Java Program Anatomy

Package là gì ?

Một **package (gói) trong java** là một nhóm các kiểu tương tự của các lớp, giao diện và các package con. Package được sử dụng để phân loại lớp và interface giúp dễ dàng bảo trì, cung cấp bảo vệ truy cập và khắc phục được việc đặt trùng tên.

Tại sao cần import package ?

import trong Java là từ khóa sử dụng để nạp các Package chứa class cần dùng vào một chương trình Java.

Để sử dụng được các class có sẵn, hoặc là class do người dùng định nghĩa nhưng đã được đóng gói vào các Package, chúng ta có 2 phương pháp là dùng **tên class đủ điều kiện** của chúng, hoặc là nạp các package chứa chúng vào trong chương trình để sử dụng, thông qua lệnh import.

Sự khác nhau giữa tham số và đối số ?

Tham số là một biến được xác định bởi một hàm nhận một giá trị mà khi một hàm được gọi.

(Chỉ định biến được khai báo trong định nghĩa phương thức)

Đối số là một giá trị được truyền vào thời điểm gọi một hàm.

(Chỉ định dữ liệu được truyền cho các tham số phương thức)

Tóm lại, sự khác biệt giữa đối số và tham số là **đối số** là dữ liệu được truyền vào lúc gọi một hàm và **tham số** là một biến được xác định bởi hàm sẽ nhận giá trị khi hàm được gọi.

Trình biên dịch và thực thi trong JVM (Java Virtual Machine)

Giải thích quá trình biên dịch và thực thi ?

Trình biên dịch tiến hành compile (translate) mã nguồn thành mã máy dưới dạng **byte code**.(Tức là từ mà nguồn dạng [tên-class].java sẽ được biên dịch thành [tên-class].class.)

Cuối cùng, file Java dạng [tên-class].class sẽ được thực thi tại bất cứ hệ điều hành nào, từ Windows, Linux đến MacOs, chỉ cần có cài sẵn **máy ảo JVM** là được.

Sự khác nhau giữa source code và bytecode ?

Sự khác biệt giữa **mã nguồn** và mã **bytecode** là **mã nguồn** là tập hợp các lệnh máy tính được viết bằng ngôn ngữ lập trình có thể đọc được của con người trong khi **mã bytecode** là mã trung gian giữa mã nguồn và mã máy được thực thi bởi máy ảo.

JVM là gì ? Giống và khác nhau với hệ mở ?

JVM (Java Virtual Machine) là 1 máy ảo **java** – nó được dùng để thực thi các chương trình Java hay hiểu nôm na là trình thông dịch của Java. Nó cung cấp môi trường để code java có thể được thực thi. Chương trình Java khi biên dịch sẽ tạo ra các mã máy gọi là bytecodes. Tóm gọn lại là mỗi nền tảng/hệ điều hành khác nhau (Windows, Android, Linux…) lại có một loại JVM khác nhau được cài đặt.

JVM thực hiện các công việc chính sau đây: Tải code (các class, resource) - Kiểm tra code (kiểm tra code có đúng cú pháp không, có bị lỗi không, tất nhiên nếu code có lỗi thì sẽ không chạy được chương trình rồi) - Thực thi code - Cung cấp môi trường runtime

JVM có 3 thành phần chính

Class Loader: Tìm kiếm và load các file *.class vào vùng nhớ của java dưới dạng bytecode

Data Area : vùng nhớ hệ thống cấp phát cho Java Virtual Machine

Execution Engine: chuyển các lệnh của JVM trong file *.class thành các lệnh của máy, hệ điều hành tương ứng và thực thi chúng.

Data Type

Primitive Type

Primitive types are byte, boolean, char, short, int, float, long and double.

Primitive types always have a value; **if not assigned**, will have a default value.

A **long** value is suffixed with L (or l) to differentiate it from int.

A **float** value is suffixed with F (or f) to differentiate it from double. Similarly **double** is suffixed(hậu tố) with D (or d)

A **char** is unsigned and represent an Unicode values.

