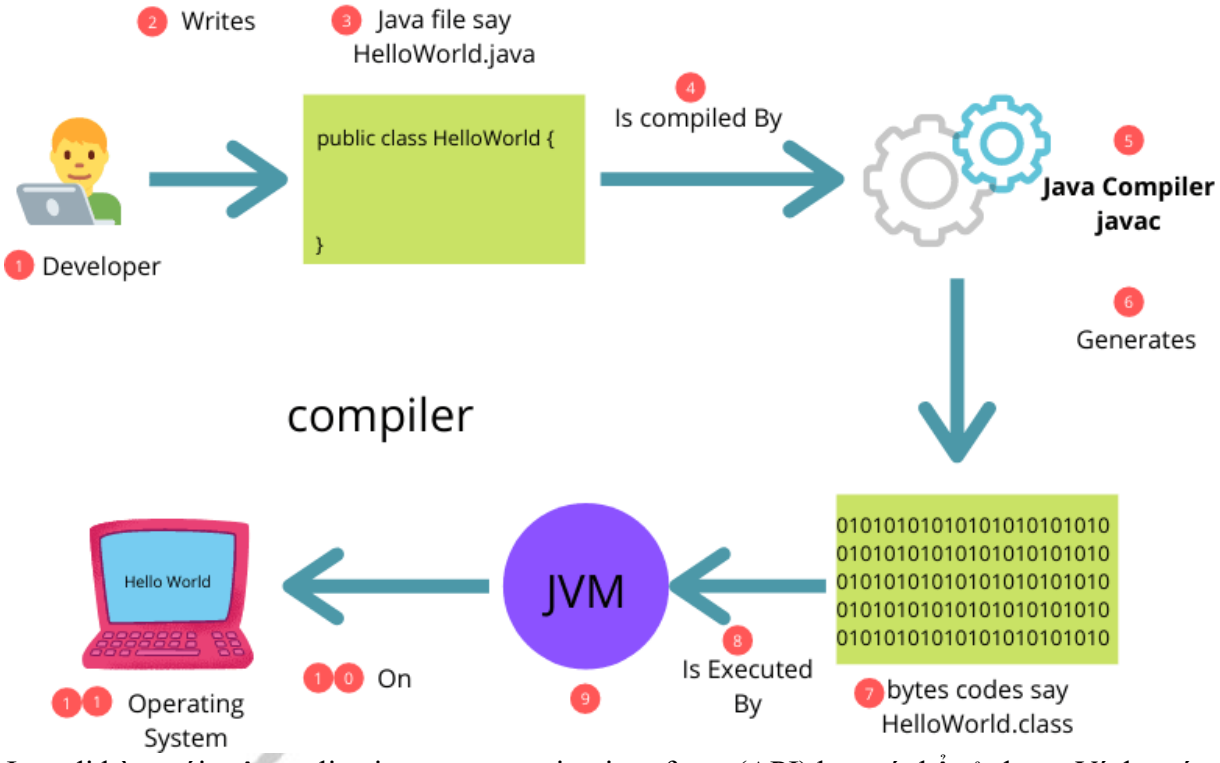
Chapter 1: Welcome to Java

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:

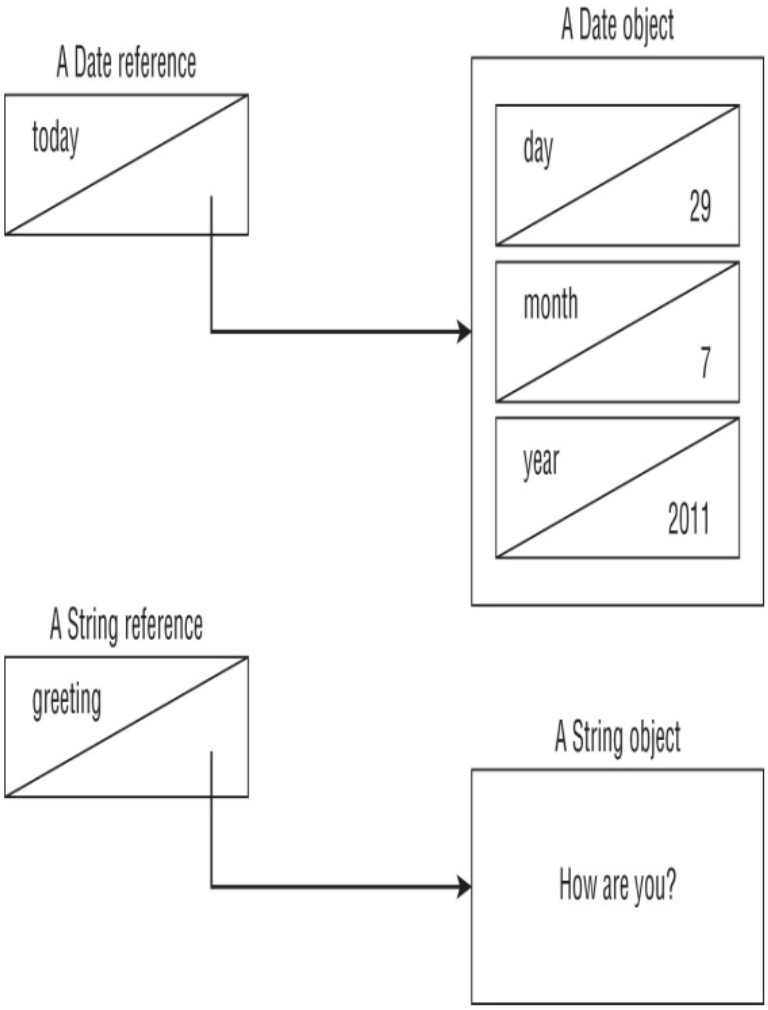
```
java.util.Date today;
String greeting;
```

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, interface 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 override 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 interface 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++;
}
```

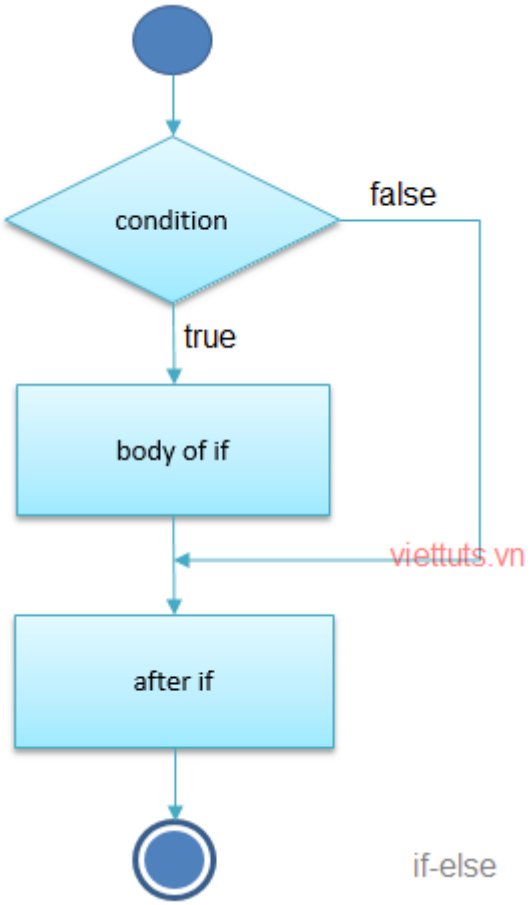
a) Statement if-else

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:

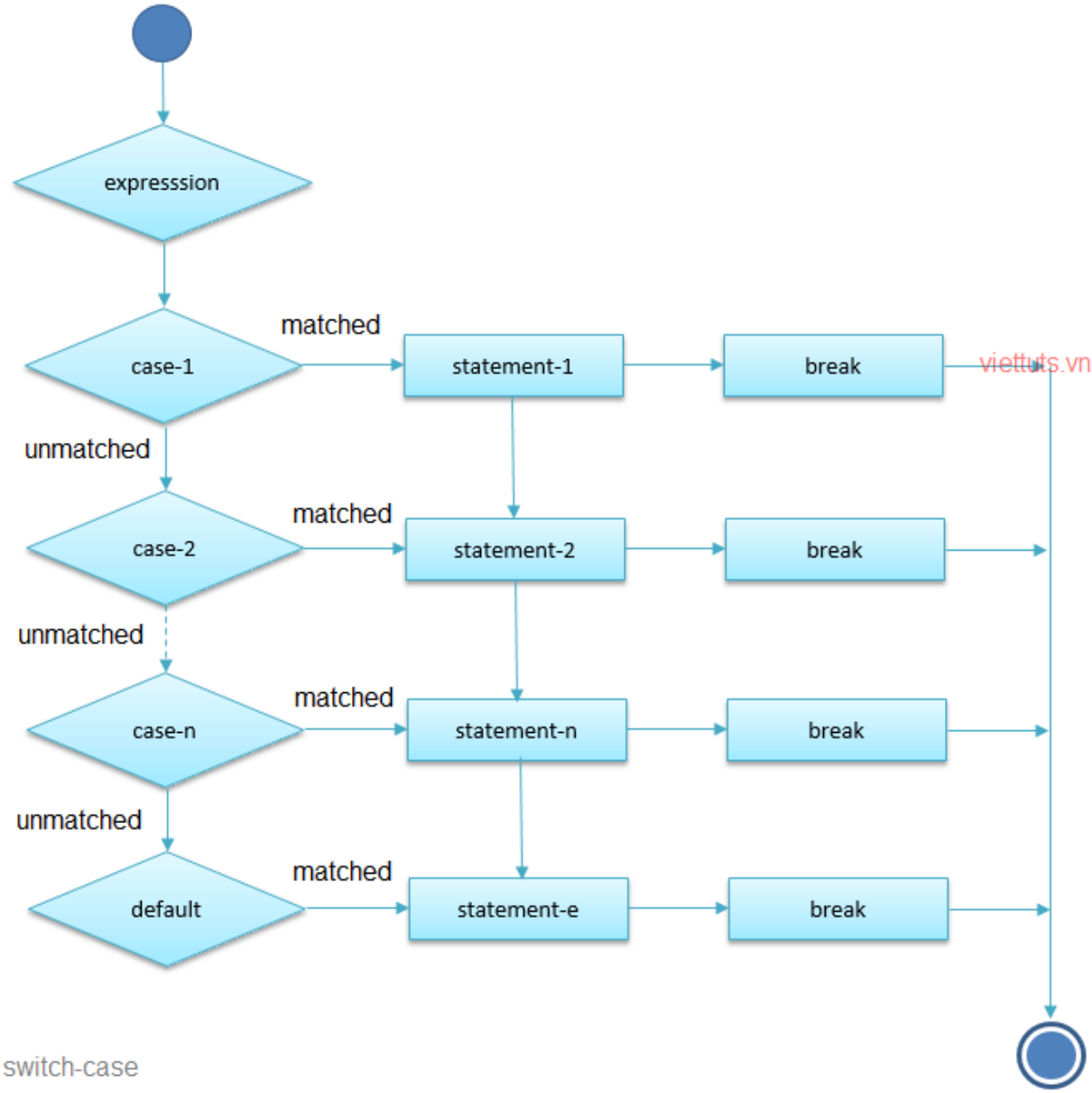
```
if (condition1) {
    // khối lệnh này được thực thi
    // nếu condition1 là true
} else if (condition2) {
    // khối lệnh này được thực thi
    // nếu condition2 là true
}  else if (condition3) {
    // khối lệnh này được thực thi
    // nếu condition3 là true
}
...
else {
    // khối lệnh này 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++;
}
```

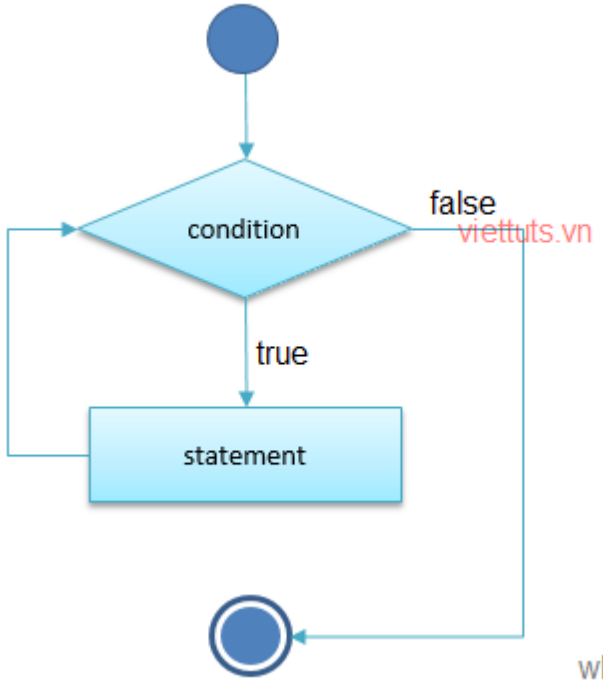
⇨ In ra **counter * 10,** tăng giá trị của counter lên 1, và kết thúc vòng lặp nếu counter >= 10

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.

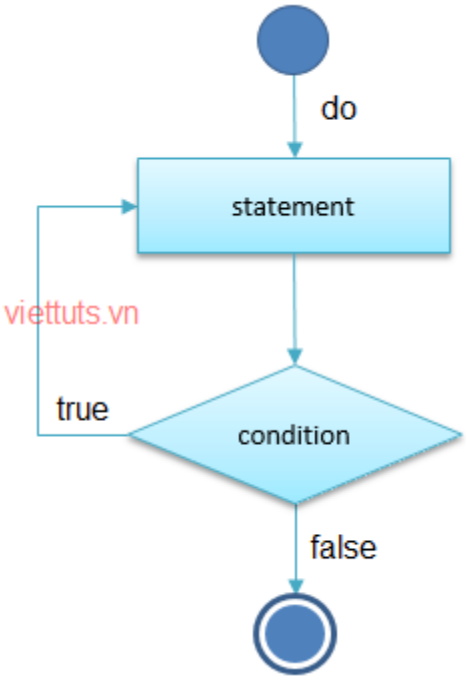
```
int roomInBelly = 5;
public void eatCheese(int bitesOfCheese) {
    while (bitesOfCheese > 0 && roomInBelly > 0) {
        bitesOfCheese--;
        roomInBelly--;
    }
    System.out.println(bitesOfCheese+" pieces of cheese left");
}
```

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:

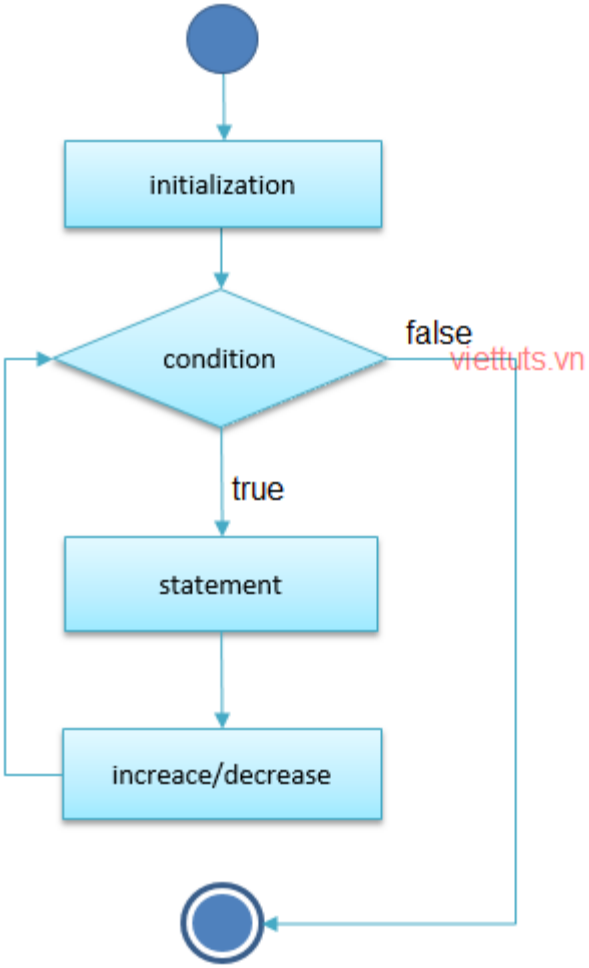
```
int pen = 2;
int pigs = 5;
while(pen < 10)
    pigs++;
```

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 + " ");
}
```

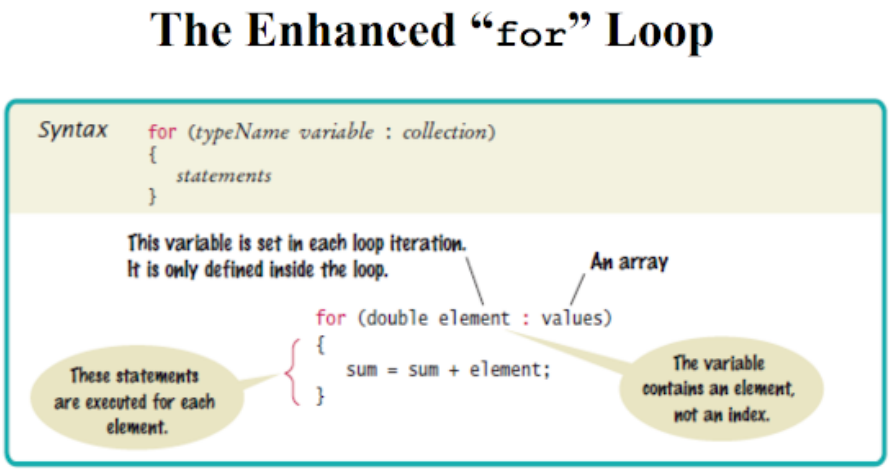
⇨ trong ví dụ này **y và x** có các loại khác nhau nên mã sẽ không biên dịch được

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

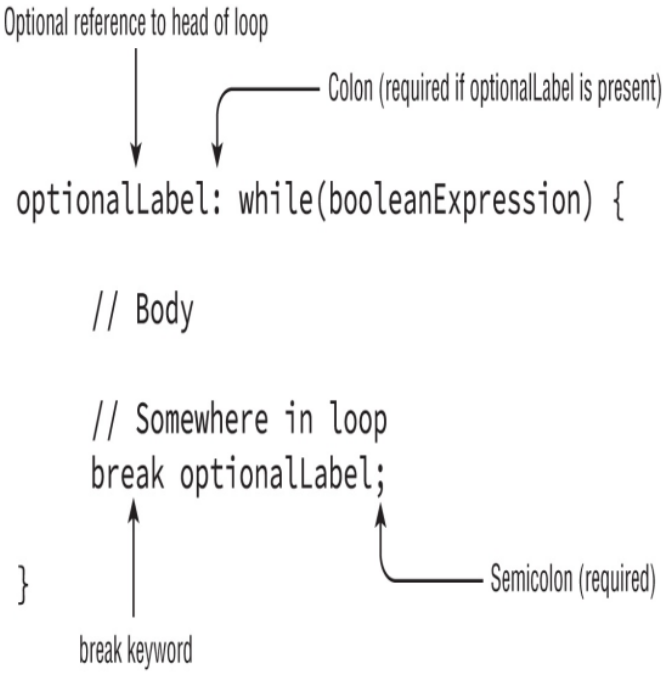
Label là một con trỏ tùy chọn tới phần đầu của câu lệnh cho phép luồng ứng dụng chuyển đến hoặc thoát khỏi nó. Nó là một identifier duy nhất được bắt đầu bằng dấu hai chấm (:). Ví dụ: có thể thêm nhãn tùy chọn vào một trong các ví dụ trước:

```
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:

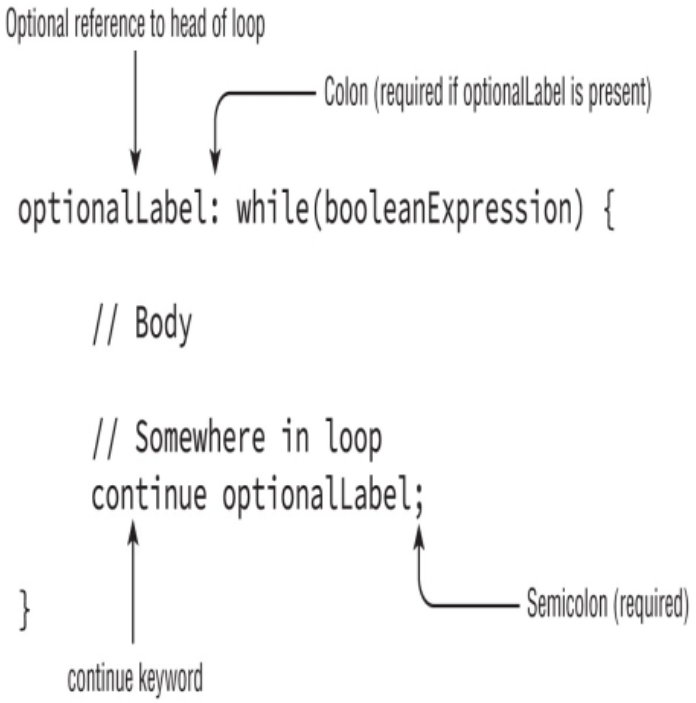
```
public class FindInMatrix {
    public static void main(String[] args) {
        int[][] list = {{1,13},{5,2},{2,2}};
        int searchValue = 2;
        int positionX = -1;
        int positionY = -1;
        PARENT_LOOP: for(int i=0; i<list.length; i++) {
            for(int j=0; j<list[i].length; 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 override 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 constructor 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:

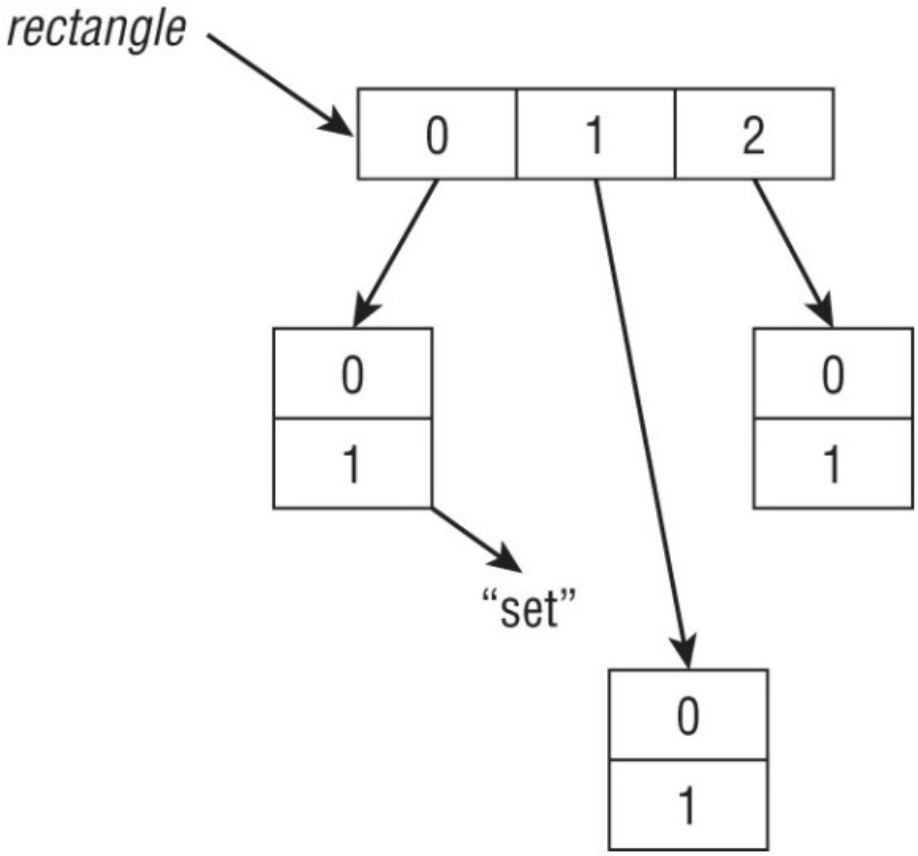
```
int[][] vars1; // 2D array
int vars2 [][]; // 2D array
int[] vars3[]; // 2D array
int[] vars4 [], space [][]; // a 2D AND a 3D array
```

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

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

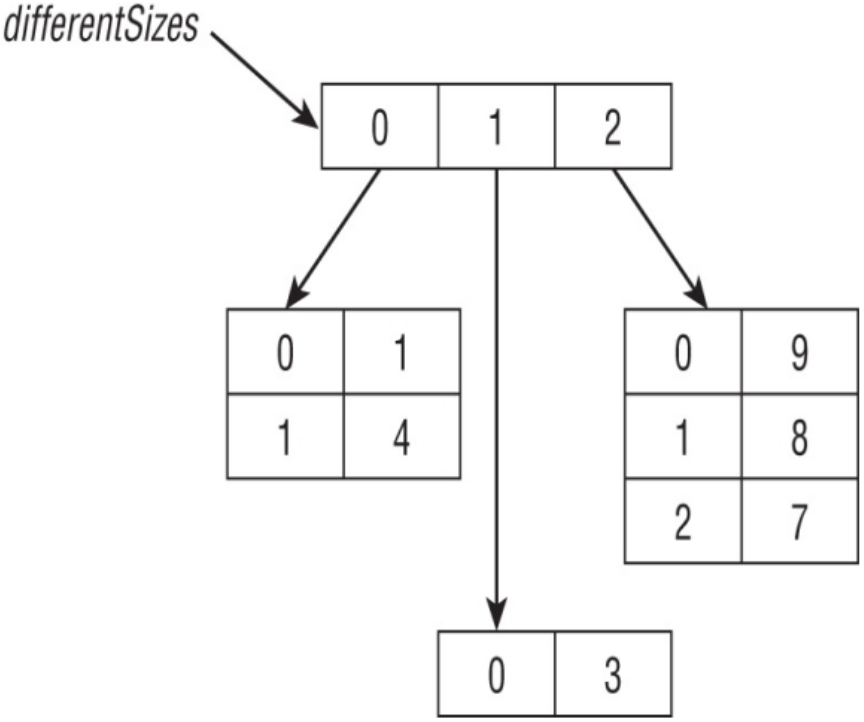
Bây giờ giả sử chúng ta set một trong các giá trị sau:

```
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; }
```

