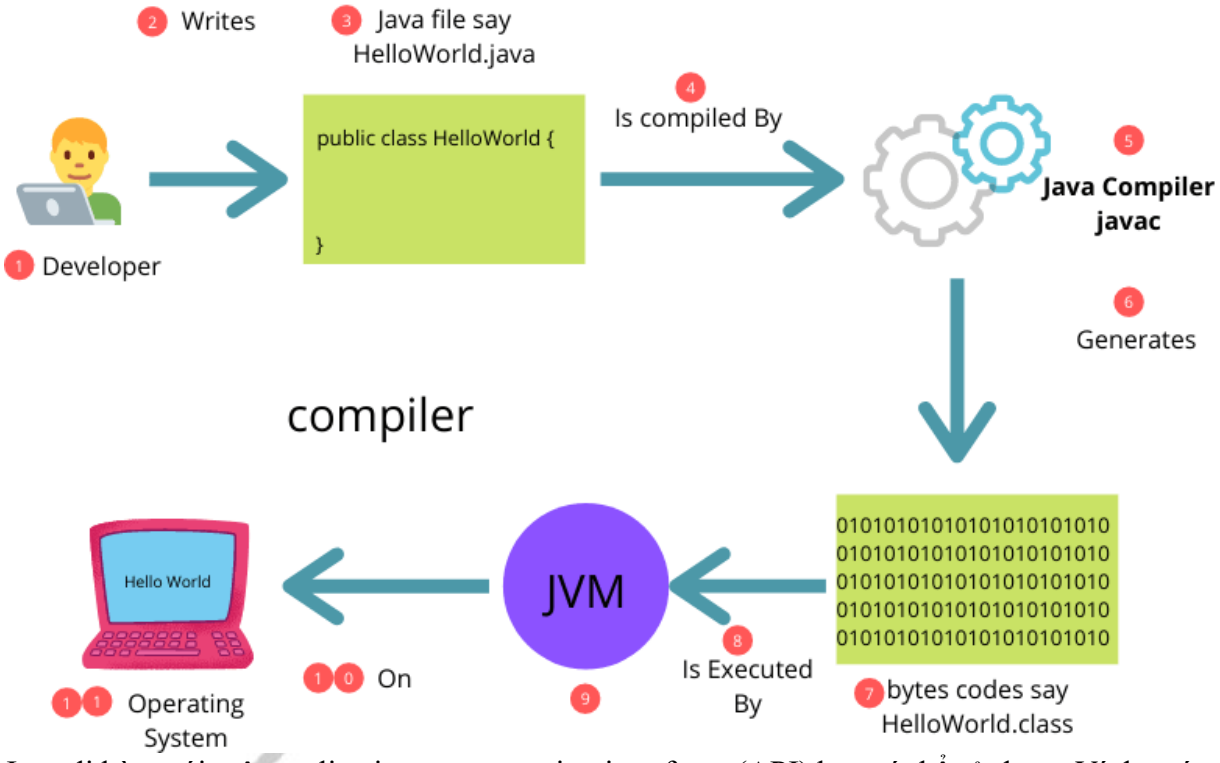
Chapter 1: Welcome to Java

1. Learning About the Java Environment

a) Các thành phần chính của JAVA

Java Development Kit (JDK) chứa phần mềm tối thiểu bạn cần để phát triển Java. Các phần chính bao gồm trình biên dịch (javac), giúp chuyển đổi các tệp .java thành các tệp .class và trình khởi chạy java, tạo ra máy ảo và thực thi chương trình. Chúng ta sẽ sử dụng cả hai ở phần sau của chương này khi chạy các chương trình ở dòng lệnh.

JDK cũng chứa các công cụ khác bao gồm lệnh archiver (jar), có thể đóng package các tệp lại với nhau và lệnh tài liệu API (javadoc) để tạo tài liệu. Chương trình javac tạo ra các hướng dẫn ở định dạng đặc biệt mà lệnh java có thể chạy được gọi là bytecode. Sau đó java khởi chạy Máy ảo Java (JVM) trước khi chạy mã. JVM biết cách chạy mã byte trên máy thực tế mà nó đang chạy.



Java đi kèm với một application programming interfaces (API) bạn có thể sử dụng. Ví dụ: có một lớp StringBuilder để tạo một chỗi và một phương thức trong Collections để sắp xếp danh sách. Khi viết một chương trình, sẽ rất hữu ích nếu bạn xem xét những phần nhiệm vụ nào của bạn có thể được thực hiện bằng các API hiện có.

2. Benefits of Java

- **Object Oriented**- Java là ngôn ngữ hướng đối tượng, có nghĩa là tất cả mã được xác định trong các lớp và hầu hết các lớp đó có thể được khởi tạo thành các đối tượng. Java cho phép lập trình hàm trong một lớp, nhưng hướng đối tượng vẫn là tổ chức chính của mã.
- **Encapsulation** - Java hỗ trợ các công cụ sửa đổi truy cập để bảo vệ dữ liệu khỏi sự truy cập và sửa đổi ngoài ý muốn. Hầu hết mọi người coi tính đóng package là một khía cạnh của ngôn ngữ hướng đối tượng.
- **Nền tảng độc lập** - Java là ngôn ngữ thông dịch được biên dịch thành bytecode. Lợi ích chính là mã Java được biên dịch một lần thay vì cần phải biên dịch lại cho các hệ điều hành khác nhau. Điều này được gọi là "viết một lần, chạy khắp nơi". Tính di động cho phép bạn dễ dàng chia sẻ các phần mềm được biên dịch sẵn.
- **Mạnh mẽ** - Một trong những ưu điểm chính của Java so với C++ là nó ngăn ngừa rò rỉ bộ nhớ. Java tự quản lý bộ nhớ và tự động thu gom rác. Quản lý bộ nhớ kém trong C++ là nguyên nhân chính gây ra lỗi trong chương trình.
- **Đơn giản -** Java được thiết kế để dễ hiểu hơn C++. Ngoài việc loại bỏ con trỏ, nó còn loại bỏ tình trạng quá tải toán tử. Trong C++, bạn có thể viết a + b và hiểu nó có nghĩa là hầu hết mọi thứ.
- **Bảo mật** - Mã Java chạy bên trong JVM. Điều này tạo ra một hộp cát khiến mã Java khó thực hiện những điều xấu đối với máy tính đang chạy nó.
- **Đa luồng** - Java được thiết kế để cho phép nhiều đoạn mã chạy cùng một lúc. Ngoài ra còn có nhiều API để hỗ trợ nhiệm vụ này.
- **Khả năng tương thích ngược** - Các kiến trúc sư ngôn ngữ Java chú ý cẩn thận đến việc đảm bảo các chương trình cũ sẽ hoạt động với các phiên bản Java mới hơn. Mặc dù điều này không phải lúc nào cũng xảy ra nhưng những thay đổi sẽ phá vỡ khả năng tương thích ngược sẽ diễn ra từ từ và được thông báo trước.

3. Java Class Structure

Trong các chương trình Java, các class là các khối xây dựng cơ bản. Khi xác định một class, bạn mô tả tất cả các bộ phận và đặc điểm của một trong những khối xây dựng đó. Để sử dụng hầu hết các class, bạn phải tạo các đối tượng.

a) Fields và methods

Các lớp Java có hai thành phần chính: các phương thức, thường được gọi là function hoặc thủ tục trong các ngôn ngữ khác và các trường, thường được gọi là biến. Các biến giữ trạng thái của chương trình và các method hoạt động trên trạng thái đó.

Java class đơn giản nhất có thể viết:

```
public class Animal {
    String name;

    public String getName() {
        return name;
    }
    public void setName(String newName) {
        name = newName;
    }
}
```

b) Comment trong Java

Trong Java, chúng ta có 2 cách comment:

- Comment trên một dòng
- Comment trên nhiều dòng

Comment trên một dòng

Comment trên một dòng có nghĩa là nội dung comment bắt đầu và kết thúc trên cùng một dòng. Để viết comment trên một dòng, chúng ta sẽ sử dụng 2 ký tự //.

```
// "Hello, World!"

class Main {
    public static void main(String[] args) {
        // Xuất ra màn hình "Hello, World!"
        System.out.println("Hello, World!");
    }
}
```

Comment trên nhiều dòng

Khi chúng ta muốn comment trên nhiều dòng, thì chúng ta có thể sử dụng cặp dấu /*….*/. Các comment sẽ được đặt bên trong chúng.

```
/*
Đây là một ví dụ comment nhiều dòng
 * Chương trình sẽ in ra "Hello, World!" trên màn hình console.
 */

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

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

c) Classes vs. Files

Mỗi lớp Java được định nghĩa trong tệp .java của riêng nó. Nó thường được public, có nghĩa là bất kỳ mã nào cũng có thể gọi nó. Điều thú vị là Java không yêu cầu lớp này phải được public. Ví dụ:

```
class Animal {
    String name;
}
```

Bạn thậm chí có thể đặt hai lớp vào cùng một tệp. Khi bạn làm như vậy, tối đa một trong các lớp trong tệp được phép public. Điều đó có nghĩa là một tệp chứa nội dung sau cũng được:

```
public class Animal {
}

class Animal2 {
    String name;
}
```

d) Running a program in one line

Bắt đầu từ Java 11, bạn có thể chạy một chương trình mà không cần biên dịch nó trước—à, không cần gõ lệnh javac. Hãy tạo một lớp mới:

```
public class SingleFileZoo {
    public static void main(String[] args) {
        System.out.println("Single file: " + args[0]);
    }
}
```

Chúng ta có thể chạy ví dụ SingleFileZoo mà không cần phải biên dịch nó.

```
java SingleFileZoo.java Cleveland
```

Tính năng này được gọi là khởi chạy các chương trình single-file source-code, nó chỉ có thể được sử dụng nếu chương trình của bạn là một tệp. Điều này có nghĩa là nếu chương trình của bạn có hai tệp .java, bạn vẫn cần sử dụng javac.

Full command	Single-file source-code command
javac HelloWorld.java java HelloWorld	java HelloWorld.java
Tạo class file	Hoàn toàn trong bộ nhớ
Đối với mọi chương trình	Đối với các chương trình có một tệp
Có thể nhập mã bằng bất kỳ Java library nào có sẵn	Chỉ có thể nhập library đi kèm với JDK

4. Hiểu các khai báo package và imports

Java có hàng nghìn lớp được tích hợp sẵn và còn vô số lớp khác từ các nhà phát triển. Với tất cả các lớp đó, Java cần một cách để tổ chức chúng. Java đặt các lớp trong các **package**. Đây là các nhóm logic cho các class.

```
public class ImportExample {
    public static void main(String[] args) {
        Random r = new Random(); // DOES NOT COMPILE
        System.out.println(r.nextInt(10));
    }
}
```

Trình biên dịch Java sẽ cung cấp cho bạn một lỗi hữu ích như thế này:

```
Random cannot be resolved to a type
```

Nguyên nhân khác của lỗi này là do thiếu câu lệnh import cần thiết. Các câu lệnh import cho Java biết **package** nào cần tìm cho các class. Vì bạn không cho Java biết nơi tìm Random nên nó không có manh mối. Việc thử lại lần nữa với quá trình import sẽ cho phép bạn biên dịch.

```
import java.util.Random; // import tells us where to findRandom
public class ImportExample {
    public static void main(String[] args) {
        Random r = new Random();
        System.out.println(r.nextInt(10)); // print a number 0 - 9
    }
}
```

Nếu **package** bắt đầu bằng java hoặc javax, điều này có nghĩa là nó đi kèm với JDK.

Nếu nó bắt đầu bằng một cái gì đó khác, nó có thể hiển thị nguồn gốc của việc sử dụng tên trang web ngược lại. Ví dụ: com.amazon.javabook cho chúng ta biết mã đến từ Amazon.com.

Sau tên trang web, bạn có thể thêm bất cứ điều gì bạn muốn. Ví dụ: com.amazon.java.my.name cũng đến từ Amazon.com. Java gọi các package con chi tiết hơn. Package com.amazon.javabook là package con của com.amazon.

a) Wildcards

Các class trong cùng một package thường được import cùng nhau. Bạn có thể sử dụng shortcut để nhập tất cả các class trong một package

```
import java.util.*; // imports java.util.Random among other things
public class ImportExample {
    public static void main(String[] args) {
        Random r = new Random();
        System.out.println(r.nextInt(10));
    }
}
```

* là ký tự đại diện khớp với tất cả các class trong package. Mọi class trong package java.util đều có sẵn cho chương trình này khi Java biên dịch nó. Nó không import các child package, fields hoặc method con; nó chỉ import các class. (Có một kiểu nhập đặc biệt gọi là static-import: import các thành phần static trong lớp)

Bạn có thể nghĩ rằng việc bao gồm quá nhiều class sẽ làm chậm quá trình thực thi chương trình của bạn, nhưng thực tế không phải vậy. Trình biên dịch sẽ tìm ra những gì thực sự cần thiết. Việc liệt kê các class được sử dụng giúp mã dễ đọc hơn, đặc biệt đối với những người mới lập trình. Việc sử dụng ký tự đại diện có thể rút ngắn danh sách nhập.

b) Redundant imports

Có một package đặc biệt trong thế giới Java có tên là java.lang. Package này đặc biệt ở chỗ nó được import tự động. Bạn có thể import package này vào câu lệnh nhập, nhưng bạn không cần phải làm vậy. Trong đoạn mã sau, bạn nghĩ có bao nhiêu lần nhập là dư thừa?

```
import java.lang.System;
import java.lang.*;
import java.util.Random;
import java.util.*;
public class ImportExample {
    public static void main(String[] args) {
        Random r = new Random();
        System.out.println(r.nextInt(10));
    }
}
```

Câu trả lời là ba trong số hàng import đều dư thừa. Dòng 1 và 2 là dư thừa vì mọi thứ trong java.lang đều tự động import. Dòng 4 cũng dư thừa trong ví dụ này vì Random đã được import từ java.util.Random.

Một trường hợp dư thừa khác liên quan đến việc import một class nằm trong cùng package với class đang import nó. Java tự động tìm kiếm các class khác trong package hiện tại.

```
import java.nio.file.Files;
import java.nio.file.Paths;

//Bây giờ hãy xem xét một số mục import không hoạt động.

import java.nio.*; // NO GOOD - ký tự đại diện chỉ khớp với tên lớp chứ không
khớp với package "file.Files"
import java.nio.*.*; // NO GOOD - bạn chỉ có thể có một ký tự đại diện và nó
phải ở cuối
import java.nio.file.Paths.*; // NO GOOD - bạn không thể, chỉ import tên lớp
của các phương thức
public class Animal {
    public static void main(String[] args) {
        Files
    }
}
```

c) Naming conflicts

Một trong những lý do sử dụng package là để tên class không nhất thiết phải là duy nhất trên toàn bộ Java. Điều này có nghĩa là đôi khi bạn sẽ muốn import một class có thể tìm thấy ở nhiều nơi. Một ví dụ phổ biến về điều này là lớp Date. Java cung cấp các triển khai java.util.Date và java.sql.Date.

Có thể sử dụng tính năng import nào nếu muốn có java.util.Date?

```
public class Conflicts {
    Date date;
// some more code
}
```

Có thể viết **import java.util.***; hoặc **import java.util.Date**;

```
import java.util.*;
import java.sql.*; // causes Date declaration to not compile
```

Khi tìm thấy class này trong nhiều package, Java sẽ báo lỗi trình biên dịch.

```
import java.util.Date;
import java.sql.*;
```

Nếu bạn nhập tên class một cách rõ ràng, nó sẽ được ưu tiên hơn bất kỳ ký tự đại diện nào hiện có. Java nghĩ: “Lập trình viên thực sự muốn tôi đảm nhận việc sử dụng lớp java.util.Date”.

d) Creating a new package

```
package packagea;
public class ClassA {

}

package packageb;
import packagea.ClassA;
public class ClassB {
    public static void main(String[] args) {
        ClassA a;
        System.out.println("Got it");
    }
}
```

Chapter 2: Java Building Blocks

1. Creating Objects

a) Calling constructors

Để tạo một instance của một class, phải viết new trước tên class và thêm dấu ngoặc đơn sau nó. Đây là một ví dụ:

Park p = new Park();

Park() trông giống như một phương thức vì nó được theo sau bởi dấu ngoặc đơn. Nó được gọi là constructor, là một loại phương thức đặc biệt để tạo một đối tượng mới.

```
package com.ttmars;

public class Chick {
    public Chick() {
        System.out.println("in constructor");
    }
}
```

Có hai điểm chính cần lưu ý về constructor: **tên của constructor khớp với tên của lớp và không có kiểu trả về.**

```
public class Chick {
    public void Chick() { } // NOT A CONSTRUCTOR
}
```

Mục đích của constructor là khởi tạo các trường, mặc dù bạn có thể đặt bất kỳ mã nào vào đó. Một cách khác để khởi tạo các trường là thực hiện trực tiếp trên dòng mà chúng được khai báo. Ví dụ này cho thấy cả hai cách tiếp cận:

```
package com.ttmars;

public class Chicken {
    int numEggs = 12; // initialize on line
    String name;
    public Chicken() {name = "Duke"; // initialize in constructor
    }
}
```

Đối với hầu hết các lớp, bạn không cần phải viết mã constructor—trình biên dịch sẽ cung cấp constructor mặc định "không làm gì" cho bạn. Có một số trường hợp yêu cầu bạn phải khai báo constructor.

b) Reading and writing member fields

Có thể đọc và ghi các biến instance trực tiếp từ caller.

```
public class Swan {
    int numberEggs; // instance variable
    public static void main(String[] args) {
        Swan mother = new Swan();
        mother.numberEggs = 1; // set variable
        System.out.println(mother.numberEggs); // read variable
    }
}
```

"Caller" trong trường hợp này là phương thức main(), có thể ở cùng một class hoặc ở một class khác. Đọc một biến được gọi là get. Class này lấy trực tiếp numberEggs để in ra. Việc ghi vào một biến được gọi là set. Class này set numberEggs thành 1.

Bạn thậm chí có thể đọc giá trị của các trường đã được khởi tạo trên một dòng khởi tạo trường mới:

```
public class Name {
    String first = "Theodore";
    String last = "Moose";
    String full = first + last;
}
```

c) Executing instance initializer blocks

Khi bạn tìm hiểu về các phương thức, bạn đã thấy dấu ngoặc nhọn ({}). Code giữa các dấu ngoặc nhọn được gọi là khối mã. Bất cứ nơi nào bạn nhìn thấy dấu ngoặc nhọn đều là khối mã (code block). Đôi khi các khối mã nằm bên trong một phương thức. Chúng được chạy khi phương thức được gọi. Đôi khi, các khối mã xuất hiện bên ngoài một phương thức. Chúng được gọi là instance initializer – khi khởi tạo 1 instance.Trong Chapter 7, sẽ học cách sử dụng instance initializer.

Có bao nhiêu khối trong ví dụ sau? Có bao nhiêu instance initializer?

```
public class Bird {
    public static void main(String[] args) {
        {
            System.out.println("Feathers");
        }
    }

    {
        System.out.println("Snowy");
    }
}
```

⇨ Có 4 code block trong ví dụ, nhưng chỉ có 1 instance initializer

d) Following order of initialization

Khi viết mã khởi tạo các field ở nhiều nơi, bạn phải theo dõi thứ tự khởi tạo. Đây chỉ đơn giản là thứ tự trong đó các phương thức, constructor hoặc khối khác nhau được gọi khi một instance của class được tạo. Các quy tắc cơ bản: (các quy tắc chi tiết sẽ được nói đến ở chapter 8)

- Các field và khối khởi tạo phiên bản được chạy theo thứ tự xuất hiện trong tệp.
- Constructor chạy sau khi tất cả các trường và khối khởi tạo cá thể đã chạy.

```
public class Chick {
    private String name = "Fluffy";

    {
        System.out.println("setting field");
    }

    public Chick() {
        name = "Tiny";
        System.out.println("setting constructor");
    }

    public static void main(String[] args) {
        Chick chick = new Chick();
        System.out.println(chick.name);
    }
}
```

Output:

```
setting field
setting constructor
Tiny
```

Thứ tự quan trọng đối với các field và code block. Không thể tham chiếu đến một biến trước khi nó được xác định:

```
{ System.out.println(name); } // DOES NOT COMPILE
private String name = "Fluffy";
```

Đoạn mã này in ra là gì?

```
public class Egg {
    public Egg() {
        number = 5;
    }

    public static void main(String[] args) {
        Egg egg = new Egg();
        System.out.println(egg.number);
    }

    private int number = 3;

    { number = 4; }
}
```

⇨ 5

2. Understanding Data Types

Java chứa hai loại dữ liệu: kiểu nguyên thủy(primitive) và kiểu tham chiếu(reference).

a) Using primitive types

Java có tám kiểu dữ liệu tích hợp, được gọi là kiểu nguyên thủy của Java. Tám kiểu dữ liệu này đại diện cho các building block cho các đối tượng Java, bởi vì tất cả các đối tượng Java chỉ là một tập hợp phức tạp của các kiểu dữ liệu nguyên thủy này.Nguyên thủy chỉ là một giá trị duy nhất trong bộ nhớ, chẳng hạn như số hoặc ký tự.

The Primitive Types:

Keyword	Type	Example
boolean	true or false	TRUE
byte	8-bit integral value	123
short	16-bit integral value	123
int	32-bit integral value	123
long	64-bit integral value	123L
float	32-bit floating-point value	123.45f
double	64-bit floating-point value	123.456
char	16-bit Unicode value	'a'

Hãy xem xét một số điểm chính:

- Kiểu float và double được sử dụng cho các giá trị dấu phẩy động (thập phân).
- float yêu cầu chữ f theo sau số để Java biết đó là float.
- Các kiểu byte, short, int và long được sử dụng cho các số không có dấu thập phân. Trong toán học, tất cả những giá trị này đều được gọi là integral value (tích phân)
- Mỗi loại số sử dụng số bit gấp đôi so với loại tương tự nhỏ hơn. Ví dụ: short sử dụng số bit gấp đôi so với byte.
- Tất cả các loại số đều được ký bằng Java. Điều này có nghĩa là họ dự trữ một trong các bit của mình để thể hiện phạm vi âm. Ví dụ: byte có phạm vi từ -128 đến 127. Bạn có thể ngạc nhiên khi phạm vi này không phải là -128 đến 128. Đừng quên, 0 cũng cần được tính trong phạm vi đó.
- short và char được lưu trữ dưới dạng kiểu tích phân có cùng độ dài 16 bit. Sự khác biệt chính là short được ký, có nghĩa là nó chia phạm vi của nó thành các số nguyên dương và âm. char không được ký, có nghĩa là phạm vi hoàn toàn dương bao gồm 0.

Writing Literals

Theo mặc định, Java giả định bạn đang xác định một giá trị int bằng một numeric literal (chữ số). Trong ví dụ sau, số được khai báo lớn hơn số lớn nhất với một kiểu int.

```
long max = 3123456789; // DOES NOT COMPILE
```

Java thông báo số lượng nằm ngoài phạm vi. Và đó là—đối với một int.