When a primitive type is assigned to another variable, a copy is created.

Reference Types

All **non-primitive** types are **reference** types.

Reference types are also usually known as objects. A reference type also refers to an object in memory. Objects of reference type will have **null** as default value, when it's **unassigned**.

Objects have **variables** and **methods**, which define its state(trạng thái) and the behaviour(hành vi).

When a reference **is assigned to another reference**, both points to the **same object**.

Access Modifier

Access Modifer trong Java xác định phạm vi có thể truy cập của biến, phương thức, constructor hoặc lớp.

	Lớp	Package	Ngoài package bởi lớp con	Ngoài package
Private	*			
Default	*	*		
Protected	*	*	*	
Public	*	*	*	*

Các thuộc tính, lớp hoặc phương thức không được khai báo bằng bất kỳ **access modifier** nào, tức là **default access modifier**, chỉ có thể truy cập được trong cùng một package.

Static

Static members does not have a copy and are stored only at a single location is memory. These members should be accessed using class name.

Static method does not have access to instance methods or properties, because static members belong to the class and not the class instances.

Static method không thể ghi đè Vì phương thức static được ràng buộc với class, còn phương thức instance được ràng buộc với đối tượng. Static thuộc về vùng nhớ class còn instance thuộc về vùng nhớ heap.

Biến static là biến được cấp phát 1 lần trong bộ nhớ, tất cả đối tượng đều dùng chung, khi thay đổi thì các đối tượng đều nhận biết được sự thay đổi đó (biến static đk tạo ra trước khi đối tượng được tạo ra).

Biến instance được tạo ra cùng với quá trình tạo đối tượng

Static method không thể truy cập được đến **instance member** vì **static method** được tạo ra trước khi một object được tạo ra, vì thế nó không thể truy cập được đến instance member – một thành viên chưa chắc chắn có tồn tại hay không.

Final

final class cannot be extended, which makes the class secure and efficient.

final method cannot be overridden, which prevents any possibility of introducing any unexpected behaviour to the class.

final variable reference cannot be changed, but the content of the mutable object, that the final variable is referencing, can be changed.(nó không thể tham chiếu đến 1 đối tượng khác, nhưng nội dung mà nó tham chiếu đến có thể thay đổi)

blank final variable - variable which is not initialized at the point of declaration.(blank final variable needs to be initialized in the constructor of the class.)

Pass by value or Pass by reference

Pass-by-value được hiểu là khi bạn thay đổi biến trong hàm thì ngoài hàm sẽ không bị ảnh hưởng. Nó giống như bạn copy giá trị của biến vào biến khác rồi truyền vào hàm.

Pass-by-reference là khi bạn thay đổi biến trong hàm cũng làm ngoài hàm bị ảnh hưởng. Nó giống như bạn truyền đúng địa chỉ của biến đó vào hàm.

Trong Java - method arguments, primitive hoặc đối tượng, luôn được truyền bằng giá trị và quyền truy cập vào một đối tượng chỉ được phép **thông qua một tham chiếu chứ không phải trực tiếp**. Trong khi truyền một đối tượng cho một phương thức, đó là **bản sao của tham chiếu** được truyền chứ không phải bản thân đối tượng. Mọi thay đổi được thực hiện đối với tham chiếu đối tượng là thay đổi **nội dung đối tượng**, không phải giá trị của tham chiếu

Naming Convention

Quy ước đặt tên là một tập hợp các quy tắc đặt tên cho giao diện, lớp, phương thức, biến và các thực thể khác. Việc lựa chọn và thực hiện các quy ước đặt tên thường trở thành vấn đề tranh luận.

Quy ước đặt tên tiêu chuẩn cải thiện khả năng đọc code, giúp xem xét và hiểu tổng thể về code.

Camel Case là thực hành viết các từ kết hợp sao cho chữ cái đầu tiên trong mỗi từ được viết hoa, như BorderLength; nó còn được gọi là **Pascal Case** hoặc **Upper Camel Case**. Nhưng trong lập trình, trường hợp Camel thường bắt đầu bằng chữ cái thường, như borderLength; nó còn được gọi là **Lower Camel Case**. Đối với Java, chúng ta hãy xem xét định dạng BorderLength là Pascal Case và định dạng borderLength là Camel Case.