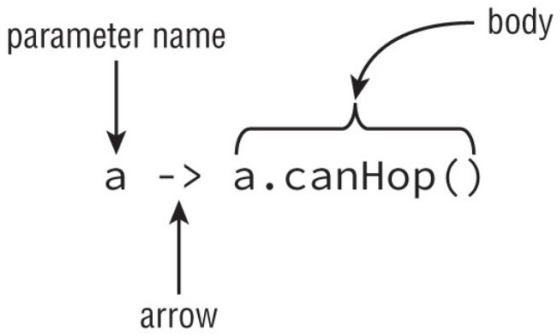
```
}


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 abstract method. 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:

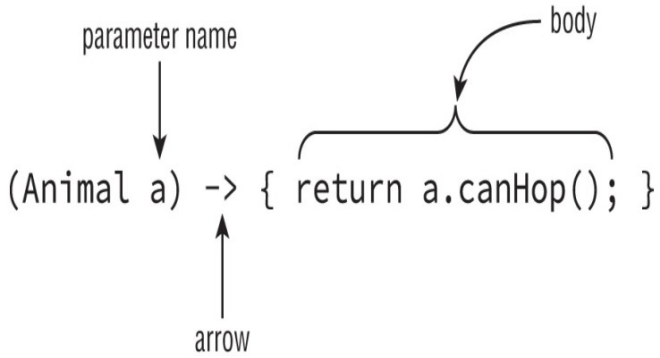
```
a -> a.canHop()
(Animal a) -> { return a.canHop(); }
```

- 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 abstract method. 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 abstract method 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 abstract method 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 abstract method 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 abstract method 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. Interface 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:

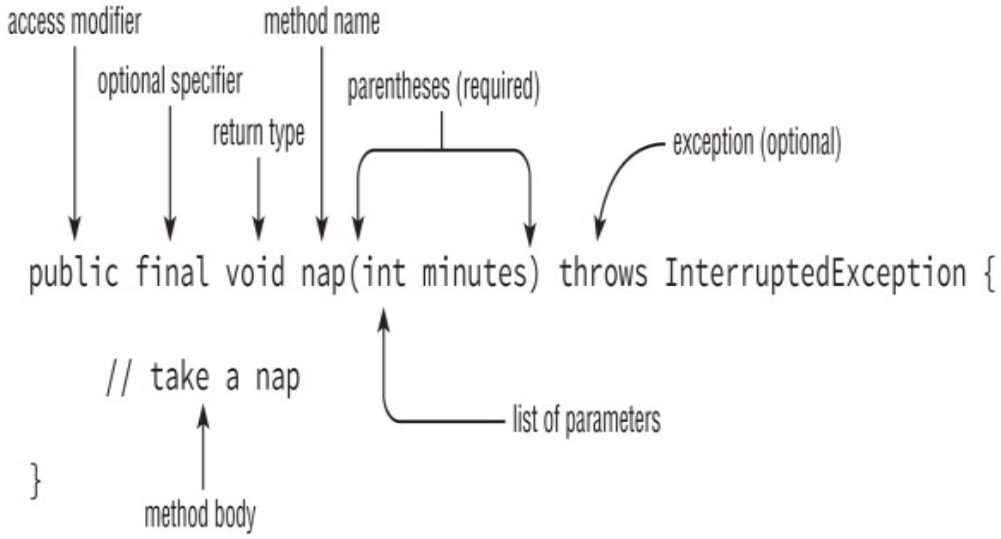
```
List<String> students = new ArrayList<>();
students.add("Anh");
students.add("Yến");
students.add("Thảo");
System.out.println(students); // [Anh, Yến, Thảo]
students.forEach(b -> System.out.println(b));
```

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

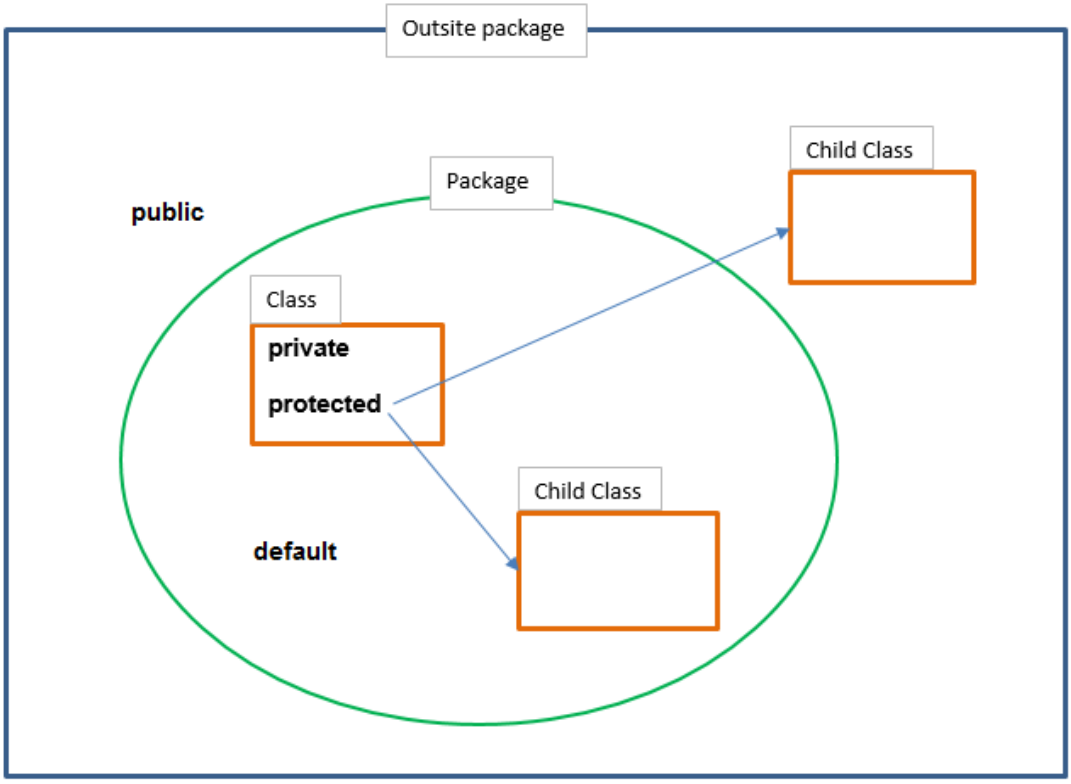
Java cung cấp bốn lựa chọn về access modifier:

- **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();
    }
}
```

b) Optional specifiers

Không giống như access modifier, bạn có thể có nhiều specifier trong cùng một phương thức (mặc dù không phải tất cả các kết hợp đều hợp lệ). Khi điều này xảy ra, bạn có thể chỉ định chúng theo bất kỳ thứ tự nào.

- **static:** được sử dụng cho các phương thức lớp và sẽ được đề cập ở phần sau của chương này.
- **abstract:** được sử dụng khi phần thân phương thức không được cung cấp.
- **final**: Công cụ sửa đổi cuối cùng được sử dụng khi một phương thức không được phép override bởi một lớp con.
- **synchronized**: được sử dụng với code đa luồng.
- **native** được sử dụng khi tương tác với mã được viết bằng ngôn ngữ khác, chẳng hạn C++.
- **strictfp:** được sử dụng để thực hiện các phép tính dấu phẩy động.

Một lần nữa, bây giờ chỉ tập trung vào cú pháp. Tại sao những thứ này biên dịch hoặc không biên dịch không?

```
public void walk1() {}
public final void walk2() {}
public static final void walk3() {}
public final static void walk4() {}
public modifier void walk5() {} // DOES NOT COMPILE
public void final walk6() {} // DOES NOT COMPILE
final public void walk7() {}
```

⇨ Phương thức walk7() được biên dịch. Java cho phép các optional specifier xuất hiện trước access modifier.

c) Return type

Kiểu trả về có thể là Java type thực tế như String hoặc int. Nếu không có kiểu trả về thì từ khóa void sẽ được sử dụng. Kiểu trả về đặc biệt này xuất phát từ tiếng Anh: void có nghĩa là không có nội dung. Trong Java, không có kiểu dữ liệu ở đó

Khi kiểm tra kiểu trả về, bạn cũng phải nhìn vào bên trong thân phương thức. Các phương thức có kiểu trả về khác với void bắt buộc phải có câu lệnh trả về bên trong body phương thức. Câu lệnh trả về này phải bao gồm primitive hoặc object được trả về. Các phương thức có kiểu trả về là void được phép có câu lệnh trả về không có giá trị trả về hoặc bỏ qua câu lệnh return.

Bạn có thể giải thích tại sao các phương pháp này biên dịch hay không?

```
public void walk1() {}
public void walk2() { return; }
public String walk3() { return ""; }
public String walk4() {} // DOES NOT COMPILE
public walk5() {} // DOES NOT COMPILE
public String int walk6() { } // DOES NOT COMPILE
String walk7(int a) { if (a == 4) return ""; } // DOES NOT COMPILE
```

Khi trả về một giá trị, nó cần được gán cho kiểu trả về. Hãy tưởng tượng có một biến local thuộc loại đó được gán trước khi được trả về. Bạn có thể nghĩ cách thêm một dòng mã với biến local trong hai phương thức này không?

```
int integer() {
    return 9;
}

int longMethod() {
    return 9L; // DOES NOT COMPILE
}
```

d) Method name

Hãy nhớ lại chương 2, identifier chỉ có thể chứa các chữ cái, số, $ hoặc _. Ngoài ra, **ký tự đầu tiên** không được phép là **số** và các **reserved word** không được phép. Cuối cùng, ký tự gạch dưới đơn không được phép. Theo quy ước, các phương thức bắt đầu bằng một chữ cái viết thường nhưng không bắt buộc.

```
public void walk1() {}public void 2walk() {} // DOES NOT COMPILE
public walk3 void() {} // DOES NOT COMPILE
public void Walk_$() {}
public _() {} // DOES NOT COMPILE
public void() {} // DOES NOT COMPILE
```

e) Parameter list

Mặc dù danh sách tham số là bắt buộc nhưng nó không nhất thiết phải chứa bất kỳ tham số nào. Điều này có nghĩa là bạn chỉ có thể có một cặp dấu ngoặc đơn trống sau tên phương thức, như sau:

```
void nap(){}
```

Nếu bạn có nhiều tham số, bạn phân tách chúng bằng dấu phẩy. Bây giờ, chúng ta hãy thực hành khai báo phương thức với các tham số "thông thường":

```
public void walk1() {}
public void walk2 {} // DOES NOT COMPILE
public void walk3(int a) {}
public void walk4(int a; int b) {} // DOES NOT COMPILE
public void walk5(int a, int b) {}
```

f) Optional exception list

Trong Java, mã có thể chỉ ra rằng đã xảy ra lỗi bằng cách đưa ra một exception. Exception là tùy chọn và nó sẽ nằm ở đâu trong phần khai báo phương thức nếu có. Ví dụ: InterruptedException là một loại Exception. Bạn có thể liệt kê bao nhiêu loại exception tùy thích trong mệnh đề này, được phân tách bằng dấu phẩy. Đây là một ví dụ:

```
public void zeroExceptions() {}
public void oneException() throws IllegalArgumentException {}
public void twoExceptions() throws IllegalArgumentException, InterruptedException {}
```

g) Method body

Phần cuối cùng của khai báo phương thức là nội dung phương thức (ngoại trừ các abstract methods và interfaces). Phần thân phương thức chỉ đơn giản là một code block. Nó có các dấu ngoặc nhọn chứa 0 hoặc nhiều câu lệnh Java.

Tại sao những câu lệnh này biên dịch hoặc không biên dịch:

```
public void walk1() {}
public void walk2() // DOES NOT COMPILE
public void walk3(int a) { int name = 5; }
```

⇨ Phương thức walk2() không biên dịch được vì nó thiếu dấu ngoặc nhọn xung quanh phần thân phương thức trống.

2. Working with Varargs

Tham số varargs phải là thành phần cuối cùng trong danh sách tham số của method. Điều này có nghĩa là bạn chỉ được phép có một tham số varargs cho mỗi phương thức. Bạn có thể xác định lý do tại sao mỗi thứ này biên dịch hoặc không biên dịch không?

```
public void walk1(int... nums) {}
public void walk2(int start, int... nums) {}
public void walk3(int... nums, int start) {} // DOES NOT COMPILE
public void walk4(int... start, int... nums) {} // DOES NOT COMPILE
```

Phương thức walk1() là một khai báo hợp lệ với một tham số varargs. Phương thức walk2() là một khai báo hợp lệ với một tham số int và một tham số varargs. Các phương thức walk3() và walk4() không biên dịch vì chúng có tham số varargs ở vị trí không phải là vị trí cuối cùng.

Khi gọi một phương thức có tham số varargs, bạn có một lựa chọn. Bạn có thể truyền vào một mảng hoặc liệt kê các phần tử của mảng và để Java tạo nó cho bạn. Bạn thậm chí có thể bỏ qua các giá trị varargs trong lệnh gọi phương thức và Java sẽ tạo một mảng có độ dài bằng 0 cho bạn.

Tại sao mỗi cuộc gọi phương thức lại đưa ra kết quả như thế nào không?

```
15:     public static void walk(int start, int... nums) {
16:         System.out.println(nums.length);
17:     }
18:     public static void main(String[] args) {
19:         walk(1); // 0
20:         walk(1, 2); // 1
21:         walk(1, 2, 3); // 2
22:         walk(1, new int[] {4, 5}); // 2
23:     }
```

Bạn đã thấy rằng Java sẽ tạo một mảng trống nếu không có tham số nào được truyền cho một vararg. Tuy nhiên, vẫn có thể truyền null một cách rõ ràng:

```
walk(1, null); // throws a NullPointerException in walk()
```

Truy cập tham số varargs cũng giống như truy cập một mảng. Nó sử dụng index của mảng. Đây là một ví dụ:

```
16:     public static void run(int... nums) {
17:         System.out.println(nums[1]);
18:     }
19:     public static void main(String[] args) {
20:         run(11, 22); // 22
21:     }
```

3. Applying the static Keyword

a) Designing static methods and fields

Ngoại trừ phương thức main(), chúng ta đang xem xét các phương thức instance, các phương thức tĩnh không yêu cầu một thể hiện của class. Chúng được chia sẻ giữa tất cả người dùng của class. Bạn có thể coi biến static là thành viên của một đối tượng class duy nhất tồn tại độc lập với bất kỳ instance nào của class đó.

Phương thức main() là phương thức static. Điều đó có nghĩa là bạn có thể gọi nó bằng tên class:

```
public class Koala {
    public static int count = 0; // static variable
    public static void main(String[] args) { // static method
        System.out.println(count);
    }
}
```

Về cơ bản, JVM gọi Koala.main() để bắt đầu chương trình.

Ngoài các phương thức main(), các phương thức static còn có hai mục đích chính:

- Dành cho các phương thức utility hay helper không yêu cầu bất kỳ trạng thái đối tượng nào. Vì không cần truy cập vào các biến instance, nên việc có các phương thức static sẽ loại bỏ nhu cầu người gọi khởi tạo một đối tượng chỉ để gọi phương thức đó.
- Dành cho trạng thái được chia sẻ bởi tất cả các instance của một class, như bộ đếm. Tất cả các instance phải chia sẻ cùng một trạng thái. Các phương thức chỉ sử dụng trạng thái đó cũng phải ở trạng thái static.

b) Accessing a static variable or method

Thông thường, việc truy cập một thành viên static như count rất dễ dàng. Bạn chỉ cần đặt tên class trước phương thức hoặc biến và thế là xong. Đây là một ví dụ:

```
System.out.println(Koala.count);
Koala.main(new String[0]);
```

Có một quy tắc phức tạp hơn. Bạn có thể sử dụng một instance của đối tượng để gọi một phương thức static. Trình biên dịch kiểm tra kiểu của reference và sử dụng loại tham chiếu đó thay vì đối tượng. Mã này là hoàn toàn hợp leej:

```
5:      Koala k = new Koala();
6:      System.out.println(k.count); // k is a Koala
7:      k = null;
8:      System.out.println(k.count); // k is still a Koala
```

Dòng 6 thấy k là Koala và count là biến static nên nó đọc biến static đó. Dòng 8 cũng làm tương tự. Java không quan tâm đến việc k có giá trị null. Vì chúng ta đang tìm kiếm biến static nên điều đó không thành vấn đề.

Một lần nữa vì điều này thực sự quan trọng: kết quả sau đây là gì?

```
Koala.count = 4;
Koala koala1 = new Koala();
Koala koala2 = new Koala();
koala1.count = 6;
koala2.count = 5;
```

⇨ 5

c) Static vs. Instance

Một thành viên static không thể gọi một thành viên instance mà không tham chiếu đến một instance của lớp.

Một phương thức static hoặc phương thức instance có thể gọi một phương thức static vì các phương thức statickhông yêu cầu một đối tượng để sử dụng. Chỉ một phương thức instance mới có thể gọi một phương thức instance khác trên cùng một lớp mà không cần sử dụng biến reference, bởi vì các phương thức instance yêu cầu một đối tượng. Logic tương tự áp dụng cho các biến instance và biến static.

```
public class Giraffe {
    public void eat(Giraffe g) {}
    public void drink() {};
    public static void allGiraffeGoHome(Giraffe g) {}
    public static void allGiraffeComeOut() {}
}
```

Static vs. instance calls

Type	Calling	Legal?
allGiraffeGoHome()	allGiraffeComeOut()	Yes
allGiraffeGoHome()	drink()	No
allGiraffeGoHome()	g.eat()	Yes
eat()	allGiraffeComeOut()	Yes
eat()	drink()	Yes
eat()	g.eat()	Yes

d) Static variables

Một số biến static có nghĩa là thay đổi khi chương trình chạy. Counters là một ví dụ phổ biến về điều này. Chúng tôi muốn số lượng tăng lên theo thời gian. Cũng giống như các biến instance, bạn có thể khởi tạo một biến static trên dòng được khai báo:

```
public class Initializers {
    private static int counter = 0; // initialization
}
```

Loại biến static khác có nghĩa là không bao giờ thay đổi trong suốt chương trình. Loại biến này được gọi là constant. Nó sử dụng **final modifier** để đảm bảo biến không bao giờ thay

đổi. Các hằng số sử dụng **modifier static final** và quy ước đặt tên khác với các biến khác. Họ sử dụng tất cả các chữ cái viết hoa có dấu gạch dưới giữa các "từ". Đây là một ví dụ:

```
public class Initializers {
    private static final int NUM_BUCKETS = 45;
    public static void main(String[] args) {
        NUM_BUCKETS = 5; // DOES NOT COMPILE
    }
}
```

Trình biên dịch sẽ đảm bảo rằng bạn không vô tình cố cập nhật biến final. Điều này có thể trở nên thú vị. Bạn có nghĩ rằng các biên dịch sau đây?

```
    private static final ArrayList<String> values = new ArrayList<>();

    public static void main(String[] args) {
        values.add("changed");
    }
```

Nó thực sự biên dịch vì giá trị là một biến tham chiếu (reference). Chúng ta được phép gọi các phương thức trên các biến reference. Tất cả những gì trình biên dịch có thể làm là kiểm tra xem chúng ta có cố gắng gán lại các giá trị cuối cùng để trỏ đến một đối tượng khác không.

e) Static initialization

Họ thêm từ khóa static để chỉ định chúng sẽ được chạy khi class được tải lần đầu tiên. Đây là một ví dụ:

```
private static final int NUM_SECONDS_PER_MINUTE;
private static final int NUM_MINUTES_PER_HOUR;
private static final int NUM_SECONDS_PER_HOUR;
static {
    NUM_SECONDS_PER_MINUTE = 60;NUM_MINUTES_PER_HOUR = 60;
}
static {
    NUM_SECONDS_PER_HOUR = NUM_SECONDS_PER_MINUTE * NUM_MINUTES_PER_HOUR;
}
```

Tất cả các static initializer đều chạy khi lớp được sử dụng lần đầu theo thứ tự chúng được xác định. Các câu lệnh trong đó chạy và gán bất kỳ biến static nào nếu cần. Có điều gì đó thú vị về ví dụ này. Chúng tôi vừa mới nói rằng các biến final không được phép gán lại. Chìa khóa ở đây là trình khởi tạo static là nhiệm vụ đầu tiên. Và vì nó xảy ra phía trước nên không sao cả.

Hãy thử một ví dụ khác để đảm bảo bạn hiểu được sự khác biệt:

```
14:      private static int one;
15:      private static final int two;
16:      private static final int three = 3;
17:      private static final int four; // DOES NOT COMPILE
18:      static {
19:          one = 1;
20:          two = 2;
21:          three = 3; // DOES NOT COMPILE
22:          two = 4; // DOES NOT COMPILE
23:      }
```

f) Static imports

Chúng ta có thể import một class cụ thể hoặc tất cả các class trong một package:

```
import java.util.ArrayList;
import java.util.*;
```

Chúng ta có thể sử dụng cách này để import hai lớp:

```
import java.util.List;
import java.util.Arrays;
public class Imports {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("one", "two");
    }
}
```

Static import là để **import** các thành viên static của class. Cũng giống như import thông thường, bạn có thể sử dụng ký tự **wildcard** hoặc nhập một thành viên cụ thể. Ý tưởng là bạn không cần phải chỉ định từng phương thức hoặc biến static đến từ đâu mỗi khi bạn sử dụng nó. Một ví dụ về thời điểm hoạt động **static import** phát huy hiệu quả là khi bạn đang đề cập đến nhiều hằng số trong một class khác:

Ví dụ bên trên có một lệnh gọi phương thức static: Arrays.asList. Viết lại mã để sử dụng tính năng **static import** mang lại kết quả như sau:

```
import java.util.List;
import static java.util.Arrays.asList; // static import
public class StaticImports {
    public static void main(String[] args) {
        List<String> list = asList("one", "two"); // no Arrays.
    }
}
```

Trong ví dụ này, chúng tôi đang nhập cụ thể phương thức asList. Điều này có nghĩa là bất cứ khi nào chúng ta đề cập đến asList trong lớp, nó sẽ gọi Arrays.asList().

Một trường hợp thú vị là điều gì sẽ xảy ra nếu chúng ta tạo một phương thức asList trong lớp StaticImports. Java sẽ ưu tiên nó hơn phương thức đã import và phương thức chúng tôi đã code sẽ được sử dụng.

```

import static java.util.Arrays; // DOES NOT COMPILE
import static java.util.Arrays.asList;
static import java.util.Arrays.*; // DOES NOT COMPILE
public class BadStaticImports {
    public static void main(String[] args) {
        Arrays.asList("one"); // DOES NOT COMPILE
    }
}
```

Dòng 1 cố gắng sử dụng tính năng import static để import một lớp. Hãy nhớ rằng import static chỉ dành cho import thành viên static. import thông thường là để nhập một class. Dòng 3 thử xem bạn có chú ý đến thứ tự **keywords** hay không. Cú pháp là import static chứ không phải ngược lại. Dòng 6 là không hợp lệ.

Ví dụ sau kết quả là gì:

```
import java.util.List;

import static java.util.Arrays.asList;
class BadStaticImports {

    static List asList(String tmp){
        System.out.println("1995Masr");
        return null;
    }

    public static void main(String[] args) {
        asList("one"); // DOES NOT COMPILE
    }
}
```

⇨ 1995Mars

Việc import hai class có cùng tên sẽ gây ra lỗi trình biên dịch. Điều này cũng đúng với import static.

```
import static statics.A.TYPE;
import static statics.B.TYPE; // DOES NOT COMPILE
```

4. Passing Data among Methods

Java là ngôn ngữ "**pass-by-value**". Điều này có nghĩa là một bản sao của biến được tạo và phương thức nhận được bản sao đó. Hãy xem một ví dụ:

```
2:      public static void main(String[] args) {
3:          int num = 4;
4:          newNumber(num);
5:          System.out.println(num); // 4
6:      }
7:      public static void newNumber(int num) {
8:          num = 8;
9:      }
```

⇨ Output: 4

Bây giờ bạn đã thấy các kiểu nguyên thủy, hãy thử một ví dụ với kiểu reference type. Bạn nghĩ kết quả của đoạn mã sau là gì?