```
long max = 3123456789L; // now Java knows it is a long
```

Một cách khác để chỉ định số là thay đổi "cơ số". Khi học đếm, bạn đã học các chữ số từ 0–9. Hệ đếm này được gọi là cơ số 10 vì có 10 số. Nó còn được gọi là hệ thống số thập phân. Java cho phép bạn chỉ định các chữ số theo một số định dạng khác:

- Bát phân (các chữ số 0–7), sử dụng số 0 làm tiền tố—ví dụ: 017
- Hệ thập lục phân (các chữ số 0–9 và các chữ cái A–F/a–f), sử dụng 0x hoặc 0X làm tiền tố—ví dụ: 0xFF, 0xff, 0XFf. Hệ thập lục phân không phân biệt chữ hoa chữ thường nên tất cả các ví dụ này đều có cùng giá trị.
- Nhị phân (chữ số 0–1), sử dụng số 0 theo sau là b hoặc B làm tiền tố — ví dụ: 0b10, 0B10

Literals and the Underscore Character

Điều cuối cùng bạn cần biết về chữ số là bạn có thể có dấu gạch dưới trong số để dễ đọc hơn:

```
int million1 = 1000000;
int million2 = 1_000_000;
```

Bạn có thể thêm dấu gạch dưới ở bất cứ đâu ngoại trừ ở đầu chữ, ở cuối chữ, ngay trước dấu thập phân hoặc ngay sau dấu thập phân. Bạn thậm chí có thể đặt nhiều ký tự gạch dưới cạnh nhau.

double notAtStart = _1000.00;
double notAtEnd = 1000.00_;
double notByDecimal = 1000_.00;
double annoyingButLegal = 1_00_0.0_0;
double reallyUgly = 1_____2;
	// DOES NOT COMPILE
	// DOES NOT COMPILE
	// DOES NOT COMPILE
	// Ugly, but compiles
	// Also compiles

b) Using reference types

Kiểu **reference** đề cập đến một đối tượng (một instance của một class). Không giống như các kiểu nguyên thủy giữ các giá trị của chúng trong bộ nhớ nơi biến được phân bổ, các tham chiếu không giữ giá trị của đối tượng mà chúng tham chiếu đến. Thay vào đó, một tham chiếu "trỏ" đến một đối tượng bằng cách lưu trữ địa chỉ bộ nhớ nơi đặt đối tượng đó, một khái niệm được gọi là con trỏ.

Chúng ta hãy xem một số ví dụ khai báo và khởi tạo các kiểu tham chiếu. Giả sử chúng ta khai báo một reference kiểu java.util.Date và một reference kiểu String:

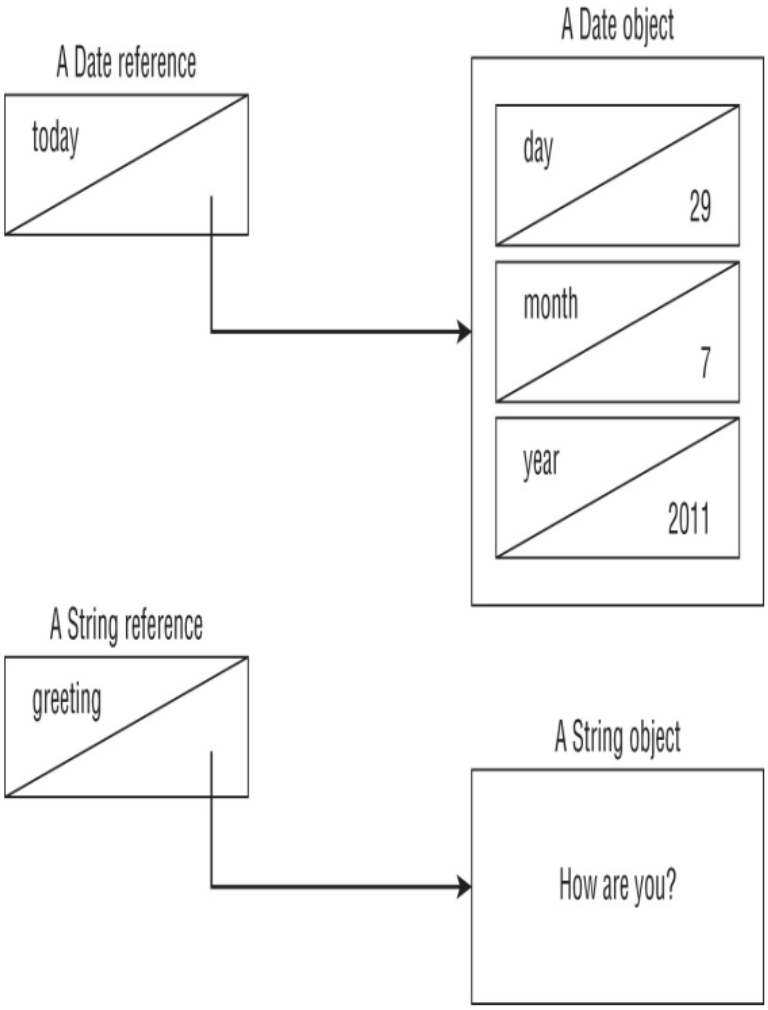
```
java.util.Date today;
String greeting;
```

Một giá trị được gán cho một tham chiếu theo một trong hai cách:

- Một tham chiếu có thể được gán cho một đối tượng khác cùng loại hoặc loại tương thích.
- Một tham chiếu có thể được gán cho một đối tượng mới bằng từ khóa new.

Ví dụ: câu lệnh sau gán các tham chiếu này cho các đối tượng mới:

```
today = new java.util.Date();
greeting = new String("How are you?");
```



c) Phân biệt giữa primitives và reference types

Có một số khác biệt quan trọng mà bạn nên biết giữa kiểu nguyên thủy và kiểu tham chiếu.

Đầu tiên, các kiểu tham chiếu có thể được gán null, có nghĩa là chúng hiện không tham chiếu đến một đối tượng. Các kiểu nguyên thủy sẽ gây ra lỗi trình biên dịch nếu bạn cố gán chúng là null.

```
int value = null; // DOES NOT COMPILE
String s = null;
```

Tiếp theo, các kiểu tham chiếu có thể được sử dụng để gọi các phương thức, giả sử tham chiếu không rỗng. Nguyên thủy không có phương thức được khai báo trên chúng.

Cuối cùng, lưu ý rằng tất cả các kiểu nguyên thủy đều có tên kiểu chữ thường. Tất cả các lớp đi kèm với Java đều bắt đầu bằng chữ hoa. Mặc dù không bắt buộc nhưng đây là thông lệ tiêu chuẩn và cũng nên tuân theo quy ước này đối với các class bạn tạo.

3. Declaring Variables

a) Identifying identifiers

Java có các quy tắc chính xác về tên định danh. Mã định danh - identifier là tên của một biến, phương thức, class, interface hoặc package. Chỉ có bốn quy tắc cần nhớ đối với identifier hợp pháp:

Identifier phải bắt đầu bằng một chữ cái, ký hiệu $ hoặc ký hiệu _.

Identifier có thể bao gồm các số nhưng không bắt đầu bằng chúng.

Kể từ Java 9, một dấu gạch dưới _ không được phép làm identifier.

Bạn không thể sử dụng cùng tên với một từ dành riêng cho Java. Từ dành riêng là từ đặc biệt mà Java đã giữ lại để bạn không được phép sử dụng nó. Hãy nhớ rằng Java phân biệt chữ hoa chữ thường, vì vậy bạn có thể sử dụng các phiên bản của từ khóa chỉ khác nhau về chữ hoa chữ thường.

Reserved words - Từ dành riêng

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

Các ví dụ sau đây là hợp lệ:

```
long okidentifier;
float $OK2Identifier;
boolean _alsoOK1d3ntifi3r;
char __SStillOkbutKnotsonice$;
```

Những ví dụ này không hợp lệ:

```
int 3DPointClass; // identifiers không thể bắt đầu bằng số
byte hollywood@vine; // @ không phải là chữ cái, chữ số, $ hoặc _
String *$coffee; // * không phải là chữ cái, chữ số, $ hoặc _
double public; // public là reserved word
short _; // một dấu gạch dưới không được phép
```

Style: camelCase

Java có các quy ước để mã dễ đọc và nhất quán. Tính nhất quán này bao gồm cả camel case - lạc đà, thường được viết là camel case để nhấn mạnh. Trong CamelCase, chữ cái đầu tiên của mỗi từ được viết hoa. Định dạng CamelCase giúp mã định danh dễ đọc hơn. Bạn thích đọc cái nào hơn: Tên Thisismyclass hay tên ThisIsMyClass?

Khi bạn nhìn thấy số nhận dạng không chuẩn, hãy nhớ kiểm tra xem nó có hợp pháp hay không.

Các nhà phát triển Java đều tuân theo các quy ước sau về tên định danh:

- Tên **phương thức** và **biến** được viết bằng CamelCase với chữ cái đầu tiên là chữ thường
- Tên **class** và **interface** được viết bằng CamelCase với chữ cái đầu tiên là chữ hoa. Ngoài ra, không bắt đầu bất kỳ tên class bằng $, vì trình biên dịch sử dụng ký hiệu này cho một số tệp.

Style: snake_case

Nó chỉ đơn giản sử dụng dấu gạch dưới (_) để phân tách các từ, thường hoàn toàn bằng chữ thường. Ví dụ trước sẽ được viết dưới dạng tên this_is_my_class trong snake_case

Các giá trị **static final** đổi thường được viết bằng snake_case, chẳng hạn như THIS_IS_A_CONSTANT. Ngoài ra, các giá trị **enum** có xu hướng được viết bằng snake_case, như trong Color.RED, Color.DARK_GRAY, v.v.

b) Declaring multiple variables

Bạn cũng có thể khai báo và khởi tạo nhiều biến trong cùng một câu lệnh.

```
void sandFence() {
    String s1, s2;
    String s3 = "yes", s4 = "no";
    int i1, i2, i3 = 0;
}
```

Đoạn code sau không được phép:

```
int num, String value; // DOES NOT COMPILE
```

Mã này không biên dịch được vì nó cố gắng khai báo nhiều biến thuộc các loại khác nhau trong cùng một câu lệnh.

4. Initializing Variables

a) Creating local variables

Biến cục bộ - **local variable** là một biến được xác định trong constructor, phương thức hoặc initializer block.

Biến cục bộ không có giá trị mặc định và phải được khởi tạo trước khi sử dụng. Hơn nữa, trình biên dịch sẽ báo lỗi nếu bạn cố đọc một giá trị chưa được khởi tạo. Ví dụ: đoạn mã sau tạo ra lỗi trình biên dịch:

```
public int notValid() {
    int y = 10;
    int x;
    int reply = x + y; // DOES NOT COMPILE
    return reply;
}
```

Tuy nhiên, vì x không được khởi tạo trước khi nó được sử dụng trong biểu thức nên trình biên dịch sẽ thông báo lỗi.

Trình biên dịch đủ thông minh để nhận ra các biến đã được khởi tạo sau khi khai báo nhưng trước khi chúng được sử dụng. Đây là một ví dụ:

```
public int valid() {
    int y = 10;
    int x; // x is declared here
    x = 3; // and initialized here
    int reply = x + y;
    return reply;
}
```

b) Passing constructor and method parameters

Các biến được truyền cho constructor hoặc phương thức tương ứng được gọi là tham số constructor hoặc tham số phương thức. Các tham số này là các biến cục bộ đã được khởi tạo trước. Nói cách khác, chúng giống như các biến cục bộ đã được khởi tạo trước khi phương thức được gọi bởi người gọi.

Các quy tắc khởi tạo tham số của constructor và phương thức là như nhau, vì vậy chúng ta sẽ tập trung chủ yếu vào các tham số của phương thức. Trong ví dụ trước, **check** là một tham số phương thức:

```
public void findAnswer(boolean check) {}
```

Hãy xem phương thức checkAnswer() sau đây trong cùng một lớp:

```
public void checkAnswer() {
    boolean value;
    findAnswer(value); // DOES NOT COMPILE
}
```

Lệnh gọi findAnswer() không biên dịch được vì nó cố gắng sử dụng một biến chưa được khởi tạo. Trong khi người gọi phương thức checkAnswer() cần quan tâm đến biến đang được khởi tạo, thì khi ở trong phương thức findAnswer(), chúng ta có thể giả sử biến cục bộ đã được khởi tạo thành một giá trị nào đó.

c) Defining instance and class variables

Các biến không phải là biến cục bộ được định nghĩa là biến instance hoặc biến class. Một biến instance, thường được gọi là field, là một giá trị được xác định trong một instance cụ thể của một đối tượng. Giả sử chúng ta có một lớp Person với tên biến name là kiểu String. Mỗi instance của class sẽ có giá trị tên riêng, chẳng hạn như Elysia hoặc Sarah. Hai instance có thể có cùng giá trị về name, nhưng việc thay đổi giá trị của một instance sẽ không sửa đổi instance kia.

Mặt khác, biến class là biến được xác định ở cấp độ clss và được chia sẻ giữa tất cả các instance của lớp. Nó thậm chí có thể được truy cập public đối với các lớp bên ngoài lớp mà không yêu cầu sử dụng instance. Chỉ cần biết rằng một biến là biến class nếu nó có từ khóa static trong phần khai báo.

Các biến instance và class không yêu cầu bạn khởi tạo chúng. Ngay khi bạn khai báo các biến này, chúng sẽ có giá trị mặc định.

Variable type	Default initialization value
boolean	FALSE
byte, short, int, long	0
float, double	0
char	'\u0000' (NUL)
All object references (everything else)	null

d) Var

Bắt đầu từ Java 10, có tùy chọn sử dụng từ khóa var thay vì loại cho các biến cục bộ trong một số điều kiện nhất định. Để sử dụng tính năng này, bạn chỉ cần gõ var thay vì kiểu nguyên thủy hoặc kiểu tham chiếu. Đây là một ví dụ:

```
public void whatTypeAmI() {
    var name = "Hello";
    var size = 7;
}
```

Chỉ có thể sử dụng tính năng này cho các biến cục bộ:

```
public class VarKeyword {
    var tricky = "Hello"; // DOES NOT COMPILE
}
```

Biến **tricky** là một biến instance. Nên không được phép sử dụng var.

Type Inference of var

Khi bạn sử dụng var, bạn đang hướng dẫn trình biên dịch xác định kiểu dữ liệu cho bạn. Trình biên dịch xem code trên dòng khai báo và sử dụng nó để suy ra kiểu. Hãy xem ví dụ này:

```
7:  public void reassignment() {
8:      var number = 7;
9:      number = 4;
10:     number = "five"; // DOES NOT COMPILE
11: }
```

Ở dòng 8, trình biên dịch xác định rằng chúng ta muốn khai báo một biến int. Ở dòng 9, chúng ta không gặp khó khăn gì khi gán một biến int khác cho nó. Ở dòng 10, Java có vấn đề. Chúng tôi đã yêu cầu nó gán Chuỗi cho biến int. Điều này không được phép.

Vì vậy, kiểu của var không thể thay đổi trong thời gian chạy, nhưng còn giá trị thì sao? Hãy xem đoạn mã sau:

```
var apples = (short)10;
apples = (byte)5;
apples = 1_000_000; // DOES NOT COMPILE – giá trị vượt quá giới hạn
```

*Examples with **var***

```
3:  public void doesThisCompile(boolean check) {
4:      var question;
5:      question = 1;
6:      var answer;
7:      if (check) {
8:          answer = 2;
9:      } else {
10:         answer = 3;
11:     }
12:     System.out.println(answer);
13: }
```

Mã không biên dịch. Tại sao lại không biên dịch được ?

Ví dụ khác ? Bạn có thể hiểu tại sao hai câu lệnh này không biên dịch được không?

```
4: public void twoTypes() {
5:      int a, var b = 3; // DOES NOT COMPILE
6:      var n = null; // DOES NOT COMPILE
7: }
```

Dòng 5 sẽ không hoạt động ngay cả khi bạn thay thế var bằng float. Tất cả các kiểu được khai báo trên một dòng phải cùng kiểu và có chung một khai báo. Chúng ta không thể viết int a, int v = 3; hoặc. Tương tự như vậy, điều này không được phép:

```
5: var a = 2, b = 3; // DOES NOT COMPILE
```

Nói cách khác, Java không cho phép var khai báo nhiều biến. Dòng 6, trình biên dịch đang được yêu cầu suy ra kiểu dữ liệu của null. Đây có thể là bất kỳ loại reference type. Lựa chọn duy nhất mà trình biên dịch có thể thực hiện là Object. Tuy nhiên, đó gần như chắc chắn không phải là điều mà dev của đoạn mã dự định. Các nhà thiết kế của Java đã quyết định rằng tốt hơn là không cho phép var null hơn là phải đoán ý định.

Lưu ý 2 trường hợp dưới đây là hợp lệ:

```
13:     var n = "myData";
14:     n = null;
15:     var o = (String)null;
```

Hãy thử một ví dụ khác. Bạn có thấy tại sao điều này không biên dịch?

```
public int addition(var a, var b) { // DOES NOT COMPILE
    return a + b;
}
```

Trong ví dụ này, a và b là các tham số của phương thức. Đây không phải là các biến cục bộ.

Ví dụ sau hợp lệ:

```
package var;
public class Var {
    public void var() {
        var var = "var";
    }
    public void Var() {
        Var var = new Var();
    }
}
```

Mặc dù var không phải là một reserved word và được phép sử dụng làm identifier nhưng nó được coi là reserved type.

Reserved type có nghĩa là nó không thể được sử dụng để xác định một kiểu dữ liệu, chẳng hạn như class, interface hoặc enum.

Review of var Rules

- var được sử dụng làm biến cục bộ trong constructor, method, hoặc initializer block.
- Không thể sử dụng var trong tham số constructor, tham số method, biến instance hoặc biến class.
- var luôn được khởi tạo trên cùng một dòng (hoặc câu lệnh) nơi nó được khai báo.
- Giá trị của biến có thể thay đổi nhưng kiểu thì không.
- var không thể được khởi tạo với giá trị null nếu không có loại.
- Không được phép sử dụng var trong khai báo nhiều biến.
- var là reserved type nhưng không phải là reserved word, nghĩa là nó có thể được sử dụng làm identifier ngoại trừ tên lớp, giao diện hoặc tên enum.

5. Managing Variable Scope

a) Limiting scope

Các biến cục bộ không bao giờ có phạm vi lớn hơn phương thức mà chúng được xác định. Tuy nhiên, chúng có thể có phạm vi nhỏ hơn. Hãy xem xét ví dụ này:

```
3:  public void eatIfHungry(boolean hungry) {
4:      if (hungry) {
5:          int bitesOfCheese = 1;
6:      } // bitesOfCheese goes out of scope here
7:      System.out.println(bitesOfCheese); // DOES NOT COMPILE
8:  }
```

hungry có phạm vi của toàn bộ phương thức, trong khi biến **bitesOfCheese** có phạm vi nhỏ hơn. Nó chỉ có sẵn để sử dụng trong câu lệnh if vì nó được khai báo bên trong nó.

b) Nesting scope

Hãy nhớ rằng các block có thể chứa các block khác. Các block nhỏ hơn này có thể tham chiếu các biến được xác định trong các block có phạm vi lớn hơn, nhưng không phải ngược lại. Đây là một ví dụ:

```
16:  public void eatIfHungry(boolean hungry) {
17:     if (hungry) {
18:         int bitesOfCheese = 1;
19:         {
20:             var teenyBit = true;
21:             System.out.println(bitesOfCheese);
22:         }
23:     }
24:     System.out.println(teenyBit); // DOES NOT COMPILE
25:  }
```

c) Applying scope to classes

Quy tắc dành cho các biến instance dễ dàng hơn: chúng có sẵn ngay khi được xác định và tồn tại trong toàn bộ thời gian tồn tại của đối tượng.

Quy tắc dành cho class, hay còn gọi là static, các biến thậm chí còn dễ dàng hơn: chúng đi vào phạm vi khi được khai báo giống như các loại biến khác. Tuy nhiên, chúng vẫn nằm trong phạm vi của toàn bộ vòng đời của chương trình.

```
1:      public class Mouse {
2:          final static int MAX_LENGTH = 5;
3:          int length;
4:          public void grow(int inches) {
5:              if (length < MAX_LENGTH) {
6:                  int newSize = length + inches;
7:                  length = newSize;
8:              }
9:          }
10:     }
```

- Biến MAX_LENGTH là biến class vì nó có từ khóa static trong khai báo. Trong trường hợp này, MAX_LENGTH nằm trong phạm vi ở dòng 2 nơi nó được khai báo. Nó vẫn ở trong phạm vi cho đến khi chương trình kết thúc.
- Tiếp theo, length nằm trong phạm vi ở dòng 3 nơi nó được khai báo. Nó vẫn nằm trong phạm vi miễn là đối tượng Mouse này tồn tại.
- inches đi vào phạm vi được khai báo ở dòng 4. Nó nằm ngoài phạm vi ở cuối phương thức trên dòng 9.
- newSize đi vào phạm vi được khai báo ở dòng 6. Vì nó được xác định bên trong block câu lệnh if, nên nó vượt quá phạm vi khi block đó kết thúc ở dòng 8.

d) Reviewing scope

- Biến cục bộ: Trong phạm vi từ khai báo đến cuối block
- Biến Instance: Trong phạm vi từ khai báo cho đến khi đối tượng đủ điều kiện để thu gom rác
- Biến class: Trong phạm vi từ khi khai báo cho đến khi kết thúc chương trình

6. Destroying Objects

a) Understanding garbage collection

Garbage collection - Thu gom rác đề cập đến quá trình tự động giải phóng bộ nhớ trên heap bằng cách xóa các đối tượng không còn truy cập được trong chương trình của bạn. Có nhiều thuật toán khác nhau để thu thập rác, một thuật toán là giữ một bộ đếm về số lượng vị trí mà một đối tượng có thể truy cập vào bất kỳ thời điểm nào và đánh dấu nó đủ điều kiện để thu gom rác nếu bộ đếm đạt tới 0.

Điều kiện để Garbage Collection

Trong Java và các ngôn ngữ khác, đủ điều kiện để thu gom rác đề cập đến trạng thái của một đối tượng không còn có thể truy cập được trong chương trình và do đó có thể được thu gom rác.

Quá trình thu dọn rác trong Java được thực hiện bởi bộ thu dọn rác (garbage collector) của máy ảo Java (JVM). Garbage collector tự động quét và thu dọn các đối tượng không còn được sử dụng trong bộ nhớ để giải phóng tài nguyên và tái sử dụng không gian bộ nhớ.

Garbage collector hoạt động theo các bước sau:

- Phát hiện các đối tượng không còn được tham chiếu: Garbage collector kiểm tra tất cả các đối tượng trong bộ nhớ và xác định xem chúng có còn được tham chiếu từ các đối tượng khác hay không. Nếu một đối tượng không còn được tham chiếu từ bất kỳ đối tượng nào, nó được coi là không còn sử dụng và sẽ được đánh dấu để thu dọn.
- Đánh dấu các đối tượng không còn được sử dụng: Các đối tượng không còn được sử dụng được đánh dấu bằng cách thêm thông tin đánh dấu vào chúng. Điều này giúp garbage collector biết rằng các đối tượng này có thể được thu dọn.
- Thu dọn các đối tượng không còn được sử dụng: Các đối tượng đã được đánh dấu không còn sử dụng sẽ được thu dọn bằng cách giải phóng không gian bộ nhớ mà chúng chiếm giữ. Quá trình này gọi là thu dọn rác. Garbage collector sẽ tự động xóa các đối tượng không còn sử dụng và giải phóng không gian bộ nhớ.

Điều này có nghĩa là một đối tượng đủ điều kiện để thu gom rác sẽ được thu gom rác ngay lập tức phải không? Chắc chắn không phải. Quá trình thu dọn rác sẽ được kích hoạt sau đó theo một lịch trình hoặc khi một số điều kiện xảy ra. Các cách thức kích hoạt quá trình thu dọn rác có thể khác nhau tùy thuộc vào thuật toán được sử dụng bởi garbage collector.

Là một lập trình viên, điều quan trọng nhất bạn có thể làm để hạn chế vấn đề hết bộ nhớ là đảm bảo các đối tượng đủ điều kiện để thu gom rác khi chúng không còn cần thiết nữa. Trách nhiệm của JVM là thực sự thực hiện việc thu thập rác.

Calling System.gc()

Java bao gồm một phương thức tích hợp sẵn để giúp hỗ trợ việc thu gom rác có thể được gọi bất cứ lúc nào.