Inteface: Name should be Pascal Case and an adjective if it defines behaviour, otherwise noun.

Class: Name should be Pascal Case and a noun, as a class represents some real world object.

Method: Name should be Camel Case

Variable: Name should be Camel Case

Constant: Name should be all uppercase letters. Compound words should be separated by underscores.(DEFAULT_CAPACITY, DAY_IN_MONTH)

Enum: (một tập hợp các giá trị có thể có của 1 thuộc tính) Enum set name should be all uppercase letters (SECOND, MINUTE, HOUR, PI)

Acronyms: Mặc dù các từ viết tắt thường được đại diện bởi tất cả các chữ cái viết hoa, nhưng trong Java chỉ chữ cái đầu tiên của các từ viết tắt phải là chữ hoa và phần còn lại là chữ thường.

Polymorphism(Đa hình)

Là khả năng một đối tượng có thể thực hiện một tác vụ theo nhiều cách khác nhau. Trong Java, ta sử dụng nạp chồng phương thức (method overloading) và ghi đè phương thức (method overriding) để thu được tính đa hình.

Lợi ích: Có thể sử dụng lại code thông qua tính đa hình

Hỗ trợ một tên biến duy nhất cho nhiều kiểu dữ liệu (VD: Shape shape = new Shape(), shape = new Rectangle(), shape = new Triangle(), …)

Subtype polymorphism

In Subtype polymorphism, also known as inclusion polymorphism, the parameter definition of a function supports any argument of a type or its subtype. (các tham số của method hỗ trợ các đối số là kiểu truyền vào hoặc kiểu con của nó)

```
public void printWheelsCount(Vehicle vehicle) {
    print(vehicle.getWheelsCount());
}
```

Ta có thể truyền đối số là newVehicle() hoặc newCar(), newBike(), … vào phương thức trên.

Overriding

Method overriding: Là lớp con định nghĩa lại một phương thức cùng tên, cùng tham số, cùng kiểu trả về với lớp cha.

Tuy nhiên **không thể ghi đè được các phương thức static**, do các phương thức static được ràng buộc với lớp của nó, còn phương thức instance thì được ràng buộc với đối tượng. Static thuộc về vùng nhớ của lớp còn instance thuộc về vùng nhớ heap

Ghi đè phương thức được sử dụng để thu được tính đa hình tại **runtime**.
Đa hình động (thời gian chạy) là đa hình tồn tại trong thời gian chạy. Ở đây, trình biên dịch Java không hiểu phương thức nào được gọi tại thời điểm biên dịch. Chỉ JVM quyết định phương thức nào được gọi vào thời gian chạy.
- **Đa hình tại thực thi (runtime):**
Là khi việc gọi phương thức được quyết định bởi JVM chứ không phải Compiler.
Là phương thức đã được ghi đè trong thời gian thực thi chương trình.
Nguyên tắc ghi đè phương thức:

Phương thức phải có tên giống với lớp cha.
Phương thức phải có tham số giống với lớp cha.
Lớp con và lớp cha có mối quan hệ kế thừa.

Overloading

Khả năng cho phép **một lớp có nhiều phương thức có cùng tên**, nhưng **khác nhau ở tham số** truyền vào.

Ta không thể nạp chồng phương thức bằng cách chỉ thay đổi kiểu trả về mà vẫn giữ nguyên tham số truyền vào ở 2 phương thức, vì như thế JVM sẽ không biết phương thức nào được gọi.

Từ việc overload các phương thức ta sẽ thu được đa hình hàm.

Đa hình tĩnh (thời gian biên dịch) là đa hình được thể hiện tại thời gian biên dịch. Ở đây, trình biên dịch Java biết phương thức nào được gọi. Phương thức nạp chồng và phương thức ghi đè bằng phương thức tĩnh; ghi đè phương thức bằng các phương thức riêng tư hoặc cuối cùng là các ví dụ cho đa hình tĩnh

Đa hình tại biên dịch (compile):
Là trường hợp đối tượng bị ràng buộc với chức năng của chúng ngay tại thời gian chương trình đang biên dịch.