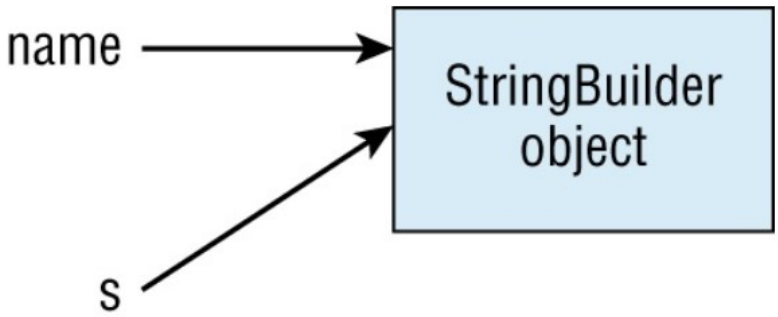
```
public static void main(String[] args) {
    String name = "Webby";
    speak(name);
    System.out.println(name);
}
public static void speak(String name) {
    name = "Sparky";
}
```

⇨ **Webby.** Giống như trong ví dụ cơ bản, việc gán biến chỉ dành cho tham số của phương thức và không ảnh hưởng đến người gọi.

Ví dụ: đây là gọi một phương thức trên StringBuilder được truyền vào phương thức đó:

```
public static void main(String[] args) {
    StringBuilder name = new StringBuilder();
    speak(name);
    System.out.println(name); // Webby
}
public static void speak(StringBuilder s) {
    s.append("Webby");
}
```

Trong trường hợp này, output là **Webby** vì phương thức này chỉ gọi một phương thức trên tham số. Nó không gán lại tên cho một đối tượng khác.



Bạn có thể thấy giá trị truyền theo vẫn được sử dụng như thế nào. Biến s là bản sao của biến name. Cả hai đều trỏ đến cùng một StringBuilder, có nghĩa là những thay đổi được thực hiện đối với StringBuilder đều có sẵn cho cả hai tham chiếu.

Java sử dụng **pass-by-value** theo từng giá trị để lấy dữ liệu vào một phương thức. Việc gán một tham số gốc hoặc tham chiếu mới cho một tham số không làm thay đổi người gọi. Các phương thức gọi trên một tham chiếu đến một đối tượng có thể ảnh hưởng đến người gọi. Lấy lại dữ liệu từ một phương thức dễ dàng hơn. Một bản sao được tạo từ primitive hoặc reference và được trả về từ phương thức.

5. Overloading Methods

Overloading method xảy ra khi các phương thức có cùng tên nhưng chữ ký phương thức khác nhau, có nghĩa là chúng khác nhau về các tham số của phương thức. **Overloading** cũng cho phép số lượng tham số khác nhau. Mọi thứ khác ngoài tên phương thức có thể khác nhau đối với các phương thức nạp chồng (overloading method). Điều này có nghĩa là có thể có sự khác nhau về access modifiers, specifiers (giống static), return types, và exception lists.

Đây đều là các phương thức overloading hợp lệ:

```
    public void fly(int numMiles) {}
    public void fly(short numFeet) {}
    public boolean fly() { return false; }
    void fly(int numMiles, short numFeet) {}
    public void fly(short numFeet, int numMiles) throws Exception {}
```

Như bạn có thể thấy, chúng ta có thể overloading bằng cách thay đổi bất cứ thứ gì trong danh sách tham số. Chúng ta có thể có một kiểu khác, nhiều kiểu dữ liệu hơn hoặc cùng một kiểu

nhưng theo thứ tự khác. Cũng lưu ý rằng kiểu trả về, access modifier, và exception list không liên quan đến overloading. Bây giờ hãy xem một ví dụ về overloading không hợp lệ:

```
    public void fly(int numMiles) {}
    public int fly(int numMiles) {} // DOES NOT COMPILE
```

Phương thức này không biên dịch vì nó chỉ khác với phương thức ban đầu ở kiểu trả về. Các danh sách tham số giống nhau, vì vậy chúng là các phương thức trùng lặp đối với Java. Còn hai method này thì sao? Tại sao thứ hai không biên dịch?

```
   public void fly(int numMiles) {}
   public static void fly(int numMiles) {} // DOES NOT COMPILE
```

Một lần nữa, danh sách tham số là như nhau. Bạn không thể có các phương thức mà điểm khác biệt duy nhất là một phương thức là instance method và một phương thức là static method.

Ví dụ: hãy xem hai phương pháp sau:

```
public void fly(int numMiles) {
    System.out.println("int");
}

public void fly(short numFeet) {
    System.out.println("short");
}
```

Khi gọi fly((short) 1) in short. Nó tìm kiếm các kiểu phù hợp và gọi phương thức thích hợp. Tất nhiên, nó có thể phức tạp hơn thế này.

a) Varargs - biến thể

Bạn nghĩ phương thức nào sẽ được gọi nếu chúng ta truyền int[]?

```
public void fly(int[] lengths) {}
public void fly(int... lengths) {} // DOES NOT COMPILE
```

Hãy nhớ rằng Java xử lý các biến thể như thể chúng là một mảng. Điều này có nghĩa là signature phương thức giống nhau cho cả hai phương thức. Mặc dù mã trông không giống nhau nhưng nó vẫn biên dịch thành cùng một danh sách tham số.

Không có gì ngạc nhiên khi bạn có thể gọi một trong hai phương thức bằng cách truyền một mảng:

```
fly(new int[] { 1, 2, 3 });
```

Tuy nhiên, bạn chỉ có thể gọi phiên bản varargs với các tham số độc lập:

```
fly(1, 2, 3);
```

b) Autoboxing

Java sẽ chuyển đổi một int nguyên thủy thành một số nguyên đối tượng để thêm nó vào ArrayList thông qua tính năng **autoboxing**. Điều này cũng hoạt động với code sau:

```
public void fly(Integer numMiles) {}
```

Điều này có nghĩa là gọi fly(3) sẽ gọi phương thức trước đó như mong đợi. Tuy nhiên, điều gì sẽ xảy ra nếu bạn có cả phiên bản nguyên thủy và phiên bản integer?

```
public void fly(int numMiles) {}
public void fly(Integer numMiles) {}
```

Java sẽ gọi int numMiles. Java cố gắng sử dụng danh sách tham số cụ thể nhất mà nó có thể tìm thấy. Khi phiên bản int nguyên thủy không có, nó sẽ tự động chuyển sang chế độ

autoboxing. Tuy nhiên, khi phiên bản int nguyên thủy được cung cấp, không có lý do gì để Java thực hiện thêm công việc **autoboxing**.

c) Reference types

Đưa ra quy tắc về việc Java chọn phiên bản cụ thể nhất của một phương thức mà nó có thể, bạn nghĩ đoạn mã này sẽ đưa ra kết quả gì?

```
public class ReferenceTypes {
    public void fly(String s) {
        System.out.print("string");
    }

    public void fly(Object o) {
        System.out.print("object");
    }

    public static void main(String[] args) {
        ReferenceTypes r = new ReferenceTypes();
        r.fly("test");
        System.out.print("-");
        r.fly(56);
    }
}
```

Hãy thử một cái khác. Cái này in cái gì?

```
public static void print(Iterable i) {
    System.out.print("I");
}
public static void print(CharSequence c) {
    System.out.print("C");
}
public static void print(Object o) {
    System.out.print("O");
}
public static void main(String[] args){
    print("abc");
    print(new ArrayList<>());
    print(LocalDate.of(2019, Month.JULY, 4));
}
```

⇨ CIO

d) Primitives

Primitive hoạt động theo cách tương tự như các biến reference. Java cố gắng tìm ra phương thức overloading phù hợp cụ thể nhất. Bạn nghĩ điều gì sẽ xảy ra ở đây?

```
public class Plane {
    public void fly(int i) {
        System.out.print("int");
    }

    public void fly(long l) {
        System.out.print("long");
    }

    public static void main(String[] args) {
        Plane p = new Plane();
        p.fly(123);
        System.out.print("-");
        p.fly(123L);
    }
}
```

⇨ int-long

Lưu ý rằng Java chỉ có thể chấp nhận các kiểu rộng hơn (widing). Một int có thể được truyền cho một phương thức lấy tham số long. Java sẽ không tự động chuyển đổi sang kiểu hẹp hơn. Nếu bạn muốn truyền long cho một phương thức lấy tham số int, bạn phải thêm một ép kiểu để nói rõ ràng việc thu hẹp (narrowing) là được.

e) Generics

Bạn có thể ngạc nhiên khi biết rằng đây không phải là tình trạng overloading hợp lệ:

```
public void walk(List<String> strings) {}
public void walk(List<Integer> integers) {} // DOES NOT COMPILE
```

Java có một khái niệm gọi là xóa kiểu trong đó các tổng quát chỉ được sử dụng tại thời điểm biên dịch. Điều đó có nghĩa là mã được biên dịch trông như thế này:

```
public void walk(List strings) {}
public void walk(List integers) {} // DOES NOT COMPILE
```

Rõ ràng chúng ta không thể có hai phương thức có cùng chữ ký phương thức, vì vậy phương thức này không được biên dịch. Hãy nhớ rằng overloading phương thức phải khác nhau ở ít nhất một trong các tham số của phương thức.

f) Arrays

Không giống như ví dụ trước, mã này vẫn ổn:

```
public static void walk(int[] ints) {}
public static void walk(Integer[] integers) {}
```

Mảng đã xuất hiện từ thời đầu của Java. Họ chỉ định loại thực tế của họ và không tham gia vào việc xóa kiểu.

g) Putting it all together

Tất cả các quy tắc khi gọi một phương thức overloading phải hợp lý. Java gọi phương thức cụ thể nhất có thể. Khi một số loại tương tác, các quy tắc Java tập trung vào khả năng tương thích ngược. Cách đây rất lâu, autoboxing và varargs không tồn tại. Vì mã cũ vẫn cần hoạt động, điều này có nghĩa là tính năng autoboxing và các varargs sẽ xuất hiện sau cùng khi Java xem xét các phương thức bị overloading.

Thứ tự mà Java sử dụng để chọn phương thức overloading phù hợp:

Rule	**Example of what will be chosen for glide(1,2)**
Exact match by type	String glide(int i, int j)
Larger primitive type	String glide(long i, long j)
Autoboxed type	String glide(Integer i, Integer j)
Varargs	String glide(int... nums)

6. Encapsulating Data

Bạn đã thấy một ví dụ về một class có trường không phải là private:

```
public class Swan {
    int numberEggs; // instance variable
}
```

Vì có default (package-private), điều đó có nghĩa là bất kỳ lớp nào trong package đều có thể đặt numberEggs. Chúng tôi không còn quyền kiểm soát những gì được thiết lập trong class. Người gọi thậm chí có thể viết điều này:

```
mother.numberEggs = -1;
```

Encapsulation - Đóng gói có nghĩa là chỉ các phương thức trong class có các biến mới có thể tham chiếu đến các biến instance. Người gọi được yêu cầu sử dụng các phương pháp này. Chúng ta hãy xem class Swan mới được đóng gói:

```
1:  public class Swan {
2:      private int numberEggs; // private
3:      public int getNumberEggs() { // getter
4:          return numberEggs;
5:      }
6:  public void setNumberEggs(int newNumber) { // setter
7:      if (newNumber >= 0) // guardcondition
8:      numberEggs = newNumber;
9:  } }
```

Để đóng gói, hãy nhớ rằng dữ liệu (một biến instance) là private và getters/setters là public. Java định nghĩa quy ước đặt tên cho getters và setters được liệt kê dưới đây:

Rule	**Example**

Các phương thức Getter thường bắt đầu bằng **is** nếu thuộc tính là boolean.	public boolean isHappy() {
	  return happy;
	}

Các phương thức Getter bắt đầu bằng **get** nếu thuộc tính không phải là boolean.	public int getNumberEggs() {
	  return numberEggs;
	}

Các phương thức setter bắt đầu bằng set.	public void setHappy(boolean _happy) {
	  happy = _happy;
	}

Dòng nào tuân theo các quy ước đặt tên sau không:

```
12:     private boolean playing;
13:     private String name;
14:     public boolean isPlaying() { return playing; }
15:     public String name() { return name; }
16:     public void updateName(String n) { name = n; }
17:     public void setName(String n) { name = n; }
```

⇨ 14, 17

Để dữ liệu được đóng gói, bạn không cần phải cung cấp getters và setters. Miễn là các biến thể hiện là private, vẫn ổn. Ví dụ: đây là một class được đóng gói tốt:

```
public class Swan {
    private int numEggs;
    public void layEgg() {
        numEggs++;
    }
    public void printEggCount() {
        System.out.println(numEggs);
    }
}
```

Chapter 8: Class Design

1. Understanding inheritance

Khi tạo một class mới trong Java, bạn có thể định nghĩa class đó là kế thừa từ một class hiện có. Kế thừa là quá trình trong đó một subclass tự động bao gồm bất kỳ thành viên **public** hoặc **protected** nào của lớp, bao gồm các primitives, objects, hay methods, được xác định trong parent class.

Khi một class kế thừa từ parent class, tất cả các thành viên public và protected đều tự động có sẵn như một phần của child class. Package-private của package sẽ khả dụng nếu child class nằm trong cùng package với parent class.

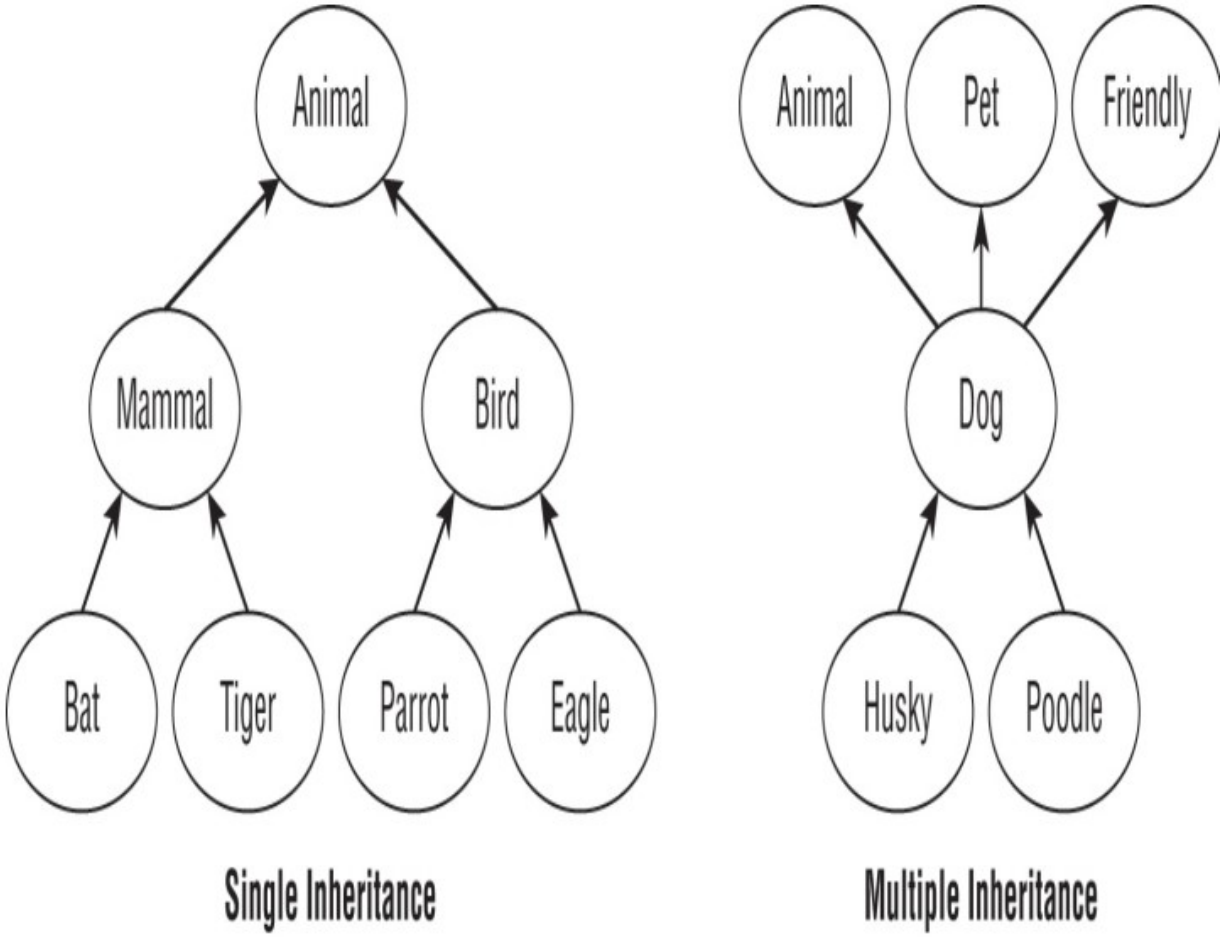
Cuối cùng nhưng không kém phần quan trọng, các private bị giới hạn trong class mà chúng được định nghĩa và không bao giờ có sẵn thông qua tính kế thừa. Điều này không có nghĩa là parent class không có các thành viên riêng có thể chứa dữ liệu hoặc sửa đổi một đối tượng; nó chỉ có nghĩa là subclass không có tham chiếu trực tiếp đến chúng. Chúng ta hãy xem một ví dụ đơn giản với các class BigCat và Jaguar. Trong ví dụ này, Jaguar là subclass hoặc child của BigCat, khiến BigCat trở thành superclass hay parent class của Jaguar.

```
public class BigCat {
    public double size;
}

public class Jaguar extends BigCat {
    public Jaguar() {
        size = 10.2;
    }
    public void printDetails() {
        System.out.println(size);
    }
}
```

a) Single vs. Multiple inheritance

Java hỗ trợ kế thừa đơn, theo đó một class chỉ có thể kế thừa từ một parent class trực tiếp. Java cũng hỗ trợ đa kế thừa, theo đó một class có thể mở rộng class khác, từ đó mở rộng class khác. Bạn có thể có bất kỳ cấp độ kế thừa nào, cho phép mỗi con cháu có quyền truy cập vào các thành viên của tổ tiên nó.

Để thực sự hiểu được tính kế thừa đơn, có thể hữu ích nếu so sánh nó với tính đa kế thừa, theo đó một class có thể có nhiều cha mẹ trực tiếp. Theo thiết kế, Java không hỗ trợ đa kế thừa trong ngôn ngữ vì đa kế thừa có thể dẫn đến các mô hình dữ liệu phức tạp, thường khó bảo trì. Java cho phép một ngoại lệ đối với quy tắc kế thừa duy nhất mà bạn sẽ thấy interface.



Một phần nguyên nhân khiến việc đa kế thừa trở nên phức tạp là việc xác định giá trị từ cha mẹ nào sẽ kế thừa trong trường hợp xảy ra xung đột. Ví dụ: nếu bạn có một đối tượng hoặc phương thức được xác định trong tất cả các bậc cha mẹ, thì child class sẽ kế thừa cái nào? Đó là lý do tại sao Java tránh được những vấn đề này bằng cách không cho phép đa kế thừa hoàn toàn.

Trong Java có thể ngăn chặn việc mở rộng một class bằng cách đánh dấu class đó bằng **final**.

b) Inheriting object

Trong Java, tất cả các lớp đều kế thừa từ một class duy nhất: java.lang.Object hoặc gọi tắt là Object. Hơn nữa, Object là lớp duy nhất không có super class. Bạn có thể thắc mắc, "Không có class nào tôi viết cho đến nay đều extend Object, vậy làm thế nào để tất cả các class kế thừa từ nó?" Câu trả lời là trình biên dịch đã tự động chèn mã vào bất kỳ class nào bạn viết mà không mở rộng một lớp cụ thể. Ví dụ, hãy xem xét hai định nghĩa lớp tương đương sau:

```
public class Zoo { }
public class Zoo extends java.lang.Object { }
```

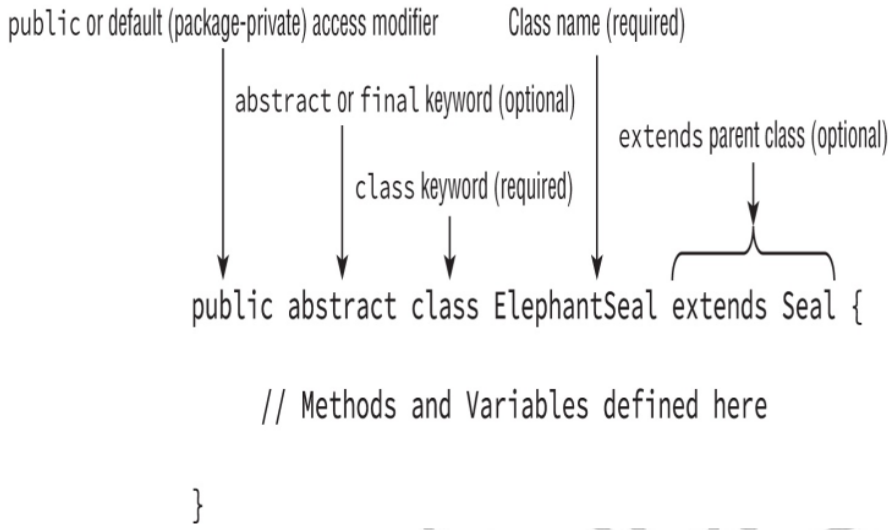
Khi Java thấy bạn định nghĩa một class không extend một class khác, nó sẽ tự động thêm cú pháp mở rộng java.lang.Object vào định nghĩa lớp. Kết quả là mọi class đều có quyền truy cập vào bất kỳ phương thức nào có thể truy cập được trong lớp Object. Ví dụ: các phương thức toString() và equals() có sẵn trong Object; do đó, chúng có thể truy cập được ở tất cả các class. Tuy nhiên, nếu không override trong một subclass, chúng có thể không đặc biệt hữu ích.

2. Creating classes

Bây giờ chúng ta đã thiết lập cách kế thừa trong Java, chúng ta có thể sử dụng nó để định nghĩa và tạo các mối quan hệ class phức tạp.

a) Extending a class

Cú pháp đầy đủ của việc xác định và mở rộng một lớp bằng từ khóa extend được hiển thị dưới đây:



Hãy tạo hai file là Animal.java và Lion.java, trong đó class Lion extend class Animal. Giả sử chúng nằm trong cùng một package thì không cần phải có câu lệnh import trong Lion.java để truy cập vào lớp Animal. Dưới đây là nội dung của Animal.java:

```
public class Animal {
    private int age;
    protected String name;
    public int getAge() {
        return age;}
    public void setAge(int newAge) {
        age = newAge;
    }
}
```

Và đây là nội dung của Lion.java:

```
public class Lion extends Animal {
    public void setProperties(int age, String n) {
        setAge(age);
        name = n;
    }
    public void roar() {
        System.out.print(name + ", age " + getAge() + ", says:
                Roar!");
    }
    public static void main(String[] args) {
        var lion = new Lion();
        lion.setProperties(3, "kion");
        lion.roar();
    }
}
```

Từ khóa extend được sử dụng để diễn tả rằng class Lion kế thừa lớp Animal. Khi được thực thi, chương trình Lion sẽ in ra kết quả sau:

```
kion, age 3, says: Roar!
```

Chúng ta hãy nhìn vào các thành viên của class Lion. Biến instance age được đánh dấu là private và không thể truy cập trực tiếp từ class Lion.

b) Accessing the this reference

Điều gì xảy ra khi một tham số phương thức có cùng tên với một biến instance hiện có? Chúng ta hãy xem một ví dụ. Bạn nghĩ chương trình sau in ra cái gì?

```
public class Flamingo {
    private String color;
    public void setColor(String color) {
        color = color;
    }
    public static void main(String... unused) {
        Flamingo f = new Flamingo();
        f.setColor("PINK");
        System.out.println(f.color);
    }
}
```

⇨ Sẽ là null. Java sử dụng phạm vi chi tiết nhất nên khi nhìn thấy color = color, nó cho rằng bạn đang gán giá trị tham số phương thức cho chính nó.

Cách khắc phục khi bạn có biến local có cùng tên với biến instance là sử dụng tham chiếu hoặc từ khóa **this**. Tham chiếu **this** đề cập đến phiên bản hiện tại của lớp và có thể được sử dụng để truy cập bất kỳ thành viên nào của class, bao gồm cả các thành viên được kế thừa. Nó có thể được sử dụng trong bất kỳ instance method, constructor, và instance initializer block. Nó không thể được sử dụng khi không có phiên bản tiềm ẩn của lớp, chẳng hạn như trong một phương thức static hoặc static initializer block.

Chúng tôi áp dụng điều này cho việc triển khai phương pháp trước đây của mình như sau:

```
public class Flamingo {
    private String color;
    public void setColor(String color) {
        this.color = color;
    }
    public static void main(String... unused) {
        Flamingo f = new Flamingo();
        f.setColor("PINK");
        System.out.println(f.color);
    }
}
```

⇨ PINK

c) Calling the *super* reference

Trong Java, một biến hoặc phương thức có thể được định nghĩa trong cả lớp cha và lớp con. Khi điều này xảy ra, làm cách nào để chúng tôi tham chiếu phiên bản trong lớp cha thay vì lớp hiện tại?