```
public static void main(String[] args) {
    System.gc();
}
```

Lệnh System.gc() được đảm bảo thực hiện những gì? Thực ra không có gì. Nó chỉ gợi ý rằng JVM khởi động việc thu thập rác. JVM có thể thực hiện thu thập rác tại thời điểm đó hoặc có thể nó đang bận và chọn không thực hiện. JVM có quyền bỏ qua yêu cầu.

Khi nào System.gc() được đảm bảo được JVM gọi? Thực ra là không bao giờ. Mặc dù JVM có thể sẽ chạy nó theo thời gian khi bộ nhớ khả dụng giảm đi nhưng không đảm bảo nó sẽ thực sự chạy được. Trên thực tế, ngay trước khi một chương trình hết bộ nhớ và ném ra OutOfMemoryError, JVM sẽ cố gắng thực hiện thu gom rác nhưng không đảm bảo sẽ thành công.

b) Tracing eligibility

Một đối tượng sẽ vẫn còn trên heap cho đến khi không thể truy cập được nữa. Một đối tượng không thể truy cập được nữa khi một trong hai trường hợp xảy ra:

- Đối tượng không còn có bất kỳ tham chiếu nào trỏ đến nó nữa.
- Tất cả các tham chiếu đến đối tượng đã vượt quá phạm vi.

Ví dụ:

```
1:      public class Scope {
2:          public static void main(String[] args) {
3:              String one, two;
4:              one = new String("a");
5:              two = new String("b");
6:              one = two;
7:              String three = one;
8:              one = null;
9:          }
10:     }
```

Ở dòng 6, "a" đủ điều kiện để thu gom rác. "b" không nằm ngoài phạm vi cho đến khi kết thúc phương thức ở dòng 9.

Chapter 3: Operators

1. Understanding Java Operators

a) Types of operators

Nói chung, có ba loại toán tử có sẵn trong Java: đơn nguyên, nhị phân và ba ngôi. Những loại toán tử này có thể được áp dụng tương ứng cho một, hai hoặc ba toán hạng.

```
int cookies = 4;
double reward = 3 + 2 * --cookies;
System.out.print("Zoo animal receives: "+reward+" reward points");
```

b) Operator precedence – Độ ưu tiên

Trong toán học, một số toán tử nhất định có thể ghi đè các toán tử khác và được đánh giá trước. Việc xác định toán tử nào được đánh giá theo thứ tự nào được gọi là độ ưu tiên của toán tử. Theo cách này, Java tuân thủ chặt chẽ hơn các quy tắc toán học.

```
var perimeter = 2 * height + 2 * length;
```

Toán tử nhân (*) có độ ưu tiên cao hơn toán tử cộng (+) nên chiều cao và chiều dài đều được nhân với 2 trước khi cộng lại với nhau. Toán tử gán (=) có thứ tự ưu tiên thấp nhất, do đó việc gán cho biến chu vi được thực hiện sau cùng.

Thứ tự ưu tiên của toán tử

Operator	**Symbols and examples**
Post-unary operators	*expression*++, *expression*--
Pre-unary operators	++*expression*, --*expression*
Other unary operators	-, !, ~, +, (*type*)
Multiplication/division/modulus	*, /, %
Addition/subtraction	+, -
Shift operators	<<, >>, >>>
Relational operators	<, >, <=, >=, instanceof
Equal to/not equal to	==, !=
Logical operators	&, ^, |
Short-circuit logical operators	&&, ||
Ternary operators	*boolean expression* ? *expression1* : *expression2*
Assignment operators	=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=

2. Applying Unary Operators

Theo định nghĩa, toán tử một ngôi là toán tử yêu cầu chính xác một toán hạng hoặc biến để hoạt động.

Operator	Description
!	Đảo ngược giá trị logic của boolean
+	Cho biết một số là dương
-	Cho biết một số bằng chữ là âm hoặc phủ định một biểu thức
++	Tăng giá trị lên 1
--	Giảm giá trị đi 1
(type)	Truyền một giá trị tới một loại cụ thể.

a) Logical complement and negation

Toán tử bổ sung logic (!) lật giá trị của biểu thức boolean. Ví dụ: nếu giá trị là true thì nó sẽ được chuyển thành false và ngược lại. Để minh họa điều này, hãy so sánh kết quả đầu ra của các câu lệnh sau:

```
boolean isAnimalAsleep = false;
System.out.println(isAnimalAsleep); // false
isAnimalAsleep = !isAnimalAsleep;
System.out.println(isAnimalAsleep); // true
```

Tương tự, toán tử phủ định, -, đảo ngược dấu của biểu thức số, như được hiển thị trong các câu lệnh sau:

```
double zooTemperature = 1.21;
System.out.println(zooTemperature); // 1.21
zooTemperature = -zooTemperature;
System.out.println(zooTemperature); // -1.21
zooTemperature = -(-zooTemperature);
System.out.println(zooTemperature); // -1.21
```

b) Increment and decrement operators

Các toán tử tăng và giảm, ++ và --, tương ứng, có thể được áp dụng cho các biến số và có thứ tự ưu tiên cao hơn so với các toán tử nhị phân.

Các toán tử tăng và giảm cần được chú ý đặc biệt vì thứ tự chúng được gắn vào biến liên quan có thể tạo ra sự khác biệt trong cách xử lý một biểu thức.

Nếu toán tử được đặt trước toán hạng, được gọi là *pre-increment operator* và *predecrement operator*, thì toán tử được áp dụng trước và giá trị trả về là giá trị mới của biểu thức.

Ngoài ra, nếu toán tử được đặt sau toán hạng, được gọi là *post-increment operator* và *post-decrement operator*, thì giá trị ban đầu của biểu thức sẽ được trả về, với toán tử được áp dụng sau khi giá trị được trả về.

Đoạn mã sau minh họa sự khác biệt này:

```
int parkAttendance = 0;
System.out.println(parkAttendance); // 0
System.out.println(++parkAttendance); // 1
System.out.println(parkAttendance); // 1
System.out.println(parkAttendance--); // 1
System.out.println(parkAttendance); // 0
```

3. Working with Binary Arithmetic Operators

a) Arithmetic operators

Operator	Description
+	Adds two numeric values
-	Subtracts two numeric values
*	Multiplies two numeric values
/	Divides one numeric value by another
%	Modulus operator returns the remainder after division of one numeric value by another

b) Numeric promotion

Numeric Promotion Rules – Nguyên tắc thăng cấp kiểu dữu liệu

1. Nếu hai giá trị có kiểu dữ liệu khác nhau, Java sẽ tự động nâng cấp một trong các giá trị lên giá trị lớn hơn trong hai kiểu dữ liệu.
2. Nếu một trong các giá trị là nguyên và giá trị còn lại là dấu phẩy động, Java sẽ tự động thăng cấp giá trị nguyên thành kiểu dữ liệu của giá trị dấu phẩy động.
3. Các kiểu dữ liệu nhỏ hơn, cụ thể là byte, short và char, trước tiên được thăng cấp thành int bất cứ khi nào chúng được sử dụng với toán tử số học nhị phân Java, ngay cả khi cả hai toán hạng đều không phải là int.
4. Sau khi tất cả việc thăng hạng đã diễn ra và các toán hạng có cùng kiểu dữ liệu, giá trị thu được sẽ có cùng kiểu dữ liệu với các toán hạng được thăng hạng của nó.
   Một số ví dụ cho mục đích minh họa:

- Kiểu dữ liệu của x * y là gì?

```
int x = 1;
long y = 33;
var z = x * y;
```

Nếu chúng ta tuân theo quy tắc đầu tiên, giá trị kết quả là long.

- Kiểu dữ liệu của x + y?

```
double x = 39.21;
float y = 2.1f;
var z = x + y;
```

⇨ Áp dụng quy tắc số 2, kết quả là double

- Kiểu dữ liệu của x * y?

```
short x = 10;
short y = 3;
var z = x * y;
```

⇨ Áp dụng quy tắc thứ ba, cụ thể là x và y đều sẽ được thăng cấp thành int trước phép nhân, dẫn đến kết quả đầu ra là int.

- Kiểu dữ liệu của x * y?

```
short w = 14;
float x = 13;
double y = 30;
var z = w * x / y;
```

⇨ Áp dụng tất cả các quy tắc. Đầu tiên, w sẽ tự động được thăng cấp thành int. Giá trị w được thăng cấp sau đó sẽ tự động được thăng cấp thành số float để có thể nhân với x. Kết quả của w * x sau đó sẽ tự động được thăng cấp lên double để có thể chia cho y

4. Assigning Values

a) Casting values

Casting là việc gán giá trị của một biến có kiểu dữ liệu này sang biến khác có kiểu dữ liệu khác. **Casting** là tùy chọn và không cần thiết khi chuyển đổi sang loại dữ liệu lớn hơn hoặc mở rộng, nhưng nó là bắt buộc khi chuyển đổi sang loại dữ liệu nhỏ hơn hoặc thu hẹp. Nếu không **casting**, trình biên dịch sẽ tạo ra lỗi khi cố gắng đặt kiểu dữ liệu lớn hơn vào kiểu dữ liệu nhỏ hơn. Việc truyền được thực hiện bằng cách đặt kiểu dữ liệu được đặt trong dấu ngoặc đơn ở bên trái giá trị bạn muốn truyền.

Ví dụ:

```
int fur = (int)5;
int hair = (short) 2;
String type = (String) "Bird";
short tail = (short)(4 + 10);
long feathers = 10(long); // DOES NOT COMPILE
```

Nới rộng (widening)

Nới rộng (widening): Là quá trình làm tròn số từ kiểu dữ liệu có kích thước nhỏ hơn sang kiểu có kích thước lớn hơn. Kiểu biến đổi này không làm mất thông tin. Ví dụ chuyển từ int sang float. Chuyển kiểu loại này có thể được thực hiện ngầm định bởi trình biên dịch.

byte -> short -> int -> long -> float -> double

```
public class TestWidening {
    public static void main(String[] args) {
        int i = 100;
        long l = i;    // không yêu cầu chỉ định ép kiểu
        float f = l;   // không yêu cầu chỉ định ép kiểu
        System.out.println("Giá trị Int: " + i); // Giá trị Int: 100
        System.out.println("Giá trị Long: " + l); // Giá trị Long: 100
```

```
        System.out.println("Giá trị Float:  " + f); // Giá trị Float: 100.0
    }
}
```

Thu hẹp (narrowwing)

Thu hẹp (narrowwing): Là quá trình làm tròn số từ kiểu dữ liệu có kích thước lớn hơn sang kiểu có kích thước nhỏ hơn. Kiểu biến đổi này có thể làm mất thông tin như ví dụ ở trên. Chuyển kiểu loại này không thể thực hiện ngầm định bởi trình biên dịch, người dùng phải thực hiện chuyển kiểu tường minh.

double -> float -> long -> int -> short -> byte

```
public class TestNarrowwing {
    public static void main(String[] args) {
        double d = 100.04;
        long l = (long) d; // yêu cầu chỉ định kiểu dữ liệu (long)
        int i = (int) l; // yêu cầu chỉ định kiểu dữ liệu (int)

        System.out.println("Giá trị Double: " + d);
        System.out.println("Giá trị Long: " + l);
        System.out.println("Giá trị Int:  " + i);
    }
}
```

b) Compound assignment operators

Operator	Description
+=	Cộng giá trị bên phải vào biến ở bên trái và gán tổng cho biến
-=	Trừ giá trị bên phải cho biến ở bên trái và gán hiệu cho biến
*=	Nhân giá trị ở bên phải với biến ở bên trái và gán tích cho biến đó
/=	Chia biến ở bên trái cho giá trị ở bên phải và gán thương cho biến

c) Assignment operator return value

Một điều cuối cùng cần biết về toán tử gán là kết quả của phép gán là một biểu thức bên trong và của chính nó, bằng giá trị của phép gán. Ví dụ: đoạn mã sau đây hoàn toàn hợp lệ:

```
long wolf = 5;
long coyote = (wolf=3);
System.out.println(wolf); // 3
System.out.println(coyote); // 3
boolean healthy = false;
if(healthy = true)
   System.out.print("Good!"); // Good
```

5. Comparing Values

a) Equality operators

Operator	Apply to primitives	Apply to objects
==	Trả về true nếu hai giá trị đại diện cho cùng một giá trị	Trả về true nếu hai giá trị tham chiếu đến cùng một đối tượng
!=	Trả về true nếu hai giá trị đại diện cho các giá trị khác nhau	Trả về true nếu hai giá trị không tham chiếu cùng một đối tượng

Các toán tử đẳng thức được sử dụng trong một trong ba trường hợp:

- So sánh hai kiểu nguyên thủy số hoặc ký tự. Nếu các giá trị số thuộc các loại dữ liệu khác nhau thì các giá trị đó sẽ tự động được thăng cấp. Ví dụ: 5 == 5,00 trả về true vì vế trái được tăng gấp đôi.
- So sánh hai giá trị boolean
- So sánh hai đối tượng, bao gồm giá trị null và Chuỗi

Ví dụ: mỗi code sau đây sẽ dẫn đến lỗi trình biên dịch:

```
boolean monkey = true == 3; // DOES NOT COMPILE
boolean ape = false != "Grape"; // DOES NOT COMPILE
boolean gorilla = 10.2 == "Koko"; // DOES NOT COMPILE
```

b) Relational operators

Operator	Description
<	"Trả về true nếu giá trị bên trái nhỏ hơn giá trị bên phải"
<=	Trả về true nếu giá trị bên trái nhỏ hơn hoặc bằng giá trị bên phải
>	Trả về true nếu giá trị bên trái lớn hơn giá trị bên phải
>=	Trả về true nếu giá trị bên trái là lớn hơn hoặc bằng giá trị ở bên phải
a instance of b	Trả về true nếu tham chiếu mà a trỏ tới là một thể hiện của một lớp, lớp con hoặc lớp thực hiện một giao diện cụ thể, như được đặt tên trong b

```
public static void openZoo(Number time) {
    if(time instanceof Integer)
        System.out.print((Integer)time + " O'clock");
    else
        System.out.print(time);
}
```

c) Logical operators

Các toán tử logic, (&), (|) và (^), có thể được áp dụng cho cả kiểu dữ liệu số và kiểu dữ liệu boolean;

Operator	Description
&	Logic AND chỉ đúng nếu cả hai giá trị đều đúng.
|	OR là đúng nếu ít nhất một trong các giá trị là đúng.
^	XOR chỉ đúng nếu một giá trị là đúng và giá trị kia là sai.

d) Short-circuit operators

Operator	Description
&&	Short-circuit AND chỉ đúng nếu cả hai giá trị đều đúng. Nếu bên trái sai thì bên phải sẽ không được đánh giá.
||	Short-circuit OR là đúng nếu ít nhất một trong các giá trị là đúng. Nếu vế trái đúng thì vế phải sẽ không được đánh giá.

Short-circuit gần giống với các toán tử logic & và |, ngoại trừ việc phía bên phải của biểu thức có thể không bao giờ được đánh giá nếu kết quả cuối cùng có thể được xác định bởi phía bên trái của biểu thức.

6. Making Decisions with the Ternary Operator

Toán tử cuối cùng mà bạn nên biết là toán tử có điều kiện, ? :, còn được gọi là toán tử ba ngôi. Điều đáng chú ý là nó là toán tử duy nhất có ba toán hạng. Toán tử 3 ngôi có dạng sau:

```
booleanExpression ? expression1 : expression2
```

Toán hạng đầu tiên phải là biểu thức boolean, toán hạng thứ hai và thứ ba có thể là bất kỳ biểu thức nào trả về một giá trị. Phép toán ba ngôi thực sự là một dạng cô đọng của câu lệnh if và else kết hợp trả về một giá trị.

Ví dụ:

```
int owl = 5;
int food;
if(owl < 2) {
    food = 3;
} else {
    food = 4;
}
System.out.println(food); // 4
```

Tương đương:

```
int owl = 5;
int food = owl < 2 ? 3 : 4;
System.out.println(food); // 4
```

Chapter 4: Making Decisions

1. Creating Decision-Making Statements

Câu lệnh Java là một đơn vị thực thi hoàn chỉnh trong Java, được kết thúc bằng dấu chấm phẩy (;). Các câu lệnh luồng điều khiển chia nhỏ luồng thực thi bằng cách sử dụng tính năng ra quyết định, lặp và phân nhánh, cho phép ứng dụng thực thi có chọn lọc các đoạn mã cụ thể. Những câu lệnh này có thể được áp dụng cho các biểu thức đơn lẻ cũng như một khối mã Java.

Một block code trong Java là một nhóm gồm 0 hoặc nhiều câu lệnh giữa các dấu ngoặc nhọn ({}) và có thể được sử dụng ở bất kỳ nơi nào cho phép một câu lệnh. Ví dụ: hai đoạn mã sau là tương đương nhau, đoạn mã đầu tiên là một biểu thức đơn và đoạn mã thứ hai là một khối câu lệnh:

```
// Single statement
patrons++;

// Statement inside a block
{
    patrons++;
}
```

Một statement hoặc block thường đóng vai trò là mục tiêu của một câu lệnh ra quyết định. Ví dụ: chúng ta có thể thêm câu lệnh if ra quyết định vào hai ví dụ sau:

```
// Single statement
if(ticketsTaken > 1)
    patrons++;

// Statement inside a block
if(ticketsTaken > 1)
{
    patrons++;
}
```

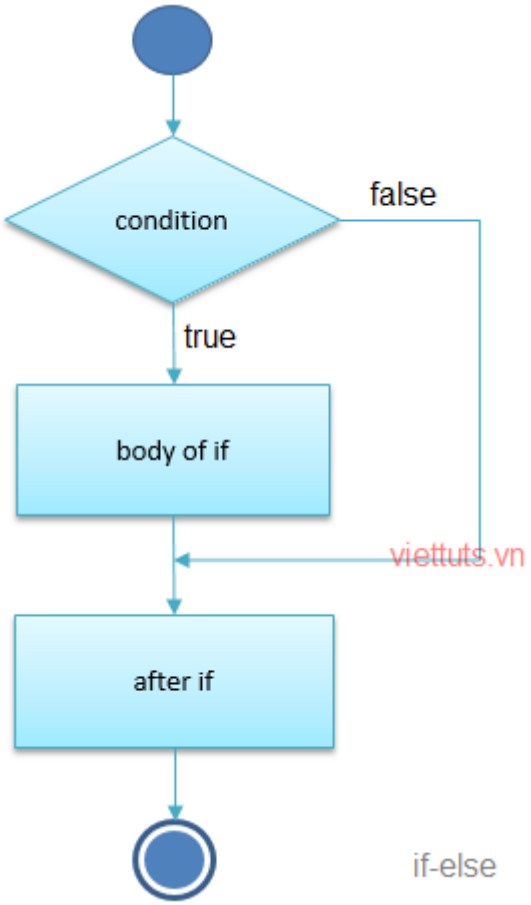
a) Statement if-else

Mệnh đề if

Mệnh đề if được sử dụng để kiểm tra giá trị dạng boolean của điều kiện. Khối lệnh sau if được thực thi nếu giá trị của điều kiện là **True**

Cú pháp:

```
if (condition) {
    // khối lệnh này thực thi
    // nếu condition = true
}
```



if-else

Mệnh đề if-else

Mệnh đề if-else cũng kiểm tra giá trị dạng boolean của điều kiện. Nếu giá trị điều kiện là **True** thì chỉ có khối lệnh sau if sẽ được thực hiện, nếu là **False** thì chỉ có khối lệnh sau else được thực hiện.

Cú pháp:

```
if (condition) {
    // khối lệnh này được thực thi
    // nếu condition = true
} else {
    // khối lệnh này được thực thi
    // nếu condition = false
}
```

Mệnh đề if-else-if

Mệnh đề if-else-if cũng kiểm tra giá trị dạng boolean của điều kiện. Nếu giá trị điều kiện if là **True** thì chỉ có khối lệnh sau if sẽ được thực hiện. Nếu giá trị điều kiện if else nào là **True** thì chỉ có khối lệnh sau else if đó sẽ được thực hiện... Nếu tất cả điều kiện của if và else if là **False** thì chỉ có khối lệnh sau else sẽ được thực hiện.

Cú pháp:

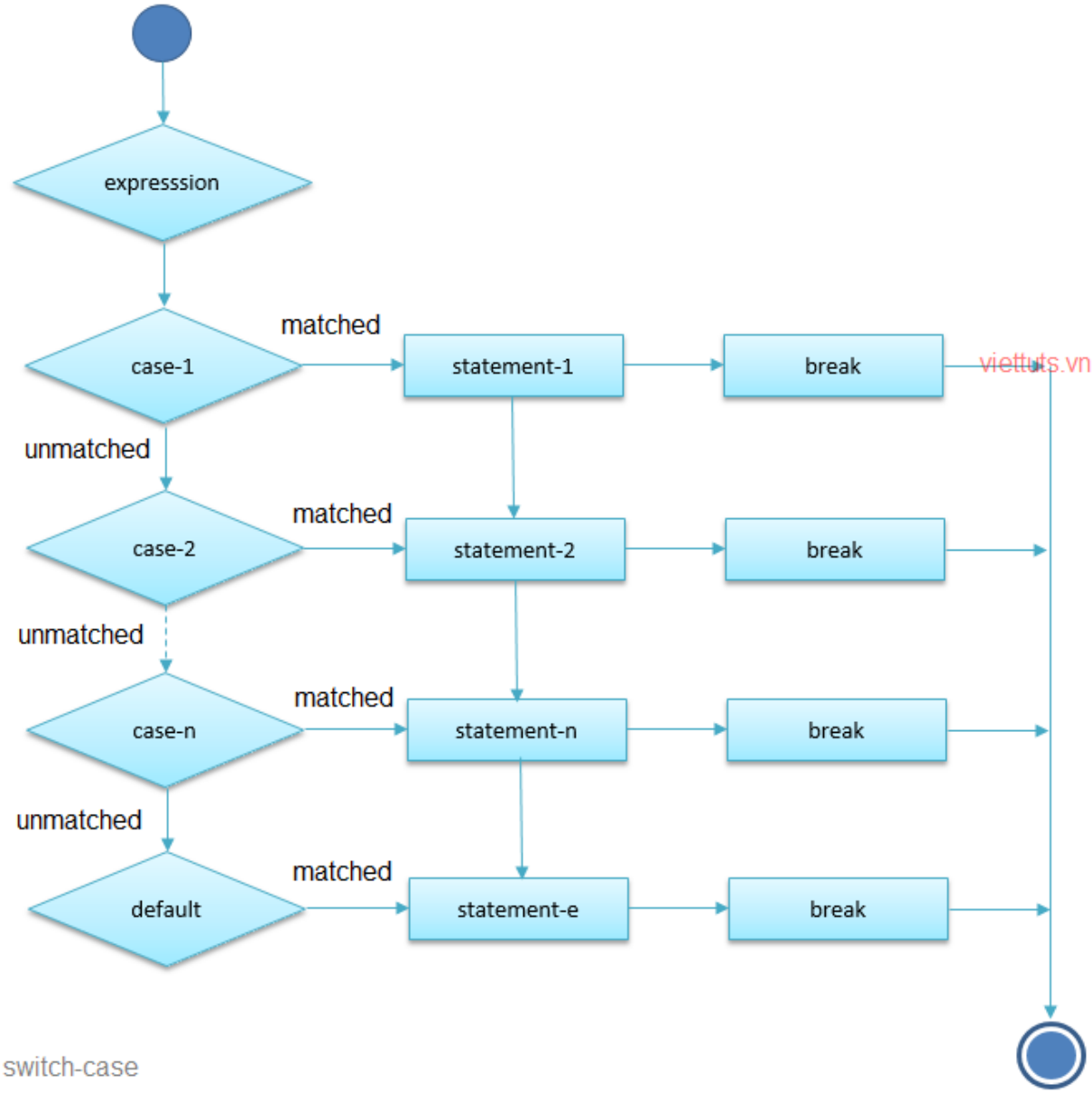
```
if (condition1) {
    // khối lệnh này được thực thi
    // nếu condition1 là true
} else if (condition2) {
    // khối lệnh này được thực thi
    // nếu condition2 là true
}  else if (condition3) {
    // khối lệnh này được thực thi
    // nếu condition3 là true
}
...
else {
    // khối lệnh này được thực thi
    // nếu tất cả những điều kiện trên là false
}
```

b) Switch statement

Mệnh đề switch-case trong java được sử dụng để thực thi 1 hoặc nhiều khối lệnh từ nhiều điều kiện.

Cú pháp:

```
switch (bieu_thuc) {
    case gia_tri_1:
        // Khối lệnh 1
        break;  //tùy chọn
    case gia_tri_2:
        // Khối lệnh 2
        break;  //tùy chọn
    ......
    case gia_tri_n:
        // Khối lệnh n
        break;  //tùy chọn
    default:
        // Khối lệnh này được thực thi
        // nếu tất cả các điều kiện trên không thỏa mãn
}
```



switch-case

Switch Data Types

Trước Java 5.0, **switch** chỉ có thể là giá trị int hoặc những giá trị có thể được thăng cấp thành int, cụ thể là byte, short, char hoặc int (kiểu số nguyên thủy).

Câu lệnh switch cũng hỗ trợ bất kỳ wrapper class nào của các kiểu số nguyên thủy này, chẳng hạn như Byte, Short, Character hoặc Integer.

Khi enum, được thêm vào Java 5.0, enum đã được thêm vào để câu lệnh **switch** hỗ trợ các giá trị enum. **enum** là một tập hợp cố định các giá trị không đổi, cũng có thể bao gồm các phương thức và biến class, tương tự như định nghĩa class.

Trong Java 7, các câu lệnh switch đã được cập nhật thêm để cho phép khớp các giá trị String. Trong Java 10, var được phân giải là một trong những loại được câu lệnh switch hỗ trợ, thì var cũng có thể được sử dụng trong câu lệnh switch

Sau đây là danh sách tất cả các kiểu dữ liệu được hỗ trợ bởi câu lệnh switch:

- int và Integer
- byte và Byte
- short và Short
- char và Character
- String
- enum
- var (nếu var có kiểu dữ liệu của các kiểu trên)\

Switch Control Flow

Hãy xem một ví dụ chuyển đổi đơn giản sử dụng ngày trong tuần, với 0 cho Chủ Nhật, 1 cho Thứ Hai, v.v.:

```
int dayOfWeek = 5;
switch(dayOfWeek) {
    default:
        System.out.println("Weekday");
        break;
    case 0:
        System.out.println("Sunday");
        break;
    case 6:
        System.out.println("Saturday");
        break;
}
```

Với giá trị dayOfWeek là 5, mã này sẽ xuất ra kết quả như sau: **Weekday**

Câu lệnh break chấm dứt câu lệnh switch và trả lại điều khiển luồng cho câu lệnh kèm theo. Nếu bạn bỏ qua câu lệnh break, luồng sẽ tự động tiếp tục đến trường hợp tiến hành tiếp theo hoặc default block. Một điều khác bạn có thể nhận thấy là default block không nằm ở cuối câu lệnh switch. Không có yêu cầu nào về trường hợp hoặc câu default block phải theo một thứ tự cụ thể.

```
var dayOfWeek = 5;
switch(dayOfWeek) {
    case 0:
        System.out.println("Sunday");
    default:
        System.out.println("Weekday");
    case 6:
        System.out.println("Saturday");
        break;
}
```

Với giá trị đã cho của dayOfWeek, 5, code sẽ nhảy đến default block và sau đó thực thi tất cả các câu lệnh tiến hành theo thứ tự cho đến khi tìm thấy câu lệnh break hoặc kết thúc câu lệnh switch. Output:

Weekday
Saturday

Acceptable Case Values

Không phải bất kỳ biến hoặc giá trị nào cũng có thể được sử dụng trong câu lệnh swith! Trước hết, các giá trị trong mỗi câu lệnh swith phải là các giá trị **hằng số** thời gian biên dịch có cùng kiểu dữ liệu với giá trị **switch**. Điều này có nghĩa là bạn chỉ có thể sử dụng hằng số, hằng số enum hoặc biến hằng số final của cùng một kiểu dữ liệu.

Ví dụ:

```
static final int getCookies() { return 4; }

void feedAnimals() {
    final int bananas = 1;
    int apples = 2;
    int numberOfAnimals = 3;
    final int cookies = getCookies();
    switch (numberOfAnimals) {
        case bananas:
        case apples: // DOES NOT COMPILES
        case getCookies(): // DOES NOT COMPILE
        case cookies : // DOES NOT COMPILE
        case 3 * 5 :
    }
}
```

Với các giá trị getCookies() và cookie, không biên dịch vì các phương thức không được đánh giá cho đến khi chạy, vì vậy chúng không thể được sử dụng làm giá trị của câu lệnh switch, ngay cả khi một trong các giá trị được lưu trữ trong biến final.

Một ví dụ phức tạp hơn:

```
private int getSortOrder(String firstName, final String lastName) {
    String middleName = "Patricia";
    final String suffix = "JR";
    int id = 0;
    switch(firstName) {
        case "Test":
            return 52;
        case middleName: // DOES NOT COMPILE
            id = 5;
            break;
        case suffix:
            id = 0;
            break;
        case lastName: // DOES NOT COMPILE
            id = 8;
            break;
        case 5: // DOES NOT COMPILE
            id = 7;
            break;
```

```
        case 'J': // DOES NOT COMPILE
            id = 10;
            break;
        case java.time.DayOfWeek.SUNDAY: // DOES NOT COMPILE
            id=15;
            break;
        }
    return id;
}
```

Numeric Promotion and Casting

```
short size = 4;
final int small = 15;
final int big = 1_000_000;
switch(size) {
    case small:
    case 1+2 :
    case big: // DOES NOT COMPILE
}
```

Trình biên dịch có thể dễ dàng chuyển kiểu small từ int sang short tại thời gian biên dịch vì giá trị 15 đủ nhỏ để vừa với một short. Tương tự như vậy, nó có thể chuyển đổi biểu thức 1+2 từ int thành short tại thời điểm biên dịch. Mặt khác, 1_000_000 quá lớn để có thể nhét vừa vào bên trong short, do đó câu lệnh trường hợp cuối cùng không được biên dịch

2. Writing while Loops

Loop - Vòng lặp là một cấu trúc điều khiển lặp đi lặp lại có thể thực thi một câu lệnh mã nhiều lần liên tiếp. Bằng cách sử dụng các biến có thể được gán giá trị mới, mỗi lần lặp lại câu lệnh có thể khác nhau.

```
int counter = 0;
while (counter < 10) {
    double price = counter * 10;
    System.out.println(price);
    counter++;
}
```

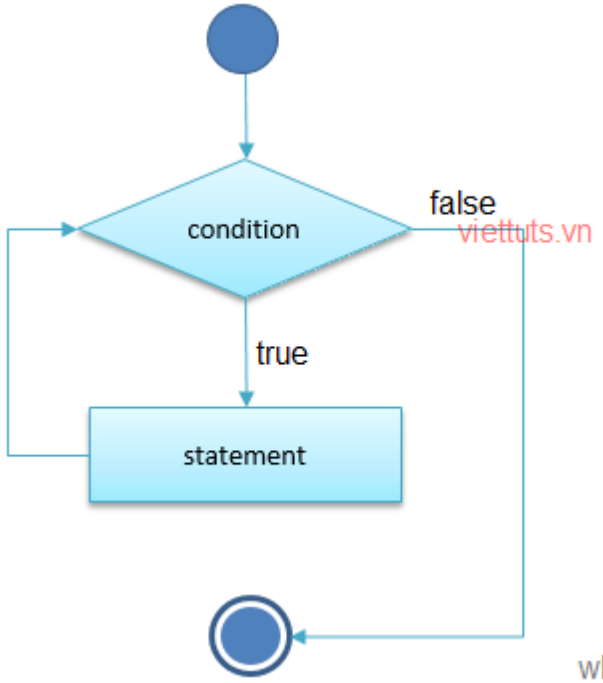
⇨ In ra **counter * 10,** tăng giá trị của counter lên 1, và kết thúc vòng lặp nếu counter >= 10

a) The while statement

Vòng lặp while trong java được sử dụng để lặp một phần của chương trình một vài lần. Nếu số lần lặp không được xác định trước thì vòng lặp lặp while được khuyến khích sử dụng trong trường hợp này.

Cú pháp:

```
while(condition) {
    // Khối lệnh được lặp lại cho đến khi condition = False
}
```



while-loop

Vòng lặp while tương tự như câu lệnh if ở chỗ nó bao gồm một biểu thức boolean và một câu lệnh hoặc một khối câu lệnh. Trong quá trình thực thi, biểu thức boolean được đánh giá trước mỗi lần lặp của vòng lặp và thoát ra nếu đánh giá trả về sai.

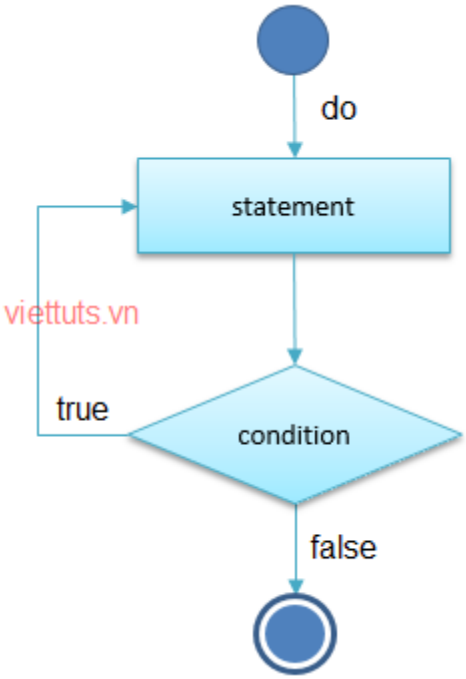
```
int roomInBelly = 5;
public void eatCheese(int bitesOfCheese) {
    while (bitesOfCheese > 0 && roomInBelly > 0) {
        bitesOfCheese--;
        roomInBelly--;
    }
    System.out.println(bitesOfCheese+" pieces of cheese left");
}
```

b) The *do/while* statement

Vòng lặp do-while trong java được sử dụng để lặp một phần của chương trình một vài lần. Tương tự như vòng lặp while, ngoại trừ do-while thực hiện lệnh ít nhất một lần ngay cả khi điều kiện là False.

Cú pháp:

```
do {
    // Khối lệnh được thực thi
} while(condition);
```



do-while

Ví dụ:

```
public class DoWhileExample1 {
    public static void main(String[] args) {
        int a = 1, sum = 0;
        do {
            sum += a;
            a++;
        } while (a <= 5);
        System.out.println("Sum of 1 to 5  is " + sum);
    }
}
```

Kết quả:

```
Sum of 1 to 5 is 15
```

c) Comparing while and do/while loops

Trong thực tế, có thể khó xác định khi nào bạn nên sử dụng vòng lặp while và khi nào bạn nên sử dụng vòng lặp do/while. Câu trả lời ngắn gọn là nó không thực sự quan trọng. Bất kỳ vòng lặp while nào cũng có thể được chuyển đổi thành vòng lặp do/while và ngược lại.

Ví dụ: so sánh vòng lặp while này:

```
while(llama > 10) {
    System.out.println("Llama!");
    llama--;
}
```

và vòng lặp do/while này:

```
if(llama > 10) {
    do {
        System.out.println("Llama!");
        llama--;
    } while(llama > 10);
}
```

Nên sử dụng vòng lặp while khi mã sẽ thực thi 0 hoặc nhiều lần và vòng lặp do/while khi mã sẽ thực thi một hoặc nhiều lần. Nói cách khác, bạn nên sử dụng vòng lặp do/while khi bạn muốn vòng lặp của mình thực thi ít nhất một lần. Điều đó cho thấy, việc xác định xem bạn nên sử dụng vòng lặp while hay vòng lặp do/while trong thực tế đôi khi phụ thuộc vào sở thích cá nhân và khả năng dễ đọc của code.

d) Infinite loops

Điều quan trọng nhất bạn cần lưu ý khi sử dụng bất kỳ cấu trúc kiểm soát sự lặp lại nào là đảm bảo chúng luôn kết thúc! Việc không kết thúc vòng lặp có thể dẫn đến nhiều vấn đề trong thực tế bao gồm overflow exceptions, rò rỉ bộ nhớ, hiệu suất chậm. Hãy xem một ví dụ:

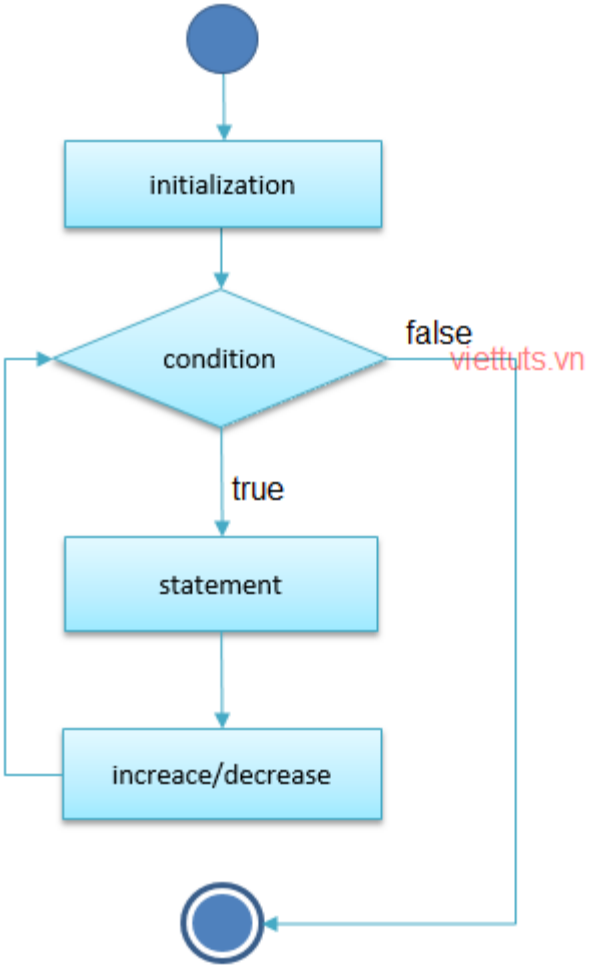
```
int pen = 2;
int pigs = 5;
while(pen < 10)
    pigs++;
```

Kết quả là vòng lặp sẽ không bao giờ kết thúc, tạo ra cái thường được gọi là vòng lặp vô hạn. Vòng lặp vô hạn là vòng lặp mà điều kiện kết thúc không bao giờ đạt được trong suốt thời gian chạy. Bất cứ khi nào bạn viết một vòng lặp, bạn nên kiểm tra nó để xác định xem liệu điều kiện kết thúc cuối cùng có luôn được đáp ứng dưới một điều kiện nào đó hay không.

3. Constructing for Loops

Vòng lặp for cơ bản có cùng biểu thức và câu lệnh boolean có điều kiện hoặc khối câu lệnh giống như vòng lặp while, cũng như hai phần mới: khối khởi tạo và câu lệnh cập nhật.

```
for (khoi_tao_bien ; check_dieu_kien ; tang/giam_bien) {
    // Khối lệnh được thực thi
}
```



for loop

Ví dụ:

```
public class ForExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            System.out.println(i);
        }
    }
}
```

a) Printing Elements in Reverse

Giả sử bạn muốn in năm số đầu tiên từ 0 giống như chúng ta đã làm ở phần trước, nhưng lần này theo thứ tự ngược lại. Mục tiêu khi đó là in ra 4 3 2 1 0. Bạn sẽ làm điều đó như thế nào?

Bắt đầu với Java 10, có thể thấy var được sử dụng trong vòng lặp for, vì vậy hãy sử dụng nó cho ví dụ này. Quá trình triển khai ban đầu có thể trông giống như sau:

```
for (var counter = 4; counter >= 0; counter--) {
    System.out.print(counter + " ");
}
```

b) Working with for Loops

Creating an Infinite Loop

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

Mặc dù vòng lặp for này có vẻ như không biên dịch nhưng thực tế nó sẽ biên dịch và chạy mà không gặp vấn đề gì. Nó thực sự là một vòng lặp vô hạn sẽ in đi in lại cùng một câu lệnh. Ví dụ này củng cố thực tế rằng các thành phần của vòng lặp for đều là tùy chọn. Lưu ý rằng dấu chấm phẩy ngăn cách ba phần là bắt buộc, vì for( ) không có dấu chấm phẩy sẽ không biên dịch được.

Adding Multiple Terms to the for Statement

```
int x = 0;
for(long y = 0, z = 4; x < 5 && y < 10; x++, y++) {
    System.out.print(y + " "); }
System.out.print(x + " ");
```

Đoạn mã này trình bày ba biến thể của vòng lặp for mà có thể bạn chưa từng thấy. Đầu tiên, bạn có thể khai báo một biến, chẳng hạn như x trong ví dụ này, trước khi vòng lặp bắt đầu và sử dụng biến đó sau khi vòng lặp hoàn thành. Thứ hai, khối khởi tạo, biểu thức boolean và câu lệnh cập nhật của bạn có thể bao gồm các biến bổ sung có thể hoặc không thể tham chiếu lẫn nhau. Ví dụ: z được xác định trong khối khởi tạo và không bao giờ được sử dụng. Cuối cùng, câu lệnh cập nhật có thể sửa đổi nhiều biến.

Redeclaring a Variable in the Initialization Block

```
int x = 0;
for(int x = 4; x < 5; x++) { // DOES NOT COMPILE
    System.out.print(x + " ");
}
```

Sự khác biệt là x được lặp lại trong khối khởi tạo sau khi đã được khai báo trước vòng lặp, dẫn đến việc trình biên dịch dừng do khai báo biến trùng lặp.

```
int x = 0;
for(x = 0; x < 5; x++) {
    System.out.print(x + " ");
}
```

Sử dụng các kiểu dữ liệu không tương thích trong Khối khởi tạo (Initialization Block)

```
int x = 0;
for(long y = 0, int z = 4; x < 5; x++) { // DOES NOT COMPILE
    System.out.print(y + " ");
}
```

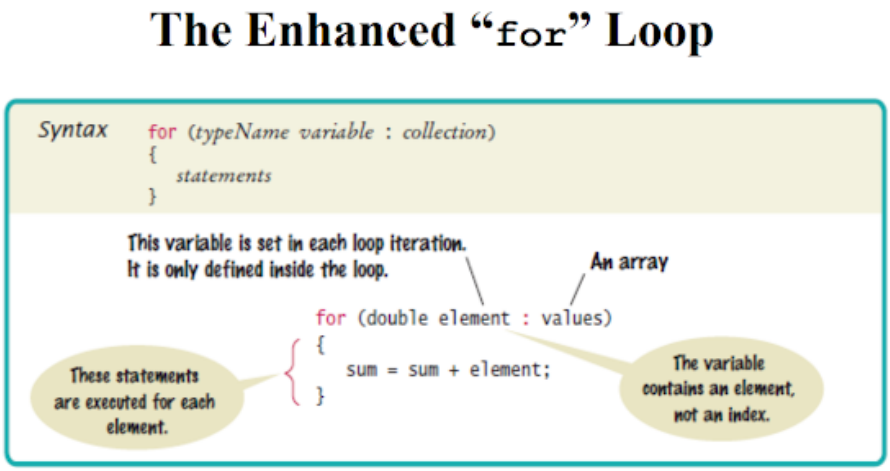
⇨ trong ví dụ này **y và x** có các loại khác nhau nên mã sẽ không biên dịch được

Using Loop Variables Outside the Loop

```
for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) {
    System.out.print(y + " ");
}
System.out.print(x); // DOES NOT COMPILE
```

c) For-each loop

Vòng lặp **for-each** được sử dụng để lặp mảng(array) hoặc collection trong java. Bạn có thể sử dụng nó dễ dàng, dễ hơn cả vòng lặp for đơn giản. Bởi vì bạn không cần phải tăng hay giảm giá trị của biến rồi check điều kiện, bạn chỉ cần sử dụng ký hiệu hai chấm ":"



Khai báo vòng lặp for-each bao gồm phần khởi tạo và một đối tượng được lặp lại. Phía bên phải của vòng lặp for-each phải là một trong những phần sau:

- Một mảng Java tích hợp
- Một đối tượng có kiểu triển khai (implements) java.lang.Iterable

Ví dụ:

```
public class ForEachExample {
    public static void main(String[] args) {
        int arr[] = { 12, 23, 44, 56, 78 };
        for (int i : arr) {
            System.out.println(i);
        }
    }
}
```

4. Controlling Flow with Branching

a) Nested loops

Vòng lặp lồng nhau là vòng lặp chứa một vòng lặp khác bao gồm các vòng lặp while, do/while, for và for-each. Ví dụ: hãy xem đoạn mã sau lặp qua một mảng hai chiều, là một mảng chứa các mảng khác làm thành viên của nó.