Có 2 cách nạp chồng phương thức trong java:
+ Thay đổi số lượng các tham số
+ Thay đổi kiểu dữ liệu của các tham số

Overloading	**Overriding**
Nạp chồng phương thức được sử dụng để tăng tính có thể đọc của chương trình	Ghi đè phương thức được sử dụng để cung cấp trình triển khai cụ thể của phương thức mà đã được cung cấp bởi lớp cha của nó
Nạp chồng phương thức được thực hiện bên trong lớp (class)	Ghi đè phương thức xuất hiện trong hai lớp mà có mối quan hệ IS-A (kế thừa)
Trong Nạp chồng phương thức, tham số phải khác nhau	Trong Ghi đè phương thức, tham số phải là giống nhau
Nạp chồng phương thức là ví dụ của đa hình tại biên dịch (**compile)**	Ghi đè phương thức là ví dụ của đa hình tại thực thi (**runtime)**

Đa hình tĩnh (Đa hình thời gian biên dịch)	**Đa hình động** (Đa hình thời gian chạy)
Quyết định phương thức nào sẽ thực thi trong **thời gian biên dịch**	Quyết định phương thức nào sẽ thực thi trong **thời gian chạy**

Overloading method là một ví dụ về đa hình tĩnh và nó được yêu cầu để xảy ra đa hình tĩnh	**Overriding method** là một ví dụ về đa hình động, và nó được yêu cầu để xảy ra đa hình động
Đa hình tĩnh đạt được thông qua liên kết tĩnh	Đa hình động đạt được thông qua liên kết động
Xảy ra trong **cùng một lớp**	Xảy ra **giữa các lớp khác nhau**
Không cần gán đối tượng cho đa hình tĩnh	Nó được yêu cầu trong đó một **đối tượng lớp con được gán cho đối tượng lớp cha** (**upcasting**)cho đa hình động
Kế thừa **không liên quan** đến đa hình tĩnh	Kế thừa **liên quan** đến đa hình động.

Trừu tượng

Cung cấp sự tổng quát hóa cho đối tượng, bỏ qua những thứ không liên quan. VD: Bài toán quản lý sinh viên thì chỉ cần quan tâm các thông tin như Họ và tên, Ngày tháng năm sinh, CMND/CCCD, ... mà không cần quan tâm các thông tin như Cân nặng, Chiều cao, ...

Lợi ích: Tránh việc để logic phức tạp trong các phương thức bị lộ ra ở phía người dùng. Như trong VD code Java ở trên, người dùng chỉ có thể nhìn thấy kết quả của phép tính mà không biết logic được implement ở trong từng hàm.

Khai báo một lớp với **tên lớp**, cùng với các thuộc tính khác như lớp cha của lớp đó, và từ khóa để chỉ tính chất của lớp đó **public**, **final** hay **abstract**.

Một lớp trừu tượng(abstract class) là một lớp không đầy đủ, tức nó có ít nhất một phương thức trừu tượng.

Tách phần giao diện(nằm ở lớp abstract) và phần cài đặt(implimentation)(nằm ở lớp kế thừa) để có thể giấu đi các thông tin ở phần cài đặt.

Abstract class:

+ Một lớp được khai báo với từ khóa **abstract** là lớp trừu tượng.
+ Lớp trừu tượng có thể có các phương thức trừu tượng hoặc không trừu tượng.
+ **Không thể khởi tạo** một đối tượng trực tiếp từ một lớp trừu tượng.
+ Một lớp kế thừa một lớp trừu tượng thì **không cần phải implement các phương thức không trừu tượng của nó**, nhưng **bắt buộc phải ghi đè các phương thức trừu tượng**. Trừ khi lớp con cũng là trừu tượng.