Để đạt được điều này, bạn có thể sử super. Tham chiếu super tương tự như tham chiếu this, ngoại trừ việc nó loại trừ bất kỳ thành viên nào được tìm thấy trong lớp hiện tại. Nói cách khác, thành viên phải có thể truy cập được thông qua tính kế thừa. Lớp sau đây cho thấy cách áp dụng super để sử dụng hai biến có cùng tên trong một phương thức:

```
class Mammal {
    String type = "mammal";
}
public class Bat extends Mammal {
    String type = "bat";
    public String getType() {
        return super.type + ":" + this.type;
    }
    public static void main(String... zoo) {
        System.out.print(new Bat().getType());
    }
}
```

⇨ mammal:bat

Bạn nghĩ điều gì sẽ xảy ra nếu tham chiếu super bị loại bỏ? Chương trình sau đó sẽ in **bat:bat**. Java sử dụng phạm vi hẹp nhất có thể—trong trường hợp này là biến **type** được xác định trong class Bat.

Vì **super** bao gồm các thành viên được kế thừa, nên bạn thường chỉ sử dụng super khi có xung đột đặt tên thông qua tính kế thừa. Ví dụ: bạn có một phương thức hoặc biến được xác định trong lớp hiện tại khớp với một phương thức hoặc biến trong lớp cha. Điều này thường xuất hiện trong override phương thức ẩn và biến ẩn.

3. Declaring constructors

a) Creating a constructor

Hãy bắt đầu với một constructor đơn giản:

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

Tên của **constructor**, Bunny, trùng với tên của lớp, Bunny, và không có kiểu trả về, thậm chí không có void. Điều đó làm cho cái này trở thành một constructor.

Một class có thể có nhiều constructor, miễn là mỗi constructor có một chữ ký duy nhất. Trong trường hợp này, điều đó có nghĩa là các tham số của constructor phải khác biệt. Giống như các phương thức có cùng tên nhưng khác nhau về chữ ký, việc khai báo nhiều constructor với các chữ ký khác nhau được gọi là ***constructor overloading***. Lớp Turtle sau đây có bốn constructor được overloading riêng biệt:

```
public class Turtle {
    private String name;
    public Turtle() {
        name = "John Doe";
    }
    public Turtle(int age) {}
    public Turtle(long age) {}
    public Turtle(String newName, String... favoriteFoods) {
        name = newName;
    }
}
```

Constructor được sử dụng khi tạo một đối tượng mới. Quá trình này được gọi là khởi tạo vì nó tạo ra một instance mới của lớp. Một constructor được gọi khi chúng ta viết new theo sau là tên của lớp mà chúng ta muốn khởi tạo. Đây là một ví dụ:

```
new Turtle()
```

b) Default constructor

Mỗi lớp trong Java đều có một constructor cho dù bạn có viết mã hay không. Nếu bạn không bao gồm bất kỳ constructor nào trong lớp, Java sẽ tạo một constructor cho bạn mà không có bất kỳ tham số nào. Constructor do Java tạo này được gọi là constructor mặc định và được thêm vào bất cứ khi nào một lớp được khai báo mà không có bất kỳ constructor nào. Chúng tôi thường gọi nó là constructor không có đối số mặc định cho rõ ràng. Đây là một ví dụ:

```
public class Rabbit {
    public static void main(String[] args) {
        Rabbit rabbit = new Rabbit(); // Calls default constructor
    }
}
```

Trong lớp Rabbit, Java nhận thấy không có constructor nào và tạo một constructor đó. Constructor mặc định này tương đương với việc viết cái này:

```
public Rabbit() {}
```

Hãy nhớ rằng constructor mặc định chỉ được cung cấp nếu không có constructor nào. Bạn nghĩ lớp nào trong số này có constructor mặc định?

```
public class Rabbit1 {}
public class Rabbit2 {
public Rabbit2() {}
}
public class Rabbit3 {
    public Rabbit3(boolean b) {}
}
public class Rabbit4 {
    private Rabbit4() {}
}
```

⇨ Chỉ Rabbit1 mới có constructor không có đối số mặc định.

Chúng ta hãy xem nhanh cách gọi các constructor này:

```
1:      public class RabbitsMultiply {
2:      public static void main(String[] args) {
3:      Rabbit1 r1 = new Rabbit1();
4:      Rabbit2 r2 = new Rabbit2();
5:      Rabbit3 r3 = new Rabbit3(true);
6:      Rabbit4 r4 = new Rabbit4(); // DOES NOT COMPILE
7:      } }
```

⇨ Dòng 6 không biên dịch. Rabbit4 đặt constructor ở chế độ private để các lớp khác không thể gọi nó.

c) Calling overloaded constructors with this()

Hãy nhớ rằng, một class có thể có nhiều constructor. Điều này được gọi là constructor overloading vì tất cả các constructor đều có cùng tên vốn có nhưng có chữ ký khác nhau. Chúng ta hãy xem xét vấn đề này chi tiết hơn bằng cách sử dụng class Hamster.

```
public class Hamster {
    private String color;
    private int weight;
    public Hamster(int weight) { // First constructor
        this.weight = weight;
        color = "brown";
    }
    public Hamster(int weight, String color) { // Second constructor
        this.weight = weight;
        this.color = color;
    }
}
```

Điều gì xảy ra khi viết:

```
public Hamster(int weight) {
    Hamster(weight, "brown"); // DOES NOT COMPILE
}
```

Điều này sẽ không hoạt động. Constructor chỉ có thể được gọi bằng cách viết new trước tên của constructor. Chúng không giống như các phương thức thông thường mà bạn có thể gọi. Điều gì xảy ra nếu chúng ta dán new trước tên constructor?

```
public Hamster(int weight) {
    new Hamster(weight, "brown"); // Compiles, but incorrect
}
```

⇨ Điều này biên dịch. Tuy nhiên, nó không làm được những gì chúng ta muốn.

Java cung cấp một giải pháp: this(). Khi this() được sử dụng với dấu ngoặc đơn, Java sẽ gọi một constructor khác trên cùng một instance của class.

```
public Hamster(int weight) {
    this(weight, "brown");
}
```

Gọi this() có một quy tắc đặc biệt bạn cần biết. Nếu bạn gọi nó, lệnh gọi this() phải là câu lệnh đầu tiên trong constructor.

```
3:  public Hamster(int weight) {
4:  System.out.println("in constructor");5: // Set weight and default color
6:  this(weight, "brown"); // DOES NOT COMPILE
7:  }
```

d) Calling parent constructors with *super()*

Trong Java, câu lệnh đầu tiên của mọi constructor là lệnh gọi đến constructor khác trong lớp bằng cách sử dụng this() hoặc lệnh gọi tới constructor trong lớp cha trực tiếp bằng cách sử dụng super().

Nếu constructor cha nhận các đối số thì lệnh gọi super() cũng nhận các đối số. Để đơn giản trong phần này, chúng tôi gọi lệnh super() như bất kỳ constructor gốc nào, ngay cả những hàm lấy đối số. Chúng ta hãy xem lớp Animal và lớp con Zebra của nó và xem các constructor của chúng có thể được viết chính xác như thế nào để gọi lẫn nhau:

```
public class Animal {
    private int age;
    public Animal(int age) {
        super(); // Refers to constructor in java.lang.Object
        this.age = age;
    }
}
public class Zebra extends Animal {
    public Zebra(int age) {
        super(age); // Refers to constructor in Animal
    }
    public Zebra() {
        this(4); // Refers to constructor in Zebra with int argument
    }
}
```

Giống như gọi this(), gọi super() chỉ có thể được sử dụng làm câu lệnh đầu tiên của constructor. Ví dụ: ba định nghĩa lớp sau sẽ không được biên dịch:

```
public class Zoo {
    public Zoo() {
        System.out.println("Zoo created");
        super(); // DOES NOT COMPILE
    }
}
public class Zoo {
    public Zoo() {
        super();
        System.out.println("Zoo created");
        super(); // DOES NOT COMPILE
    }
}
public class Zoo {
    public Zoo() {
        super();
        this(); // DOES NOT COMPILE
    }
}
```

Nếu lớp cha có nhiều hơn một constructor, lớp con có thể sử dụng bất kỳ constructor cha hợp lệ nào trong định nghĩa của nó, như trong ví dụ sau:

```
public class Animal {
    private int age;
    private String name;
    public Animal(int age, String name) {
        super();
        this.age = age;
        this.name = name;
    }
    public Animal(int age) {
        super();
        this.age = age;
        this.name = null;
    }
}
public class Gorilla extends Animal {
    public Gorilla(int age) {
        super(age,"Gorilla");
    }
    public Gorilla() {
        super(5);
    }
}
```

Understanding compiler enhancements

Chúng ta đã nói dòng đầu tiên của mọi constructor là lệnh gọi tới this() hoặc super(). Trình biên dịch Java sẽ tự động chèn lệnh gọi tới constructor không có đối số super() nếu bạn không gọi rõ ràng this() hoặc super() là dòng đầu tiên của constructor. Ví dụ, ba định nghĩa lớp và constructor sau đây là tương đương, vì trình biên dịch sẽ tự động chuyển đổi tất cả chúng sang ví dụ cuối cùng:

```
public class Donkey {}
public class Donkey {
    public Donkey() {}
}
public class Donkey {
    public Donkey() {
        super();
    }
}
```

Missing a Default No-Argument Constructor

Điều gì xảy ra nếu lớp cha không có constructor không có đối số? Hãy nhớ lại rằng constructor không có đối số mặc định là không bắt buộc và chỉ được trình biên dịch chèn vào nếu không có constructor nào được xác định trong lớp. Ví dụ, bạn có thấy tại sao khai báo lớp Elephant sau đây không biên dịch được không?

```
public class Mammal {
    public Mammal(int age) {}
}
public class Elephant extends Mammal { // DOES NOT COMPILE
}
```

Vì Elephant không định nghĩa bất kỳ constructor nào nên trình biên dịch Java sẽ cố gắng chèn một constructor không có đối số mặc định. Nó cũng sẽ tự động chèn lệnh gọi tới super() làm dòng đầu tiên của constructor không có đối số mặc định. Khai báo Elephant trước đó của chúng tôi sau đó được trình biên dịch chuyển đổi thành khai báo sau:

```
public class Elephant extends Mammal {
    public Elephant() {
      super(); // DOES NOT COMPILE
    }
}
```

Vì lớp Mammal có ít nhất một constructor được khai báo nên trình biên dịch không chèn constructor không có đối số mặc định. Do đó, lệnh gọi super() trong khai báo lớp Elephant không được biên dịch.

Code sau sẽ hoạt động:

```
public class Elephant extends Mammal {
    public Elephant() {
       super(5);
    }
}
```

e) Constructors and final fields

final static phải được gán một giá trị chính xác một lần. Bạn đã thấy điều này xảy ra trong dòng khai báo và trong static initializer. Các biến instance được đánh dấu final tuân theo các quy tắc tương tự. Chúng có thể được gán giá trị trong dòng mà chúng được khai báo hoặc trong instance initializer.

```
public class MouseHouse {
    private final int volume;
    private final String name = "The Mouse House";
    {
        volume = 10;
    }
}
```

Còn một nơi nữa mà chúng có thể được gán giá trị—constructor. Constructor là một phần của quá trình khởi tạo nên nó được phép gán các biến final instance trong đó.

```
public class MouseHouse {
    private final int volume;
    private final String type;
    public MouseHouse() {
        this.volume = 10;
        type = "happy";
    }
}
```

Các biến final instance phải được gán một giá trị. Giá trị mặc định không được sử dụng cho các biến này. Nếu chúng không được gán một giá trị trong dòng nơi chúng được khai báo hoặc trong instance initializer thì chúng phải được gán một giá trị trong phần khai báo constructor. Nếu không làm như vậy sẽ dẫn đến lỗi trình biên dịch trên dòng khai báo constructor

```
public class MouseHouse {
    private final int volume;
    private final String type;
    {
        this.volume = 10;
    }
    public MouseHouse(String type) {
        this.type = type;
    }
    public MouseHouse() { // DOES NOT COMPILE
        this.volume = 2; // DOES NOT COMPILE
    }
}
```

f) Order of initialization

Class Initialization

Trước tiên, bạn cần khởi tạo lớp, bao gồm việc gọi tất cả các thành viên static trong hệ thống phân cấp lớp, bắt đầu từ lớp cha cao nhất và đi xuống. Điều này thường được gọi là tải lớp. JVM kiểm soát thời điểm lớp được khởi tạo. Lớp có thể được khởi tạo khi chương trình khởi động lần đầu tiên, khi một thành viên static của lớp được tham chiếu hoặc ngay trước khi một instance của lớp được tạo. Quy tắc quan trọng nhất với việc khởi tạo lớp là nó xảy ra nhiều nhất một lần cho mỗi lớp. Lớp này cũng có thể không bao giờ được tải nếu nó không được sử dụng trong chương trình. Chúng tôi tóm tắt thứ tự khởi tạo cho một lớp như sau:

Initialize Class X

- Nếu có **superclass Y** của X thì hãy khởi tạo lớp **Y trước**.
- Xử lý tất cả các khai báo biến static theo thứ tự chúng xuất hiện trong lớp.
- Xử lý tất cả các trình khởi tạo staic theo thứ tự chúng xuất hiện trong lớp

Hãy xem một ví dụ, chương trình sau in ra cái gì?

```
public class Animal {
    static { System.out.print("A"); }
}
public class Hippo extends Animal {
    static { System.out.print("B"); }
    public static void main(String[] grass) {
        System.out.print("C");
        new Hippo();
        new Hippo();
        new Hippo();
    }
}
```

⇨ ABC

Instance Initialization

Một **instance** được khởi tạo bất cứ lúc nào từ khóa **new** được sử dụng. Việc khởi tạo **instance** phức tạp hơn một chút so với khởi tạo class, bởi vì một class hoặc superclass có thể có nhiều constructor được khai báo nhưng chỉ một số ít được sử dụng như một phần của quá trình khởi tạo **instance**.

Chúng tôi tóm tắt thứ tự khởi tạo cho một instance như sau:

Initialize Instance of X

- Nếu có superclass Y của X thì trước tiên hãy khởi tạo instance của Y.
- Xử lý tất cả các khai báo biến instance theo thứ tự chúng xuất hiện trong lớp.
- Xử lý tất cả các trình khởi tạo instance theo thứ tự chúng xuất hiện trong lớp.
- Khởi tạo constructor bao gồm mọi overloading constructor được tham chiếu bằng this()

```
public class ZooTickets {
    private String name = "BestZoo";

    {
        System.out.print(name + "-");
    }

    private static int COUNT = 0;

    static {
        System.out.print(COUNT + "-");
    }

    static {
        COUNT += 10;
        System.out.print(COUNT + "-");
    }

    public ZooTickets() {
        System.out.print("z-");
    }

    public static void main(String... patrons) {
        new ZooTickets();
    }
```

⇨ 0-10-BestZoo-z-

Tiếp theo, hãy thử một ví dụ đơn giản về tính kế thừa.

```
class Primate {
    public Primate() {
        System.out.print("Primate-");
    }
}

class Ape extends Primate {
    public Ape(int fur) {
        System.out.print("Ape1-");
    }

    public Ape() {
        System.out.print("Ape2-");
    }
}

public class Chimpanzee extends Ape {
    public Chimpanzee() {
        super(2);
        System.out.print("Chimpanzee-");
    }

    public static void main(String[] args) {
        new Chimpanzee();
    }
}
```

⇨ Primate-Ape1-Chimpanzee

Lưu ý rằng chỉ một trong hai constructor Ape() được gọi. Bạn cần bắt đầu bằng lệnh gọi new Chimpanzee() để xác định constructor nào sẽ được thực thi. Hãy nhớ rằng, các constructor được thực thi từ dưới lên, nhưng vì dòng đầu tiên của mỗi constructor là lệnh gọi đến một constructor khác nên luồng thực sự kết thúc bằng parent constructor được thực thi trước child constructor.

Ví dụ tiếp theo khó hơn một chút. Bạn nghĩ điều gì sẽ xảy ra ở đây?

```
1:  public class Cuttlefish {
2:      private String name = "swimmy";
3:      { System.out.println(name); }
4:      private static int COUNT = 0;
5:      static { System.out.println(COUNT); }
6:      { COUNT++; System.out.println(COUNT); }
7:
8:  public Cuttlefish() {
9:      System.out.println("Constructor");
10: }
11:
12:  public static void main(String[] args) {
13:          System.out.println("Ready");
14:          new Cuttlefish();
15:      }
16:  }
```

Output:

```
0
Ready
swimmy
1Constructor
```

⇨ Không có superclass được khai báo nên có thể bỏ qua bất kỳ bước nào liên quan đến kế thừa.

Sẵn sàng cho một ví dụ khó khăn hơn? Kết quả sau đây là gì?

```
1: class GiraffeFamily {
2:      static { System.out.print("A"); }
3:      { System.out.print("B"); }
4:
5:      public GiraffeFamily(String name) {
6:          this(1);
7:          System.out.print("C");
8:      }
9:
10:     public GiraffeFamily() {
11:         System.out.print("D");
12:     }
13:
14:     public GiraffeFamily(int stripes) {
15:         System.out.print("E");
16:     }
17: }
18: public class Okapi extends GiraffeFamily {
19:     static { System.out.print("F"); }
20:
21:     public Okapi(int stripes) {
22:         super("sugar");
23:         System.out.print("G");
24:     }
25:     { System.out.print("H"); }
26:
27:     public static void main(String[] grass) {
28:         new Okapi(1);
29:         System.out.println();
30:         new Okapi(2);
31:     }
32: }
```

⇨ Output:
AFBECHG
BECHG

g) Reviewing constructor rules

- Câu lệnh đầu tiên của mọi constructor là lệnh gọi tới một overloading constructor thông qua this() hoặc parent constructor trực tiếp thông qua super().
- Nếu câu lệnh đầu tiên của constructor không phải là lệnh gọi this() hoặc super() thì trình biên dịch sẽ chèn super() không có đối số làm câu lệnh đầu tiên của constructor.
- Gọi this() và super() sau câu lệnh đầu tiên của constructor sẽ dẫn đến lỗi trình biên dịch.
- Nếu lớp cha không có constructor không có đối số thì mọi constructor trong lớp con phải bắt đầu bằng lệnh gọi constructor this() hoặc super() rõ ràng.
- Nếu lớp cha không có constructor không có đối số và lớp con không định nghĩa bất kỳ constructor nào thì lớp con sẽ không biên dịch.
- Nếu một lớp chỉ định nghĩa các private constructor thì lớp đó không thể được mở rộng bởi lớp cấp cao nhất.

- Tất cả các final instance phải được gán một giá trị chính xác một lần vào cuối constructor. Bất kỳ biến final instance cùng nào không được gán giá trị sẽ được báo cáo là lỗi trình biên dịch trên dòng mà constructor được khai báo

4. Inheriting members

a) Calling inherited members

Các lớp Java có thể sử dụng bất kỳ thành viên **public** hoặc **protected** nào của parent class, bao gồm các phương thức, primitive hoặc object reference. Nếu lớp cha và lớp con là một phần của cùng một package thì lớp con cũng có thể sử dụng bất kỳ thành viên package-private nào được xác định trong lớp cha.

Cuối cùng, một lớp con không bao giờ có thể truy cập vào một thành viên private của lớp cha, ít nhất là không thông qua bất kỳ tham chiếu trực tiếp nào.

Để tham chiếu một thành viên trong lớp cha, bạn chỉ cần gọi nó trực tiếp, như trong ví dụ sau với function displaySharkDetails():

```
class Fish {
    protected int size;
    private int age;
    public Fish(int age) {
        this.age = age;
    }
    public int getAge() {
        return age;
    }
}
public class Shark extends Fish {
    private int numberOfFins = 8;
    public Shark(int age) {
        super(age);
        this.size = 4;
    }
    public void displaySharkDetails() {
        System.out.print("Shark with age: "+getAge());
        System.out.print(" and "+size+" meters long");
        System.out.print(" with "+numberOfFins+" fins");
    }
}
```

Trong lớp con, chúng ta sử dụng phương thức **public getAge()** và **protected int size** để truy cập các giá trị trong lớp cha. Hãy nhớ rằng, bạn có thể sử dụng **this** để truy cập các thành viên lớp hiện tại hoặc lớp cha và bạn có thể sử dụng super để truy cập các thành viên của lớp cha.

```
public void displaySharkDetails() {
    System.out.print("Shark with age: "+super.getAge());
    System.out.print(" and "+super.size+" meters long");
    System.out.print(" with "+this.numberOfFins+" fins");
}
```

b) Inheriting methods

Overriding a Method

Điều gì sẽ xảy ra nếu có một phương thức được định nghĩa trong cả lớp cha và lớp con có cùng signature? Giải pháp là override - override phương thức trong lớp con. Trong Java, việc override một phương thức xảy ra khi một lớp con khai báo một triển khai mới cho một phương thức được kế thừa có cùng chữ ký và kiểu trả về tương thích. Hãy nhớ rằng chữ ký phương thức bao gồm tên của phương thức và các tham số của phương thức.

Khi bạn override một phương thức, có thể tham chiếu phiên bản gốc của phương thức đó bằng từ khóa super. Theo cách này, từ khóa this và super cho phép bạn chọn giữa phiên bản hiện tại và phiên bản gốc của một phương thức tương ứng. Chúng tôi minh họa điều này bằng ví dụ sau:

```
public class Canine {
    public double getAverageWeight() {
        return 50;
    }
}public class Wolf extends Canine {
    public double getAverageWeight() {
        return super.getAverageWeight()+20;
    }
    public static void main(String[] args) {
        System.out.println(new Canine().getAverageWeight());
        System.out.println(new Wolf().getAverageWeight());
    }
}
```

Trong ví dụ này, trong đó lớp con Wolf override lớp cha Canine, phương thức getAverageWeight() sẽ chạy và chương trình hiển thị như sau:

```
50.0
70.0
```

Để override một phương thức, bạn phải tuân theo một số quy tắc. Trình biên dịch thực hiện các bước kiểm tra sau khi bạn override một phương thức:

- Phương thức trong lớp con phải có cùng chữ ký với phương thức trong lớp cha.
- Phương thức trong lớp con ít nhất phải có khả năng truy cập được như phương thức trong lớp cha.
- Phương thức trong lớp con không được khai báo một checked exception mới hoặc rộng hơn lớp của bất kỳ ngoại lệ nào được khai báo trong phương pháp lớp cha
- Nếu phương thức trả về một giá trị thì giá trị đó phải giống hoặc là một kiểu con của phương thức trong lớp cha, được gọi là covariant return types - hiệp biến.

Quy tắc đầu tiên của việc override một phương thức có phần dễ hiểu. Nếu hai phương thức có cùng tên nhưng khác chữ ký thì phương thức đó bị overload không bị override.

Mục đích của **quy tắc thứ hai** về access modifier là gì? Hãy thử một ví dụ minh họa:

```
public class Camel {
    public int getNumberOfHumps() {
        return 1;
    }
}
public class BactrianCamel extends Camel {
    private int getNumberOfHumps() { // DOES NOT COMPILE
        return 2;
    }
```

```
}

public class Rider {
    public static void main(String[] args) {
        Camel c = new BactrianCamel();
        System.out.print(c.getNumberOfHumps());
    }
}
```

⇨ BactrianCamel cố gắng override phương thức getNumberOfHumps() được xác định trong lớp cha nhưng không thành công vì access modifier private hạn chế hơn phương thức được xác định trong phiên bản cha của phương thức.

Quy tắc thứ ba nói rằng việc override một phương thức không thể khai báo các checked exceptions or checked exception rộng hơn phương thức được kế thừa. Nói cách khác, bạn có thể kết thúc với một đối tượng hạn chế hơn loại tham chiếu mà nó được gán, dẫn đến một ngoại lệ được kiểm tra không được xử lý hoặc khai báo.

Hiện tại, bạn chỉ cần biết nếu một checked exception tra rộng hơn được khai báo trong phương thức override thì code sẽ không được biên dịch. Hãy thử một ví dụ:

```
public class Reptile {
    protected void sleepInShell() throws IOException {}
    protected void hideInShell() throws NumberFormatException {}
    protected void exitShell() throws FileNotFoundException {}
}
public class GalapagosTortoise extends Reptile {
    public void sleepInShell() throws FileNotFoundException {}
    public void hideInShell() throws IllegalArgumentException {}
    public void exitShell() throws IOException {} // DOES NOTCOMPILE
}
```

⇨ Phương thức exitShell() được override, khai báo IOException, là superclass của exception được khai báo trong phương thức kế thừa, FileNotFoundException. Vì đây là checked exception và IOException rộng hơn nên phương thức exitShell() được overridden không biên dịch trong lớp GalapagosTortoise.

Quy tắc thứ tư và cuối cùng xung quanh việc override một phương thức có lẽ là phức tạp nhất, vì nó yêu cầu biết mối quan hệ giữa các kiểu trả về. Phương thức override phải sử dụng kiểu trả về hiệp biến (giống kiểu hoặc là một kiểu con) với kiểu trả về của phương thức được kế thừa.

```
public class Rhino {
    protected CharSequence getName() {
        return "rhino";
    }
    protected String getColor() {
        return "grey, black, or white";
    }
}

class JavanRhino extends Rhino {
    public String getName() {
        return "javan rhino";
    }
    public CharSequence getColor() { // DOES NOT COMPILE
        return "grey";
```

```
    }
}
```

⇨ String triển khai interface CharSequence, biến String thành một kiểu con của CharSequence. Do đó, kiểu trả về của getName() trong JavanRhino là hiệp biến với kiểu trả về của getName() trong Rhino.