```
int[][] myComplexArray = {{5,2,1,3},{3,9,8,9},{5,7,12,7}};

for(int[] mySimpleArray : myComplexArray) {
    for(int i=0; i<mySimpleArray.length; i++){
        System.out.print(mySimpleArray[i]+"\t");
    }
    System.out.println();
}
```

Lưu ý rằng sử dụng cả vòng lặp for và for-each trong ví dụ này. Vòng lặp bên ngoài sẽ thực hiện tổng cộng ba lần. Mỗi lần vòng lặp bên ngoài thực thi, vòng lặp bên trong được thực hiện bốn lần. Khi thực thi mã này, chúng tôi thấy kết quả đầu ra sau:

```
5   2   1   3
3   9   8   9
5   7   12  7
```

Các vòng lặp lồng nhau có thể bao gồm while và do/while, như trong ví dụ này. Xem liệu có thể xác định code này sẽ xuất ra output là gì:

```
int hungryHippopotamus = 8;
while(hungryHippopotamus>0) {
    do {
        hungryHippopotamus -= 2;
    } while (hungryHippopotamus>5);
    hungryHippopotamus--;
    System.out.print(hungryHippopotamus+", ");
}
```

Output:

3, 0,

b) Adding optional labels

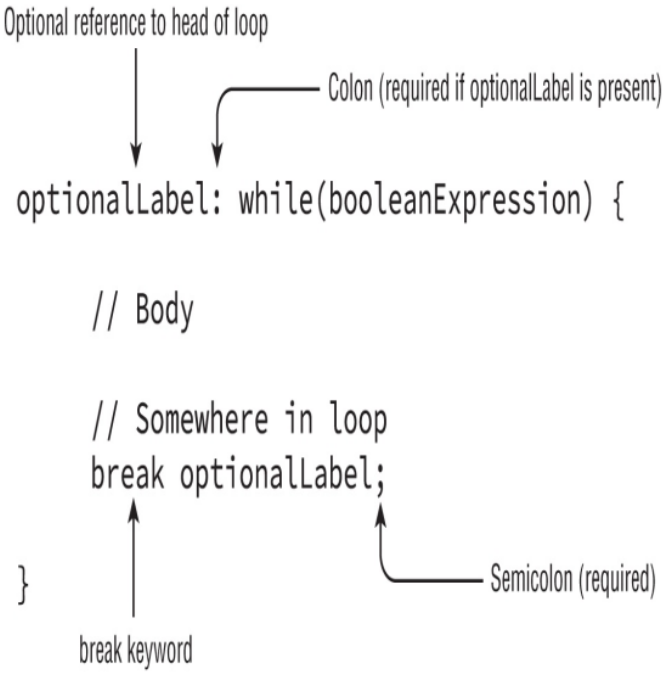
Label là một con trỏ tùy chọn tới phần đầu của câu lệnh cho phép luồng ứng dụng chuyển đến hoặc thoát khỏi nó. Nó là một identifier duy nhất được bắt đầu bằng dấu hai chấm (:). Ví dụ: có thể thêm nhãn tùy chọn vào một trong các ví dụ trước:

```
int[][] myComplexArray = {{5,2,1,3},{3,9,8,9},{5,7,12,7}};
OUTER_LOOP: for(int[] mySimpleArray : myComplexArray) {
    INNER_LOOP: for(int i=0; i<mySimpleArray.length; i++) {
        System.out.print(mySimpleArray[i]+"\t");
    }
    System.out.println();
}
```

Label tuân theo các quy tắc tương tự để định dạng dưới dạng identifier. Để dễ đọc, chúng thường được biểu thị bằng chữ in hoa, có dấu gạch dưới giữa các từ, để phân biệt chúng với các biến thông thường. Khi chỉ xử lý một vòng lặp, label không thêm bất kỳ giá trị nào, chúng cực kỳ hữu ích trong các cấu trúc lồng nhau

c) Break statement

Với câu lệnh switch, câu lệnh break chuyển luồng điều khiển sang câu lệnh kèm theo. Điều tương tự cũng đúng đối với câu lệnh break xuất hiện bên trong vòng lặp while, do/while hoặc for, vì nó sẽ kết thúc vòng lặp sớm.



Câu lệnh break có thể nhận một tham số label tùy chọn. Nếu không có tham số label, câu lệnh break sẽ chấm dứt vòng lặp bên trong gần nhất mà nó hiện đang trong quá trình thực thi. Tham số label tùy chọn cho phép chúng ta thoát ra khỏi vòng lặp bên ngoài cấp cao hơn. Trong ví dụ sau, chúng tôi tìm kiếm vị trí chỉ mục mảng (x, y) đầu tiên của một số trong mảng hai chiều chưa được sắp xếp:

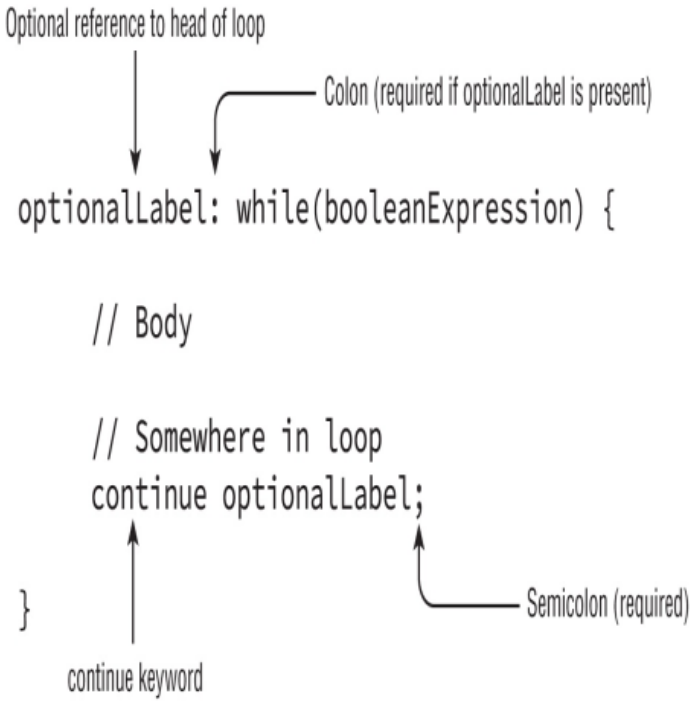
```
public class FindInMatrix {
    public static void main(String[] args) {
        int[][] list = {{1,13},{5,2},{2,2}};
        int searchValue = 2;
        int positionX = -1;
        int positionY = -1;
        PARENT_LOOP: for(int i=0; i<list.length; i++) {
            for(int j=0; j<list[i].length; j++) {
                if(list[i][j]==searchValue) {
                    positionX = i;
                    positionY = j;
                    break PARENT_LOOP;
                }
            }
        }
        if(positionX==-1 || positionY==-1) {
            System.out.println("Value "+searchValue+" not found");
        } else {
            System.out.println("Value "+searchValue+" found at: "
                    +
                    "("+positionX+","+positionY+")");
        }
    }
}
```

Output:

Value 2 found at: (1,1)

d) Continue statement

Điều khiển vòng lặp nâng cao bằng câu lệnh continue, một câu lệnh khiến luồng kết thúc việc thực thi vòng lặp hiện tại:



Bạn có thể nhận thấy cú pháp của câu lệnh continue giống cú pháp của câu lệnh break. Trên thực tế, các statement giống hệt nhau về cách sử dụng nhưng mang lại kết quả khác nhau. Trong khi câu lệnh break chuyển điều khiển sang câu lệnh kèm theo, câu lệnh continue chuyển điều khiển sang biểu thức boolean để xác định xem vòng lặp có nên tiếp tục hay không. Nói cách khác, nó kết thúc vòng lặp hiện tại.

Ngoài ra, giống như câu lệnh break, câu lệnh continue được áp dụng cho vòng lặp bên trong gần nhất đang được thực thi bằng cách sử dụng câu lệnh label tùy chọn để ghi đè hành vi này.

Ví dụ:

```
public static void main(String[] args) {
    first:
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j< 3; j++){
            if(i == 1){
                continue first;
            }
            System.out.println(" [i = " + i + ", j = " + j + "] ");
        }
    }
}
```

Output:

```
[i = 0, j = 0]
[i = 0, j = 1]
[i = 0, j = 2]
[i = 2, j = 0]
[i = 2, j = 1]
[i = 2, j = 2]
```

Nếu không sử dụng label thì kết quả là gì ?

e) Unreachable code – code không thể truy cập

Một khía cạnh của break, continue, và return mà bạn cần lưu ý là bất kỳ code nào được đặt ngay sau chúng trong cùng một khối đều được coi là không thể truy cập được và sẽ không được biên dịch. Ví dụ: đoạn mã sau không biên dịch được:

```
int checkDate = 0;
while(checkDate<10) {
    checkDate++;
    if(checkDate>100) {
        break;
        checkDate++; // DOES NOT COMPILE
    }
}
```

Mặc dù về mặt logic, câu lệnh if không thể đánh giá là đúng trong code này, nhưng trình biên dịch sẽ thông báo rằng bạn có các câu lệnh ngay sau break và sẽ không biên dịch được với lý do là "mã không thể truy cập". Điều này cũng đúng với các câu lệnh continue và return, như trong hai ví dụ sau:

```
int minute = 1;
WATCH: while(minute>2) {
    if(minute++>2) {continue WATCH;
        System.out.print(minute); // DOES NOT COMPILE
    }
}
int hour = 2;
switch(hour) {
    case 1: return; hour++; // DOES NOT COMPILE
    case 2:
}
```

f) Eviewing branching

	Allows optional labels	**Allows *break* statement**	**Allows *continue* statement**
while	Yes	Yes	Yes
do while	Yes	Yes	Yes
for	Yes	Yes	Yes
switch	Yes	Yes	No

Chapter 5: Core Java APIs

1. Creating and Manipulating Strings

String Class là một lớp cơ bản đến mức bạn sẽ khó có thể viết code mà không có nó. **string** về cơ bản là một chuỗi các ký tự; đây là một ví dụ:

```
String name = "Fluffy";
```

Bạn cũng đã biết rằng các loại tham chiếu được tạo bằng từ khóa new. Ví dụ trước còn thiếu một cái gì đó: Nó không có gì mới trong đó! Trong Java, cả hai đoạn mã này đều tạo ra một String:

```
String name = "Fluffy";
String name = new String("Fluffy");
```

Cả hai đều cung cấp cho bạn một biến tham chiếu có tên tên trỏ đến đối tượng String "Fluffy". Nhưng có sự khác nhau.

a) Concatenation – nối chuỗi

Các quy tắc khi cộng String với các đối tượng khác:

- Nếu cả hai toán hạng đều là số, + có nghĩa là phép cộng số.
- Nếu một trong hai toán hạng là Chuỗi, + có nghĩa là nối.
- Biểu thức được đánh giá từ trái sang phải.

Bây giờ hãy xem một số ví dụ:

```
System.out.println(1 + 2); // 3
System.out.println("a" + "b"); // ab
System.out.println("a" + "b" + 3); // ab3
System.out.println(1 + 2 + "c"); // 3c
System.out.println("c" + 1 + 2); // c12
```

Chỉ còn một điều nữa cần biết về phép nối, nhưng đó là một điều dễ dàng. Trong ví dụ này, bạn chỉ cần nhớ += làm gì. s +="2" có nghĩa tương tự như s = s + "2".

```
String s = "1"; // s currently holds "1"
s += "2"; // s currently holds "12"
s += 3; // s currently holds "123"
System.out.println(s); // 123
```

b) Immutability – Bất biến

Khi một đối tượng String được tạo, nó không được phép thay đổi. String là bất biến.

Ví dụ:

```
public class Testimmutablestring {
    public static void main(String args[]) {
        String s = "Hello";
        s.concat(" Java");//phương thức concat() để nối thêm chuỗi vào đuôi chuỗi s.
        System.out.println(s);//sẽ chỉ in ra "Hello" vì các chuỗi này là đối tượng
không thể thay đổi.
    }
}
```

Output:

Hello

c) Important string methods

Lớp String có rất nhiều phương thức. Đối với tất cả các phương thức này, bạn cần nhớ rằng một chuỗi là một chuỗi các ký tự và Java được tính từ 0 khi được lập chỉ mục.

a	n	i	m	a	l	s
0	1	2	3	4	5	6

toUpperCase() và toLowerCase()

Phương thức toUpperCase() để chuyển đổi chuỗi thành chữ hoa và phương thức toLowerCase() để chuyển đổi chuỗi thành chữ thường. Ví dụ:

```
String s="Tt.tung";
System.out.println(s.toUpperCase());//Chuyen doi thanh HOCLAPTRINH
System.out.println(s.toLowerCase());//Chuyen doi thanh hoclaptrinh
System.out.println(s);//Hoclaptrinh(khong co thay doi nao)
```

Chương trình trên sẽ cho kết quả:

```
TT.TUNG
tt.tung
Tt.tung
```

trim()

Phương thức trim() trong Java loại bỏ các khoảng trống trắng ở trước và sau chuỗi (leading và trailing). Ví dụ:

```
String s="  1995mars  ";
System.out.println(s);//in ra chuoi nhu ban dau  1995mars  (van con khoang trang)
System.out.println(s.trim());//in ra chuoi sau khi da cat cac khoang trong trang: 199
5mars
```

startWith() và endsWith()

```
String s="1995mars";
 System.out.println(s.startsWith("19"));//true
 System.out.println(s.endsWith("k"));//false
```

charAt()

Phương thức charAt() trả về ký tự tại chỉ mục đã cho. Ví dụ:

```
String s="1995mars";
System.out.println(s.charAt(0));//tra ve 1
System.out.println(s.charAt(3));//tra ve 9
```

length()

Phương thức length() trả về độ dài của chuỗi. Ví dụ:

```
String s="1995mars";
System.out.println(s.length());//tra ve do dai la 8
```

intern

Ban đầu, một Pool của các chuỗi là trống, được duy trì riêng cho lớp String. Khi phương thức intern được gọi, nếu Pool đã chứa một chuỗi bằng với đối tượng String như khi được xác định bởi phương thức equals(object), thì chuỗi từ Pool được trả về. Nếu không thì, đối tượng String này được thêm vào Pool và một tham chiếu tới đối tượng String này được trả về. Ví dụ:

```
String s1 = "1995mars";
String s2 = new String("1995mars");
String s3 = s2.intern();
System.out.println(s3);//tra ve 1995mars
System.out.println(s1==s2); //false
System.out.println(s1==s3); //true
```

indexOf(String subString)

Phương thức indexOf(String subString) trả về vị trí đầu tiên của chuỗi con subString trong chuỗi gốc, hoặc -1 nếu không tìm thấy.

```
String day = "Sunday";
int index1 = day.indexOf('n');      // 2
int index2 = day.indexOf("Sun");    // 0
int index3 = day.indexOf('z', 2);   // -1
```

subString

Phương thức subString() trả về chuỗi con của một chuỗi.

Chúng ta truyền chỉ số bắt đầu và chỉ số kết thúc cho phương thức subString(), với chỉ số bắt đầu tính từ 0 và chỉ số kết thúc tính từ 1.

Cú pháp:

```
public String substring(int startIndex)
public String substring(int startIndex, int endIndex)
```

Ví dụ phương thức subString trong Java String:

```
String s1 = "hellojava";
System.out.println(s1.substring(3, 7));// "loja"
System.out.println(s1.substring(3));// "lojava"
System.out.println(s1.substring(9,9));// empty string
System.out.println(s1.substring(3,2));// throws exception
System.out.println(s1.substring(3,10));// throws exception
```

equals và equalsIgnoreCase

- Phương thức **equals**() so sánh hai chuỗi đưa ra dựa trên nội dung của chuỗi. Nếu hai chuỗi khác nhau nó trả về false. Nếu hai chuỗi bằng nhau nó trả về true.
- Phương thức **equalsIgnoreCase**() so sánh hai chuỗi đưa ra dựa trên nội dung của chuỗi không phân biệt chữ hoa và chữ thường. Nếu hai chuỗi khác nhau nó trả về false. Nếu hai chuỗi bằng nhau nó trả về true.

Cú pháp:

```
public boolean equals(Object obj)

public boolean equalsIgnoreCase(String str)
```

Ví dụ:

```
System.out.println("abc".equals("ABC")); // false
System.out.println("ABC".equals("ABC")); // true
System.out.println("abc".equalsIgnoreCase("ABC")); // true
```

replace()

Phương thức replace() được sử dụng để thay thế tất cả các ký tự hoặc chuỗi cũ thành ký tự hoặc chuỗi mới.

Cú pháp:

```
public String replace(char oldChar, char newChar)
public String replace(CharSequence target, CharSequence replacement)
```

Ví dụ:

```
System.out.println("abcabc".replace('a', 'A')); // AbcAbc
System.out.println("abcabc".replace("a", "A")); // AbcAbc
```

contains

Phương thức contains() tìm kiếm chuỗi ký tự trong chuỗi này. Nó trả về true nếu chuỗi các giá trị char được tìm thấy trong chuỗi này, nếu không trả về false.

```
public boolean contains(CharSequence sequence)
```

Ví dụ:

```
System.out.println("abc".contains("b")); // true
System.out.println("abc".contains("B")); // false
```

trim(), strip(), stripLeading(), và stripTrailing()

Tiếp theo là xóa khoảng trống ở đầu và hoặc cuối String. Các phương thức Strip() và Trim() loại bỏ khoảng trắng ở đầu và cuối String. Về mặt bài kiểm tra, khoảng trắng bao gồm các khoảng trắng cùng với các ký tự \t (tab) và \n (dòng mới). Các ký tự khác, chẳng hạn như \r (carriage return), cũng được bao gồm trong những gì được cắt bớt. Phương thức Strip() là phương thức mới trong Java 11.

Ngoài ra, các phương thức StripLeading() và StripTrailing() đã được thêm vào Java 11. Phương thức StripLeading() loại bỏ khoảng trắng ở đầu String và để nó ở cuối. Phương thức StripTrailing() thực hiện ngược lại. Nó loại bỏ khoảng trắng ở cuối String và để lại ở đầu String.

```
String text = " abc  ";
System.out.println("abc".strip()); // abc
System.out.println("\t a b c\n".strip()); // a b c String text = " abc\t ";
System.out.println(text.trim().length()); // 3
System.out.println(text.strip().length()); // 3
System.out.println(text.stripLeading().length()); // 5
System.out.println(text.stripTrailing().length());// 4
```

d) Method chaining

Thường gọi nhiều phương thức như được hiển thị ở đây:

```
String start = "AniMaL ";
String trimmed = start.trim(); // "AniMaL"
String lowercase = trimmed.toLowerCase(); // "animal"
String result = lowercase.replace('a', 'A'); // "AnimAl"
System.out.println(result);
```

Chúng ta có thể sử dụng các phương thức liên tục như sau:

```
String result = "AniMaL ".trim().toLowerCase().replace('a','A');
System.out.println(result);
```

2. Using the StringBuilder Class

Một chương trình nhỏ có thể tạo rất nhiều đối tượng String một cách nhanh chóng. Ví dụ: bạn nghĩ đoạn mã này tạo ra bao nhiêu?

```
10:     String alpha = "";
11:     for(char current = 'a'; current <= 'z'; current++)
12:     alpha += current;
13:     System.out.println(alpha);
```

Các đối tượng String liên tục được tạo và đủ điều kiện thu dọn rác. Chuỗi sự kiện này tiếp tục diễn và lặp lại. Điều này rất kém hiệu quả. May mắn thay, Java có giải pháp. Lớp StringBuilder tạo một Chuỗi mà không lưu trữ tất cả các giá trị Chuỗi tạm thời đó. Không giống như lớp String, StringBuilder không phải là bất biến.

```
15:     StringBuilder alpha = new StringBuilder();
16:     for(char current = 'a'; current <= 'z'; current++)
17:     alpha.append(current);
18:     System.out.println(alpha);
```

Code này sử dụng lại cùng một StringBuilder mà không tạo String tạm thời mỗi lần.

a) Mutability and chaining

StringBuilder không phải là bất biến. Trên thực tế, chúng tôi đã cung cấp cho nó 27 giá trị khác nhau trong ví dụ. StringBuilder thay đổi trạng thái của chính nó và trả về một tham chiếu đến chính nó. Hãy xem một ví dụ để làm rõ điều này:

```
4:      StringBuilder sb = new StringBuilder("start");
5:      sb.append("+middle"); // sb ="start+middle"
6:      StringBuilder same = sb.append("+end"); //"start+middle+end"
```

Bạn nghĩ ví dụ này in ra cái gì?

```
4:      StringBuilder a = new StringBuilder("abc");
5:      StringBuilder b = a.append("de");
6:      b = b.append("f").append("g");
7:      System.out.println("a=" + a);
8:      System.out.println("b=" + b);
```

⇨ a=abcdefg
b=abcdefg

b) Creating a stringbuilder

Có ba cách để khởi tạo StringBuilder:

```
StringBuilder sb1 = new StringBuilder();
StringBuilder sb2 = new StringBuilder("animal");
StringBuilder sb3 = new StringBuilder(10);
```

c) Important stringbuilder methods

*charAt(), indexOf(), length(), **and** substring()*

Bốn phương thức này hoạt động giống hệt như trong lớp String. Hãy chắc chắn rằng bạn có thể xác định output của ví dụ này:

```
StringBuilder sb = new StringBuilder("animals");
String sub = sb.substring(sb.indexOf("a"), sb.indexOf("al"));
int len = sb.length();
char ch = sb.charAt(6);
System.out.println(sub + " " + len + " " + ch);
```

⇨ anim 7 s

append()

Phương thức append() của lớp StringBuilder nối thêm tham số vào cuối chuỗi.
Cú pháp:

```
StringBuilder append(String str)
```

Ví dụ:

```
StringBuilder sb = new StringBuilder().append(1).append('c');
sb.append("-").append(true);
System.out.println(sb); // 1c-true
```

insert()

Phương thức insert() của lớp StringBuilder chèn chuỗi vào chuỗi này từ vị trí quy định.
Cú pháp:

```
StringBuilder insert(int offset, String str)
```

Ví dụ:

```
3:      StringBuilder sb = new StringBuilder("animals");
4:      sb.insert(7, "-"); // sb = animals-
5:      sb.insert(0, "-"); // sb = -animals-
6:      sb.insert(4, "-"); // sb = -ani-mals-
7:      System.out.println(sb);
```

delete() và deleteCharAt()

Phương thức delete() ngược lại với phương thức insert(). Nó xóa các ký tự khỏi chuỗi và trả về một tham chiếu đến StringBuilder hiện tại. Phương thức deleteCharAt() thuận tiện khi bạn chỉ muốn xóa một ký tự. Các cú pháp như sau:

```
StringBuilder delete(int startIndex, int endIndex)
StringBuilder deleteCharAt(int index)
```

Đoạn mã sau đây cho thấy cách sử dụng các phương pháp này:

```
StringBuilder sb = new StringBuilder("abcdef");
sb.delete(1, 3); // sb = adef
sb.deleteCharAt(5); // throws an exception
```

Phương thức delete() linh hoạt hơn một số phương thức khác khi nói đến index mảng. Nếu bạn chỉ định tham số thứ hai vượt quá phần cuối của StringBuilder, Java sẽ chỉ cho rằng bạn muốn nói đến phần cuối. Điều đó có nghĩa là mã này là hợp pháp:

```
StringBuilder sb = new StringBuilder("abcdef");
sb.delete(1, 100); // sb = a
```

replace()

Phương thức replace() của lớp StringBuilder thay thế chuỗi bằng chuỗi khác từ vị trị bắt đầu và kết thúc được quy định.

```
StringBuilder replace(int startIndex, int endIndex, String newString)
```

Đoạn mã sau cho thấy cách sử dụng phương pháp này:

```
StringBuilder builder = new StringBuilder("pigeon dirty");
builder.replace(3, 6, "sty");
System.out.println(builder); // pigsty dirty
```

reverse()

Phương thức reverse() của lớp StringBuilder đảo ngược chuỗi hiện tại.
Ví dụ:

```
StringBuilder sb = new StringBuilder("ABC");
sb.reverse();
System.out.println(sb); // CBA
```

toString()

Phương thức cuối cùng chuyển đổi StringBuilder thành String.
Đoạn mã sau đây cho biết cách sử dụng phương thức này:

```
StringBuilder sb = new StringBuilder("ABC");
String s = sb.toString();
```

3. Understanding Equality

a) Comparing equals() and ==

Hãy xem xét đoạn mã sau sử dụng == với các đối tượng:

```
StringBuilder one = new StringBuilder();
StringBuilder two = new StringBuilder();
StringBuilder three = one.append("a");
System.out.println(one == two); // false
System.out.println(one == three); // true
```

one và two đều là các đối tượng StringBuilder hoàn toàn riêng biệt, cho chúng ta hai đối tượng. Hãy nhớ cách các phương thức StringBuilder muốn trả về tham chiếu hiện tại cho chuỗi.

Nếu một lớp không có phương thức equals, Java sẽ xác định xem các tham chiếu có trỏ đến cùng một đối tượng hay không - đó chính xác là những gì == thực hiện.

Bạn có thể đoán xem tại sao mã không biên dịch được?

