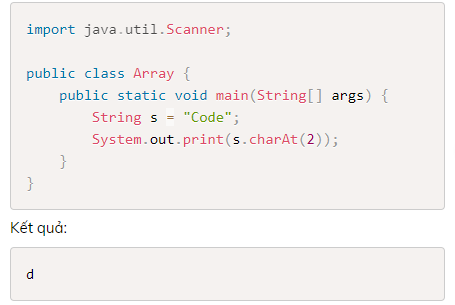
**Xử lý chuỗi**

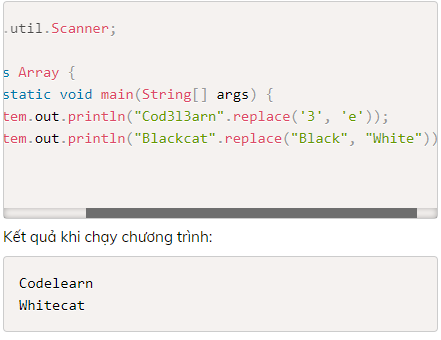
**charAt():** trả về một ký tự trong string

**VD:**



**Replace():** thay thế chuỗi/ký tự được tìm thấy thành chuỗi/ký tự khác.

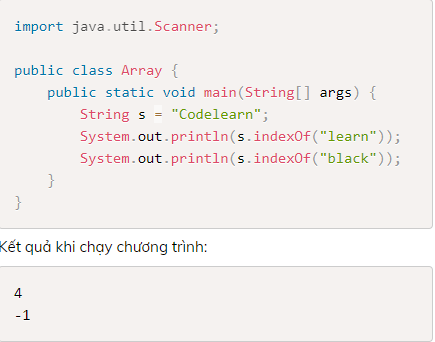
VD:



**toUpperCase/toLowerCase()**: chuyển các ký tự từ in thường thành in hoa và ngược lại.

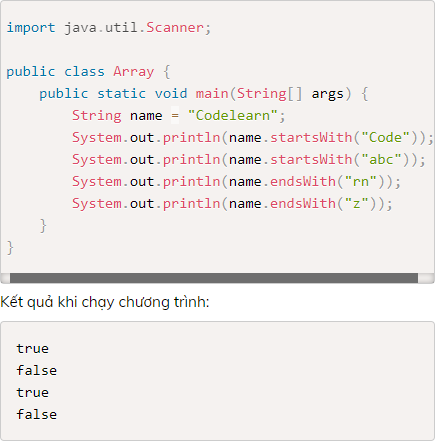
**indexOf():** trả về vị trí xuất hiện đầu tiên của 1 chuỗi trong 1 chuỗi khác, nếu không tìm thấy trả về -1

VD:



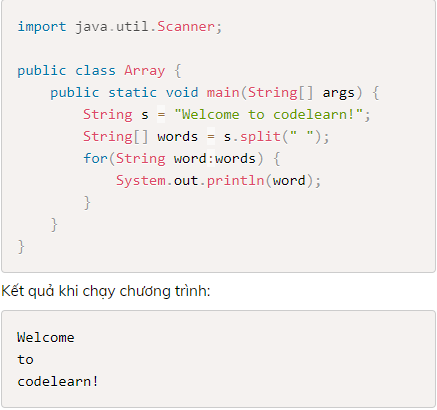
**startsWith / endsWith :** dùng để kiểm tra một xâu có bắt đầu hoặc kết thúc bằng một chuỗi khác hay không

VD:



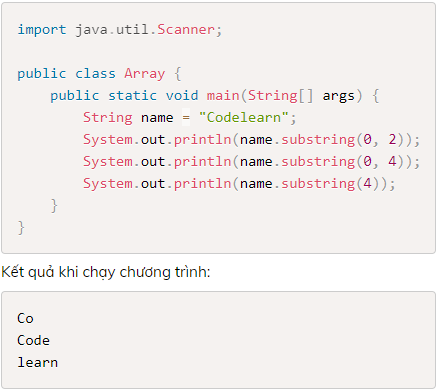
**Split():** tách 1 xâu ra thành mảng các xâu dựa trên 1 xâu cho trước

VD:



**Substring():** lấy ra 1 xâu con dựa trên chỉ số bắt đầu và chỉ số kết thúc của 1 xâu khác

VD:



**Overloading**(nạp chồng phương thức): là 1 phương thức cùng tên nhưng khác tham số, kiểu tham số

**Overide** (ghi đè): Nếu lớp con có cùng phương thức như đã được khai báo trong lớp cha, thì đó gọi là Ghi đè phương thức

Qui tắc cho ghi đè phương thức

Phương thức phải có cùng tên như trong lớp cha.

Phương thức phải có cùng tham số như trong lớp cha.

Phải là quan hệ IS-A (kế thừa).

**Đa hình:** Khi một tác vụ được thực hiện theo nhiều cách khác nhau được gọi là tính đa hình. Chúng ta có thể thực hiện tính đa hình trong Java bởi nạp chồng phương thức và ghi đè phương thức.

**Trừu tượng (abstract):**

Phương thức: khai báo bằng từ khóa abstract phía trước tên phương thức và phương thức không có hành động (chỉ khai báo tên thôi).