Overriding a Generic Method

Generic Method Parameters

Mặt khác, bạn có thể override một phương thức bằng các **generic parameter** - tham số chung, nhưng bạn phải khớp chính xác chữ ký bao gồm **generic type**. Ví dụ: phiên bản này của lớp Anteater biên dịch vì nó sử dụng cùng một **generic type** trong phương thức được override như kiểu được định nghĩa trong lớp cha:

```
class LongTailAnimal {
    protected void chew(List<String> input) {}
}
class Anteater extends LongTailAnimal {
    protected void chew(List<String> input) {}
}
```

Các **generic type parameter** phải khớp, nhưng còn generic class hay interface thì sao? Hãy xem ví dụ sau. Từ những gì bạn biết cho đến nay, bạn có nghĩ các class này sẽ được biên dịch không?

```
public class LongTailAnimal {
    protected void chew(List<Object> input) {}
}
public class Anteater extends LongTailAnimal {
    protected void chew(ArrayList<Double> input) {}
}
```

⇨ Những class này có biên dịch. Tuy nhiên, chúng được coi là phương thức overload chứ không phải phương thức bị override vì chữ ký không giống nhau. Việc xóa kiểu không thay đổi thực tế rằng một trong các đối số của phương thức là Danh sách và đối số còn lại là ArrayList.

Generic Return Types

Khi bạn đang làm việc với các phương thức override trả về tổng quát, các giá trị trả về phải là hiệp biến. Về mặt tổng quát, điều này có nghĩa là kiểu trả về của class hoặc interface được khai báo trong phương thức override phải là subclass được định nghĩa trong lớp cha. Loại tham số chung phải khớp chính xác với loại tham số gốc của nó.

Ví dụ:

```
public class Mammal {
    public List<CharSequence> play() { ... }
    public CharSequence sleep() { ... }
}
public class Monkey extends Mammal {
    public ArrayList<CharSequence> play() { ... }
}
public class Goat extends Mammal {
    public List<String> play() { ... } // DOES NOT COMPILE
    public String sleep() { ... }
}
```

⇨ Phương thức play() trong lớp Goat không biên dịch. Để các kiểu trả về có tính hiệp biến, tham số kiểu chung phải khớp. Mặc dù String là một kiểu con của CharSequence, nhưng

nó không khớp chính xác với kiểu chung được xác định trong lớp **Mammal**. Vì vậy, đây được coi là ghi đè không hợp lệ.

Redeclaring private Methods

Trong Java, bạn không thể ghi đè các phương thức riêng tư vì chúng không được kế thừa. Chỉ vì lớp con không có quyền truy cập vào phương thức cha không có nghĩa là lớp con không thể xác định phiên bản phương thức riêng của nó. Nói đúng ra, điều đó chỉ có nghĩa là phương thức mới không phải là phiên bản bị ghi đè của phương thức của lớp cha.

Java cho phép bạn khai báo lại một phương thức mới trong lớp con có chữ ký giống hoặc được sửa đổi như phương thức trong lớp cha. Phương thức này trong lớp con là một phương thức riêng biệt và độc lập, không liên quan đến phương thức của phiên bản cha, do đó không có quy tắc nào cho các phương thức ghi đè được gọi.

```
public class Camel {
    private String getNumberOfHumps() {
        return "Undefined";
    }
}
public class DromedaryCamel extends Camel {
    private int getNumberOfHumps() {
        return 1;
    }
}
```

Hiding Static Methods

Phương thức ẩn – hidden method xảy ra khi một lớp con định nghĩa một phương thức static có cùng tên và chữ ký với một phương thức static kế thừa được xác định trong lớp cha. Hidden method cũng tương tự nhưng không hoàn toàn giống như ghi đè phương thức. Bốn quy tắc trước đó để ghi đè một phương thức phải được tuân theo khi một phương thức là hidden method. Ngoài ra, một quy tắc mới được thêm vào để ẩn một phương thức:

- Phương thức được định nghĩa trong lớp con phải được đánh dấu là static nếu nó được đánh dấu là static trong lớp cha.

Nói một cách đơn giản, đó là ẩn phương thức nếu hai phương thức được đánh dấu là static và ghi đè phương thức nếu chúng không được đánh dấu là static. Nếu một cái được đánh dấu là static còn cái kia thì không, lớp sẽ không biên dịch.

Hãy xem lại một số ví dụ về quy tắc mới:

```
public class Bear {
    public static void eat() {
        System.out.println("Bear is eating");
    }
}
public class Panda extends Bear {
    public static void eat() {
        System.out.println("Panda is chewing");
    }
    public static void main(String[] args) {
        eat();
    }
}
```

⇨ Trong ví dụ này, mã biên dịch và chạy.

Creating final Methods

Bằng cách đánh dấu một phương thức là final, bạn cấm một lớp con ghi đè phương thức này. Quy tắc này được áp dụng cả khi bạn ghi đè một phương thức và khi bạn ẩn một phương thức. Nói cách khác, bạn không thể ẩn một phương thức static trong lớp con nếu nó được đánh dấu là final trong lớp cha.

```
public class Bird {
    public final boolean hasFeathers() {
        return true;
    }
    public final static void flyAway() {}
}
public class Penguin extends Bird {
    public final boolean hasFeathers() { // DOES NOT COMPILE
        return false;
    }
    public final static void flyAway() {} // DOES NOT COMPILE
}
```

c) Hiding variables

Trên thực tế, Java không cho phép ghi đè các biến. Tuy nhiên, các biến có thể bị ẩn. Biến ẩn xảy ra khi lớp con định nghĩa một biến có cùng tên với biến kế thừa được định nghĩa trong lớp cha. Điều này tạo ra hai bản sao riêng biệt của biến trong một instance của lớp con: một instance được định nghĩa trong lớp cha và một instance được định nghĩa trong lớp con.

Giống như khi ẩn một phương thức static, bạn không thể ghi đè một biến; bạn chỉ có thể che giấu nó. Chúng ta hãy xem xét một biến ẩn. Bạn nghĩ gì về ứng dụng sau đây?

```
class Carnivore {
    protected boolean hasFur = false;
}
public class Meerkat extends Carnivore {
    protected boolean hasFur = true;
    public static void main(String[] args) {
        Meerkat m = new Meerkat();
        Carnivore c = m;
        System.out.println(m.hasFur);
        System.out.println(c.hasFur);
    }
}
```

⇨ Nó in true theo sau là false.

5. Understanding polymorphism

Java hỗ trợ tính đa hình (**polymorphism**), thuộc tính của một đối tượng có nhiều dạng khác nhau. Nói một cách chính xác hơn, một đối tượng Java có thể được truy cập bằng cách sử dụng một tham chiếu có cùng loại với đối tượng, một tham chiếu là **superclass** của đối tượng hoặc một tham chiếu xác định interface mà đối tượng thực hiện, trực tiếp hoặc thông qua **superclass**. Hơn nữa, không cần phải ép kiểu nếu đối tượng đang được gán lại cho **superclass** hoặc **interface** của đối tượng.

Hãy minh họa thuộc tính đa hình này bằng ví dụ sau:

```
public class Primate {
    public boolean hasHair() {
        return true;
    }
}
public interface HasTail {
    public abstract boolean isTailStriped();
}
public class Lemur extends Primate implements HasTail {
    public boolean isTailStriped() {
        return false;
    }
    public int age = 10;
    public static void main(String[] args) {
        Lemur lemur = new Lemur();
        System.out.println(lemur.age);
        HasTail hasTail = lemur;
        System.out.println(hasTail.isTailStriped());
        Primate primate = lemur;
        System.out.println(primate.hasHair());
    }
}
```

Code sẽ biên dịch và in kết quả đầu ra sau:

```
10
false
true
```

Tính đa hình cho phép một instance của Lemur được gán lại hoặc chuyển sang một phương thức bằng cách sử dụng một trong các supertypes của nó, chẳng hạn như Primate hoặc HasTail.

Khi đối tượng đã được gán cho một kiểu tham chiếu mới, chỉ các phương thức và biến có sẵn cho kiểu tham chiếu đó mới có thể gọi được trên đối tượng mà không cần truyền rõ ràng. Ví dụ: các đoạn mã sau sẽ không được biên dịch:

```
HasTail hasTail = lemur;
System.out.println(hasTail.age); // DOES NOT COMPILE
Primate primate = lemur;
System.out.println(primate.isTailStriped()); // DOES NOT COMPILE
```

a) Object vs. Reference

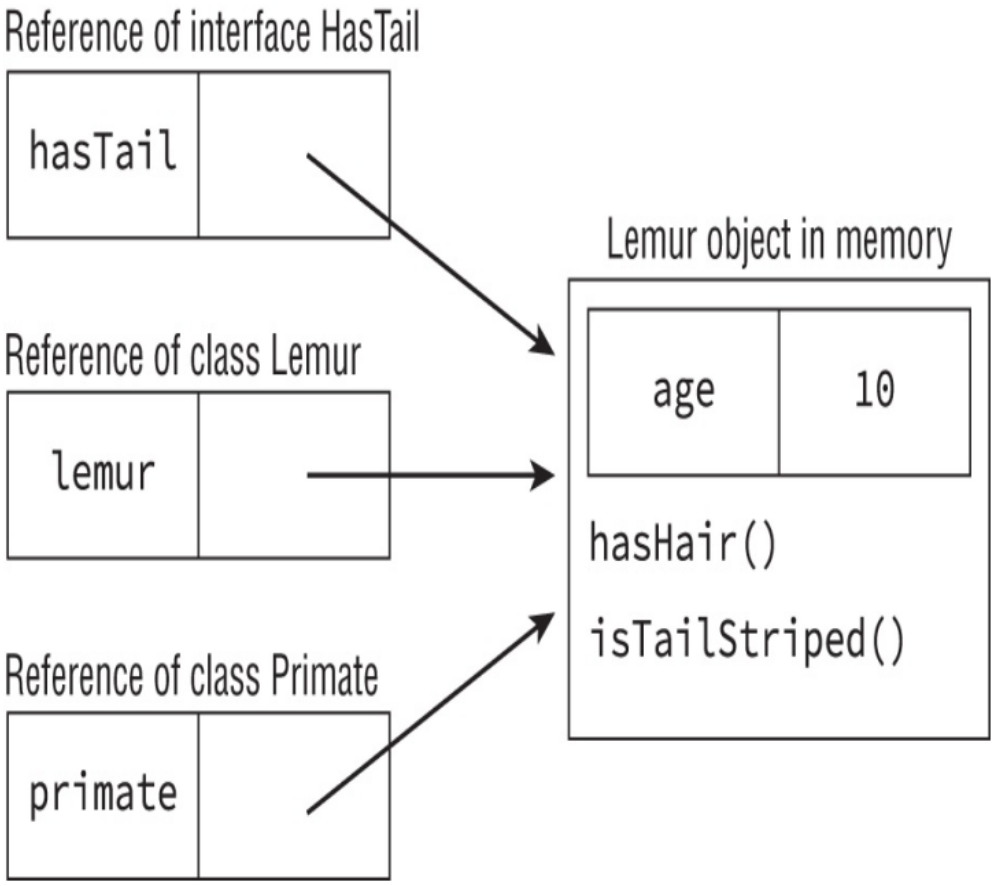
Trong Java, tất cả các đối tượng đều được truy cập bằng tham chiếu, vì vậy, với tư cách là nhà phát triển, bạn không bao giờ có quyền truy cập trực tiếp vào chính đối tượng đó. Tuy nhiên, về mặt khái niệm, bạn nên coi đối tượng là thực thể tồn tại trong bộ nhớ, được phân bổ bởi Java runtime environment. Bất kể loại tham chiếu nào bạn có cho đối tượng trong bộ nhớ, bản thân đối tượng đó sẽ không thay đổi. Ví dụ: vì tất cả các đối tượng đều kế thừa java.lang.Object nên tất cả chúng đều có thể được gán lại cho java.lang.Object, như trong ví dụ sau:

```
Lemur lemur = new Lemur();
Object lemurAsObject = lemur;
```

Chúng ta có thể tóm tắt nguyên tắc này bằng hai quy tắc sau:

- Loại đối tượng xác định thuộc tính nào tồn tại trong đối tượng trong bộ nhớ.
- Kiểu tham chiếu đến đối tượng xác định phương thức và biến nào có thể truy cập được đối với chương trình Java

Minh họa thuộc tính này bằng ví dụ trước



Cùng một đối tượng tồn tại trong bộ nhớ bất kể tham chiếu nào đang trỏ đến nó. Tùy thuộc vào loại tham chiếu, chúng ta có thể chỉ có quyền truy cập vào một số phương thức nhất định. Ví dụ: hasTail reference có quyền truy cập vào phương thức isTailStriped() nhưng không có quyền truy cập vào biến age được xác định trong lớp Lemur.

b) Casting objects

Khi thay đổi loại tham chiếu, sẽ mất quyền truy cập vào các thành viên cụ thể hơn được xác định trong lớp con vẫn tồn tại trong đối tượng. Chúng ta có thể lấy lại các tham chiếu đó bằng cách chuyển đổi tượng trở lại lớp con cụ thể mà nó xuất phát:

```
Primate primate = new Lemur(); // Implicit Cast - ngầm
Lemur lemur2 = primate; // DOES NOT COMPILE
System.out.println(lemur2.age);
Lemur lemur3 = (Lemur)primate; // Explicit Cast
System.out.println(lemur3.age);
```

Tóm tắt những khái niệm này thành một bộ quy tắc:

- Việc truyền một tham chiếu từ một subtype đến một supertype không yêu cầu một kiểu truyền rõ ràng.
- Việc truyền một tham chiếu từ supertype sang một subtype yêu cầu một kiểu truyền rõ ràng.
- Trình biên dịch không cho phép chuyển sang một lớp không liên quan.
- Trong thời gian chạy, việc truyền tham chiếu không hợp lệ đến loại không liên quan sẽ dẫn đến ngoại lệ ClassCastException bị ném ra.

Hãy xem xét ví dụ này:

```
public class Bird {}
public class Fish {
    public static void main(String[] args) {
        Fish fish = new Fish();
        Bird bird = (Bird)fish; // DOES NOT COMPILE
    }
}
```

Casting không phải là không có những hạn chế của nó. Mặc dù hai lớp chia sẻ một hệ thống phân cấp liên quan, điều đó không có nghĩa là một instance của một lớp có thể tự động được chuyển sang lớp khác. Đây là một ví dụ:

```
public class Rodent {}
public class Capybara extends Rodent {
    public static void main(String[] args) {
        Rodent rodent = new Rodent();
        Capybara capybara = (Capybara)rodent; //ClassCastException
    }
}
```

Code tạo một instance của Rodent và sau đó cố gắng chuyển nó sang một subclass của Rodent, Capybara. Mặc dù code này sẽ biên dịch nhưng nó sẽ ném ra một ngoại lệ ClassCastException khi chạy vì đối tượng được tham chiếu không phải là một instance của lớp Capybara. Điều cần lưu ý trong ví dụ này là đối tượng Rodent được tạo không extends lớp Capybara dưới bất kỳ hình thức nào.

c) The *instanceof* operator

Toán tử instanceof, có thể được sử dụng để kiểm tra xem một đối tượng có thuộc về một class hoặc interface cụ thể hay không và để ngăn chặn ClassCastExceptions khi chạy. Không giống như ví dụ trước, đoạn code sau không đưa ra ngoại lệ trong thời gian chạy và chỉ thực hiện việc truyền nếu toán tử instanceof trả về true:

```
if(rodent instanceof Capybara) {
    Capybara capybara = (Capybara)rodent;
}
```

Giống như trình biên dịch không cho phép truyền một đối tượng sang các kiểu không liên quan, nó cũng không cho phép sử dụng instanceof với các kiểu không liên quan. Ví dụ:

```
class Bird {}
class Fish {
    public static void main(String[] args) {
        Fish fish = new Fish();
        if (fish instanceof Bird) { // DOES NOT COMPILE
            Bird bird = (Bird) fish; // DOES NOT COMPILE}
        }
    }
}
```

d) Polymorphism and method overriding

Trong Java, tính đa hình cho biết rằng khi bạn ghi đè một phương thức, bạn sẽ thay thế tất cả các lệnh gọi đến nó, ngay cả những lệnh gọi được xác định trong lớp cha. Ví dụ: bạn nghĩ kết quả đầu ra của đoạn mã sau đây là gì?

```
class Penguin {
    public int getHeight() { return 3; }
    public void printInfo() {
        System.out.print(this.getHeight());
    }
}
public class EmperorPenguin extends Penguin {
    public int getHeight() { return 8; }
    public static void main(String []fish) {
        new EmperorPenguin().printInfo();
    }
}
```

⇨ 8.

Khía cạnh đa hình thay thế các phương thức thông qua ghi đè là một trong những thuộc tính quan trọng nhất trong tất cả Java. Nó cho phép bạn tạo các mô hình kế thừa phức tạp, với các lớp con có cách triển khai các phương thức ghi đè tùy chỉnh của riêng chúng. Điều đó cũng có nghĩa là lớp cha không cần phải cập nhật để sử dụng phương thức tùy chỉnh hoặc ghi đè. Nếu phương thức được ghi đè đúng cách thì phiên bản bị ghi đè sẽ được sử dụng ở tất cả những nơi mà nó được gọi

e) Overriding vs. Hiding members

Mặc dù ghi đè phương thức sẽ thay thế phương thức ở mọi nơi nó được gọi, nhưng static method và variable hiding thì không. Nói đúng ra, các hiden member không phải là một dạng đa hình vì các phương thức và biến duy trì các thuộc tính riêng của chúng. Không giống như ghi đè phương thức, việc ẩn thành viên rất nhạy cảm với loại tham chiếu và vị trí nơi thành viên đang được sử dụng

```
class Penguin {
    public static int getHeight() { return 3; }
    public void printInfo() {
        System.out.println(this.getHeight());
    }
}
class CrestedPenguin extends Penguin {
    public static int getHeight() { return 8; }
    public static void main(String... fish) {
        new CrestedPenguin().printInfo();
    }
}
```

⇨ nó in 3 thay vì 8. Phương thức getHeight() là static và do đó bị ẩn, không bị ghi đè. Kết quả là việc gọi getHeight() trong CrestedPenguin trả về một giá trị khác với việc gọi nó trong Penguin, ngay cả khi đối tượng cơ bản giống nhau.

Bên cạnh vị trí, loại tham chiếu cũng có thể xác định giá trị bạn nhận được khi làm việc với các hidden member. Hãy thử một ví dụ phức tạp hơn:

```
class Marsupial {
    protected int age = 2;
    public static boolean isBiped() {return false;
    }
}
class Kangaroo extends Marsupial {
    protected int age = 6;
    public static boolean isBiped() {
        return true;
    }
    public static void main(String[] args) {
        Kangaroo joey = new Kangaroo();
        Marsupial moey = joey;
        System.out.println(joey.isBiped());
        System.out.println(moey.isBiped());
        System.out.println(joey.age);
        System.out.println(moey.age);
    }
}
```

Chương trình in ra như sau:

```
true
false
62
```

Chapter 9: Advanced Class Design

1. Creating abstract classes

a) Introducing abstract classes

Abstract class - **abstract classe** là lớp không thể khởi tạo và có thể chứa các abstract method – **abstract method**. abstract method là phương thức không xác định cách triển khai khi nó được khai báo. Cả abstract classes và abstract methods đều được biểu thị bằng abstract modifier.

```
abstract class Bird {
    public abstract String getName();
    public void printName() {
        System.out.print(getName());
    }
}
public class Stork extends Bird {
    public String getName() { return "Stork!"; }
    public static void main(String[] args) {
        new Stork().printName();}
}
```

Ví dụ: cách triển khai sau đây không được biên dịch vì Stork không ghi đè abstract method getName() được yêu cầu:

```
public class Stork extends Bird {} // DOES NOT COMPILE
```

abstract class được sử dụng phổ biến nhất khi bạn muốn một lớp khác kế thừa các thuộc tính của một lớp cụ thể, nhưng bạn muốn lớp con triển khai chi tiết. Trong ví dụ của chúng ta, class Bird đã viết phương thức printName() nhưng không biết nó sẽ làm gì trong thời gian chạy vì việc triển khai getName() vẫn chưa được cung cấp.

abstract class là lớp không thể khởi tạo được. Điều này có nghĩa là nếu bạn cố gắng khởi tạo nó, trình biên dịch sẽ báo cáo một exception, như trong ví dụ này:

```
abstract class Alligator {
    public static void main(String... food) {
        var a = new Alligator(); // DOES NOT COMPILE
    }
}
```

b) Defining abstract methods

abstract method có thể bao gồm các phương thức không trừu tượng, trong trường hợp này là phương thức printName(). Trong thực tế, một **abstract class** có thể bao gồm tất cả các thành viên giống như một lớp nonabstract, bao gồm các biến, static và instance method cũng như các lớp bên trong (inner class).

Một trong những tính năng quan trọng nhất của abstract class là nó thực sự không bắt buộc phải bao gồm bất kỳ abstract method nào. Ví dụ: đoạn mã sau biên dịch ngay cả khi nó không xác định bất kỳ abstract method nào:

```
public abstract class Llama {
    public void chew() {}
}
```

Mặc dù một abstract class không phải khai báo bất kỳ abstract method nào, nhưng một abstract method chỉ có thể được định nghĩa trong một abstract class (hoặc interface).

```
public class Egret { // DOES NOT COMPILE
    public abstract void peck();
}
```

Giống như **final** modifier, **abstract** có thể được đặt trước hoặc sau access modifier trong các khai báo class và method, như được hiển thị trong class Tiger này:

```
abstract public class Tiger {
    abstract public int claw();
}
```

Constructors in Abstract Classes

Mặc dù các **abstract class** không thể được khởi tạo nhưng chúng vẫn được các **subclass** của chúng khởi tạo thông qua constructor. Ví dụ: chương trình sau có biên dịch không?

```
abstract class Bear {
    abstract CharSequence chew();
    public Bear() {
        System.out.println(chew()); // Does this compile?
    }
}
class Panda extends Bear {
    String chew() { return "yummy!"; }
    public static void main(String[] args) {
        new Panda();
    }
}
```

⇨ Trình biên dịch chèn một default no-argument constructor vào lớp Panda, lớp này đầu tiên gọi super() trong lớp Bear. Constructor Bear chỉ được gọi khi **abstract class** đang được khởi tạo thông qua một **subclass**; do đó, có một triển khai chew() tại thời điểm constructor được gọi. Mã này biên dịch và in "yummy!" trong thời gian chạy.

Invalid Abstract Method Declarations

Ví dụ: bạn có thể hiểu tại sao mỗi phương thức sau không biên dịch được không?

```
public abstract class Turtle {
    public abstract long eat() // DOES NOT COMPILE
    public abstract void swim() {}; // DOES NOT COMPILE
    public abstract int getAge() { // DOES NOT COMPILE
        return 10;
    }
    public void sleep; // DOES NOT COMPILE
    public void goInShell(); // DOES NOT COMPILE
}
```

- Phương thức đầu tiên, **eat(),** không biên dịch vì nó được đánh dấu là abstract nhưng không kết thúc bằng dấu chấm phẩy (;).
- Hai phương thức tiếp theo, **swim()** và **getAge(),** không biên dịch vì chúng được đánh dấu là abstract, nhưng chúng cung cấp một khối triển khai được đặt trong dấu ngoặc nhọn ({}).
- Phương thức tiếp theo, **sleep()**, không biên dịch vì nó thiếu dấu ngoặc đơn, ().
- Phương thức cuối cùng, goInShell(), không biên dịch vì nó không được đánh dấu là abstract và do đó phải cung cấp phần thân được đặt trong dấu ngoặc nhọn

c) Creating a concrete class

abstract class sẽ có thể sử dụng được khi nó được mở rộng bởi một lớp con cụ thể - **concrete subclass**. **Concrete subclass** là một lớp không trừu tượng. Cần có concrete subclass đầu tiên mở rộng một abstract class để triển khai tất cả các abstract method được kế thừa. Điều này bao gồm việc triển khai bất kỳ abstract method kế thừa nào từ các inherited interface.

Bạn có thể hiểu tại sao lớp Walrus sau không biên dịch được không?

```
public abstract class Animal {
    public abstract String getName();
}
public class Walrus extends Animal { // DOES NOT COMPILE
}
```

Animal được đánh dấu là abstract còn Walrus thì không, khiến Walrus trở thành một **concrete subclass** của Animal. Vì Walrus là concrete subclass đầu tiên nên nó phải triển khai tất cả các phương thức trừu tượng được kế thừa —getName() trong ví dụ này. Bởi vì không phải vậy, trình biên dịch sẽ báo lỗi với việc khai báo Walrus.

abstract class có thể extend một non-abstract class và ngược lại. Bất cứ khi nào một **concrete subclass** extend một abstract class, nó phải triển khai tất cả các phương thức được kế thừa dưới dạng trừu tượng. Hãy minh họa điều này bằng một tập hợp các abstract class:

```
abstract class Mammal {
    abstract void showHorn();
    abstract void eatLeaf();
}
abstract class Rhino extends Mammal {
    void showHorn() {}
}
class BlackRhino extends Rhino {
    void eatLeaf() {}
}
```

class BlackRhino là lớp con cụ thể đầu tiên, trong khi các lớp Mammal và Rhino là abstract class. class BlackRhino kế thừa phương thức eatLeaf() dưới dạng trừu tượng và do đó bắt buộc phải cung cấp một cách triển khai.