```
String string = "a";
StringBuilder builder = new StringBuilder("a");
System.out.println(string == builder); //DOES NOT COMPILE
```

Hãy nhớ rằng == đang kiểm tra sự bình đẳng tham chiếu đối tượng. Trình biên dịch đủ thông minh để biết rằng hai tham chiếu không thể trỏ đến cùng một đối tượng khi chúng có kiểu hoàn toàn khác nhau.

b) The string pool

Vì các chuỗi có ở khắp mọi nơi trong Java nên chúng sử dụng rất nhiều bộ nhớ. Trong một số ứng dụng product, chúng có thể sử dụng một lượng lớn bộ nhớ trong toàn bộ chương trình. Java nhận ra rằng có nhiều chuỗi lặp lại trong chương trình và giải quyết vấn đề này bằng cách sử dụng lại các chuỗi phổ biến.

String pool, còn được gọi là intern pool, là một vị trí trong máy ảo Java (JVM) thu thập tất cả các chuỗi này. String pool chứa các giá trị bằng chữ và các hằng số xuất hiện trong chương trình của bạn. Ví dụ: "tên" là một chữ và do đó nằm trong string pool. myObject.toString() là một chuỗi nhưng không phải là một chuỗi ký tự, vì vậy nó không đi vào string pool.

Bây giờ chúng ta hãy xem kịch bản phức tạp và khó hiểu hơn, đẳng thức String, được thực hiện một phần là do cách JVM sử dụng lại các chuỗi ký tự chuỗi.

```
String x = "Hello World";
String y = "Hello World";
System.out.println(x == y); // true
```

Hãy nhớ rằng String là bất biến và các chữ được gộp lại. JVM chỉ tạo một chữ trong bộ nhớ. x và y đều trỏ đến cùng một vị trí trong bộ nhớ; do đó, câu lệnh cho ra kết quả đúng. Nó thậm chí còn phức tạp hơn. Hãy xem xét mã này:

```
String x = "Hello World";
String z = " Hello World".trim();
System.out.println(x == z); // false
```

Trong ví dụ này, chúng ta không có hai chuỗi ký tự giống nhau. Mặc dù x và z tình cờ đánh giá cùng một chuỗi, nhưng một chuỗi được tính khi chạy. Vì nó không giống nhau tại thời điểm biên dịch nên một đối tượng String mới sẽ được tạo. Hãy thử một cái khác. Bạn nghĩ đầu ra ở đây là gì?

```
String singleString = "hello world";
String concat = "hello ";
concat += "world";
System.out.println(singleString == concat);
```

Output là false. Ghép nối cũng giống như gọi một phương thức và tạo ra một String mới. Bạn thậm chí có thể giải quyết vấn đề bằng cách tạo String mới:

```
String x = "Hello World";
String y = new String("Hello World");
System.out.println(x == y); // false
```

⇨ x sẽ được chỏ đến string poll còn y tạo ra đối tượng mới

Phương thức intern() sẽ sử dụng một đối tượng trong string pool nếu có. Nếu chữ đó chưa có trong string pool, Java sẽ thêm nó vào lúc này.

```
String name = "Hello World";
String name2 = new String("Hello World").intern();
System.out.println(name == name2); // true
```

Đầu tiên chúng ta yêu cầu Java sử dụng string bool thông thường cho name. Sau đó, đối với name2, chúng tôi yêu cầu Java tạo một đối tượng mới bằng cách sử dụng constructor nhưng vẫn thực hiện nó và sử dụng string bool. Vì cả hai biến đều trỏ đến cùng một tham chiếu trong string bool nên chúng ta có thể sử dụng toán tử ==. Hãy thử một cái khác. Bạn nghĩ cái này in ra cái gì? Hãy cẩn thận. Thật khó khăn

```
15: String first = "rat" + 1;
16: String second = "r" + "a" + "t" + "1";
17: String third = "r" + "a" + "t" + new String("1");
18: System.out.println(first == second);
19: System.out.println(first == second.intern());
20: System.out.println(first == third);
21: System.out.println(first == third.intern());
```

Ở dòng 15, chúng ta có một hằng số thời gian biên dịch tự động được đặt trong string pool dưới dạng "rat1". Ở dòng 16, chúng ta có một biểu thức phức tạp hơn cũng là hằng số thời gian biên dịch. Do đó, thứ nhất và thứ hai chia sẻ cùng một tham chiếu string pool. Điều này làm cho dòng 18 và 19 in ra đúng. Ở dòng 17, chúng ta có hàm tạo String. Điều này có nghĩa là chúng tôi không còn có hằng số thời gian biên dịch nữa và third không trỏ đến tham chiếu trong string pool. Vì vậy, dòng 20 in sai. Trên dòng 21, lệnh gọi intern() sẽ tìm trong string pool. Java thông báo rằng đầu tiên trỏ đến cùng một Chuỗi và in ra giá trị đúng.

4. Understanding Java Arrays

a) Creating an array of primitives

Cách phổ biến nhất để tạo một mảng trông như thế này:

```
int[] numbers1 = new int[3];
```

⇨ Tạo ra mảng có giá trị mặc định là [0,0,0]

Java cho phép bạn viết điều này:

```
int[] numbers2 = {42, 55, 99};
```

Cuối cùng, bạn có thể nhập [] trước hoặc sau tên và việc thêm dấu cách là tùy chọn. Điều này có nghĩa là cả năm câu lệnh này đều thực hiện cùng một công việc

```
int[] numAnimals;
int [] numAnimals2;
int []numAnimals3;
int numAnimals4[];
int numAnimals5 [];
```

Bạn nghĩ đoạn mã sau sẽ tạo ra những loại biến tham chiếu nào?

```
int[] ids, types;
```

Câu trả lời đúng là hai biến kiểu int[]. Điều này có vẻ đủ logic. Rốt cuộc, int a, b; đã tạo hai biến int.

b) Creating an array with reference variables

Bạn có thể chọn bất kỳ kiểu Java nào làm kiểu của array. Điều này bao gồm các class bạn tự tạo. Ví dụ:

```
public class ArrayType {
    public static void main(String args[]) {
        String [] bugs = { "cricket", "beetle", "ladybug" };
        String [] alias = bugs;
        System.out.println(bugs.equals(alias)); // true
        System.out.println(
                bugs.toString()); //[Ljava.lang.String;@160bc7c0
    }
}
```

Chúng ta có thể gọi bằng() vì mảng là một đối tượng. Hãy nhớ rằng, điều này cũng sẽ hoạt động ngay cả trên int[]. int là nguyên thủy; int[] là một đối tượng.

Chúng ta có thể ép kiểu lớn hơn thành kiểu nhỏ hơn. Bạn cũng có thể làm điều đó với mảng:

```
3: String[] strings = { "stringValue" };
4: Object[] objects = strings;
5: String[] againStrings = (String[]) objects;
6: againStrings[0] = new StringBuilder(); // DOES NOTCOMPILE
7: objects[0] = new StringBuilder(); // careful!
```

c) Using an array

Bây giờ bạn đã biết cách tạo một mảng, hãy thử truy cập:

```
4: String[] mammals = {"monkey", "chimp", "donkey"};
5: System.out.println(mammals.length); // 3
6: System.out.println(mammals[0]); // monkey
7: System.out.println(mammals[1]); // chimp
8: System.out.println(mammals[2]); // donkey
```

Việc sử dụng vòng lặp khi đọc hoặc ghi vào một mảng là rất phổ biến. Vòng lặp này đặt mỗi phần tử của số cao hơn chỉ số hiện tại là 5: (có thể sử dụng for-each)

```
5: int[] numbers = new int[10];
6: for (int i = 0; i < numbers.length; i++)
7:       numbers[i] = i + 5;
```

d) Sorting

Java giúp bạn dễ dàng sắp xếp một mảng bằng cách cung cấp một phương thức sắp xếp—hay đúng hơn là một loạt các phương thức sắp xếp: (**Quicksort(primitive) – MergeSort (reference)**

```
Arrays.sort()
```

Ví dụ đơn giản này sắp xếp ba số:

```
int[] numbers = { 6, 9, 1 };
Arrays.sort(numbers);
for (int i = 0; i < numbers.length; i++)
System.out.print(numbers[i] + " ");
```

Kết quả là 1 6 9

Hãy thử lại điều này với String:

```
String[] strings = { "10", "9", "100" };
Arrays.sort(strings);
for (String string : strings)
    System.out.print(string + " ");
```

Output:
10 100 9

⇨ Làm các nào để sắp xếp chính xác ?

e) Searching

Java cũng cung cấp một cách thuận tiện để tìm kiếm—nhưng chỉ khi mảng đã được sắp xếp.

Scenario	**Result**
Phần tử mục tiêu được tìm thấy trong mảng được sắp xếpy	Index của phần tử
Không tìm thấy phần tử trong mảng được sắp xếp	Giá trị âm hiển thị một giá trị nhỏ hơn giá trị âm của chỉ mục, trong đó cần chèn kết quả khớp để duy trì thứ tự sắp xếp
Mảng chưa sắp xếp	A surprise—this result isn't predictable

Hãy thử các quy tắc này bằng một ví dụ:

```
3:      int[] numbers = {2,4,6,8};
4:      System.out.println(Arrays.binarySearch(numbers, 2)); // 0
5:      System.out.println(Arrays.binarySearch(numbers, 4)); // 1
6:      System.out.println(Arrays.binarySearch(numbers, 1)); // -1
7:      System.out.println(Arrays.binarySearch(numbers, 3)); // -28:
System.out.println(Arrays.binarySearch(numbers, 9)); // -5
```

Bạn nghĩ điều gì sẽ xảy ra trong ví dụ này?

```
5:      int[] numbers = new int[] {3,2,1};
6:      System.out.println(Arrays.binarySearch(numbers, 2));
7:      System.out.println(Arrays.binarySearch(numbers, 3));
```

Lưu ý rằng ở dòng 5, mảng chưa được sắp xếp. Điều này có nghĩa là đầu ra sẽ không thể dự đoán được.

f) Comparing

Java cũng cung cấp các phương thức để so sánh hai mảng để xác định mảng nào "nhỏ hơn". Đầu tiên chúng ta sẽ đề cập đến phương thức compare() và sau đó chuyển sang phương thức mismatch().

compare()

Đầu tiên bạn cần tìm hiểu giá trị trả về có nghĩa là gì.

- Số âm có nghĩa là mảng đầu tiên nhỏ hơn mảng thứ hai.
- Số 0 có nghĩa là các mảng bằng nhau.
- Số dương có nghĩa là mảng đầu tiên lớn hơn mảng thứ hai.

Bây giờ, hãy xem cách so sánh các mảng có độ dài khác nhau:

- Nếu cả hai mảng có cùng độ dài và có cùng giá trị ở mỗi vị trí theo cùng thứ tự, hãy trả về 0.
- Nếu tất cả các phần tử đều giống nhau nhưng mảng thứ hai có thêm phần tử ở cuối, hãy trả về số âm.
- Nếu tất cả các phần tử đều giống nhau nhưng mảng đầu tiên có thêm phần tử ở cuối, hãy trả về một số dương.
- Nếu phần tử đầu tiên khác biệt nhỏ hơn trong mảng đầu tiên, hãy trả về số âm.
- Nếu phần tử đầu tiên khác biệt lớn hơn trong mảng đầu tiên, hãy trả về số dương.

Cuối cùng, nhỏ hơn có nghĩa là gì? Dưới đây là một số quy tắc khác áp dụng ở đây và cho compareTo()

- null nhỏ hơn bất kỳ giá trị nào khác.
- number, thứ tự số bình thường được áp dụng.
- Đối với chuỗi, một giá trị nhỏ hơn nếu nó là tiền tố của chuỗi khác. Đối với chuỗi/ký tự, số nhỏ hơn chữ cái.
- Đối với chuỗi/ký tự, chữ hoa nhỏ hơn chữ thường

Arrays.compare() examples

First array	Second array	Result	Reason
new int[] {1, 2}	new int[] {1}	Số dương	Phần tử đầu tiên giống nhau nhưng mảng đầu tiên dài hơn.
new int[] {1, 2}	new int[] {2}	Số âm	So sánh phần tử đầu tiên của 2 mảng
new int[] {1, 2}	new int[] {1, 2}	0	Giống nhau
new String[] {"a"}	new String[] {"aa"}	Số âm	Phần tử đầu tiên là chuỗi con của phần tử thứ hai.
new String[] {"a"}	new String[] {"A"}	Số dương	Chữ hoa nhỏ hơn chữ thường.
new String[] {"a"}	new String[] {null}	Số dương	null nhỏ hơn một chữ cái

Cuối cùng, mã này không biên dịch được vì các kiểu khác nhau. Khi so sánh hai mảng, chúng phải có cùng kiểu mảng.

```
System.out.println(Arrays.compare(
    new int[] {1}, new String[] {"a"})); // DOES NOT COMPILE
```

mismatch()

Nếu các mảng bằng nhau, mismatch() trả về -1. Nếu không, nó sẽ trả về chỉ mục đầu tiên nơi chúng khác nhau.

```
System.out.println(Arrays.mismatch(new int[] {1}, new int[]{1}));  // -1
System.out.println(Arrays.mismatch(new String[]{"a"}, new String[] {"A"}));  //0
System.out.println(Arrays.mismatch(new int[] {1, 2}, new int[]{1})); //1
```

Equality vs. comparison vs. mismatch

Method	When arrays are the same	When arrays are different
equals()	true	FALSE
compare()	0	Positive or negative number
mismatch()	-1	Zero or positive index

g) Varargs

Dưới đây là ba ví dụ với phương thức main():

```
public static void main(String[] args)
public static void main(String args[])
public static void main(String... args) // varargs
```

⇨ Varargs thay cho mảng khi khai báo và bắt buộc ở cuối (sẽ nói chi tiết ở chapter sau)

h) Multidimensional arrays

Creating a Multidimensional Array

Tất cả những gì cần có để phân tách nhiều mảng là khai báo mảng có nhiều chiều. Bạn có thể định vị chúng bằng tên loại hoặc tên biến trong phần khai báo, giống như dưới đây:

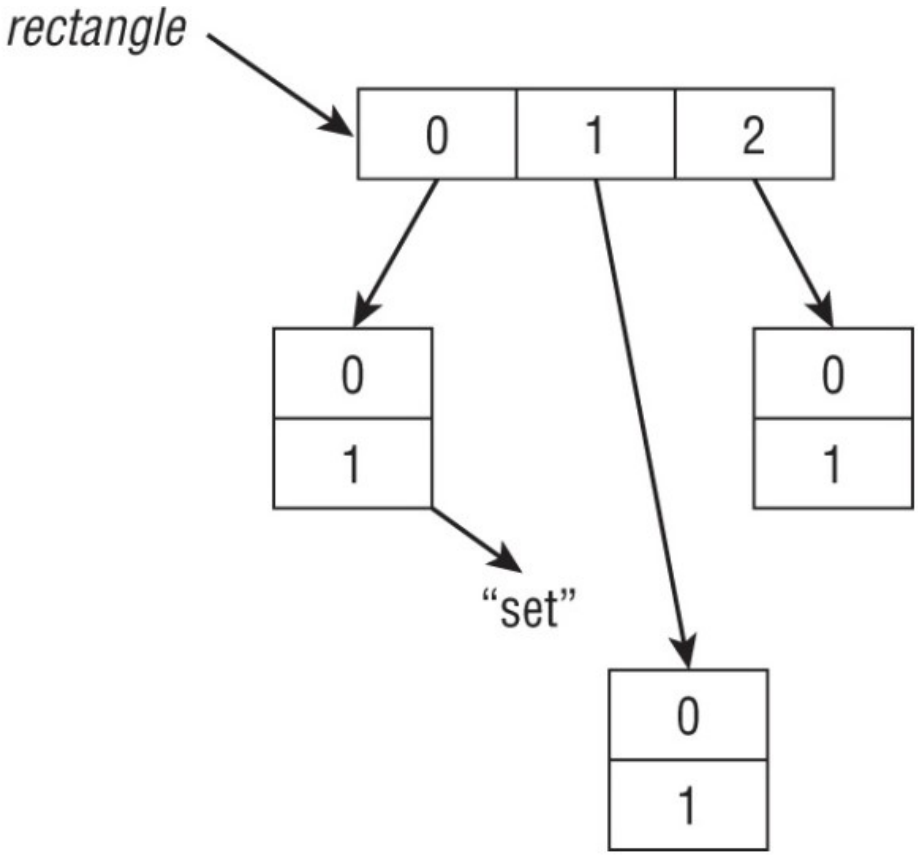
```
int[][] vars1; // 2D array
int vars2 [][]; // 2D array
int[] vars3[]; // 2D array
int[] vars4 [], space [][]; // a 2D AND a 3D array
```

Bạn có thể chỉ định kích thước của mảng đa chiều trong phần khai báo nếu muốn:

```
String [][] rectangle = new String[3][2];
```

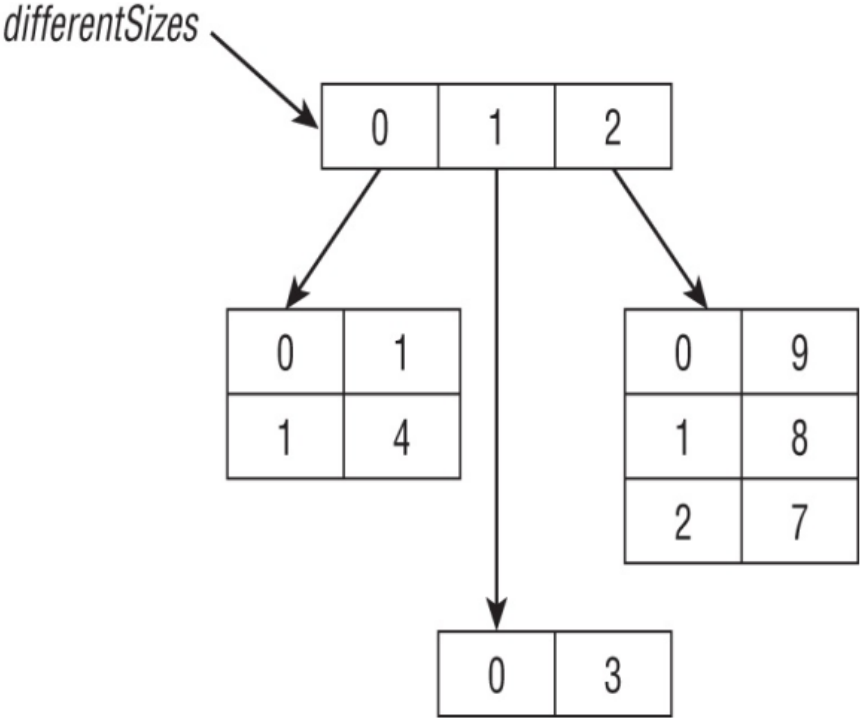
Bây giờ giả sử chúng ta set một trong các giá trị sau:

```
rectangle[0][1] = "set";
```



Hãy xem xét điều này:

```
int[][] differentSizes = {{1, 4}, {3}, {9,8,7}};
```



Một cách khác để tạo một mảng bất đối xứng là chỉ khởi tạo kích thước đầu tiên của mảng và xác định kích thước của từng thành phần mảng trong một câu lệnh riêng:

```
int [][] args = new int[4][];
args[0] = new int[5];
args[1] = new int[3];
```

Using a Multidimensional Array

Hoạt động phổ biến nhất trên mảng nhiều chiều là lặp lại nó. Ví dụ này in ra một mảng 2D:

```
int[][] twoD = new int[3][2];
for (int i = 0; i < twoD.length; i++) {
    for (int j = 0; j < twoD[i].length; j++)
        System.out.print(twoD[i][j] + " "); // print element
    System.out.println(); // time for a new row
}
```

Chúng ta có thể sử dụng for-each để code dễ đọc hơn:

```
for (int[] inner : twoD) {
    for (int num : inner)
        System.out.print(num + " ");
    System.out.println();
}
```

5. Understanding an ArrayList

a) Creating an arraylist

Có ba cách để tạo ArrayList:

```
ArrayList list1 = new ArrayList();
ArrayList list2 = new ArrayList(10);
ArrayList list3 = new ArrayList(list2);
```

Ví dụ cuối cùng cho Java biết rằng chúng ta muốn tạo một bản sao của một ArrayList khác. Chúng ta sao chép cả kích thước lẫn nội dung của ArrayList đó.

Bạn vẫn cần phải tìm hiểu một số biến thể của nó. Các ví dụ trước đây là cách tạo ArrayList cũ trước Java 5. Chúng vẫn hoạt động và bạn vẫn cần biết chúng hoạt động. Bạn cũng cần biết cách làm mới và cải tiến hơn. Java 5 đã giới thiệu các generics, cho phép bạn chỉ định loại lớp mà ArrayList sẽ chứa.

```
ArrayList<String> list4 = new ArrayList<String>();
ArrayList<String> list5 = new ArrayList<>();
```

Bắt đầu từ Java 7, bạn thậm chí có thể bỏ qua loại đó ở phía bên phải. Tuy nhiên, < và > vẫn được yêu cầu. Đây được gọi là toán tử kim cương vì <> trông giống như một viên kim cương.

Using var with arraylist

Bây giờ var có thể được sử dụng để che khuất các kiểu dữ liệu. Hãy xem xét mã này:

```
var strings = new ArrayList<String>();
strings.add("a");
for (String s: strings) { }
```

⇨ Kiểu dữ liệu của var là: ArrayList<String>

Điều gì sẽ xảy ra nếu chúng ta sử dụng toán tử kim cương với var?

```
var list = new ArrayList<>();// var là ArrayList<Object>
```

Bây giờ có thể hiểu tại sao điều này không biên dịch được không?

```
var list = new ArrayList<>();
list.add("a");
for (String s: list) { } // DOES NOT COMPILE
```

⇨ Tại sao ?

b) Using an arraylist

ArrayList có nhiều phương thức, nhưng chỉ cần biết một số ít trong số đó—thậm chí còn ít hơn so với những gì đã làm với String và StringBuilder.

add()

Các phương thức add() chèn giá trị mới vào ArrayList. Các cú pháp phương thức như sau:

```
boolean add(E element)
void add(int index, E element)
```

Hãy bắt đầu với trường hợp đơn giản nhất:

```
ArrayList list = new ArrayList();
list.add("hawk"); // [hawk]
list.add(Boolean.TRUE); // [hawk, true]
System.out.println(list); // [hawk, true]
```

Bây giờ, hãy sử dụng generic để báo cho trình biên dịch biết chúng ta chỉ muốn cho phép các đối tượng String trong ArrayList của mình:

```
ArrayList<String> safer = new ArrayList<>();
safer.add("sparrow");
safer.add(Boolean.TRUE); // DOES NOT COMPILE
```

⇨ Lần này trình biên dịch biết rằng chỉ các đối tượng String mới được phép thêm và ngăn chặn nỗ lực thêm Boolean.

