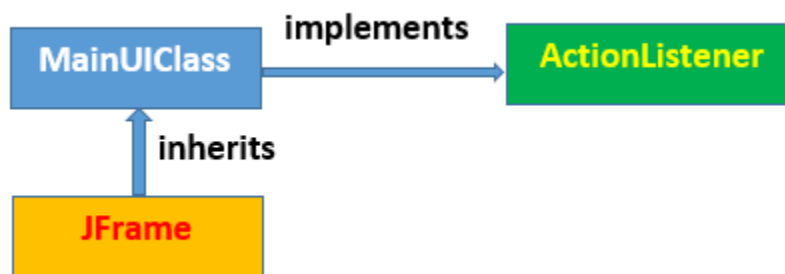
ActionListener. Ngoài **ActionListener**, **Swing** còn cung cấp một danh sách các giao diện lắng nghe các sự kiện có thể tham khảo tại

https://www.tutorialspoint.com/swing/swing_event_listeners.htm

Như vậy, một lớp giao diện đồ họa người dùng trong Java có hai đặc điểm cơ bản:

- Kế thừa lớp **JFrame**
- Thực thi giao diện **ActionListener**

Một cách hình dung trực quan như hình sau đây:



Lớp **MainClass** của chúng ta sẽ thực thi giao diện **ActionListener** như sau:

```

import java.awt.*;
import javax.swing.*;

public class MainClass extends JFrame implements ActionListener{

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        MainClass mainWindow = new MainClass();
    }
    public MainClass() {

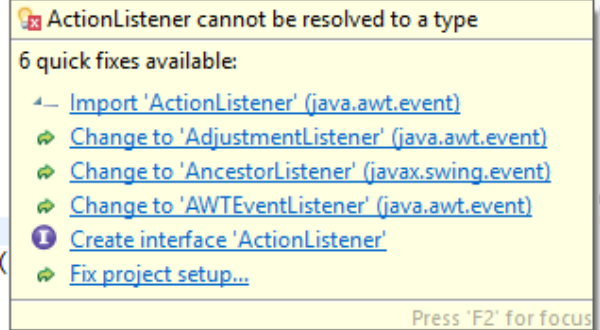
        //...
    }
}
  
```

Khi chúng ta gõ từ khóa **implements ActionListener** sẽ có lỗi trong như sau:

```
public class MainClass extends JFrame implements ActionListener{

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        MainClass mainWindow = new MainClass();
    }
    public MainClass() {

        // tạo một đối tượng JPanel
        JPanel panel = new JPanel(new FlowLayout(
```



Nhấn dòng liên kết đầu tiên để **import** các lớp phù hợp:

```
import java.awt.event.ActionListener;
```

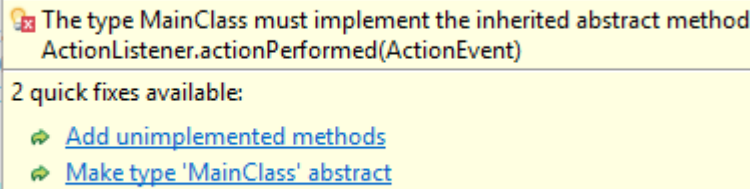
Lúc này lớp MainClass sẽ bị lỗi:

```
public class MainClass extends JFrame implements ActionListener{

    public st
    // TO
    MainC

}
public Ma

// ta
```



Vẫn nhấn vào dòng liên kết đầu tiên sẽ xuất hiện dòng mã **import**:

```
import java.awt.event.ActionEvent;
```

Và xuất hiện một phương thức **actionPerformed** trong lớp **MainClass**:

```
@Override
public void actionPerformed(ActionEvent e) {
    // TODO Auto-generated method stub
}
```

Đoạn mã hoàn chỉnh cho lớp **MainClass**:

```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

public class MainClass extends JFrame implements ActionListener{
```

```

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        MainClass mainWindow = new MainClass();

    }
    public MainClass() {

        //...
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        // TODO Auto-generated method stub

    }

}

```

Bây giờ chúng ta thêm vào cửa sổ giao diện một **JLabel** và một **JButton** bằng cách thêm các đoạn mã sau vào phương thức khởi tạo của lớp **MainClass**:

```

// Thêm một JLabel
JLabel lbName = new JLabel("Test button: ");
panel.add(lbName);
//Thêm một JButton
JButton btnClickMe = new JButton("Click here to display a message!");
panel.add(btnClickMe);