Rhino, cung cấp cách triển khai showHorn(), phương thức này được kế thừa trong BlackRhino dưới dạng một phương thức không trừu tượng. Vì lý do này, class BlackRhino được phép nhưng không bắt buộc phải ghi đè phương thức showHorn(). Do đó chỉ cần ghi đè phương thức eatLeaf()

Hãy thử một ví dụ nữa. Class Lion cụ thể sau đây kế thừa hai phương thức trừu tượng, getName() và noise():

```
public abstract class Animal {
    abstract String getName();
}
public abstract class BigCat extends Animal {
    protected abstract void roar();
}
public class Lion extends BigCat {
    public String getName() {
        return "Lion";
    }
    public void roar() {
        System.out.println("The Lion lets out a loud ROAR!");
    }
}
```

d) Reviewing abstract class rules

Abstract Class Definition Rules

- Các **abstract class** không thể được khởi tạo.
- Tất cả các loại cấp cao nhất, bao gồm cả các abstract class, không thể được đánh dấu là protected or private.
- Các abstract class không thể được đánh dấu là final.
- Các abstract class có thể bao gồm 0 hoặc nhiều abstract và nonabstract method.
- Một abstract class mở rộng một abstract class khác kế thừa tất cả các abstract method của nó.
- **Concrete class** đầu tiên mở rộng một abstract class phải cung cấp cách triển khai cho tất cả các abstract method được kế thừa.
- Các abstract class constructors tuân theo các quy tắc khởi tạo tương tự như các constructor thông thường, ngoại trừ chúng chỉ có thể được gọi như một phần của quá trình khởi tạo một subclass.

Abstract Method Definition Rules

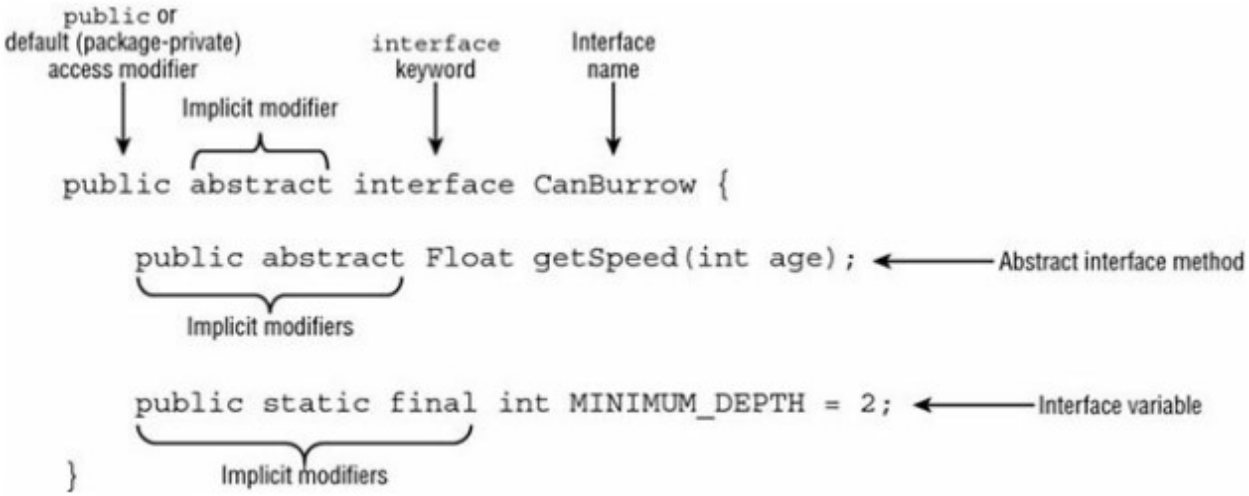
- Các **abstract method** chỉ có thể được định nghĩa trong các **abstract classes** hoặc **interface**.
- Các **abstract method** không thể được khai báo là private hoặc final.
- Các **abstract method** không được cung cấp nội dung/cách triển khai phương thức trong **abstract class** mà chúng được khai báo.
- Việc triển khai một **abstract method** trong một lớp con tuân theo các quy tắc tương tự để ghi đè một phương thức, bao gồm các kiểu trả về hiệp biến, khai báo exception, v.v.

2. Implementing interfaces

Mặc dù Java không cho đa kế thừa nhưng nó cho phép class triển khai bất kỳ số lượng interface nào. Interface là một kiểu dữ liệu trừu tượng khai báo danh sách các abstract method mà bất kỳ class nào triển khai interface đều phải cung cấp. Một interface cũng có thể bao gồm các biến hằng số. Cả hai **abstract method** và constant variable có trong interface đều được coi là public.

a) Defining an interface

Trong Java, một giao diện được định nghĩa bằng từ khóa giao diện, tương tự như từ khóa lớp được sử dụng khi định nghĩa một lớp.



Khai báo interface của chúng ta bao gồm một biến không đổi và một abstract method. Các biến interface được gọi là hằng số vì chúng được coi là public, static, và final. Chúng được khởi tạo với giá trị không đổi khi được khai báo. Vì chúng là public và static nên chúng có thể được sử dụng bên ngoài khai báo interface mà không yêu cầu instance của interface.

Một khía cạnh của khai báo interface khác với abstract class là nó chứa các **implicit modifier**. **implicit modifier** là modifier mà trình biên dịch tự động thêm vào class, interface, method hoặc khai báo biến.

```
public abstract interface WalksOnTwoLegs {}
```

Nó biên dịch vì các interface không bắt buộc phải xác định bất kỳ phương thức nào.

Hãy xem xét hai ví dụ sau, không được biên dịch:

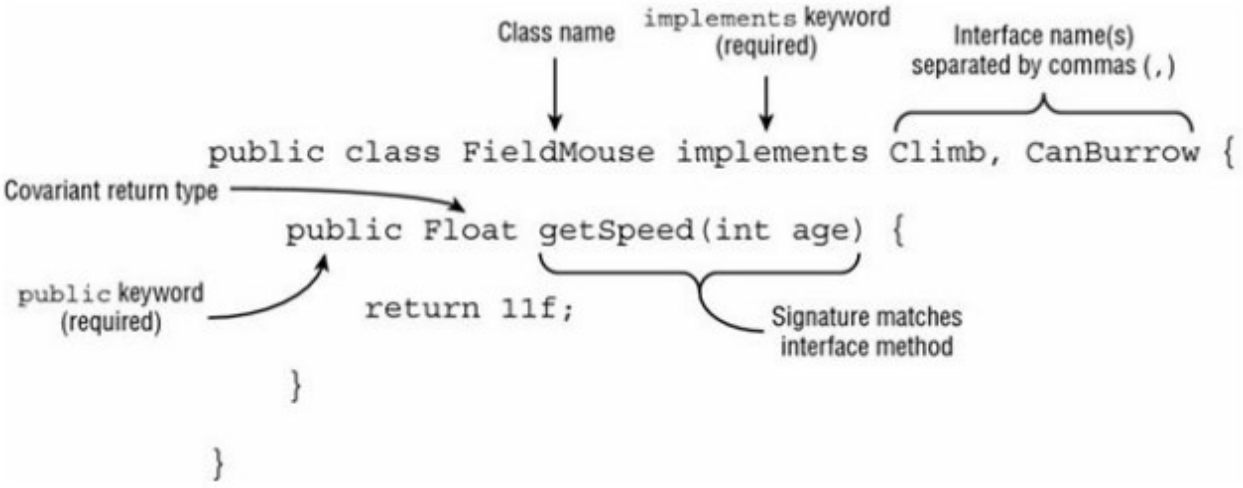
```
public class Biped {
    public static void main(String[] args) {
        var e = new WalksOnTwoLegs();  // DOES NOT COMPILE
    }
}

public final interface WalksOnEightLegs {} // DOES NOT COMPILE
```

Ví dụ đầu tiên không được biên dịch vì WalksOnTwoLegs là một interface và không thể khởi tạo được. Ví dụ thứ hai, WalksOnEightLegs, không biên dịch được vì các interface không thể được đánh dấu là final

Bạn sử dụng một interface như thế nào? Giả sử chúng ta có interface Climb, được định nghĩa như sau:

```
interface Climb {
    Number getSpeed(int age);
}
```

Tiếp theo, chúng ta có một concrete class FieldMouse gọi Climb interface bằng cách sử dụng từ khóa implements trong khai báo class:



Một class có thể triển khai nhiều interface, mỗi interface được phân tách bằng dấu phẩy (,). Nếu bất kỳ interface nào định nghĩa các abstract method thì cần có concrete class để ghi đè chúng.

Giống như một class, một interface có thể extend một interface khác bằng từ khóa **extends**.

```
interface Nocturnal {}

public interface HasBigEyes extends Nocturnal {}
```

Không giống như một class chỉ có thể mở rộng một class, một interface có thể mở rộng nhiều interface.

```
interface Nocturnal {
    public int hunt();
}

interface CanFly {
    public void flap();
}

interface HasBigEyes extends Nocturnal, CanFly {}

public class Owl implements Nocturnal, CanFly {
    public int hunt() { return 5; }
    public void flap() { System.out.println("Flap!"); }
}
```

Nhiều quy tắc khai báo class cũng áp dụng cho các interface bao gồm:

- Một file Java có thể có nhiều nhất một class hoặc interface cấp cao nhất public và nó phải khớp với tên của file.
- Class hoặc interface cấp cao nhất chỉ có thể được khai báo với quyền truy cập public hoặc gói package-private.

b) Inserting implicit modifiers

Danh sách sau đây bao gồm các implicit modifier cho các interface mà bạn cần biết:

- Các interface được coi là abstract.
- Các biến interface được coi là public, static, và final.
- Các phương thức interface không có phần thân được coi là abstract và public.\

Ví dụ: hai định nghĩa interface sau đây là tương đương, vì trình biên dịch sẽ chuyển đổi cả hai sang khai báo thứ hai:

```
public interface Soar {
    int MAX_HEIGHT = 10;
    final static boolean UNDERWATER = true;
    void fly(int speed);
    abstract void takeoff();
    public abstract double dive();
}

public abstract interface Soar {
    public static final int MAX_HEIGHT = 10;
    public final static boolean UNDERWATER = true;
    public abstract void fly(int speed);
    public abstract void takeoff();
    public abstract double dive();
}
```

Conflicting Modifiers

Ví dụ: nếu một abstract method được coi là public thì nó có thể được đánh dấu rõ ràng là protected hay private không?

```
public interface Dance {
    private int count = 4; // DOES NOT COMPILE
    protected void step(); // DOES NOT COMPILE
}
```

Cả hai khai báo thành viên giao diện này đều không được biên dịch, vì trình biên dịch sẽ áp dụng **public modifier** cho cả hai, dẫn đến xung đột.

Dòng nào trong khai báo interface không được biên dịch?

```
1:      private final interface Crawl {
2:          String distance;
3:          private int MAXIMUM_DEPTH = 100;
4:          protected abstract boolean UNDERWATER = false;
5:          private void dig(int depth);
6:          protected abstract double depth();
7:          public final void surface(); }
```

⇨ Tất cả điều ko biên dịch được.

Differences between Interfaces and Abstract Classes

Abstract class	Interface
1) Abstract class có phương thức **abstract** (không có thân hàm) và phương thức **non-abstract** (có thân hàm).	Interface chỉ có phương thức **abstract**. Từ java 8, nó có thêm **các phương thức default và static**.
2) Abstract class **không hỗ trợ đa kế thừa**.	Interface **có hỗ trợ đa kế thừa**
3) Abstract class có các biến **final, non-final, static and non-static**.	Interface chỉ có các biến **static và final**.
4) Abstract class **có thể cung cấp nội dung cài đặt cho phương thức của interface**.	Interface **không thể cung cấp nội dung cài đặt cho phương thức của abstract class**.
5) Từ khóa **abstract** được sử dụng để khai báo abstract class.	Từ khóa **interface** được sử dụng để khai báo interface.

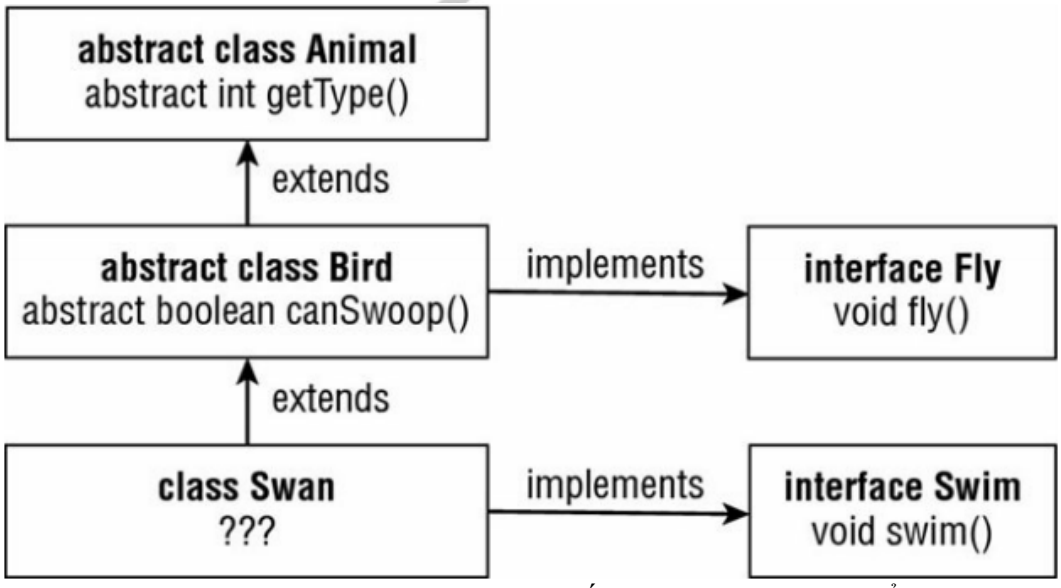
6) Ví dụ:
public abstract class Shape {
public abstract void draw();
}
	Ví dụ:
	public interface Drawable {
	void draw();
	}

c) Inheriting an interface

Một interface có thể được kế thừa theo một trong ba cách.

- Một interface có thể mở rộng một interface khác.
- Một class có thể implement một interface.
- Một class có thể mở rộng một class khác mà class trước của nó implement một interface.

Khi một interface được kế thừa, tất cả các **abstract method** đều được kế thừa. **concrete subclass** đầu tiên kế thừa giao diện phải triển khai tất cả các phương thức trừu tượng được kế thừa.



Chúng ta hãy xem một ví dụ khác liên quan đến một abstract class triển khai interface:

```
public interface HasTail {
    public int getTailLength();
}

public interface HasWhiskers {
    public int getNumberOfWhiskers();
}

public abstract class HarborSeal implements HasTail, HasWhiskers {}

public class CommonSeal extends HarborSeal { // DOES NOT COMPILE
}
```

Class HarborSeal không bắt buộc phải triển khai bất kỳ abstract method nào mà nó kế thừa từ HasTail và HasWhiskers vì nó được đánh dấu là abstract. CommonSeal là **concrete subclass** lên phải triển khai các abstract method

Duplicate Interface Method Declarations

Vì Java cho phép đa kế thừa thông qua các interface, nên sẽ xảy ra nếu bạn định nghĩa một class kế thừa từ hai interface chứa cùng một abstract method.

```
public interface Herbivore {
    public void eatPlants();
}

public interface Omnivore {
    public void eatPlants();
    public void eatMeat();
}
```

Bạn có thể định nghĩa một class đáp ứng đồng thời cả hai interface.

```
public class Bear implements Herbivore, Omnivore {
    public void eatMeat() {
        System.out.println("Eating meat");
    }
    public void eatPlants() {
        System.out.println("Eating plants");
    }
}
```

Nếu hai **abstract interface method** có hành vi giống hệt nhau - hoặc trong trường hợp này là khai báo phương thức giống nhau - bạn chỉ cần tạo một phương thức duy nhất ghi đè cả hai **abstract method** được kế thừa cùng một lúc.

Điều gì xảy ra nếu các phương thức trùng lặp có chữ ký khác nhau? Nếu tên phương thức giống nhau nhưng tham số đầu vào khác nhau thì không có xung đột vì đây được coi là method overload. Chúng tôi chứng minh nguyên tắc này trong ví dụ sau:

```
public interface Herbivore {
    public int eatPlants(int quantity);
}

public interface Omnivore {
    public void eatPlants();
}

public class Bear implements Herbivore, Omnivore {
    public int eatPlants(int quantity) {
        System.out.println("Eating plants: " + quantity);
        return quantity;
    }
    public void eatPlants() {
        System.out.println("Eating plants");
    }
}
```

Điều gì sẽ xảy ra nếu các phương thức trùng lặp có cùng chữ ký nhưng có kiểu trả về khác nhau? Trong trường hợp đó, bạn cần xem lại các quy tắc dành cho phương pháp ghi đè. Hãy thử một ví dụ:

```
interface Dances {
    String swingArms();
}

interface EatsFish {
    CharSequence swingArms();
}

public class Penguin implements Dances, EatsFish {
    public String swingArms() {
        return "swing!";
    }
}
```

⇨ class Penguin biên dịch. Vì kiểu dữ liệu hiệp biến

Chúng ta hãy xem một mẫu trong đó các kiểu trả về không hiệp biến:

```
interface Dances {
    int countMoves();
}

interface EatsFish {
    boolean countMoves();
}

public class Penguin implements Dances, EatsFish { // DOES NOT
    COMPILE
    ...
}
```

⇨ Vì không thể xác định một phiên bản của countMoves() trả về cả int và boolean, nên không có triển khai Penguin nào cho phép biên dịch khai báo này

d) Polymorphism and interfaces

Casting Interfaces

Ví dụ: những điều sau đây không được phép vì trình biên dịch phát hiện ra rằng class String và Long không thể liên quan với nhau:

```
String lion = "Bert";
Long tiger = (Long)lion;
```

Với các **interface**, có những hạn chế đối với những gì trình biên dịch có thể xác thực. Ví dụ: chương trình sau có biên dịch không?

```
1: interface Canine {}
2: class Dog implements Canine {}
3: class Wolf implements Canine {}
4:
5: public class BadCasts {
6:   public static void main(String[] args) {
7:     Canine canine = new Wolf();
8:     Canine badDog = (Dog)canine;
9:   } }
```

⇨ Mã biên dịch nhưng ném ra ngoại lệ ClassCastException khi chạy.

Interfaces and the instanceof Operator

Trình biên dịch sẽ báo lỗi nếu bạn cố gắng sử dụng toán tử instanceof với hai lớp không liên quan, như sau:

```
Number tickets = 4;
if(tickets instanceof String) {} // DOES NOT COMPILE
```

Với các interface, trình biên dịch có khả năng hạn chế trong việc thực thi quy tắc này vì mặc dù kiểu tham chiếu có thể không triển khai interface nhưng một trong các subclasses của nó có thể. Ví dụ: phần sau đây sẽ biên dịch:

```
Number tickets = 5;
if(tickets instanceof List) {}
```

Mặc dù **Number** không kế thừa **List**, nhưng có thể biến **tickets** có thể tham chiếu đến một subclass của **Number** kế thừa **List**.

Ví dụ:

```
public class MyNumber extends Number implements List
```

Điều đó có nghĩa là trình biên dịch có thể kiểm tra các interface không liên quan nếu tham chiếu là class được đánh dấu là final.

```
Integer tickets = 6;
if(tickets instanceof List) {} // DOES NOT COMPILE
```

e) Reviewing interface rules

Để định nghĩa abstractclass thì bốn quy tắc đầu tiên đều tương tự nhau.

Quy tắc định nghĩa interface

1. **Interface** không thể được khởi tạo.
2. Tất cả các loại cấp cao nhất, bao gồm cả **interface**, không thể được đánh dấu là **protected** hoặc **private**.
3. Các interface được coi là abstract và không thể được đánh dấu là final.
4. Các interface có thể bao gồm 0 hoặc nhiều abstract method.
5. Một interface có thể mở rộng bất kỳ số lượng interface nào
6. Một interface reference có thể được chuyển sang bất kỳ tham chiếu nào kế thừa interface đó, mặc dù điều này có thể tạo ra exception khi chạy nếu các class không liên quan.
7. Trình biên dịch sẽ chỉ báo cáo lỗi kiểu không liên quan đối với thao tác instanceof có interface ở bên phải nếu tham chiếu ở bên trái là class final không kế thừa interface.
8. Một interface method có phần thân phải được đánh dấu default, private, static, hoặc private static

Quy tắc Abstract Interface Method

1. Các abstract method chỉ có thể được định nghĩa trong các class hoặc abstract interface.
2. Các abstract method không thể được khai báo là private hoặc final
3. Các abstract method không được cung cấp nội dung/cách triển khai phương thức trong abstract class mà nó được khai báo.
4. Việc triển khai một abstract method trong một lớp con tuân theo các quy tắc tương tự để ghi đè một phương thức, bao gồm các kiểu trả về hiệp biến, khai báo exception, v.v.
5. Các **abstract interface** không có phần thân được coi là abstract và publi.

Interface Variables Rules

1. Các biến interface được coi là public, static, và final
2. Vì các biến interface được đánh dấu là final nên chúng phải được khởi tạo bằng một giá trị khi khai báo.

3. Introducing inner classes

a) Defining a member inner class

inner class là class được xác định ở cấp thành viên của một class (cùng cấp với các methods, instance variables, và constructors).

Sau đây là ví dụ về class bên trong 1 class:

```
public class Zoo {
    public class Ticket {}
}
```

Chúng ta có thể mở rộng điều này để bao gồm một interface.

```
public class Zoo {
    private interface Paper {}
    public class Ticket implements Paper {}
}
```

Mặc dù các class và interface cấp cao nhất chỉ có thể được đặt với quyền truy cập public hoặc package-private, các class bên trong thành viên không có cùng access modifier. Một class bên trong thành viên có thể được khai báo với tất cả các access modifier giống như một thành viên class, chẳng hạn như public, protected, default (package-private) hoặc private.