Abstract method: Là một phương thức được khai báo là **abstract** và không có phần triển khai logic.

Giao diện (Interface):

+ Là một kỹ thuật để thu được tính trừu tượng hoàn toàn và đa kế thừa trong Java.

+ Interface luôn có modifier là **public interface**.
+ Các phương thức trong interface **đều là phương thức trừu tượng**, đều có modifier là **public abstract**.
+ Các **thuộc tính** trong interface đều có modifier là **public static final**.
+ Interface không có constructor.
+ Một interface không phải là một lớp, nó là một lớp mô tả các thuộc tính và phương thức của một đối tượng. Một interface chứa các phương thức mà một lớp sẽ triển khai.
+ **Không thể khởi tạo** đối tượng là một interface.
+ Một lớp có thể **triển khai một hoặc nhiều interface** tại một thời điểm.
+ Một interface có thể kế thừa từ một interface khác.

=>> Tóm lại:

+ **Abstract class** có quan hệ với lớp con theo quan hệ IS-A. Vì vậy khi thiết kê, nếu đối tượng có quan hệ cha con thì ta nên chọn abstract class.

+ **Interface** không quan tâm đến việc tương thích dữ liệu, nó chỉ mang tính hợp đồng với lớp con.

+ Thông qua Interface ta có thể thực hiện đa thừa kế trong Java.

Đóng gói

Không cho phép người dùng trực tiếp tác động đến các dữ liệu được bảo vệ (protected, private) bên trong đối tượng, **mà phải thông qua các phương thức** mà đối tượng cung cấp (các phương thức public).
Tính chất này **đảm bảo tính toàn vẹn** của đối tượng.

- Các đặc điểm của tính đóng gói:
+ Ngăn ngừa việc gọi phương thức của lớp này tác động hay truy xuất, sửa đổi thuộc tính của đối tượng thuộc về lớp khác.
+ Thuộc tính được khai báo **private** của mỗi đối tượng được bảo vệ khỏi sự truy xuất không hợp lệ từ bên ngoài.
+ Có thể kiểm soát thuộc tính bằng cách viết hàm logic cho các điều kiện đặc biệt để truy xuất thuộc tính trong các phương thức public.
+ Cho phép thay đổi cấu trúc bên trong một lớp mà không làm ảnh hưởng đến những lớp bên ngoài có sử dụng lớp đó.

- Các mức đóng gói trong java:
+ Các thuộc tính và phương thức **private** chỉ có thể được truy cập bởi các phương thức trong cùng lớp của nó.

+ Các thuộc tính và phương thức **default** thì trong lớp (class) đó và trong gói (package) đó mới có thể nhìn thấy được.
+ Các thuộc tính và phương thức **protected** có thể được truy cập bởi các phương thức trong cùng lớp của nó và các lớp con của lớp đó.

+ Các thuộc tính và phương thức **public** có thể được truy cập từ bất kỳ phương thức từ bất kỳ lớp nào trong project.

Inheritance (Kế thừa)

Cho phép xây dựng một lớp mới dựa trên các định nghĩa của lớp đã có bằng từ khóa **extends**. Kế thừa là mối quan hệ giữa 2 lớp: lớp con (subclass/derived class/extended class/child class) và lớp cha (superclass/base class/parent class).

Khi kế thừa từ lớp cha, lớp con được sử dụng lại và sửa đổi tất cả các phương thức cùng thuộc tính **public** và **protected** của lớp cha, đồng thời có thể khai báo thêm các phương thức và thuộc tính khác.

Tính kế thừa thiết lập mối quan hệ IS-A giữa lớp con với lớp cha

Tính kế thừa có thể tái sử dụng mã nguồn chương trình một cách tối ưu.

Sử dụng tính kế thừa trong Java để override phương thức, từ đó ta có thể thu được tính đa hình tại runtime.

Để giảm thiểu sự phức tạp và đơn giản hóa ngôn ngữ, **đa kế thừa không được support** trong java (đa kế thừa chỉ được hỗ trợ thông qua interface). Nếu cố tình sử dụng đa kế thừa sẽ xảy ra lỗi Compile Time Error.