```

Khi người dùng nhấn chuột vào button, một hộp thông điệp sẽ xuất hiện. Đoạn mã xử lý yêu cầu này phải được đặt trong phương thức *actionPerformed* và sử dụng lớp **JOptionPane** (xem lại chương III):

```

@Override
public void actionPerformed(ActionEvent e) {
    // TODO Auto-generated method stub
    // Hiển thị hộp thông điệp
    JOptionPane.showMessageDialog(null, "You clicked on the button!");
}

```

Chúng ta muốn button lắng nghe và thực thi đoạn mã hiển thị hộp thông điệp. Để thực hiện điều này, chúng ta dùng phương thức *addActionListener* của đối tượng JButton như sau:

```

//lắng nghe sự kiện từ button
btnClickMe.addActionListener(this);

```

Đoạn mã hoàn chỉnh của lớp **MainClass** lúc này trông như sau:

```

import java.awt.*;

```

```

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.*;

public class MainClass extends JFrame implements ActionListener{

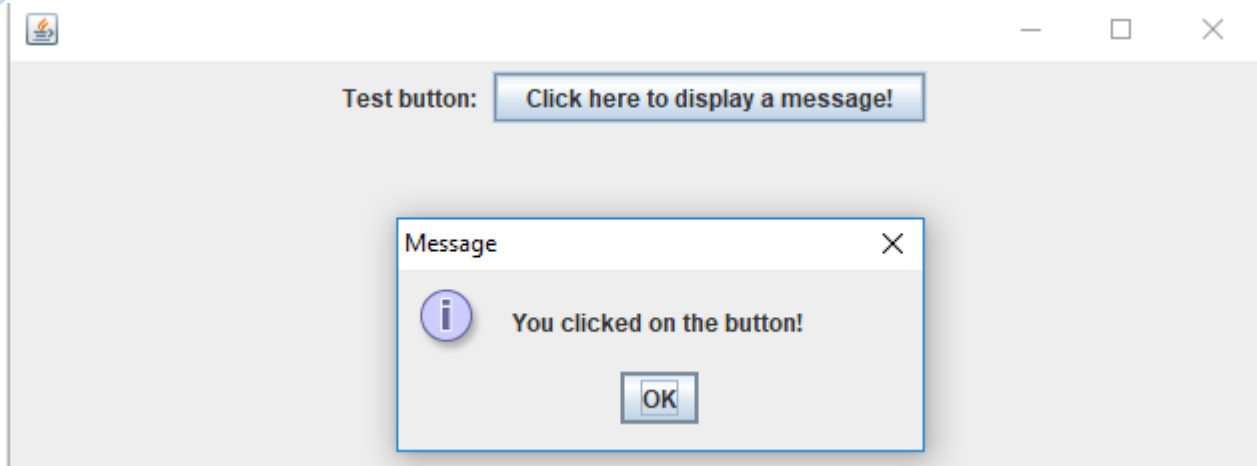
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        MainClass mainWindow = new MainClass();
    }
    public MainClass() {

        // tạo một đối tượng JPanel
        JPanel panel = new JPanel(new FlowLayout());

        // Thêm một JLabel
        JLabel lbName = new JLabel("Test button: ");
        panel.add(lbName);
        //Thêm một JButton
        JButton btnClickMe = new JButton("Click here to display a
message!");
        panel.add(btnClickMe);
        // Thiết lập nội dung hiện tại cho panel
        this.setContentPane(panel);
        //lắng nghe sự kiện từ button
        btnClickMe.addActionListener(this);
        // thiết lập kích thước cửa sổ
        this.setSize(640, 480);
        // thiết lập thao tác đóng cửa sổ
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        // cho phép cửa sổ hiển thị
        this.setVisible(true);
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        // TODO Auto-generated method stub
        // Hiển thị hộp thông điệp
        JOptionPane.showMessageDialog(null,
        "You clicked on the button!");
    }
}

```

Kết quả:



Xử lý sự kiện cho nhiều điều khiển

Cách thức thực thi phương thức *addPerformed* trong lớp **MainClass** như trên có nhược điểm là chỉ thực thi cùng một đoạn mã cho các sự kiện của nhiều điều khiển. Giả sử rằng chúng ta có hai button và khi click chuột mỗi button sẽ xuất hiện những thông điệp khác nhau, thì với cách thực thi phương thức *addPerformed* như trên sẽ không hiệu quả. Có hai giải pháp:

- Sử dụng lớp nặc danh (anonymous class)
- Tạo các lớp thực thi *addPerformed* cho các yêu cầu khác nhau

Sử dụng lớp nặc danh

Chúng ta có thể thực thi phương thức *addPerformed* của giao diện **ActionListener** bằng cách dùng lớp nặc danh. Lúc này, các lớp nặc danh sẽ đóng vai trò tham số cho phương thức *addActionListener* của các điều khiển. Khi sử dụng lớp nặc danh, chúng ta không cần khai báo *implements ActionListener* tại lớp **MainClass**.