```
List<String> birds = new ArrayList<>();
birds.add("hawk"); // [hawk]6: birds.add(1, "robin"); // [hawk, robin]
birds.add(0, "blue jay"); // [blue jay, hawk, robin]
birds.add(1, "cardinal"); // [blue jay, cardinal, hawk,robin]
System.out.println(birds); // [blue jay, cardinal, hawk,robin]
```

remove()

Phương thức **remove()** loại bỏ giá trị khớp đầu tiên trong ArrayList hoặc loại bỏ phần tử tại một chỉ mục được chỉ định. Các cú pháp phương thức như sau:

```
boolean remove(Object object)
E remove(int index)
```

Sau đây cho thấy cách sử dụng các phương pháp này:

```
List<String> birds = new ArrayList<>();
birds.add("hawk"); // [hawk]
birds.add("hawk"); // [hawk, hawk]
System.out.println(birds.remove("cardinal")); // prints false
System.out.println(birds.remove("hawk")); // prints true
System.out.println(birds.remove(0)); // prints hawk
System.out.println(birds); // []
```

⇨ Vì việc gọi remove() bằng int sử dụng index, nên một index không tồn tại sẽ đưa ra một ngoại lệ. Ví dụ: bird.remove(100) ném ra ngoại lệ IndexOutOfBounds.

set()

Phương thức set() thay đổi một trong các phần tử của ArrayList mà không thay đổi kích thước. Cú pháp phương thức như sau:

```
E set(int index, E newElement)
```

Sau đây cho thấy cách sử dụng phương pháp này:

```
List<String> birds = new ArrayList<>();
birds.add("hawk"); // [hawk]
System.out.println(birds.size()); // 1
birds.set(0, "robin"); // [robin]
System.out.println(birds.size()); // 1
birds.set(1, "robin"); //IndexOutOfBoundsException
```

isEmpty() and size()

Các phương thức isEmpty() và size() xem có bao nhiêu vị trí đang được sử dụng. Các cú pháp phương thức như sau:

```
boolean isEmpty()
int size()
```

Sau đây cho thấy cách sử dụng các phương pháp này:

```
List<String> birds = new ArrayList<>();
System.out.println(birds.isEmpty()); // true
System.out.println(birds.size()); // 0
birds.add("hawk"); // [hawk]
birds.add("hawk"); // [hawk, hawk]System.out.println(birds.isEmpty()); // false
System.out.println(birds.size()); // 2
```

clear()

Phương thức clear() cung cấp một cách dễ dàng để loại bỏ tất cả các phần tử của ArrayList. Cú pháp phương thức như sau:

```
void clear()
```

Sau đây cho thấy cách sử dụng phương pháp này:

```
List<String> birds = new ArrayList<>();
birds.add("hawk"); // [hawk]
birds.add("hawk"); // [hawk, hawk]
System.out.println(birds.isEmpty()); // false
System.out.println(birds.size()); // 2
birds.clear(); // []
System.out.println(birds.isEmpty()); // true
System.out.println(birds.size()); // 0
```

contains()

Phương thức contains() kiểm tra xem một giá trị nhất định có trong ArrayList hay không. Cú pháp phương thức như sau:

```
boolean contains(Object object)
```

Sau đây cho thấy cách sử dụng phương pháp này:

```
List<String> birds = new ArrayList<>();
birds.add("hawk"); // [hawk]
System.out.println(birds.contains("hawk")); // true
System.out.println(birds.contains("robin")); // false
```

Phương thức này gọi equals() trên mỗi phần tử của ArrayList để xem liệu có kết quả khớp nào không. Vì String thực hiện bằng() nên điều này hoạt động tốt.

equals()

ArrayList có cách triển khai tùy chỉnh equals(), vì vậy bạn có thể so sánh hai danh sách để xem liệu chúng có chứa các phần tử giống nhau theo cùng một thứ tự hay không.

```
List<String> one = new ArrayList<>();
List<String> two = new ArrayList<>();
System.out.println(one.equals(two)); // true
one.add("a"); // [a]
System.out.println(one.equals(two)); // false
two.add("a"); // [a]
System.out.println(one.equals(two)); // true
one.add("b"); // [a,b]
two.add(0, "b"); // [b,a]
System.out.println(one.equals(two)); // false
```

c) Wrapper classes

Điều gì xảy ra nếu chúng ta muốn đưa những primitive vào? Mỗi kiểu nguyên thủy có một lớp **Wrapper**, là một kiểu đối tượng tương ứng với kiểu nguyên thủy.

Wrapper classes

Primitive type	Wrapper class	Example of creating
boolean	Boolean	Boolean.valueOf(true)
byte	Byte	Byte.valueOf((byte) 1)
short	Short	Short.valueOf((short) 1)
int	Integer	Integer.valueOf(1)
long	Long	Long.valueOf(1)
float	Float	Float.valueOf((float) 1.0)
double	Double	Double.valueOf(1.0)
char	Character	Character.valueOf('c')

Mỗi wrapper classes cũng có một cóntructor. Nó hoạt động theo cách tương tự như valueOf() nhưng không được khuyến nghị.

Các wrapper classes là bất biến và cũng tận dụng một số bộ nhớ đệm. Các wrapper classes cũng có một phương thức chuyển đổi về dạng nguyên thủy.

Các phương thức parse, chẳng hạn như parsInt(), trả về một giá trị nguyên thủy và phương thức valueOf() trả về một wrapper class. Điều này rất dễ nhớ vì tên của nguyên thủy được trả về nằm trong tên phương thức. Đây là một ví dụ:

```
int primitive = Integer.parseInt("123");
Integer wrapper = Integer.valueOf("123");
```

Nếu String được truyền vào không hợp lệ đối với loại đã cho, Java sẽ đưa ra một ngoại lệ. Trong những ví dụ này, các chữ cái và dấu chấm không hợp lệ đối với giá trị số nguyên:

```
int bad1 = Integer.parseInt("a"); // throws NumberFormatException
Integer bad2 = Integer.valueOf("123.45"); // throws NumberFormatException
```

Converting from a String

Wrapper class	Converting String to a primitive	Converting String to a wrapper class
Boolean	Boolean.parseBoolean("true")	Boolean.valueOf("TRUE")
Byte	Byte.parseByte("1")	Byte.valueOf("2")
Short	Short.parseShort("1")	Short.valueOf("2")
Integer	Integer.parseInt("1")	Integer.valueOf("2")
Long	Long.parseLong("1")	Long.valueOf("2")
Float	Float.parseFloat("1")	Float.valueOf("2.2")
Double	Double.parseDouble("1")	Double.valueOf("2.2")
Character	None	None

d) Autoboxing and unboxing

Kể từ Java 5, chỉ cần nhập giá trị nguyên thủy và Java sẽ chuyển đổi nó thành wrapper classes phù hợp cho bạn. Điều này được gọi là **autoboxing**. Việc chuyển đổi ngược lại lớp bao bọc thành giá trị nguyên thủy được gọi là **unboxing**. Hãy xem một ví dụ:

```
List<Integer> weights = new ArrayList<>();
Integer w = 50;
weights.add(w); // [50]
weights.add(Integer.valueOf(60)); // [50, 60]
weights.remove(new Integer(50)); // [60]
double first = weights.get(0); // 60.0
```

Bạn nghĩ điều gì sẽ xảy ra nếu bạn cố gắng mở hộp một giá trị rỗng?

```
List<Integer> heights = new ArrayList<>();
heights.add(null);
int h = heights.get(0); // NullPointerException
```

Ngoài ra, hãy cẩn thận khi tự động **autoboxing** vào Integer. Bạn nghĩ mã này đầu ra là gì?

```
List<Integer> numbers = new ArrayList<>();
numbers.add(1);
numbers.add(2);
numbers.remove(1);
System.out.println(numbers);
```

⇨ Nó thực sự xuất ra [1]. Sau khi thêm hai giá trị, Danh sách chứa [1, 2]. Sau đó chúng tôi yêu cầu loại bỏ phần tử có index 1 (không phải phần tử 1).

e) Converting between array and list

Hãy bắt đầu với việc biến ArrayList thành một mảng:

```
List<String> list = new ArrayList<>();
list.add("hawk");
list.add("robin");
Object[] objectArray = list.toArray();
String[] stringArray = list.toArray(new String[0]);
list.clear();
System.out.println(objectArray.length); // 2
System.out.println(stringArray.length); // 2
```

⇨ Hãy lưu ý rằng sẽ xóa List ban đầu. Điều này không ảnh hưởng đến cả hai mảng. Mảng là một đối tượng mới được tạo không có mối quan hệ nào với List ban đầu. Nó chỉ đơn giản là một bản sao.

Việc chuyển đổi từ một mảng thành List thú vị hơn. Có hai phương pháp để thực hiện chuyển đổi này. Hãy chú ý cẩn thận đến các giá trị ở đây:

Arrays.asList() - fixed-size list

```
20:      String[] array = { "hawk", "robin" }; // [hawk, robin]
21:      List<String> list = Arrays.asList(array); // returns fixed size list
22:      System.out.println(list.size()); // 2
23:      list.set(1, "test"); // [hawk, test]
24:      array[0] = "new"; // [new, test]
25:      System.out.print(Arrays.toString(array));// [new, test]
26:      list.remove(1); // throws UnsupportedOperationException
```

⇨ Tại sao lại lỗi ?

Một tùy chọn khác là tạo immutable List (List bất biến). Điều đó có nghĩa là bạn không thể thay đổi giá trị hoặc kích thước của List. Bạn có thể thay đổi mảng ban đầu, nhưng những thay đổi sẽ không được phản ánh trong immutable List.

```
32:     String[] array = { "hawk", "robin" }; // [hawk,robin]
33:     List<String> list = List.of(array); // returns immutable list
34:     System.out.println(list.size()); // 2
35:     array[0] = "new";
36:     System.out.println(Arrays.toString(array)); // [new,robin]
37:     System.out.println(list); // [hawk,robin]
38:     list.set(1, "test"); // throws UnsupportedOperationException
```

f) Using varargs to create a list

Sử dụng varargs cho phép bạn tạo List:

```
List<String> list1 = Arrays.asList("one", "two");
List<String> list2 = List.of("one", "two");
```

Cả hai phương thức này đều sử dụng các biến thể, cho phép bạn chuyển vào một mảng hoặc chỉ cần gõ các giá trị String. Điều này rất hữu ích khi testing vì bạn có thể dễ dàng tạo và điền List trên một dòng. Cả hai phương pháp đều tạo mảng có kích thước cố định. Nếu sau này bạn cần thêm hoặc xóa các phần tử, bạn vẫn cần tạo ArrayList bằng cách sử dụng constructor.

	toArray()	Arrays.asList()	List.of()
Chuyển đổi từ	List	Array (or varargs)	Array (or varargs)
Kiểu dữ liệu được tạo	Array	List	List
Được phép xóa giá trị khỏi đối tượng đã tạo	No	No	No
Được phép thay đổi giá trị trong đối tượng đã tạo	Yes	Yes	No
Thay đổi giá trị trong đối tượng được tạo sẽ ảnh hưởng đến đối tượng gốc hoặc ngược lại.	No	Yes	N/A

Lưu ý rằng không có cách nào cho phép thay đổi số lượng phần tử. Nếu muốn làm điều đó, thực sự cần phải viết logic để tạo đối tượng mới. Đây là một ví dụ:

```
List<String> fixedSizeList = Arrays.asList("a", "b", "c");
List<String> expandableList = new ArrayList<>(fixedSizeList);
```

g) Sorting

Sắp xếp một ArrayList tương tự như sắp xếp một mảng. Chỉ cần sử dụng một class trợ giúp khác:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(99);
numbers.add(5);
numbers.add(81);
Collections.sort(numbers);
System.out.println(numbers); // [5, 81, 99]
```

6. Creating Sets and Maps

a) Introducing sets

Set là một interface kế thừa Collection interface trong java. Set trong java là một Collection không thể chứa các phần tử trùng lặp.

Set được triển khai bởi Hashset, LinkedHashset, Treeset hoặc EnumSet.

- **HashSet** lưu trữ các phần tử của nó trong bảng băm, là cách thực hiện tốt nhất, tuy nhiên nó không đảm bảo về thứ tự các phần tử được chèn vào.
- **TreeSet** lưu trữ các phần tử của nó trong một cây, sắp xếp các phần tử của nó dựa trên các giá trị của chúng, về cơ bản là chậm hơn HashSet.
- **LinkedHashSet** được triển khai dưới dạng bảng băm với có cấu trúc dữ liệu danh sách liên kết, sắp xếp các phần tử của nó dựa trên thứ tự chúng được chèn vào tập hợp (thứ tự chèn).
- **EnumSet** là một cài đặt chuyên biệt để sử dụng với các kiểu enum.

Để đảm bảo bạn hiểu Set, hãy xem ví dụ sau:

```
Set<Integer> set = new HashSet<>();
System.out.println(set.add(66)); // true
System.out.println(set.add(66)); // false
System.out.println(set.size()); // 1
set.remove(66);
System.out.println(set.isEmpty()); // true
```

Các phương thức của interface Set trong java

Method	Description
boolean add(Object element)	Nó được sử dụng để chèn các phần tử vào set.
boolean addAll(Collection c)	Nó được sử dụng để chèn tất cả các phần tử của c vào set.
void clear()	Xóa tất cả các phần tử khỏi set.
boolean contains(Object element)	Trả về true nếu tập hợp này chứa phần tử đã chỉ định.
boolean containsAll(Collection c)	Trả về true nếu set chứa tất cả các phần tử của collection c đã chỉ định.
boolean equals(Object o)	So sánh các đối tượng được chỉ định với set.
boolean isEmpty()	Trả về true nếu set không chứa phần tử.
int hashCode()	Trả về giá trị mã băm
Iterator iterator()	Trả về một trình vòng lặp iterator để duyệt qua các phần tử của set.
boolean remove(Object o)	Xóa phần tử đã chỉ định khỏi set.
boolean removeAll(Collection c)	Xóa khỏi set tất cả các phần tử của nó được chứa trong collection c đã chỉ định.
boolean retainAll(Collection c)	Chỉ giữ lại các phần tử trong set được chứa trong collection c đã chỉ định.
int size()	Trả về số lượng các phần tử của set.
Object[] toArray()	Trả về một mảng chứa tất cả các phần tử trong set.
T[] toArray(T[] a)	Trả về một mảng chứa tất cả các phần tử trong set, kiểu run-time của mảng trả về là kiểu đã chỉ định.

b) Introducing maps

Trong java, map được sử dụng để lưu trữ và truy xuất dữ liệu theo cặp key và value. Mỗi cặp key và value được gọi là mục nhập (entry). Map trong java chỉ chứa các giá trị key duy nhất. Map rất hữu ích nếu bạn phải tìm kiếm, cập nhật hoặc xóa các phần tử trên dựa vào các key.

George	555-555-5555
Mary	777-777-7777

Việc triển khai Map phổ biến nhất là HashMap. Một số phương thức tương tự như các phương thức trong ArrayList như clear(), isEmpty() và size()

Common Map methods

Method	Description
V get(Object key)	Trả về giá trị được ánh xạ theo key hoặc null nếu không có giá trị nào được ánh xạ
V getOrDefault(Object key, V other)	Trả về giá trị được ánh xạ theo key hoặc giá trị khác nếu không có giá trị nào được ánh xạ
V put(K key, V value)	Thêm hoặc thay thế cặp key/value. Trả về giá trị trước đó hoặc null
V remove(Object key)	Loại bỏ và trả về giá trị được ánh xạ tới key. Trả về null nếu không có
boolean containsKey(Object key)	Trả về liệu khóa có trong Map hay không
boolean containsValue(Object value)	Trả về xem giá trị có trong Map hay không
Set<K> keySet()	Trả về Set tất cả các key
Collection<V> values()	Trả về Collection tất cả các giá trị

Bây giờ hãy xem một ví dụ để xác nhận điều này là rõ ràng:

```
Map<String, String> map = new HashMap<>();
map.put("koala", "bamboo");
String food = map.get("koala"); // bamboo
String other = map.getOrDefault("ant", "leaf"); // leaf
for (String key: map.keySet())
    System.out.println(key + " " + map.get(key)); // koala bamboo
```

7. Calculating with Math APIs

a) Min() và Max()

Các phương thức min() và max() so sánh hai giá trị và trả về một trong số chúng.

Cú pháp:

```
double min(double a, double b)
float min(float a, float b)
int min(int a, int b)
long min(long a, long b)
```

Sau đây cho thấy cách sử dụng các phương pháp này:

```
int first = Math.max(3, 7); // 7
int second = Math.min(7, -9); // -9
```

b) Round()

Phương thức round() loại bỏ phần thập phân của giá trị, chọn số cao hơn tiếp theo nếu phù hợp. Nếu phần phân số bằng 0,5 hoặc cao hơn thì chúng ta làm tròn lên. Chữ ký phương thức cho round() như sau:

```
long round(double num)
int round(float num)
```

Sau đây cho thấy cách sử dụng phương pháp này:

```
long low = Math.round(123.45); // 123
long high = Math.round(123.50); // 124
int fromFloat = Math.round(123.45f); // 123
```

c) Pow()

Phương thức pow() xử lý số mũ. Cú pháp của phương pháp như sau:

```
double pow(double number, double exponent)
```

Sau đây cho thấy cách sử dụng phương pháp này:

```
double squared = Math.pow(5, 2); // 25.0
```

d) Random()

Phương thức Random() trả về giá trị lớn hơn hoặc bằng 0 và nhỏ hơn 1. Cú pháp phương thức như sau:

```
double random()
```

Sau đây cho thấy cách sử dụng phương pháp này:

```
double num = Math.random();
```

Chapter 6 Lambdas and Functional Interfaces

1. Writing simple lambdas

Thực chất Java là một ngôn ngữ hướng đối tượng. Trong Java 8, ngôn ngữ này đã thêm khả năng viết mã bằng một kiểu khác.

Functional programming là một cách viết mã mang tính khai báo hơn. Bạn chỉ định những gì bạn muốn làm hơn là xử lý trạng thái của các đối tượng. Bạn tập trung nhiều hơn vào biểu thức hơn là vòng lặp.

Functional programming sử dụng biểu thức lambda để viết mã. Biểu thức lambda là một khối mã được truyền đi khắp nơi. Bạn có thể coi biểu thức lambda là một phương thức chưa được đặt tên. Nó có các tham số và phần thân giống như các phương thức chính thức, nhưng nó không có tên như một phương thức thực. **Biểu thức Lambda** thường được gọi tắt là lambdas.

Nói cách khác, biểu thức lambda giống như một phương thức mà bạn có thể truyền vào như thể nó là một biến.

a) Lambda example

Mục tiêu của chúng tôi là in ra tất cả các Animal trong danh sách theo một số tiêu chí. Chúng ta bắt đầu với lớp Animal:

```
public class Animal {
    private String species;
    private boolean canHop;
    private boolean canSwim;
    public Animal(String speciesName, boolean hopper, boolean swimmer){
        species = speciesName;
        canHop = hopper;
        canSwim = swimmer;
    }
    public boolean canHop() { return canHop; }
    public boolean canSwim() { return canSwim; }
    public String toString() { return species; }
}
```

Tạo một interface chỉ định các phương thức mà lớp của chúng ta cần triển khai:

```
public interface CheckTrait {
    boolean test(Animal a);
}
```

Điều đầu tiên chúng tôi muốn kiểm tra là liệu Animal có thể nhảy(hop) hay không. Chúng tôi cung cấp một class có thể kiểm tra điều này:

```
public class CheckIfHopper implements CheckTrait {
    public boolean test(Animal a) {
        return a.canHop();
    }
}
```

Class có vẻ đơn giản—và đúng là như vậy. Đây thực sự là một phần vấn đề mà lambdas giải quyết. Bây giờ chúng ta đã có mọi thứ cần thiết để viết mã tìm Animals có thể nhảy:

```
1:      import java.util.*;
2:      public class TraditionalSearch {
3:          public static void main(String[] args) {
4:
5:      // list of animals
6:              List<Animal> animals = new ArrayList<Animal>();
7:              animals.add(new Animal("fish", false, true));
8:              animals.add(new Animal("kangaroo", true, false));
9:              animals.add(new Animal("rabbit", true, false));
10:             animals.add(new Animal("turtle", false, true));
11:
12:             // pass class that does check
13:             print(animals, new CheckIfHopper());
14:         }
15:         private static void print(List<Animal> animals,
16:                                   CheckTrait checker) {
17:             for (Animal animal : animals) {
18:
19:                 // the general check
20:                 if (checker.test(animal))
21:                     System.out.print(animal + " ");
22:             }
23:             System.out.println();
24:         }
25:     }
```

Bây giờ điều gì sẽ xảy ra nếu chúng ta muốn in ra các loài Animal biết bơi? Chúng ta cần viết một lớp khác, CheckIfSwims. Sau đó, chúng ta cần thêm một dòng mới bên dưới dòng 13 để khởi tạo lớp đó. Tại sao chúng ta không thể chỉ rõ logic mà chúng ta quan tâm ngay tại đây? Hóa ra chúng ta có thể làm được với biểu thức lambda.

b) Lambda syntax

Một trong những biểu thức lambda đơn giản nhất bạn có thể viết là:

```
a -> a.canHop()
```

Ví dụ:

```
public class Animal {
    private String species;
    private boolean canHop;
    private boolean canSwim;
    public Animal(String speciesName, boolean hopper, boolean swimmer){
        species = speciesName;
        canHop = hopper;
        canSwim = swimmer;
    }
    public boolean canHop() { return canHop; }
    public boolean canSwim() { return canSwim; }
    public String toString() { return species; }
```

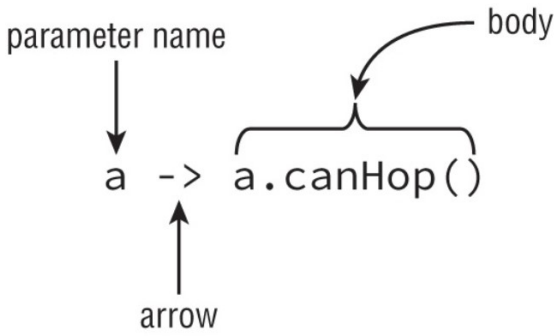
```
}


interface CheckTrait {
    boolean test(Animal a);
}

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

// list of animals
        List<Animal> animals = new ArrayList<Animal>();
        animals.add(new Animal("fish", false, true));
        animals.add(new Animal("kangaroo", true, false));
        animals.add(new Animal("rabbit", true, false));
        animals.add(new Animal("turtle", false, true));

        // pass class that does check
        print(animals,a -> a.canHop());
    }

    private static void print(List<Animal> animals,
                              CheckTrait checker) {
        for (Animal animal : animals) {

            // the general check
            if (checker.test(animal))
                System.out.print(animal + " ");
        }
        System.out.println();
    }
}
```

Lambdas hoạt động với các interface chỉ có một phương thức trừu tượng. Trong trường hợp này, Java xem xét interface CheckTrait có một phương thức. Lambda chỉ ra rằng Java nên gọi một phương thức có tham số Animal trả về giá trị boolean là kết quả của a.canHop().

Cú pháp của lambdas phức tạp vì nhiều phần là tùy chọn. Hai dòng này làm điều tương tự:

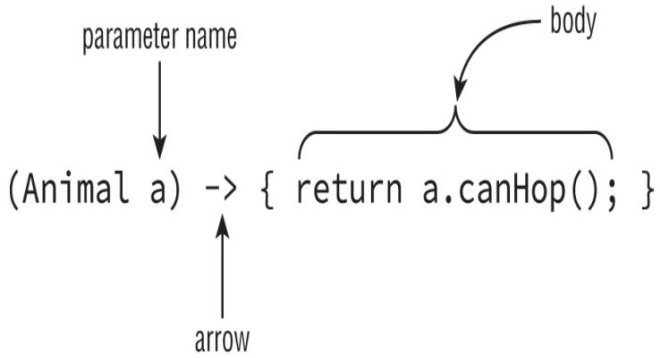
```
a -> a.canHop()
(Animal a) -> { return a.canHop(); }
```

- Một tham số duy nhất được chỉ định với tên a
- Toán tử mũi tên để phân tách tham số và nội dung
- Phần thân gọi một phương thức duy nhất và trả về kết quả của phương thức đó



Ví dụ thứ hai cho thấy lambda trả về một boolean ()

- Một tham số duy nhất được chỉ định với tên a và cho biết loại là Animal
- Toán tử mũi tên để phân tách tham số và phần thân
- Một phần thân có một hoặc nhiều dòng mã, bao gồm dấu chấm phẩy và câu lệnh trả về



Lambda hợp lệ

Lambda	**# parameters**
() -> true	0
a -> a.startsWith("test")	1
(String a) -> a.startsWith("test")	1
(a, b) -> a.startsWith("test")	2
(String a, String b) -> a.startsWith("test")	2

Lambda không hợp lệ

Invalid lambda	**Reason**
a, b -> a.startsWith("test")	Thiếu dấu ngoặc đơn
a -> { a.startsWith("test"); }	Thiếu return
a -> { return a.startsWith("test") }	thiếu dấu chấm phẩy

2. Introducing functional interfaces

Lambdas hoạt động với các interface chỉ có một phương thức trừu tượng. Chúng được gọi là ***functional interfaces***.