3 kiểu kế thừa trong Java:
+ Kế thừa đơn (Single)
+ Kế thừa nhiều cấp (Multilevel)
+ Kế thừa thứ bậc (Hierarchical)

Composition (Thành phần)

Khai báo biến tham chiếu của một class trong một class khác được gọi là composition(sự hợp thành).

Composition là một khái niệm thiết kế hướng đối tượng có liên quan chặt chẽ đến tính kế thừa, vì nó cũng liên quan đến việc sử dụng lại các lớp; nhưng nó tập trung vào việc thiết lập mối quan hệ **HAS-A** giữa các lớp thay vì **IS-A** như kế thừa. Vì vậy, không giống như Kế thừa, composition liên quan đến việc mở rộng các tính năng của một lớp, thành phần sử dụng lại một lớp bằng cách biên soạn nó. Composition đạt được bằng cách lưu trữ tham chiếu của một lớp khác như một thành viên.

Vấn đề với kế thừa là nó phá vỡ tính đóng gói khi lớp con kết hợp chặt chẽ với việc triển khai lớp cha. Vấn đề trở nên phức tạp khi một lớp không được thiết kế để giữ phạm vi kế thừa trong tương lai và bạn không có quyền kiểm soát lớp cha. Một lớp con có khả năng bị phá vỡ do những thay đổi trong lớp cha.

Vì vậy, kế thừa chỉ được sử dụng khi có mối quan hệ IS-A giữa định nghĩa lớp cha và lớp con; và trong trường hợp dễ gây nhầm lẫn, ta ưu tiên sử dụng composition hơn.

String Immutability(Không thể thay đổi)

Đối tượng String là bất biến, có nghĩa là một khi được tạo, đối tượng mà String tham chiếu đến không bao giờ có thể thay đổi. Mặc dù bạn có thể gán cùng một tham chiếu cho một đối tượng String khác.

VD: String greeting = “happy”

greeting = greeting + “birthday”

The code above creates three different String objects, “Happy“, “Birthday” and “Happy Birthday”.

Though you cannot change the value of the String object but you can change the reference variable that is referring to the object. In the above example, the String reference greeting starts referring the String object “Happy Birthday”.

Mặc dù không thể thay đổi giá trị của đối tượng String nhưng có thể thay đổi biến tham chiếu đang tham chiếu đến đối tượng.

Lưu ý rằng bất kỳ hoạt động nào được thực hiện trên Chuỗi đều dẫn đến việc tạo Chuỗi mới.

Thuận lợi:
+ Vì không cần đồng bộ hóa cho các đối tượng Chuỗi, nên có thể an toàn khi chia sẻ đối tượng Chuỗi giữa các luồng.
+ Để tận dụng thực tế này để tối ưu hóa bộ nhớ, môi trường Java lưu các ký tự Chuỗi vào một vùng đặc biệt trong bộ nhớ được gọi là String pool. Nếu một ký tự chuỗi đã tồn tại trong pool, thì ký tự chuỗi tương tự sẽ được chia sẻ.
+ Các giá trị Chuỗi bất biến bảo vệ chống lại bất kỳ thay đổi nào về giá trị trong quá trình thực thi.
+ Vì mã băm(hash-code) của đối tượng Chuỗi không thay đổi, nên có thể lưu mã băm vào bộ nhớ cache và không phải tính toán mỗi khi nó được yêu cầu.

Hạn chế:
+ Lớp chuỗi không thể được mở rộng để cung cấp các tính năng bổ sung.
+ Nếu nhiều chuỗi được tạo, đối tượng mới hoặc do bất kỳ hoạt động chuỗi nào, nó sẽ tải lên Garbage Collector.

String Literal (Chuỗi kí tự) vs String Object

String Literal là một khái niệm ngôn ngữ Java trong đó lớp String được tối ưu hóa để lưu vào bộ đệm tất cả các Chuỗi được tạo trong dấu ngoặc kép, vào một vùng đặc biệt được gọi là String pool.