Thêm hai **JLabel** và hai **JButton** đến giao diện như sau:

```
public class MainClass extends JFrame {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        MainClass mainWindow = new MainClass();
    }
    public MainClass() {

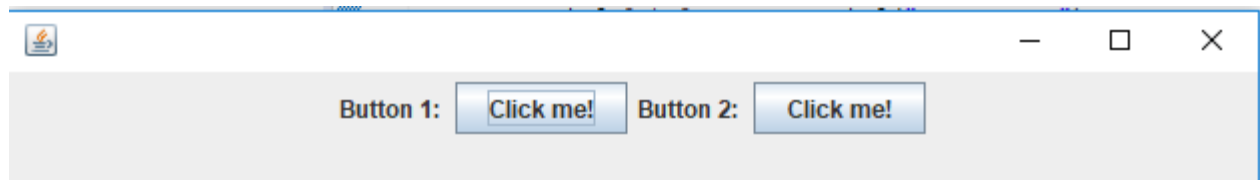
        // tạo một đối tượng JPanel
        JPanel panel = new JPanel(new FlowLayout());
```

```

        // Thêm một JLabel
        JLabel label1 = new JLabel("Button 1: ");
        panel.add(label1);
        //Thêm button 1
        JButton btn1 = new JButton("Click me!");
        panel.add(btn1);
        // Thêm một JLabel
        JLabel label2 = new JLabel("Button 2: ");
        panel.add(label2);
        //Thêm button 2
        JButton btn2 = new JButton("Click me!");
        panel.add(btn2);
        // Thiết lập nội dung hiện tại cho panel
        this.setContentPane(panel);
        // thiết lập kích thước cửa sổ
        this.setSize(640, 480);
        // thiết lập thao tác đóng cửa sổ
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        // cho phép cửa sổ hiển thị
        this.setVisible(true);
    }
}

```

Kết quả:



Lắng nghe và xử lý sự kiện *Click* cho Button 1:

```

// Thực thi ActionListener dùng lớp nặc danh
btn1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // Show a message box.
        JOptionPane.showMessageDialog(null, "I am button 1");
    }
});

```

Lắng nghe và xử lý sự kiện *Click* cho Button 2:

```

// Thực thi ActionListener dùng lớp nặc danh
btn2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {

```



```

        // Show a message box.
        JOptionPane.showMessageDialog(null, "I am button 2");
    }
});

```

(Xem mã hoàn chỉnh tại <https://github.com/TranNgocMinh/Learn-Java/blob/master/Chapter10/AnonymousClassListener.java>)

Tạo các lớp thực thi

Vì Java cho phép các lớp lồng nhau nên chúng ta có thể định nghĩa các lớp thực thi phương thức *actionPerformed* cho từng yêu cầu sự kiện khác nhau của các điều khiển. Lưu ý rằng, với mỗi lớp thực thi phải khai báo thực thi giao diện **ActionListener**. Định nghĩa hai lớp **Button1Handler** và **Button2Handler** trong **MainClass**:

```

public class MainClass extends JFrame{

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        MainClass mainWindow = new MainClass();
    }
    public MainClass() {

        ...
    }

    private class Button1Handler implements ActionListener{

        @Override
        public void actionPerformed(ActionEvent e) {
            // TODO Auto-generated method stub
            JOptionPane.showMessageDialog(null, "I am button 1");
        }
    }
    private class Button2Handler implements ActionListener{

        @Override
        public void actionPerformed(ActionEvent e) {
            // TODO Auto-generated method stub
            JOptionPane.showMessageDialog(null, "I am button 2");
        }
    }
}

```

Và để tiện sử dụng, các biến như *label1*, *label2*, *btn1*,...được khai báo như là biến thành viên của lớp **MainClass**:

```
public class MainClass extends JFrame{

    private JPanel panel;
    private JLabel label1;
    private JLabel label2;
    private JButton btn1;
    private JButton btn2;

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        MainClass mainWindow = new MainClass();
    }

    ...
}
```

Sử dụng các lớp thực thi cho từng button:

```
btn1.addActionListener(new Button1Handler());
btn2.addActionListener(new Button2Handler());
```