Có bốn ***functional interfaces*** thường được sử dụng. Đó là Predicate, Consumer, Supplier, và Comparator..

a) Predicate

Predicate<T> là một functional interface và do đó nó có thể được sử dụng với lambda expression hoặc method reference cho một mục đích cụ thể nào đó. Predicate<T> sẽ trả về giá trị true/false của một tham số kiểu T mà bạn đưa vào có thỏa với điều kiện của Predicate đó hay không, cụ thể là điều kiện được viết trong phương thức test().

Interface Predicate được khai báo trong package java.util.function như sau:

```
public interface Predicate<T> {
    boolean test(T t);
}
```

- **boolean test(T t)** : là một phương thức trừu tượng có thể được sử dụng với lambda expression hoặc method reference cho một mục đích cụ thể nào đó.
- Phương thức **test()** trả về true nếu đối số đầu vào khớp với biến predicate (điều kiện kiểm tra), nếu không trả về false.

Một số phương thức mặc định (default method) trong lớp Interface Predicate:

- **and()** : Nó thực hiện logic AND của predicate mà nó được gọi với một biến predicate khác. Ví dụ: predicate1.and (predicate2).
- **or() :** Nó thực hiện logic OR của predicate mà nó được gọi với một biến predicate khác. Ví dụ: predicate1.or(predicate2).
- **negate() :** Nó thực hiện phủ định kết quả của biến predicate được gọi. Ví dụ: predicate1.negate().

Sử dụng test()

```
Predicate<String> predicateString = s -> {
    return s.equals("1995mars");
};
System.out.println(predicateString.test("1995mars")); // true
System.out.println(predicateString.test("1995 mars")); // false

// Predicate integer
Predicate<Integer> predicateInt = i -> {
    return i > 0;
};
System.out.println(predicateInt.test(1)); // true
System.out.println(predicateInt.test(-1)); // false
```

Sử dụng and(), or(), negate(), isEqual()

```
Predicate<String> predicate = s -> {
    return s.equals("199mars");
};

// AND logical operation
Predicate<String> predicateAnd = predicate.and(s -> s.length() == 11);
System.out.println(predicateAnd.test("199mars.com")); // false

// OR logical operation
Predicate<String> predicateOr = predicate.or(s -> s.length() == 11);
System.out.println(predicateOr.test("199mars.com")); // true
```

```
    // NEGATE logical operation
    Predicate<String> predicateNegate = predicate.negate();
    System.out.println(predicateNegate.test("199mars")); // false
```

Kết hợp nhiều Predicate

```
    // Creating predicate
    Predicate<Integer> greaterThanTen = (i) -> i > 10;
    Predicate<Integer> lessThanTwenty = (i) -> i < 20;

    // Calling Predicate Chaining
    boolean result = greaterThanTen.and(lessThanTwenty).test(15);
    System.out.println(result); // true

    // Calling Predicate method
    boolean result2 = greaterThanTen.and(lessThanTwenty).negate().test(15);
    System.out.println(result2); // false
```

Sử dụng Predicate với Collection

```
    Predicate<Integer> evenNumber = (i) -> i % 2 == 0;
    List<Integer> integerList = List.of(1,2,3,4,5,6);
    integerList.stream().filter(evenNumber).forEach(System.out::println);

// Output: 2 4 6
```

b) Consumer<T>

Consumer<T> là một functional interface và do đó nó có thể được sử dụng với lambda expression hoặc method reference cho một mục đích cụ thể nào đó. **Consumer<T>** chấp nhận một tham số đầu vào, và method này không trả về gì cả.

Mục tiêu chính của Interface Consumer là thực hiện một thao tác trên đối số đã cho.

Interface Consumer được khai báo trong package **java.util.function** như sau:

```
void accept(T t)
```

Trong đó:

- void **accept(T t)** : là một phương thức trừu tượng có thể được sử dụng với lambda expression hoặc method reference cho một mục đích cụ thể nào đó.
- Phương thức **accept()** thực hiện một hành động cụ thể trên đối số đã cho.

Interface Consumer còn cung cấp một phương thức mặc định (default method) :

- **default Consumer<T> andThen(Consumer<? super T> after)** : phương thức này trả về một Consumer thực hiện hai hành động theo thứ tự, trước tiên là hành động của Consumer mà phương thức được gọi và theo sau bởi hành động của Consumer được truyền vào đối số.

Tạo consumer

```
static void printValue(int val) {
    System.out.println(val);
}

public static void main(String[] args) {
    // Create Consumer interface
    Consumer<String> consumer = new Consumer<String>() {
        @Override
        public void accept(String name) {
            System.out.println("Hello, " + name);
        }
    };
    // Calling Consumer method
    consumer.accept("gpcoder"); // Hello, gpcoder

    // Create Consumer interface with lambda expression
    Consumer<String> consumer1 = (name) -> System.out.println("Hello, " + name);
    // Calling Consumer method
    consumer1.accept("gpcoder"); // Hello, gpcoder

    // Create Consumer interface with method reference
    Consumer<Integer> consumer2 = ConsumerExample1::printValue;
    // Calling Consumer method
    consumer2.accept(12); // 12
}
```

Trong ví dụ trên, đã tạo 2 consumer:

- consumer1 : được tạo bằng cách sử dụng **lambda expression**.
- consumer2 : được tạo bằng cách sử dụng **method reference**.

Để thực thi consumer, chúng ta gọi phương thức mặc định **accept()** được cung cấp trong Interface Consumer. Phương thức này sẽ thực thi đoạn code được cài đặt thông qua lambda expression hoặc method reference.

Sử dụng phương thức mặc định andThen()

```
int testNumber = 5;
Consumer<Integer> times2 = (e) -> System.out.println(e * 2);
Consumer<Integer> squared = (e) -> System.out.println(e * e);
Consumer<Integer> isOdd = (e) -> System.out.println(e % 2 == 1);

// perform every consumer
times2.accept(testNumber); // 10
squared.accept(testNumber); // 25
isOdd.accept(testNumber); // true

// perform 3 methods in sequence
Consumer<Integer> combineConsumer = times2.andThen(squared).andThen(isOdd);
combineConsumer.accept(testNumber); // 10 25 true
```

Trong phương thức trên, đã tạo 3 consumer. Thay vì gọi phương thức **accept()** lần lượt cho từng consumer, ví dụ kết hợp chúng thông qua phương thức mặc định **andThen()**, để thực thi tất cả các consumer đó, chúng ta chỉ việc gọi một combineConsumer duy nhất.

Sử dụng Consumer với forEach loop

```
Consumer<Integer> evenNumber = (i) -> System.out.println(i);
List<Integer> integerList = List.of(1,2,3,4,5,6);
integerList.stream().forEach(evenNumber);
```

c) Supplier<T>

Supplier<T> là một functional interface và do đó nó có thể được sử dụng với lambda expression hoặc method reference cho một mục đích cụ thể nào đó. **Supplier<T>** làm ngược lại với **Consumer<T>,** nó là một phương thức trừu tượng không tham số, và trả về một đối tượng bằng cách gọi phương thức get() của nó.
Mục đích chính của Interface này là cung cấp một cái gì đó cho chúng ta khi cần.
Interface Supplier được khai báo trong package java.util.function như sau:

```
T get()
```

Trong đó:

- T get() : là một phương thức trừu tượng có thể được sử dụng với lambda expression hoặc method reference cho một mục đích cụ thể nào đó.
- Phương thức get() sẽ return một giá trị cụ thể cho chúng ta.

Tạo Supplier sử dụng Lambda Expression

```
    Supplier<String> supplier = () -> "Welcome to gpcoder.com";
    String hello = supplier.get();
    System.out.println(hello);
```

Tạo Supplier sử dụng Method Reference

```
class Programing {
    public String language;
    public int experience;

    public Programing() {
        this.language = "Java";
        this.experience = 5;
    }

    public Programing(String language, int experience) {
        this.language = language;
        this.experience = experience;
    }

    public void print() {
        System.out.println("Language: " + language + " - Experience: " +
experience);
    }

    public static String getDefaulLanguage() {
        return "Java";
    }
}

class SupplierExample2 {

    public static void main(String[] args) {

        Supplier<Programing> supplier1 = Programing::new;
        Programing lang = supplier1.get();
        lang.print();

        Supplier<String> supplier2 = Programing::getDefaulLanguage;
        String defaultLang = supplier2.get();
        System.out.println("Default Language: " + defaultLang);
    }
}
```

Comparator

Comparator trong Java được sử dụng để sắp xếp các đối tượng của lớp do người dùng định nghĩa (user-defined). Để sử dụng Comparator ta không cần phải implements Comparator cho lớp đối tượng cần được so sánh. Giao diện này thuộc về gói java.util và chứa hai phương thức là compare(Object obj1, Object obj2)

```
int compare (Object obj1, Object obj2)
```

Với interface Comparator và phương thức compare(), chúng ta có thể sắp xếp các phần tử của:

- Các đối tượng String
- Các đối tượng của lớp Wrapper
- Các đối tượng của lớp do người dùng định nghĩa (User-defined)

```
class Programing {
    public String language;
    public int experience;
    // get, set, toString
}
class SupplierExample2 {

    public static void main(String[] args) {

        Comparator<Programing> comparator = (a,b) -> a.getExperience() -
b.getExperience();
        List<Programing> programingList = List.of(
                                            new Programing("Java",5),
                                            new Programing("Python", 7)
                            );
        programingList.stream().sorted(comparator).forEach(System.out::println);
    }
}
```

3. Working with variables in lambdas

a) Parameter list

Bạn đã biết rằng việc chỉ định kiểu tham số là tùy chọn. Ngoài ra, var có thể được sử dụng thay cho kiểu cụ thể. Điều đó có nghĩa là cả ba câu lệnh này đều có thể thay thế cho nhau:

```
Predicate<String> p = x -> true;
Predicate<String> p = (var x) -> true;
Predicate<String> p = (String x) -> true;
```

Hãy thử một ví dụ khác. Bạn có thể tìm ra kiểu x không?

```
public void whatAmI() {
    consume((var x) -> System.out.print(x), 123);
}

public void consume(Consumer<Integer> c, int num) {
    c.accept(num);
}
```

⇨ X là gì ?

Bạn nghĩ loại x ở đây là gì?

```
public void counts(List<Integer> list) {
    list.sort((var x, var y) -> x.compareTo(y));
}
```

⇨ 2 trường hợp đều là Integer. Vì chúng ta đang sắp xếp một List nên chúng ta có thể sử dụng loại List để xác định loại tham số lambda.

b) Local variables inside the lambda body

Đoạn code sau thể thiện biến local bên trong lambda. Nó tạo ra một biến local có tên c nằm trong lambda block.

```
(a, b) -> { int c = 0; return 5;}
```

Bây giờ hãy thử một cái khác. Bạn có thấy điều gì sai ở đây không?

```
(a, b) -> { int a = 0; return 5;} // DOES NOT COMPILE
```

Code đã cố gắng khai báo lại a, điều này không được phép. Java không cho phép bạn tạo biến local có cùng tên với biến đã được khai báo trong phạm vi đó. Bây giờ chúng ta hãy thử một cái khó. Bạn thấy có bao nhiêu lỗi cú pháp trong phương pháp này?

```
11:     public void variables(int a) {
12:         int b = 1;
13:         Predicate<Integer> p1 = a -> {
14:             int b = 0;
15:             int c = 0;
16:         return b == c;}
17:     }
```

⇨ Có ba lỗi cú pháp. (13,14,16)

c) Variables referenced from the lambda body

Lambda body được phép tham chiếu một số biến từ mã xung quanh. Đoạn mã sau là hợp lệ:

```
public class Crow {
    private String color;
    public void caw(String name) {
        String volume = "loudly";
        Consumer<String> consumer = s -> {
            System.out.println(name + " says " + volume + " that she is " + color);
        };
    }
}
```

Điều này cho thấy lambda có thể truy cập một biến instance, tham số phương thức hoặc biến local trong một số điều kiện nhất định. Các biến instance (và biến class) luôn được cho phép. Các tham số phương thức và biến local được phép tham chiếu nếu chúng là effectively final. Điều này có nghĩa là giá trị của một biến không thay đổi sau khi nó được khởi tạo, bất kể nó có được đánh dấu rõ ràng là final hay không. Nếu bạn không chắc liệu một biến có phải là biến final hay không, hãy thêm từ khóa final. Nếu mã vẫn biên dịch thì biến đó thực sự là biến final. Bạn có thể nghĩ về nó như thể chúng tôi đã viết điều này:

```
2:      public class Crow {
3:          private String color;
4:          public void caw(String name) {
5:              String volume = "loudly";
6:              name = "Caty";
7:              color = "black";
8:
9:              Consumer<String> consumer = s ->
10:             System.out.println(name + " says "
11:                     + volume + " that she is " + color);
12:                     volume = "softly";
13:         }
14:     }
```

Name không là final vì nó được set ở dòng 6. Lỗi trình biên dịch xảy ra ở dòng 10. Khi lambda cố gắng sử dụng nó, biến không phải là biến effectively final nên lambda không được phép sử dụng biến đó. volume cũng không phải là effectively final vì nó được cập nhật ở dòng 12. Trong trường hợp này, lỗi trình biên dịch nằm ở dòng 11.

Quy tắc truy cập một biến từ lambda body bên trong một phương thức:

Variable type	**Rule**
Instance variable	Allowed
Static variable	Allowed
Local variable	Allowed if effectively final
Method parameter	Allowed if effectively final
Lambda parameter	Allowed

4. Calling apis with lambdas

a) Removeif()

List và Set khai báo một phương thức removeIf() sử dụng Predicate. Ví dụ:

```
List<String> students = new ArrayList<>();
students.add("Anh");
students.add("Yến");
students.add("Thảo");
System.out.println(students); // [Anh, Yến, Thảo]
students.removeIf(s -> s.charAt(0) != 'Y');
System.out.println(students); // [Yến]
```

b) Sort()

Mặc dù bạn có thể gọi Collections.sort(list), nhưng giờ đây bạn có thể sắp xếp trực tiếp trên đối tượng List.

```
List<String> students = new ArrayList<>();
students.add("Anh");
students.add("Yến");
students.add("Thảo");
System.out.println(students); // [Anh, Yến, Thảo]
students.sort((b1, b2) -> b1.compareTo(b2));
System.out.println(students); // [Anh, Thảo, Yến]
```

Phương thức sort() sử dụng Comparator để cung cấp thứ tự sắp xếp. Hãy nhớ rằng Comparator nhận hai tham số và trả về một int.

c) Foreach()

Phương thức cuối cùng của chúng ta là forEach(). Nó nhận một **Consumer** và gọi lambda đó cho từng phần tử gặp phải:

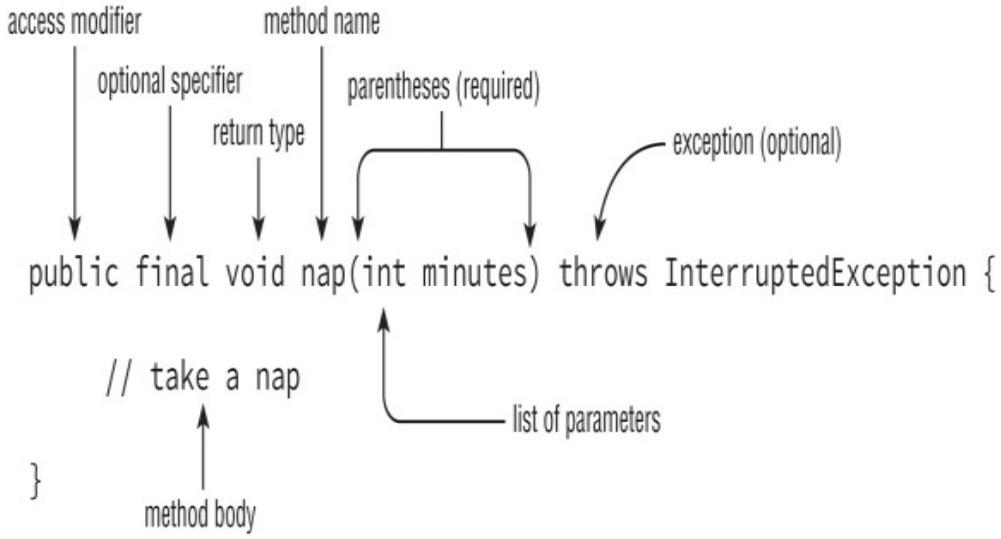
```
List<String> students = new ArrayList<>();
students.add("Anh");
students.add("Yến");
students.add("Thảo");
System.out.println(students); // [Anh, Yến, Thảo]
students.forEach(b -> System.out.println(b));
```

Output:

```
Anh
Yến
Thảo
```

Chapter 7: Methods and Encapsulation

1. Designing Methods



Đây được gọi là khai báo phương thức, trong đó chỉ định tất cả thông tin cần thiết để gọi phương thức. Hai phần—tên phương thức và danh sách tham số—được gọi là chữ ký phương thức - method signature. Bảng dưới đây là một tham chiếu ngắn gọn về các thành phần của khai báo phương thức.

Element	**Value in nap() example**	**Required?**
Access modifier	public	No
Optional specifier	final	No
Return type	void	Yes
Method name	nap	Yes
Parameter list	(int minutes)	Yes, but can be empty parentheses
Optional exception list	throws InterruptedException	No

Method body*	{
	// take a nap
	}
		Yes, but can be empty braces

a) Access modifiers

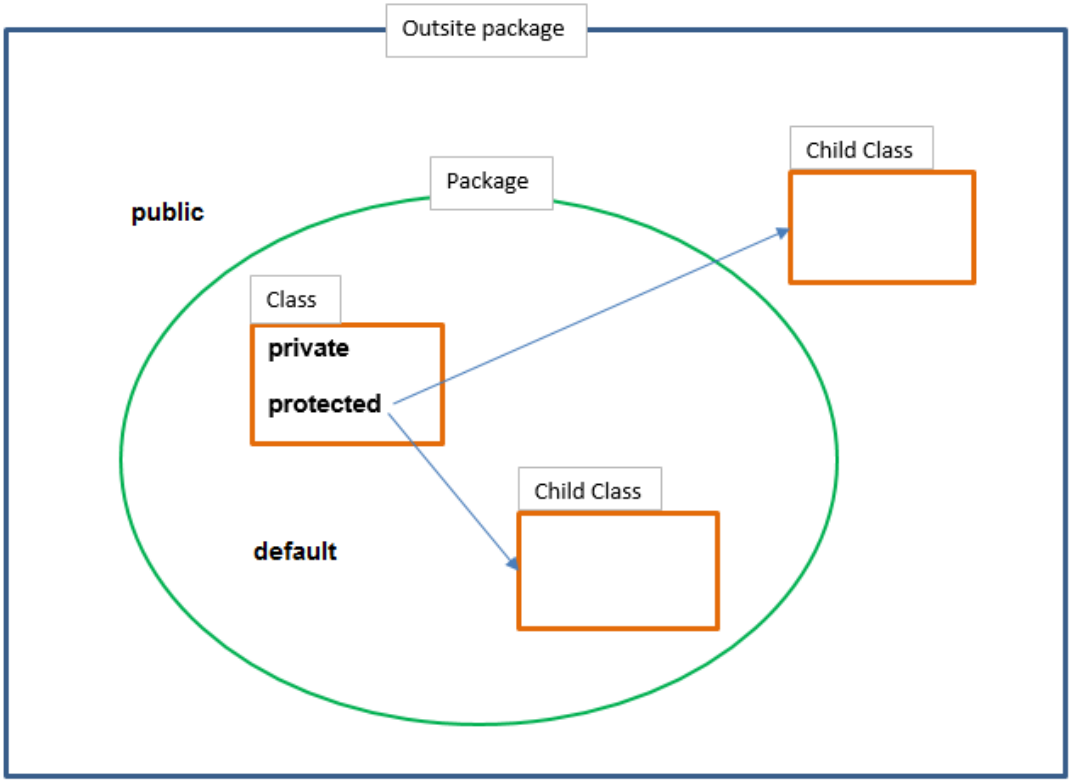
Java cung cấp bốn lựa chọn về access modifier:

- **private** có nghĩa là phương thức này chỉ có thể được gọi từ trong cùng một lớp.
- **default** (Package-Private): Với quyền truy cập default, phương thức này chỉ có thể được gọi từ các class trong cùng một package. Điều này phức tạp vì default chỉ được sử dụng trong default method trong interface.
- **protected**: có nghĩa là phương thức chỉ có thể được gọi từ các lớp trong cùng một package hoặc các class con.

Bảng dưới đây mô tả khả năng truy cập của các Access Modifier trong java:

Access Modifier	Trong lớp	Trong package	Ngoài package bởi lớp con	Ngoài package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Hình ảnh minh họa:



Hãy chú ý đến các ví dụ sau:

```
public void walk1() {}
default void walk2() {} // DOES NOT COMPILE
void public walk3() {} // DOES NOT COMPILE
void walk4() {}
```

private

Private Access Modifier chỉ được truy cập trong phạm vi lớp.

Ví dụ về private access modifier trong java

Trong ví dụ, chúng ta tạo 2 lớp A và Simple. Lớp A chứa biến và phương thức được khai bao là private. Chúng ta cố gắng truy cập chúng từ bên ngoài lớp A. Điều này dẫn đến Compile time error:

```
class A {
    private int data = 40;

    private void msg() {
        System.out.println("Hello java");
    }
}

public class Simple {
    public static void main(String args[]) {
        A obj = new A();
        System.out.println(obj.data);// Compile Time Error
        obj.msg();// Compile Time Error
    }
}
```

default

Nếu bạn không khai báo modifier nào, thì nó chính là trường hợp mặc định. Default Access Modifier là chỉ được phép truy cập trong cùng package.

Ví dụ về Default Access Modifier trong Java:

```
// Lưu file với tên A.java
package tttung.demo1;

class A {
    void msg() {
        System.out.println("Hello");
    }
}
```

```
package tttung.demo2;

import tttung.demo1.*;

public class B {
    public static void main(String args[]) {
        A obj = new A(); // Compile Time Error
        obj.msg();       // Compile Time Error
    }
}
```

protected

Protected access modifier được truy cập bên trong package và bên ngoài package nhưng phải kế thừa.

Protected access modifier có thể được áp dụng cho biến, phương thức, constructor. Nó không thể áp dụng cho lớp.

Ví dụ về protected access modifier trong Java:

```
package tttung.demo1;

public class A {
    protected void msg() {
        System.out.println("Hello");
    }
}
```

```
package tttung.demo2;

import tttung.demo1.*;

public class B extends A {
    public static void main(String args[]) {
        B obj = new B();
        obj.msg();
    }
}
```

2. Working with Varargs
3. Applying Access Modifiers
4. Applying the static Keyword
5. Passing Data among Methods
6. Overloading Methods
7. Encapsulating Data