VD: String cityName = “London”;

String pool là một vùng nhớ đặc biệt nằm trong vùng heap để lưu trữ các chuỗi immutable mà ta đề cập ở phía trên

Khi gặp đoạn code trên, JVM xác định xem chuỗi "London" đã tồn tại trong pool chưa. Nếu đã tồn tại, JVM sẽ tham chiếu biến cityName đến vùng nhớ của chuỗi này trong pool. Nếu chưa tồn tại, JVM sẽ tạo ra object chứa chuỗi này, đưa vào pool, rồi tham chiếu biến cityName đến vùng nhớ vừa tạo này.

Khi khởi tạo một **String object** vs từ khoá **new()** chúng ta sẽ luôn luôn khởi tạo một object mới vào heap memory.
VD: String cityName = new String(“London”);

String literal và String object đều là các String object, điểm khác biệt duy nhất là khởi tạo từ **new** sẽ luôn khởi tạo một object mới còn lại chúng có thể sử dụng lại các object String được khởi tạo trước đó.

Nhược điểm khi tạo một lượng lớn String literal là String literal được lưu trong String pool, mà String pool có kích thước cố định không thể mở rộng trong thời gian RunTime, nên nếu có quá nhiều String literal sẽ có khả năng gặp lỗi Out of Memory.

String Interning

String interning là một khái niệm chỉ lưu trữ một bản sao duy nhất của mỗi giá trị immutable String riêng biệt.

Khi bạn xác định bất kỳ ký tự Chuỗi mới nào, nó sẽ được thực hiện. Hằng số Chuỗi tương tự trong pool được tham chiếu cho bất kỳ ký tự Chuỗi lặp lại nào.

Các ký tự nhóm chuỗi được xác định không chỉ tại thời điểm biên dịch mà còn trong thời gian chạy. Có thể gọi phương thức intern() trên đối tượng String để thêm nó vào String pool, nếu nó chưa có.

Phương thức intern() có thể được sử dụng để trả về chuỗi từ Pool chứa hằng số chuỗi khi nó được tạo bởi từ khóa new.

Đặt một lượng cực lớn văn bản vào vùng nhớ có thể dẫn đến rò rỉ bộ nhớ và (hoặc) vấn đề về hiệu suất

Lưu ý: Thay vì sử dụng String object, hãy ưu tiên sử dụng String literal để trình biên dịch có thể tối ưu hóa nó.

Garbage Collection

Từ java 7 trở về trước JVM lưu String pool trong vùng nhớ được gọi là **PermGen** nó có kích thước cố định và không thể mở rộng tại thời điểm runtime và không được trình dọn rác thu gon.
Vì vậy chúng ta sẽ thường xuyên gặp lỗi **OutOfMemory error** từ JVM khi chúng ta có quá nhiều String trong vùng nhớ PermGen.
Thật may rằng từ java 7 trở đi, java đã lưu trữ String Pool trong vùng nhớ Heap và sẽ được trình dọn rác can thiệp khi không có một biến nào tham chiếu đến nó. Nhờ vậy chúng ta sẽ giảm bớt khả năng gặp lỗi OutOfMemory.

StringBuilder vs StringBuffer

Khi làm việc với dữ liệu kiểu text trong Java cung cấp 3 class String, StringBuffer và StringBuilder.

Cơ bản về 3 class này như sau:
+ String là không thể thay đổi (immutable), và không cho phép có class con.
+ StringBuffer, StringBuilder có thể thay đổi (mutable) (lưu trữ trong heap)
+ StringBuilder và StringBuffer là giống nhau, nó chỉ khác biệt tình huống sử dụng có liên quan tới đa luồng (Multi Thread). => về tốc độ xử lý StringBuilder là tốt nhất, sau đó StringBuffer và cuối cùng mới là String

StringBuilder và StringBuffer là khá giống nhau, điều khác biệt là tất cả các phương thức của **StringBuffer** đã được đồng bộ, nó thích hợp khi bạn làm việc với ứng dụng đa luồng, nhiều luồng có thể truy cập vào một đối tượng StringBuffer cùng lúc**. Trong khi đó StringBuilder** có các phương thức tương tự nhưng không được đồng bộ, vì vậy mà hiệu suất của nó cao hơn. Nên sử dụng StringBuilder trong ứng dụng đơn luồng, hoặc sử dụng như một biến cục bộ(local variable) trong một phương thức.