Một class bên trong thành viên có thể chứa nhiều phương thức và biến giống như class cấp cao nhất. Một số thành viên không được phép tham gia các class bên trong thành viên, chẳng hạn như thành viên static:

```
public class Zoo {
    private interface Paper {
        public String getId();
    }
    public class Ticket implements Paper {
        private String serialNumber;
        public String getId() { return serialNumber; }
    }
}
```

b) Using a member inner class

Một trong những cách có thể sử dụng thành viên **inner class** là outer class. Tiếp tục với ví dụ trước của chúng ta, hãy xác định một phương thức trong Zoo sử dụng inner class bằng phương thức sellTicket() mới.

```
public class Zoo {
    private interface Paper {
        public String getId();
    }
    public class Ticket implements Paper {
        private String serialNumber;
        public String getId() { return serialNumber; }
    }
    public Ticket sellTicket(String serialNumber) {
        var t = new Ticket();
        t.serialNumber = serialNumber;
        return t;
    }
}
```

Hãy thêm một điểm vào ví dụ này.

```
public class Zoo {
  ...
    public static void main(String...unused) {
        var z = new Zoo();
        var t = z.sellTicket("12345");
        System.out.println(t.getId() + " Ticket sold!");
    }
}
```

Chapter 10 Exceptions

1. Understanding Exceptions

Một chương trình có thể thất bại vì bất kỳ lý do gì. Đây chỉ là một vài khả năng:

- Code cố gắng kết nối với một trang web nhưng kết nối Internet bị ngắt.
- Bạn đã mắc lỗi khi code và cố truy cập vào một index không hợp lệ trong một mảng.
- Một phương thức gọi một phương thức khác với giá trị mà phương thức đó không hỗ trợ.

a) The role of exceptions

Khi bạn viết một phương thức, bạn có thể xử lý exception hoặc biến nó thành vấn đề của khi gọi code. Một phương thức có thể tự xử lý trường hợp exceptio hoặc coi đó là trách nhiệm của caller.

```
1: public class Zoo {
2:    public static void main(String[] args) {
3:      System.out.println(args[0]);
4:      System.out.println(args[1]);
5: } }
```

Sau đó, bạn đã cố gắng gọi nó mà không có đủ đối số:

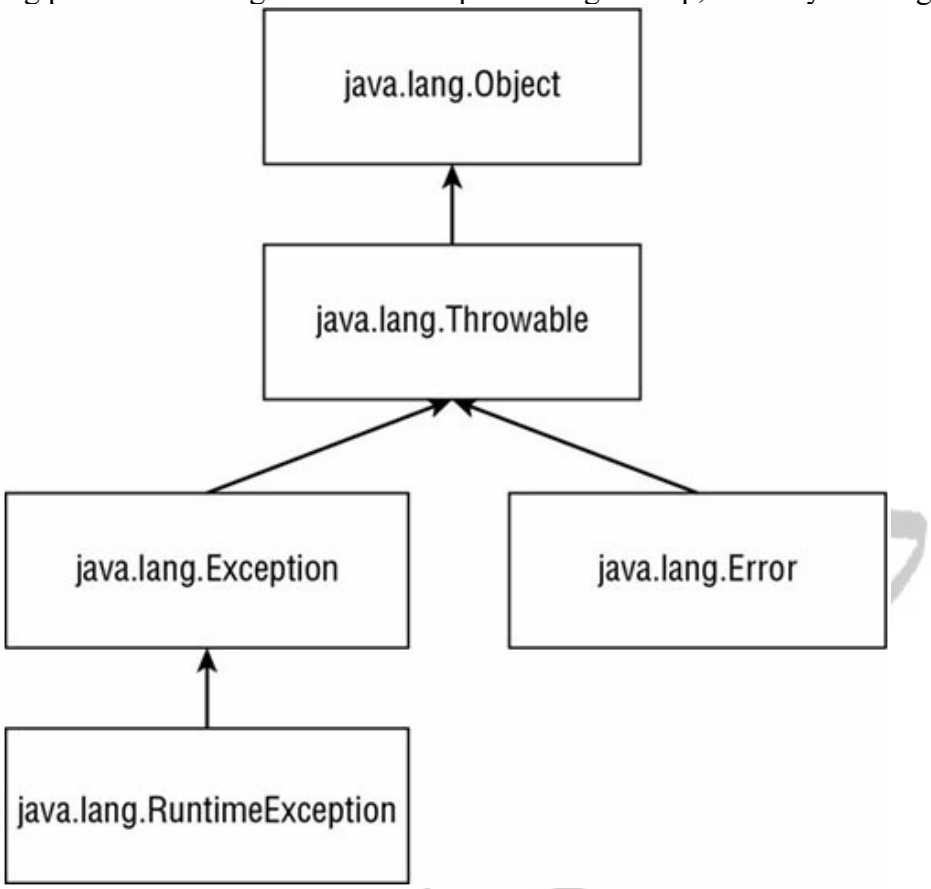
```
$ javac Zoo.java
$ java Zoo Zoo
```

Ở dòng 4, Java nhận ra rằng chỉ có một phần tử trong mảng và chỉ mục 1 không được phép. Chương trình sẽ nén ra ngoại lệ được hiển thị:

```
Zoo
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: Index 1 out of bounds
for length 1
    at Zoo.main(Zoo.java:4)
```

b) Understanding exception types

Java có Throwable superclass cho tất cả các đối tượng đại diện cho những sự kiện này. Không phải tất cả chúng đều có từ exception trong tên lớp, điều này có thể gây nhầm lẫn.



Error có nghĩa là đã xảy ra sự cố nghiêm trọng đến mức chương trình của bạn không nên cố gắng khôi phục từ lỗi đó. Ví dụ: disk drive "biến mất" hoặc chương trình hết bộ nhớ. Đây là những tình trạng bất thường mà bạn khó có thể gặp phải và không thể phục hồi.

Throwable là nó là lớp cha của tất cả các ngoại lệ, bao gồm cả lớp Error. Mặc dù bạn có thể xử lý các Throwable và Error exceptions , nhưng bạn không nên làm như vậy trong code ứng dụng của mình. Trong chương này, khi đề cập đến các exception, chúng tôi thường muốn nói đến bất kỳ lớp nào kế thừa Throwable, mặc dù chúng tôi hầu như luôn làm việc với lớp Exception hoặc các subclass của nó.

Checked Exceptions

Checked Exceptions là exception phải được khai báo hoặc xử lý bằng application code nơi nó được ném ra (throw). Trong Java, tất cả các checked exceptions đều kế thừa Exception nhưng không có RuntimeException. Các trường hợp checked exceptions có xu hướng được dự đoán trước nhiều hơn - ví dụ: cố gắng đọc một file không tồn tại. Các checked exceptions cũng bao gồm bất kỳ lớp nào extends từ Throwable, nhưng không extends từ Error hoặc RuntimeException.

Java có một quy tắc gọi là quy tắc xử lý hoặc khai báo. Quy tắc xử lý hoặc khai báo có nghĩa là tất cả các checked exceptions có thể được đưa ra trong một phương thức đều được gói trong các try và catch block hoặc được khai báo trong chữ ký phương thức.

Bởi vì các checked exceptions có xu hướng được dự đoán trước, Java thực thi quy tắc rằng lập trình viên phải làm gì đó để cho thấy exceptions đã được nghĩ đến. Có lẽ nó đã được xử lý theo phương pháp. Hoặc có thể phương thức đó tuyên bố rằng nó không thể xử lý exceptions và người khác sẽ làm việc đó.

Chúng ta hãy xem một ví dụ. Phương thức fall() sau đây khai báo rằng nó có thể throws ra IOException, đây là một checked exceptions:

```
void fall(int distance)throws IOException{
    if(distance>10){
        throw new IOException();
    }
}
```

Lưu ý rằng bạn đang sử dụng hai từ khóa khác nhau ở đây. Từ khóa throw cho Java biết rằng bạn muốn nén ra một Exception, trong khi từ khóa throws chỉ khai báo rằng phương thức này có thể ném Exception. Nó cũng có thể không.

Bây giờ bạn đã biết cách khai báo một exception, thay vào đó bạn xử lý nó như thế nào? Phiên bản thay thế sau đây của phương thức fall() xử lý exception:

```
void fall(int distance) {
    try {
        if (distance > 10) {
            throw new IOException();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Lưu ý rằng câu lệnh **catch** sử dụng Exception chứ không phải IOException. Vì IOException là một subclass của Exception nên khối catch được phép bắt nó.

Unchecked Exceptions

Unchecked exceptions là bất kỳ exception nào không cần được khai báo hoặc xử lý bởi application code nơi nó được throw ra. Các unchecked exceptions thường được gọi là runtime exception, mặc dù trong Java, các unchecked exceptions bao gồm bất kỳ lớp nào kế thừa RuntimeException hoặc Error.

Runtime Exception được định nghĩa là lớp RuntimeException và các lớp con của nó. Các trường hợp runtime exception có xu hướng bất ngờ nhưng không nhất thiết gây tạm dừng hệ thống. Ví dụ: việc truy cập một index trong mảng không hợp lệ là điều không mong đợi. Mặc dù chúng kế thừa lớp Exception nhưng chúng không phải là checked exception.

Một unchecked exceptions thường có thể xảy ra trên hầu hết mọi dòng mã vì nó không bắt buộc phải xử lý hoặc khai báo. Ví dụ: một NullPointerException có thể được ném vào phần thân của phương thức sau nếu tham chiếu đầu vào là null:

```
void fall(String input) {
    System.out.println(input.toLowerCase());
}
```

Làm việc với các đối tượng trong Java thường xuyên nên ngoại lệ NullPointerException có thể xảy ra ở hầu hết mọi nơi. Nếu bạn phải khai báo các unchecked exceptions ở mọi nơi thì mọi phương thức sẽ có sự lộn xộn đó!

c) Throwing an exception

Bất kỳ code Java nào cũng có thể đưa ra một exception; điều này bao gồm code bạn viết.

Đầu tiên là code sai sẽ throw ra exception. Đây là một ví dụ:

```
String[] animals = new String[0];
System.out.println(animals[0]);
```

Code này throw ra một ngoại lệ ArrayIndexOutOfBoundsException vì mảng không có phần tử nào.

Cách thứ hai để mã dẫn đến một exception là yêu cầu Java throw một exception một cách rõ ràng. Java cho phép bạn viết các câu lệnh như sau:

```
    throw new Exception();
    throw new Exception("Ow! I fell.");
    throw new RuntimeException();
    throw new RuntimeException("Ow! I fell.");
```

Từ khóa **throw** cho Java biết bạn muốn một phần khác của code xử lý exception. Người khác cần phải tìm ra những gì cần làm về exception.

Khi tạo một exception, bạn thường có thể truyền tham số String bằng một message hoặc bạn có thể không truyền tham số nào và sử dụng các giá trị mặc định. Chúng tôi nói thường vì đây là một quy ước. Ai đó có thể tạo một class exception không có constructor nhận message. Hai ví dụ đầu tiên tạo một đối tượng mới thuộc loại Exception và throw nó. Hai điều cuối cùng cho thấy code trông giống nhau bất kể bạn đưa ra loại exception nào.

Ngoài ra, bạn nên biết rằng Exception là Đối tượng. Điều này có nghĩa là bạn có thể lưu trữ trong một biến và điều này là hợp pháp:

```
Exception e = new RuntimeException();
throw e;
```

Bạn có thể thấy tại sao phần sau không được biên dịch không?

```
3:      try {
4:          throw new RuntimeException();
5:          throw new ArrayIndexOutOfBoundsException(); // DOES NOT COMPILE
6:      } catch (Exception e) {
7:      }
```

Vì dòng 4 đưa ra một exception nên không bao giờ có thể đạt được dòng 5 trong thời gian chạy. Trình biên dịch nhận ra điều này và báo cáo lỗi mã không thể truy cập được.

Types of exceptions and errors

Kiểu	Làm thế nào để nhận biết	Để chương trình catch?	Chương trình có cần thiết để xử lý hoặc khai báo không?
Runtime exception	Subclass c RuntimeException	Yes	No
Checked exception	Subclass of Exception but not subclass of RuntimeException	Yes	Yes
Error	Subclass of Error	No	No

2. Recognizing Exception Classes

a) RuntimeException classes

RuntimeException và các lớp con của nó là các exception không được kiểm tra và không cần phải xử lý hoặc khai báo. Chúng có thể được lập trình viên hoặc JVM throw ra. Các lớp RuntimeException phổ biến bao gồm:

- **ArithmeticException:** Được throw ra khi code cố gắng chia cho 0
- **ArrayIndexOutOfBoundsException:** throw ra khi code sử dụng index không hợp lệ để truy cập vào một mảng
- **ClassCastException:** throw ra khi cố gắng truyền một đối tượng sang một class mà nó không phải là một instance
- **NullPointerException**: throw ra khi có một tham chiếu null trong đó cần có một đối tượng
- **IllegalArgumentException:** Được lập trình viên throw ra để chỉ ra rằng một phương thức đã được thông qua một đối số không hợp lệ hoặc không phù hợp
- **NumberFormatException:** Lớp con của IllegalArgumentException được throw ra khi cố gắng chuyển đổi một chuỗi thành kiểu số nhưng chuỗi không có định dạng phù hợp

ArithmeticException

Cố gắng chia một int cho 0 sẽ cho kết quả không xác định. Khi điều này xảy ra, JVM sẽ ném ra một ngoại lệ ArithmeticException:\

```
int answer = 11 / 0;
```

Chạy mã này sẽ dẫn đến kết quả đầu ra sau:

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

ArrayIndexOutOfBoundsException

Bây giờ bạn đã biết rằng các index của mảng bắt đầu bằng 0 và nhỏ hơn length - 1 so với độ dài của mảng - điều đó có nghĩa là code này sẽ đưa ra một exception ArrayIndexOutOfBoundsException:

```
int[] countsOfMoose = new int[3];
System.out.println(countsOfMoose[-1]);
```

Đây là một vấn đề vì không có thứ gọi là index mảng âm. Chạy mã này mang lại kết quả đầu ra sau:

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException:
Index -1 out of bounds for length 3
```

Ít nhất Java cho chúng ta biết chỉ mục nào không hợp lệ. Bạn có thể thấy điều gì sai với cái này không?

```
int total = 0;
int[] countsOfMoose = new int[3];
for (int i = 0; i <= countsOfMoose.length; i++)
    total += countsOfMoose[i];
```

Vấn đề là vòng lặp for nên có < thay vì <=. Ở lần lặp cuối cùng của vòng lặp, Java cố gắng gọi countOfMoose[3], kết quả này không hợp lệ. Mảng chỉ bao gồm ba phần tử, Output trông như thế này:

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException:
Index 3 out of bounds for length 3
```

ClassCastException

Java cố gắng bảo vệ bạn khỏi những diễn biến bất khả thi. Mã này không biên dịch được vì S Integer không phải là subclass của Chuỗi:

```
String type = "moose";
Integer number = (Integer) type; // DOES NOT COMPILE
```

Code phức tạp hơn sẽ cản trở nỗ lực bảo vệ bạn của Java. Khi quá trình truyền không thành công trong runtime, Java sẽ throw ra ClassCastException:

```
String type = "moose";
Object obj = type;
Integer number = (Integer) obj;
```

Trình biên dịch sẽ thấy sự chuyển đổi từ Object sang Integer. Điều này có thể ổn. Trình biên dịch không nhận ra có một String trong Object đó. Khi mã chạy, nó mang lại kết quả đầu ra sau:

```
Exception in thread "main" java.lang.ClassCastException:
java.base/java.lang.String
cannot be cast to java.lang.base/java.lang.Integer
```

NullPointerException

Các biến instance và phương thức phải được gọi trên một tham chiếu khác null. Nếu tham chiếu là null, JVM sẽ ném ra ngoại lệ NullPointerException. Chẳng hạn như trong ví dụ sau:

```
String name;
public void printLength() {
    System.out.println(name.length());
}
```

Chạy mã này sẽ dẫn đến kết quả đầu ra này:

```
Exception in thread "main" java.lang.NullPointerException
```

IllegalArgumentException

```
public class Student {
    int m;
    public void setMarks(int marks) {
        if(marks < 0 || marks > 100)
            throw new IllegalArgumentException(Integer.toString(marks));
        else
            m = marks;
    }
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.setMarks(45);
        System.out.println(s1.m);
        Student s2 = new Student();
        s2.setMarks(101);
        System.out.println(s2.m);
    }
}
```

Output:

```
45
Exception in thread "main" java.lang.IllegalArgumentException: 101
at Student.setMarks(Student.java:5)
at Student.main(Student.java:14)
```

NumberFormatException

Java cung cấp các phương thức chuyển đổi string thành number. Khi những giá trị này được truyền một giá trị không hợp lệ, chúng sẽ ném ra một ngoại lệ NumberFormatException.

Đây là một ví dụ về việc cố gắng để chuyển đổi một cái gì đó không phải là số thành int:

```
Integer.parseInt("abc");
```

Đầu ra trông như thế này:

```
Exception in thread "main"
java.lang.NumberFormatException: For input string: "abc"
```

b) Checked exception classes

Checked exception có Exception trong hệ thống phân cấp của chúng nhưng không có RuntimeException. Chúng phải được xử lý hoặc khai báo. Các trường hợp ngoại lệ được kiểm tra phổ biến bao gồm:

- **IOException**: Được throw theo chương trình khi có sự cố khi đọc hoặc ghi tệp
- **FileNotFoundException:** Lớp con của IOException được throw theo chương trình khi mã cố gắng tham chiếu một tệp không tồn tại

c) Error classes

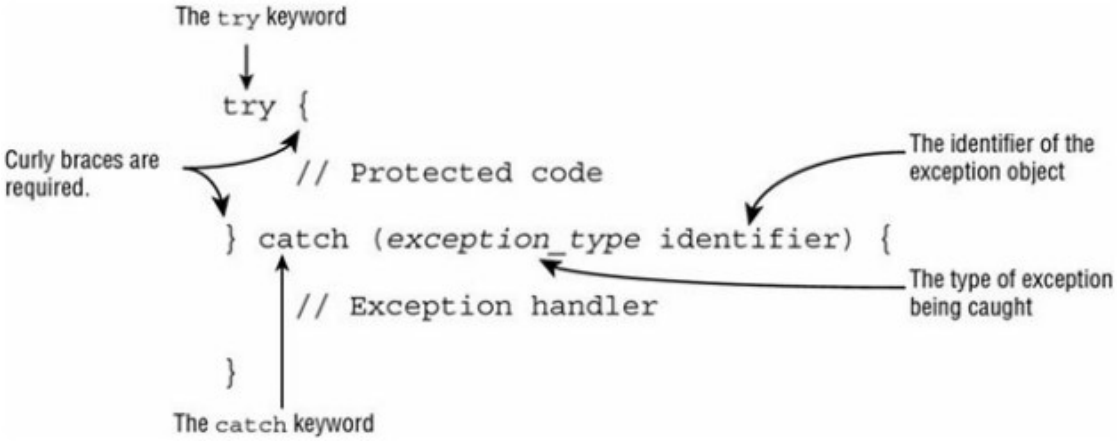
Error là các unchecked exceptions mở rộng class **Error**. Chúng được JVM throw ra và không được xử lý hoặc khai báo. Lỗi rất hiếm nhưng bạn có thể thấy những lỗi sau:

- **ExceptionInInitializerError:** Được throw ra khi trình khởi tạo tĩnh ném ra một ngoại lệ và không xử lý nó
- **StackOverflowError:** Xảy ra khi một phương thức gọi chính nó quá nhiều lần (Điều này được gọi là đệ quy vô hạn vì phương thức này thường gọi chính nó mà không có kết thúc.)
- **NoClassDefFoundError:** Được throw ra khi một class mà code sử dụng có sẵn tại thời điểm biên dịch nhưng không có sẵn tại thời điểm chạy

3. Handling Exceptions

a) Using *try* and *catch* statements

Bây giờ bạn đã biết exception là gì, hãy khám phá cách xử lý chúng. Java sử dụng câu lệnh **try** để tách logic có thể đưa ra một exception khỏi logic để xử lý exception đó.



Code trong **try block** được chạy bình thường. Nếu bất kỳ câu lệnh nào throw ra một exception có thể bị bắt bởi loại exception được liệt kê trong catch block, thì try block sẽ dừng chạy và việc thực thi sẽ chuyển sang catch block. Nếu không có câu lệnh nào trong try block đưa ra một exception có thể **catch** được thì mệnh đề catch sẽ không được chạy.

Ví dụ:

```
3:  void explore() {
4:    try {
5:      fall();
6:      System.out.println("never get here");
7:    } catch (RuntimeException e) {
8:      getUp();
9:    }
10:   seeAnimals();
11: }
12: void fall() { throw new RuntimeException(); }
```

Đầu tiên, dòng 5 gọi phương thức fall(). Dòng 12 ném một exception. Điều này có nghĩa là Java nhảy thẳng tới **catch** block, bỏ qua dòng 6. getUp() được gọi dòng 8. Bây giờ câu lệnh **try** đã kết thúc và quá trình thực thi diễn ra bình thường với dòng 10.

Cái này thì sao?

```
try { // DOES NOT COMPILE
    fall();
}
```

Code này không biên dịch được vì **try block** không có gì sau nó. Hãy nhớ rằng mục đích của câu lệnh try là để điều gì đó xảy ra nếu một exception được throw ra. Nếu không có mệnh đề khác, câu lệnh try sẽ đơn độc.

b) Chaining *catch* blocks

Bạn chỉ bắt được một loại exception. Bây giờ hãy xem điều gì sẽ xảy ra khi các loại exception khác nhau có thể được đưa ra từ cùng một try/catch block.

Đầu tiên, bạn phải có khả năng nhận biết exception đó là checked hay unchecked exception. Thứ hai, bạn cần xác định xem có bất kỳ trường hợp exception nào là subclasses của các trường hợp ngoại lệ khác không:

```
class AnimalsOutForAWalk extends RuntimeException { }
class ExhibitClosed extends RuntimeException { }
class ExhibitClosedForLunch extends ExhibitClosed { }
```

Trong ví dụ này, có ba trường hợp exception tùy chỉnh. Tất cả đều là unchecked exception vì chúng trực tiếp hoặc gián tiếp extends RuntimeException. Bây giờ chúng ta xâu chuỗi cả hai loại exception bằng hai catch block và xử lý chúng bằng cách in ra thông báo thích hợp:

```
public void visitPorcupine() {
    try {
        seeAnimal();
    } catch (AnimalsOutForAWalk e) { // first catch block
        System.out.print("try back later");
    } catch (ExhibitClosed e) { // second catch block
        System.out.print("not today");
    }
}
```

Có ba khả năng khi code này được chạy. Nếu seeAnimal() không đưa ra exception thì sẽ không có gì được in ra. Nếu throw ra AnimalsOutForAWalk exception, chỉ có catch block đầu tiên chạy. Nếu throw ra ExhibitClosed exception, chỉ catch block thứ hai chạy. Cả hai catch block không thể được thực thi khi được nối với nhau như thế này.

Có một quy tắc cho thứ tự của các catch block. Java xem xét chúng theo thứ tự chúng xuất hiện. Nếu một trong các catch block không thể được thực thi thì sẽ xảy ra lỗi trình biên dịch về code không thể truy cập được.

Ví dụ sau đây cho thấy các loại exception kế thừa lẫn nhau:

```
public void visitMonkeys() {
    try {
        seeAnimal();
    } catch (ExhibitClosedForLunch e) { // subclass exception
        System.out.print("try back later");
    } catch (ExhibitClosed e) { // superclass exception
        System.out.print("not today");
    }
}
```

Nếu ExhibitClosedForLunch exception cụ thể hơn được throw ra, catch block đầu tiên sẽ chạy. Nếu không, Java sẽ kiểm tra xem superclass ExhibitClosed exception có được throw ra hay không và bắt giữ nó. Lần này, thứ tự của các superclass có vấn đề. Điều ngược lại không hoạt động.

```
public void visitMonkeys() {
    try {
        seeAnimal();
    } catch (ExhibitClosed e) {
        System.out.print("not today");
    } catch (ExhibitClosedForLunch e) { // DOES NOT COMPILE
        System.out.print("try back later");
    }
}
```

Lần này, nếu ExhibitClosedForLunch exception cụ thể hơn được throw ra, **catch block** cho ExhibitClosed sẽ chạy-có nghĩa là không có cách nào để catch block thứ hai chạy. Java cho bạn biết chính xác rằng có một **catch block** không thể truy cập được.

Hãy thử điều này một lần nữa. Bạn có thấy tại sao mã này không biên dịch được không?

```
public void visitSnakes() {
    try {
    } catch (IllegalArgumentException e) {
    } catch (NumberFormatException e) { // DOES NOT COMPILE
    }
}
```

Hãy nhớ rằng NumberFormatException là một subclass của IllegalArgumentException? Ví dụ này là lý do tại sao ? Vì NumberFormatException là một subclass nên nó sẽ luôn bị chặn bởi **catch block** đầu tiên, khiến code không thể truy cập được ở **catch block** thứ hai và không biên dịch được.

c) Applying a multi-catch block

Thông thường, chúng ta muốn kết quả của một exception được throw ra giống nhau, bất kể exception cụ thể nào được throw ra. Ví dụ: hãy xem phương pháp này:

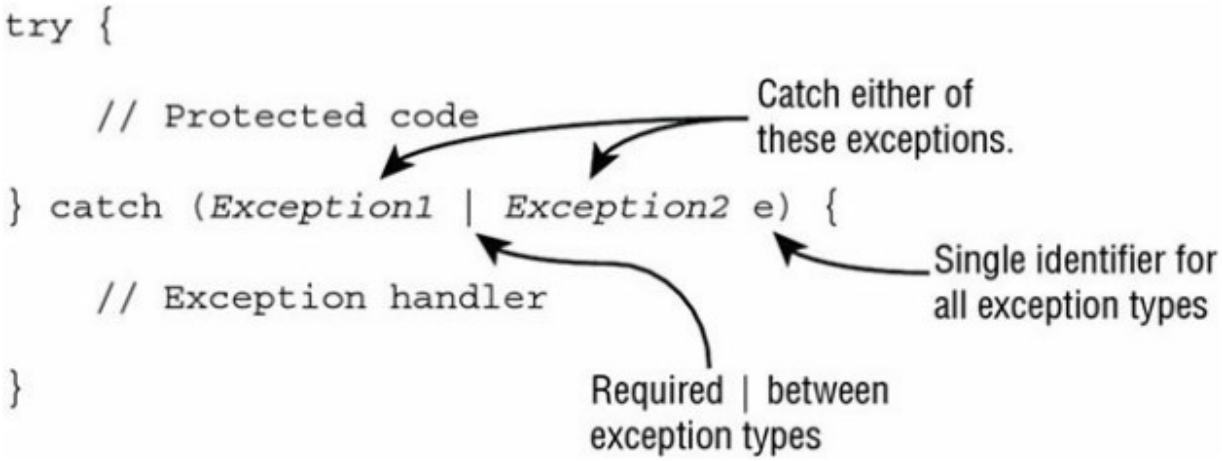
```
public static void main(String args[]) {
    try {
        System.out.println(Integer.parseInt(args[1]));
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Missing or invalid input");
    } catch (NumberFormatException e) {
        System.out.println("Missing or invalid input");
    }
}
```

Lưu ý rằng chúng ta có cùng một câu lệnh println() cho hai catch block khác nhau. Làm thế nào bạn có thể giảm mã trùng lặp

Java cung cấp một cấu trúc khác để xử lý khối này một cách tốt hơn được gọi là multi-catch block. Multi-catch block cho phép nhiều loại exception được catch bởi cùng một catch block. Hãy viết lại ví dụ trước bằng cách sử dụng khối multi-catch:

```
public static void main(String[] args) {
    try {
        System.out.println(Integer.parseInt(args[1]));
    } catch (ArrayIndexOutOfBoundsException | NumberFormatException e) {
        System.out.println("Missing or invalid input");
    }
}
```

Không có mã trùng lặp, logic chung đều ở một nơi và logic chính xác là nơi bạn mong muốn tìm thấy nó.



Cú pháp của multi-catch block

Hãy nhớ rằng các exception có thể được liệt kê theo bất kỳ thứ tự nào trong catch. Tuy nhiên, tên biến chỉ được xuất hiện một lần và ở cuối. Bạn có thấy tại sao những điều này hợp lệ hay không hợp lệ?