(Xem đoạn mã hoàn chỉnh tại <https://github.com/TranNgocMinh/Learn-Java/blob/master/Chapter10/ClassHandlerListener.java>)

Tương tác với các điều khiển

Các lớp điều khiển chứa nhiều phương thức giúp chúng ta có thể tương tác dễ dàng với các điều khiển như thay đổi định dạng kích cỡ, màu nền, kiểu chữ, v.v. của điều khiển như ví dụ chúng ta đã thực hiện với button trong phần mở đầu của chương này. Các phương thức này giúp chúng ta linh hoạt khi tương tác với các điều khiển. Ví dụ sau sẽ điều chỉnh mã thực thi khi phát sinh sự kiện *click* của hai button 1 và button 2 ở trên, thay vì hiển thị các hộp thông điệp, chúng ta sẽ thay đổi nội dung văn bản hiển thị trên hai button:

```
private class Button1Handler implements ActionListener{

    @Override
    public void actionPerformed(ActionEvent e) {
        // TODO Auto-generated method stub
    }
}
```

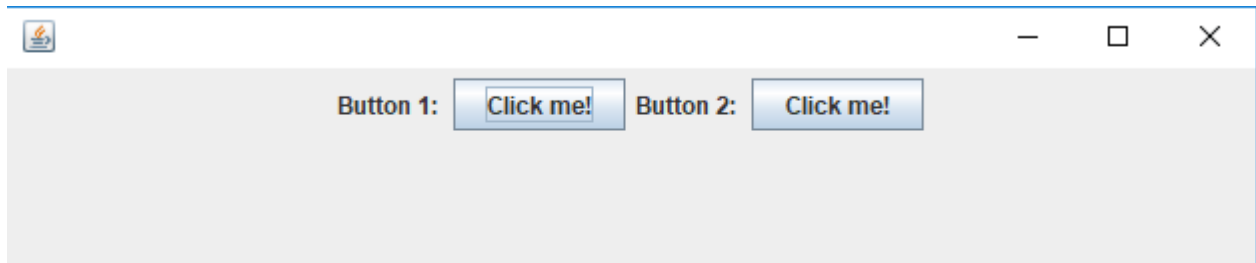
```

        btn1.setText("I am button 1");
    }
}
private class Button2Handler implements ActionListener{

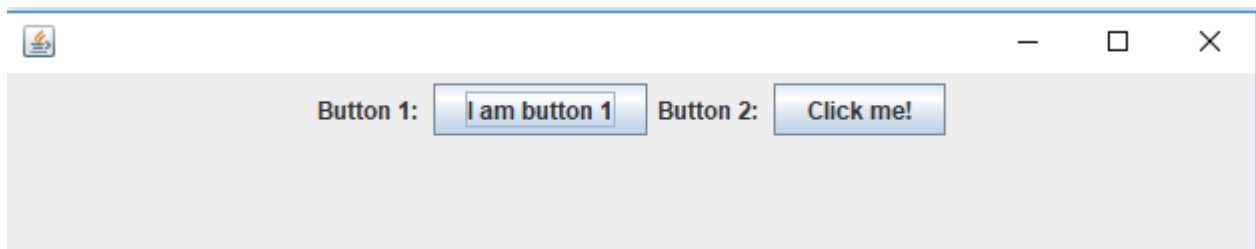
    @Override
    public void actionPerformed(ActionEvent e) {
        // TODO Auto-generated method stub
        btn2.setText("I am button 2");
    }
}

```

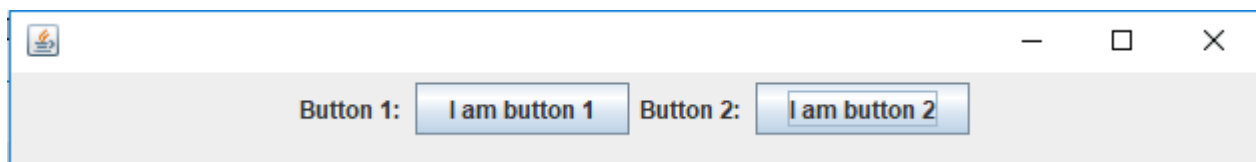
Khi thực thi chương trình:



Nhấn vào Button 1:



Nhấn vào Button 2:



Tham khảo ví dụ:

- Dùng **BorderLayout** tại <https://github.com/TranNgocMinh/Learn-Java/blob/master/Chapter10/BorderLayout.java>)
- Dùng **GridLayout** tại <https://github.com/TranNgocMinh/Learn-Java/blob/master/Chapter10/GridLayout.java>