Note: If you need to share String objects between threads then use StringBuffer, otherwise StringBuilder.

StringBuffer	StringBuilder
StringBuffer là đồng bộ (synchronized) tức là luồng an toàn. Điều này có nghĩa là **không thể có 2 luồng** cùng truy cập phương thức của lớp StringBuffer đồng thời.	StringBuilder là không đồng bộ (non-synchronized) tức là luồng không an toàn. Điều này có nghĩa là có 2 luồng cùng truy cập phương thức của lớp StringBuilder đồng thời.
StringBuffer không hiệu quả bằng StringBuilder.	StringBuilder hiệu quả hơn StringBuffer.

Stack vs Heap

Heap	Stack
Java Heap Memory là bộ nhớ được sử dụng ở runtime để lưu các Objects. Bất cứ khi nào ở đâu trong chương trình của bạn khi bạn tạo Object thì nó sẽ được lưu trong Heap (thực thi toán tử new).	**Stack Memory** là bộ nhớ để lưu các biến local trong hàm và lời gọi hàm ở runtime trong một Thread java. Các biến local bao gồm: loại nguyên thuỷ (primitive), loại tham chiếu tới đối tượng trong heap (reference), khai báo trong hàm, hoặc đối số được truyền vào hàm.
Thời gian sống của bộ nhớ Heap dài hơn so với Stack. Thời gian sống của object phụ thuộc vào Garbage Collection của java. Garbage Collection sẽ chạy trên bộ nhớ Heap để xoá các Object không được sử dụng nữa, nghĩa là object không được referece trong chương trình.	Thường có thời gian sống ngắn.
Các objects trong Heap đều được truy cập bởi **tất cả các các nơi trong ứng dụng, bởi các threads khác nhau**.	Stack chỉ được sử dụng cho **một Thread duy nhất**. Thread ngoài không thể truy cập vào được.
Cơ chế quản lý của Heap thì phức tạp hơn. Heap được phân làm 2 loại Young-Generation, Old-Generation. Đọc thêm về Garbage Collection để hiểu rõ hơn.	Cơ chế hoạt động là **LIFO** (Last-In-First-Out), chạy sau chết trước.
Dung lượng Heap thường lớn hơn Stack.	Bộ nhớ stack thường nhỏ.
Sử dụng -Xms và -Xmx để định nghĩa dung lượng bắt đầu và dung lượng tối đa của bộ nhớ heap.	Dùng -Xss để định nghĩa dung lượng bộ nhớ stack.
Khi Heap bị đầy chương trình hiện lỗi java.lang.OutOfMemoryError: Java Heap Space	Khi stack bị đầy bộ nhớ, chương trình phát sinh lỗi: java.lang.StackOverFlowError
Truy cập vùng nhớ Heap chậm hơn Stack.	Truy cập stack nhanh hơn Heap
Bộ nhớ Stack chỉ hiển thị đối với luồng chủ sở hữu(owner thread), do đó, quyền truy cập bộ nhớ sẽ diễn ra ngay lập tức. Bộ nhớ Heap được chia sẻ trên nhiều luồng trong ứng dụng; đồng bộ hóa với luồng khác nên có hậu quả về hiệu suất. =>> Heap chậm hơn Stack	
Dung lượng sử dụng của Heap sẽ tăng giảm phụ thuộc vào Objects sử dụng.	Bất cứ khi nào gọi 1 hàm, một khối bộ nhớ mới sẽ được tạo trong Stack cho hàm đó để lưu các biến local. Khi hàm thực hiện xong, khối bộ nhớ cho hàm sẽ bị xoá, và giải phóng bộ nhớ trong stack.