```
catch(Exception1 e | Exception2 e | Exception3 e) // DOES NOT COMPILE

catch(Exception1 e1 | Exception2 e2 | Exception3 e3) // DOES NOT COMPILE

catch(Exception1 | Exception2 | Exception3 e)
```

Dòng đầu tiên sai vì tên biến xuất hiện 3 lần. Chỉ vì nó có cùng tên biến thì không ổn. Dòng thứ hai sai vì đặt 3 biến tên khác nhau.

Java dự định sử dụng multi-catch cho các trường hợp exception không liên quan và nó ngăn bạn chỉ định các loại dư thừa trong multi-catch. Bạn có thấy điều gì sai ở đây không?

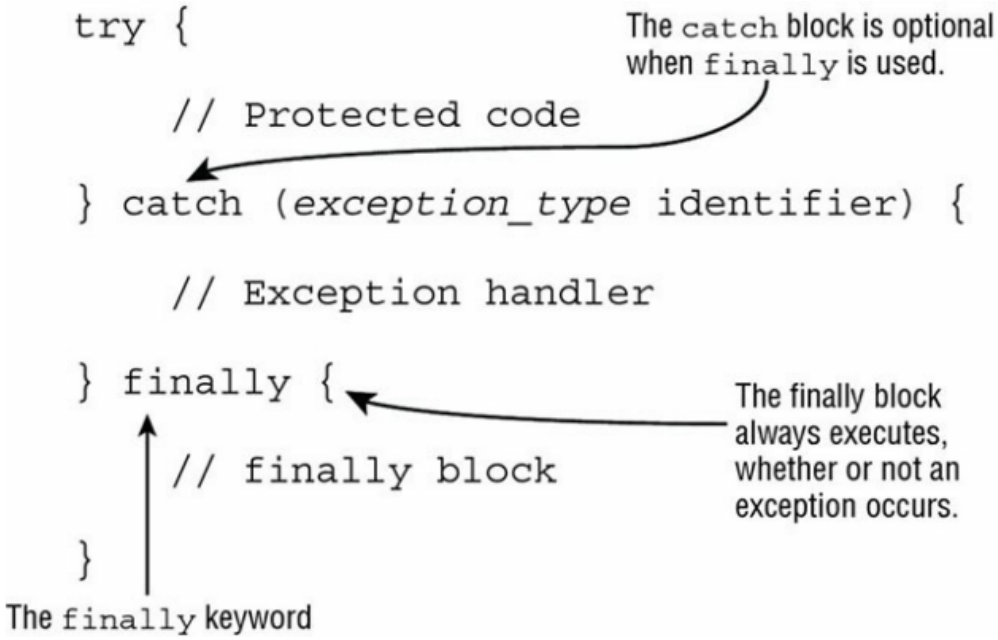
```
try {
    throw new IOException();
} catch (FileNotFoundException | IOException p) {} // DOES NOT COMPILE
```

Việc chỉ định nó trong multi-catch là không cần thiết và trình biên dịch sẽ đưa ra một thông báo như sau:

```
The exception FileNotFoundException is already caught by the
alternative IOException
```

d) Adding a *finally* block

Câu lệnh try cũng cho phép bạn chạy code ở cuối bằng finally clause bất kể có ném ngoại lệ hay không:



Nếu một exception được throw ra, finally block sẽ chạy sau catch block. Nếu không có ngoại lệ nào được đưa ra, finally block sẽ được chạy sau khi try block hoàn thành.

Hãy quay trở lại ví dụ:

```
12: void explore() {
13:   try {
14:     seeAnimals();
15:     fall();
16:   } catch (Exception e) {
17:     getHugFromDaddy();
18:   } finally {
19:     seeMoreAnimals();
20:   }
21:   goHome();
22: }
```

⇨ Dù thế nào đi nữa, finally block được thực thi và việc thực thi tiếp tục sau câu lệnh try.

Bạn có thấy tại sao những điều sau đây thực hiện hoặc không biên dịch không?

```
25: try { // DOES NOT COMPILE
26:   fall();
27: } finally {
28:   System.out.println("all better");
29: } catch (Exception e) {
30:   System.out.println("get up");
31: }
32:
33: try { // DOES NOT COMPILE
34:   fall();
35: }
36:
37: try {
38:   fall();
39: } finally {
40:   System.out.println("all better");
41: }
```

⇨ Ví dụ đầu tiên (dòng 25-31) không biên dịch được vì các catch và finally blocks không đúng thứ tự. Ví dụ thứ hai (dòng 33-35) không biên dịch vì phải có catch và finally blocks. Ví dụ thứ ba (dòng 37-41) là ổn. catch block không cần thiết nếu có finally.

Có một quy tắc bổ sung, nếu một câu lệnh try có finally blocks được nhập vào thì finally blocks sẽ luôn được thực thi, bất kể mã có hoàn thành thành công hay không.

Hãy xem phương thức goHome() sau đây. Giả sử một exception có thể được throw ra hoặc không ở dòng 14, thì phương thức này có thể in ra những giá trị nào? Ngoài ra, giá trị trả về trong mỗi trường hợp sẽ là bao nhiêu?

```
12: int goHome() {
13:   try {
14:     // Optionally throw an exception here
15:     System.out.print("1");
16:     return -1;
17:   } catch (Exception e) {
18:     System.out.print("2");
19:     return -2;
20:   } finally {
21:     System.out.print("3");
22:     return -3;
23:   }
24: }
```

⇨ Phương thức luôn in 3 giá trị cuối cùng vì nó nằm trong catch block.

Hãy cho biết output ở đây là gì ?

```
public class Animal {
    static String fall(String input) {
        try{
           throw new RuntimeException();
        }catch (RuntimeException e){
            return "RuntimeException";
        }finally {
            return "123";
        }
    }

    public static void main(String[] args) {
        System.out.println(fall("123"));
    }
}
```

e) Finally closing resources

Thông thường, ứng dụng của bạn hoạt động với các tệp, cơ sở dữ liệu và các đối tượng kết nối khác nhau. Thông thường, những nguồn dữ liệu bên ngoài này được gọi là **resources** - tài nguyên. Trong nhiều trường hợp, bạn mở một kết nối tới tài nguyên, cho dù đó là qua mạng hay trong hệ thống tệp. Sau đó bạn đọc/ghi dữ liệu bạn muốn. Cuối cùng, bạn đóng tài nguyên để cho biết bạn đã hoàn tất việc đó.

Điều gì xảy ra nếu bạn không đóng tài nguyên khi bạn đã hoàn tất việc đó? Nếu bạn đang kết nối với cơ sở dữ liệu, bạn có thể sử dụng hết tất cả các kết nối có sẵn, nghĩa là không ai có thể nói chuyện với cơ sở dữ liệu cho đến khi bạn giải phóng kết nối của mình. Mặc dù bạn thường nghe về rò rỉ bộ nhớ là nguyên nhân khiến chương trình bị lỗi, nhưng rò rỉ tài nguyên cũng tệ hại không kém và xảy ra khi một chương trình không giải phóng được kết nối của nó tới một tài nguyên, dẫn đến tài nguyên không thể truy cập được.

Viết mã để đơn giản hóa việc đóng tài nguyên chính là nội dung của phần này. Chúng ta hãy xem một phương thức mở tệp, đọc dữ liệu và đóng nó:

```
4:  public void readFile(String file) {
5:    FileInputStream is = null;
6:    try {
7:      is = new FileInputStream("myfile.txt");
8:      // Read file data
9:    } catch (IOException e) {
10:     e.printStackTrace();
11:   } finally {
12:     if(is != null) {
13:       try {
14:         is.close();
15:       } catch (IOException e2) {
16:         e2.printStackTrace();
17:       }
18:     }
19:   }
20: }
```

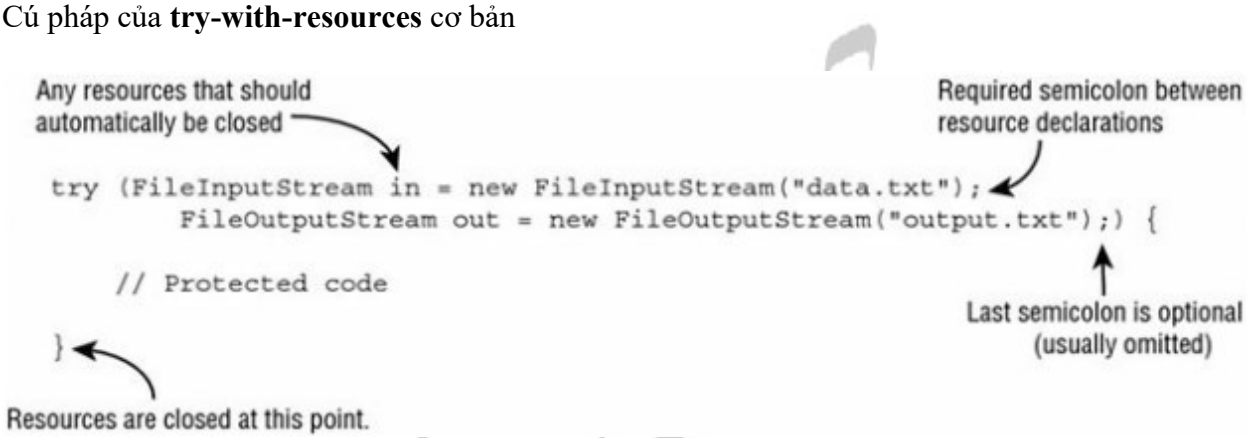
Để giải quyết vấn đề này, Java bao gồm câu lệnh try-with-resources để tự động đóng tất cả các tài nguyên được mở trong mệnh đề try. Tính năng này còn được gọi là quản lý tài nguyên tự động, vì Java tự động đảm nhiệm việc đóng.

Chúng ta hãy xem ví dụ tương tự của chúng tôi bằng cách sử dụng câu lệnh try-with-resources:

```
4:  public void readFile(String file) {
5:    try (FileInputStream is = new FileInputStream("myfile.txt")) {
6:    // Read file data
7:  } catch (IOException e) {
8:    e.printStackTrace();
9:  }
10: }
```

Basics of Try-with-Resources

Cú pháp của **try-with-resources** cơ bản



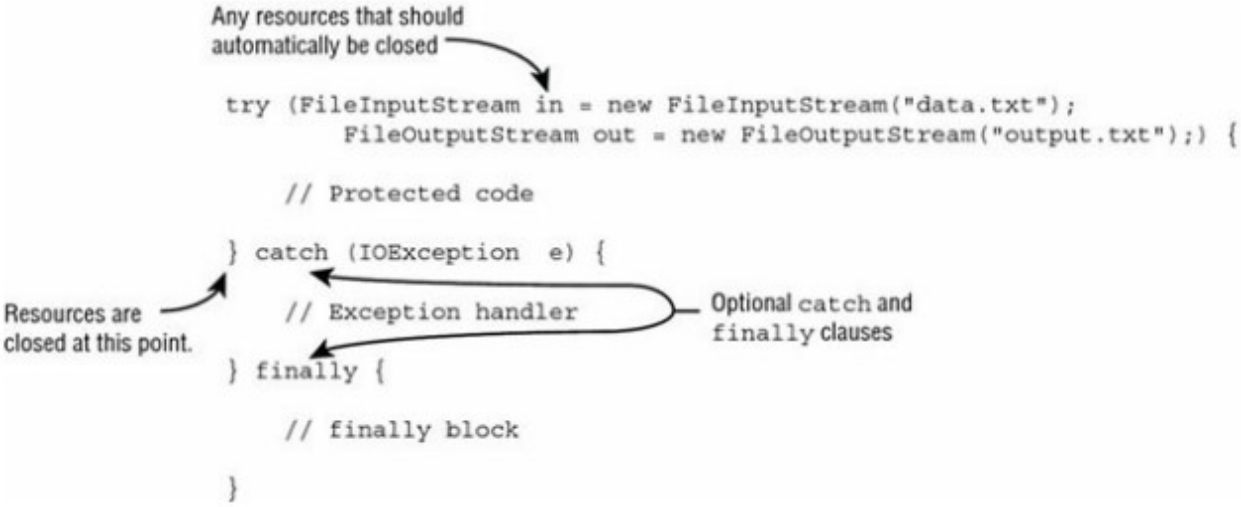
Lưu ý rằng một hoặc nhiều tài nguyên có thể được mở trong mệnh đề try. Khi có nhiều tài nguyên được mở, chúng sẽ được đóng theo thứ tự ngược lại với thứ tự chúng được tạo. Ngoài ra, hãy lưu ý rằng dấu ngoặc đơn được sử dụng để liệt kê các tài nguyên đó và dấu chấm phẩy được sử dụng để phân tách các phần khai báo. Điều này hoạt động giống như khai báo nhiều chỉ mục trong vòng lặp for.

Ví dụ: chúng ta có thể viết lại ví dụ readFile() trước đó để phương thức lấy lại exception để làm cho nó ngắn hơn nữa:

```
4: public void readFile(String file) throws IOException {
5:   try (FileInputStream is = new FileInputStream("myfile.txt")) {
6:     // Read file data
7:   }
8: }
```

Bạn đã biết rằng câu lệnh try phải có một hoặc nhiều catch blocks hoặc finally block. Điều này vẫn đúng. Mệnh đề cuối cùng tồn tại ngầm. Bạn không cần phải gõ nó.

Cú pháp của try-with-resources bao gồm cả catch/finally



Nếu code trong **try block** ném ra một checked exception không được khai báo bằng phương thức mà nó được xác định hoặc xử lý bởi một try/catch block khác, thì nó sẽ cần được xử lý bởi catch block. Ngoài ra, các catch block và finally được chạy cùng với khối ngầm đóng tài nguyên.

Cấu hình hợp lệ và không hợp lệ với câu lệnh try truyền thống:

	0 finally blocks	**1 finally block**	**2 or more finally blocks**
0 catch blocks	Not legal	Legal	Not legal
1 or more catch blocks	Legal	Legal	Not legal

Cấu hình hợp lệ và không hợp lệ với câu lệnh try-withresource

	0 finally blocks	**1 finally block**	**2 or more finally blocks**
0 catch blocks	Legal	Legal	Not legal
1 or more catch blocks	Legal	Legal	Not legal

Declaring Resources

Mặc dù try-with-resource hỗ trợ khai báo nhiều biến nhưng mỗi biến phải được khai báo trong một câu lệnh riêng. Ví dụ: phần sau không được biên dịch:

```
try (MyFileClass is = new MyFileClass(1), // DOES NOT COMPILE
    os = new MyFileClass(2)) {
}

try (MyFileClass ab = new MyFileClass(1), // DOES NOT COMPILE
    MyFileClass cd = new MyFileClass(2)) {
}
```

⇨ Câu lệnh try-with-resources không hỗ trợ khai báo nhiều biến. Ví dụ đầu tiên không biên dịch được vì nó thiếu kiểu dữ liệu và nó sử dụng dấu phẩy (,) thay vì dấu chấm phẩy (;). Ví dụ thứ hai không biên dịch được vì nó cũng sử dụng dấu phẩy (,) thay vì dấu chấm phẩy (;). Mỗi tài nguyên phải bao gồm loại dữ liệu và được phân tách bằng dấu chấm phẩy (;).

Bạn có thể khai báo một resources bằng cách sử dụng var làm kiểu dữ liệu trong câu lệnh try-with-resources, vì resource là các biến local.

```
try (var f = new BufferedInputStream(new
    FileInputStream("it.txt"))) {
    // Process file
}
```

Scope of Try-with-Resources

Các tài nguyên được tạo trong mệnh đề try chỉ nằm trong phạm vi try block. Đây là một cách khác để nhớ rằng hàm implicit finally sẽ chạy trước bất kỳ catch/finally block nào mà bạn tự viết mã. Lệnh close ngầm đã chạy và tài nguyên không còn khả dụng nữa. Bạn có hiểu tại sao dòng 6 và 8 không biên dịch được trong ví dụ này không?

```
3: try (Scanner s = new Scanner(System.in)) {
4:   s.nextLine();
5: } catch(Exception e) {
6:   s.nextInt(); // DOES NOT COMPILE
7: } finally {
8:   s.nextInt(); // DOES NOT COMPILE
9: }
```

⇨ Vấn đề là Scanner đã vượt quá phạm vi ở cuối mệnh đề try. Dòng 6 và 8 không có quyền truy cập vào nó.

Theo thứ tự hoạt động:

Hai quy tắc mới về thứ tự chạy code trong câu lệnh trywith-resources:

- Tài nguyên bị đóng sau khi mệnh đề try kết thúc và trước bất kỳ mệnh đề catch/finally nào.
- Tài nguyên được đóng theo thứ tự ngược lại với thứ tự chúng được tạo.

Hãy xem lại những nguyên tắc này bằng một ví dụ phức tạp hơn. Đầu tiên, chúng tôi xác định một class tùy chỉnh mà bạn có thể sử dụng với câu lệnh try-with-resources vì nó triển khai tính năng **AutoCloseable**.

```
public class MyFileClass implements AutoCloseable {
    private final int num;
    public MyFileClass(int num) { this.num = num; }
    public void close() {
        System.out.println("Closing: " + num);
    }
}
```

Dựa trên những quy tắc này, bạn có thể tìm ra phương thức này in ra cái gì không?

```
public static void main(String... xyz) {
    try (MyFileClass a1 = new MyFileClass(1);
         MyFileClass a2 = new MyFileClass(2)) {
        throw new RuntimeException();
    } catch (Exception e) {
        System.out.println("ex");
    } finally {
        System.out.println("finally");
    }
}
```

⇨ Closing: 2 Closing: 1 ex finally

f) Throwing additional exceptions

catch hay finally block có thể chứa bất kỳ mã Java hợp lệ nào trong đó - bao gồm cả một câu lệnh try khác. Điều gì xảy ra khi một ngoại lệ được throw bên trong catch hoặc finally block?

Để trả lời điều này, chúng ta hãy xem một ví dụ cụ thể:

```
16: public static void main(String[] a) {
17:   FileReader reader = null;
18:   try {
19:     reader = read();
20:   } catch (IOException e) {
21:     try {
22:       if (reader != null) reader.close();
23:     } catch (IOException inner) {
24:     }
25:   }
26: }
27: private static FileReader read() throws IOException {
28:   // CODE GOES HERE
29: }
```

Điều này cho thấy rằng chỉ có exception cuối cùng được đưa ra là có vấn đề:

```
26: try {
27:   throw new RuntimeException();
28: } catch (RuntimeException e) {
29:   throw new RuntimeException();
30: } finally {
31:   throw new Exception();
32: }
```

⇨ finally block chạy sau catch block. Vì finally block ném exception của chính nó vào dòng 31 nên khối này sẽ bị throw. Exception từ try block bị lãng quên.

4. Calling Methods That Throw Exceptions

Khi bạn gọi một phương thức đưa ra **exception**, các quy tắc cũng giống như trong một phương thức. Bạn có biết tại sao phần sau không được biên dịch không?

```
class NoMoreCarrotsException extends Exception {}

public class Bunny {
    public static void main(String[] args) {
        eatCarrot(); // DOES NOT COMPILE
    }
    private static void eatCarrot() throws NoMoreCarrotsException {}
}
```

Vấn đề là NoMoreCarrotsException là một checked exception. Các trường hợp checked exception phải được xử lý hoặc khai báo. Mã sẽ biên dịch nếu bạn thay đổi phương thức main() thành một trong hai phương thức sau:

```
public static void main(String[] args)
        throws NoMoreCarrotsException { // declare exception
    eatCarrot();
}

public static void main(String[] args) {
    try {
        eatCarrot();
    } catch (NoMoreCarrotsException e) { // handle exception
        System.out.print("sad rabbit");
    }
}
```

Bạn có thể nhận thấy rằng eatCarrot() thực sự không đưa ra một exception; nó chỉ tuyên bố rằng nó có thể. Điều này là đủ để trình biên dịch yêu cầu người gọi xử lý hoặc khai báo exception.

Trình biên dịch vẫn đang tìm kiếm mã không thể truy cập được. Khai báo một exception không được sử dụng không được coi là mã không thể truy cập được. Nó cung cấp cho phương thức tùy chọn thay đổi cách triển khai để đưa ra exception đó trong tương lai. Bạn có thấy vấn đề ở đây không?

```
public void bad() {
    try {
        eatCarrot();
    } catch (NoMoreCarrotsException e) { // DOES NOT COMPILE
        System.out.print("sad rabbit");
    }
}

public void good() throws NoMoreCarrotsException {
    eatCarrot();
}

private void eatCarrot() {}
```

Java biết rằng eatCarrot() không thể đưa ra một exception đã được kiểm tra - điều đó có nghĩa là không có cách nào để đạt được catch block trong bad(). Để so sánh, good() có thể tự do khai báo các exception khác.

1. Declaring and overriding methods with exceptions

Khi một lớp ghi đè một phương thức từ superclass hoặc implements một phương thức từ một interface, nó không được phép thêm các checked exceptions mới vào chữ ký phương thức. Ví dụ: mã này không được phép:

```
class CanNotHopException extends Exception {}
class Hopper {
    public void hop() {}
}
class Bunny extends Hopper {
    public void hop() throws CanNotHopException {} // DOES NOT COMPILE
}
```

Java biết hop() không được phép đưa ra bất kỳ checked exceptions nào vì phương thức hop() trong superclass Hopper không khai báo bất kỳ ngoại lệ nào.

Một phương thức bị ghi đè trong một subclass được phép khai báo ít exceptions hơn so với superclass hoặc interface. Điều này là hợp lệ vì người gọi đã xử lý chúng.

```
class Hopper {
    public void hop() throws CanNotHopException { }
}
class Bunny extends Hopper {
    public void hop() { }
}
```

Một phương thức bị ghi đè không khai báo một trong các exception do phương thức gốc đưa ra cũng tương tự như phương thức khai báo nó ném ra một exception mà nó không bao giờ thực sự ném ra. Điều này là hoàn toàn hợp pháp.

Tương tự, một lớp được phép khai báo một lớp con có kiểu ngoại lệ. Ý tưởng là như nhau. Siêu lớp hoặc giao diện đã xử lý một loại rộng hơn. Đây là một ví dụ:

```
class CanNotHopException extends Exception {}

class Hopper {
    public void hop() throws Exception { }
}

class Bunny extends Hopper {
    public void hop() throws CanNotHopException { }
}
```

Bunny có thể tuyên bố rằng nó ném Exception trực tiếp hoặc có thể tuyên bố rằng nó ném một loại Exception cụ thể hơn. Nó thậm chí có thể tuyên bố rằng nó không ném gì cả.

Quy tắc này chỉ áp dụng cho các trường hợp checked exception. Đoạn mã sau là hợp lệ vì nó có unchecked exception trong phiên bản của lớp con:

```
class Hopper {
    public void hop() { }
}
class Bunny extends Hopper {
    public void hop() throws IllegalStateException { }
}
```

2. Printing an exception

Có ba cách để in một exception. Bạn có thể để Java in nó ra, chỉ in message hoặc in nơi xuất phát dấu vết ngăn xếp - **stack trace**. Ví dụ này cho thấy cả ba cách tiếp cận:

```
5:  public static void main(String[] args) {
6:    try {
7:      hop();
8:    } catch (Exception e) {
9:      System.out.println(e);
10:     System.out.println(e.getMessage());
11:     e.printStackTrace();
12:   }
13: }
14: private static void hop() {
15:   throw new RuntimeException("cannot hop");
16: }
```

Mã này dẫn đến kết quả đầu ra sau:

```
java.lang.RuntimeException: cannot hop
cannot hop
java.lang.RuntimeException: cannot hop
    at Handling.hop(Handling.java:15)
    at Handling.main(Handling.java:7)
```

Dòng đầu tiên hiển thị những gì Java in ra theo mặc định: loại exception và message. Dòng thứ hai chỉ hiển thị message. Phần còn lại hiển thị stack trace.

Stack trace thường là dấu vết hữu ích nhất vì nó là hình ảnh kịp thời vào thời điểm exception được đưa ra. Nó hiển thị thứ bậc của các lệnh gọi phương thức đã được thực hiện để đến dòng đã đưa ra exception.

Stack trace hiển thị tất cả các phương thức trên ngăn xếp. Mỗi khi bạn gọi một phương thức, Java sẽ thêm nó vào ngăn xếp cho đến khi hoàn thành. Khi một exception được ném ra, nó sẽ đi qua ngăn xếp cho đến khi tìm thấy một phương thức có thể xử lý nó hoặc nó hết ngăn xếp.