**Kế thừa**(Java chỉ cho kế thừa đơn – chỉ kế thừa từ 1 lớp cha duy nhất): extends

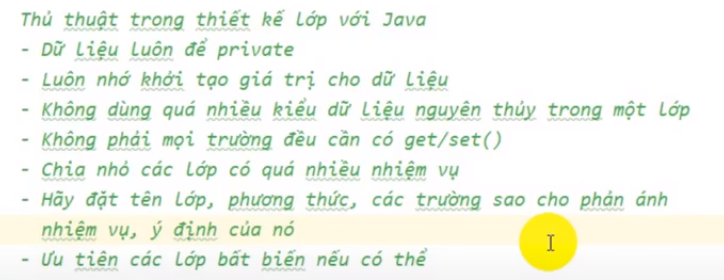
**STATIC (bộ nhớ tạm thời)**

Tạo ra dùng để sử dụng chung 1 giá trị. Là thành phần dùng chung của class

Các phương thức có cùng **static** với nhau thì mới thao tác được với nhau

Từ khóa **super :** gọi đến phương thức khởi tạo của class cha (phải gọi ở đầu phương thức), cũng có thể gọi đến phương thức của class cha bằng **super.**

**Qui chuẩn thiết kế class:**



**Regular Expression**

|  |  |  |
| --- | --- | --- |
| TT | Biểu thức chính quy | Mô tả |
| 1 | . | Khớp (match) với bất kỳ ký tự nào |
| 2 | ^regex | Biểu thức chính quy phải  khớp tại điểm bắt đầu |
| 3 | regex$ | Biểu thức chính quy phải khớp ở cuối dòng. |
| 4 | [abc] | Thiết lập định nghĩa, có thể khớp với a hoặc b hoặc c. |
| 5 | [abc][vz] | Thiết lập định nghĩa, có thể khớp với a hoặc b hoặc c theo sau là v hay z. |
| 6 | [^abc] | Khi dấu ^ xuất hiện như là nhân vật đầu tiên trong dấu ngoặc vuông, nó phủ nhận mô hình. Điều này có thể khớp với bất kỳ ký tự nào ngoại trừ a hoặc b hoặc c. |
| 7 | [a-d1-7] | Phạm vi: phù hợp với một chuỗi giữa a và điểm d và con số từ 1 đến 7. |
| 8 | X|Z | Tìm X hoặc Z. |
| 9 | XZ | Tìm X và theo sau là Z. |
| 10 | $ | Kiểm tra kết thúc dòng. |
|  | | |
| 11 | \d | Số bất kỳ, viết ngắn gọn cho [0-9] |
| 12 | \D | Ký tự không phải là số, viết ngắn gon cho [^0-9] |
| 13 | \s | Ký tự khoảng trắng, viết ngắn gọn cho [ \t\n\x0b\r\f] |
| 14 | \S | Ký tự không phải khoản trắng, viết ngắn gọn cho [^\s] |
| 15 | \w | Ký tự chữ, viết ngắn gọn cho [a-zA-Z\_0-9] |
| 16 | \W | Ký tự không phải chữ, viết ngắn gọn cho [^\w] |
| 17 | \S+ | Một số ký tự không phải khoảng trắng (Một hoặc nhiều) |
| 18 | \b | Ký tự thuộc a-z hoặc A-Z hoặc 0-9 hoặc \_, viết ngắn gọn cho [a-zA-Z0-9\_]. |
|  | | |
| 19 | \* | Xuất hiện 0 hoặc nhiều lần, viết ngắn gọn cho {0,} |
| 20 | + | Xuất hiện 1 hoặc nhiều lần, viết ngắn gọn cho {1,} |
| 21 | ? | Xuất hiện 0 hoặc 1 lần, ? viết ngắn gọn cho {0,1}. |
| 22 | {X} | Xuất hiện X lần, {} |
| 23 | {X,Y} | Xuất hiện trong khoảng X tới Y lần. |
| 24 | \*? | \* có nghĩa là xuất hiện 0 hoặc nhiều lần, thêm ? phía sau nghĩa là tìm kiếm khớp nhỏ nhất. |

Lập trình hướng đối tượng

Khi nhắc tới lập trình hướng đối tượng chắc bạn sẽ nghĩ ngay tới 4 tính chất là tính đóng gói, tính kế thừa, tính đa hình và tính trừu tượng. Thực chất thì 4 tính chất này chỉ giống như các nguyên liệu để xây dựng chương trình theo phương pháp hướng đối tượng, quan trọng nhất vẫn là cách mà bạn sử dụng các nguyên liệu này để xây dựng chương trình như thế nào.

Vậy lập trình hướng đối tượng là gì? Nó được hiểu đơn giản là một phương pháp để giải quyết bài toán lập trình mà khi áp dụng thì code sẽ trở nên dễ phát triển và dễ bảo trì hơn. Phương pháp này sẽ chia nhỏ chương trình thành các đối tượng và các mối quan hệ, mỗi đối tượng sẽ có các thuộc tính (dữ liệu) và hành vi (phương thức). Để có thể lập trình và thiết kế chương trình theo phương pháp này thì chắc chắn bạn cần hiểu rõ về 4 tính chất là là tính đóng gói, tính kế thừa, tính đa hình và tính trừu tượng.

📌Tính đóng gói (Encapsulation)

Đây là kỹ thuật giúp bạn che giấu đi những thông tin bên trong đối tượng bằng cách sử dụng phạm vi truy cập private cho các thuộc tính, muốn giao tiếp hay lấy ra các thông tin của đối tượng thì phải thông qua các phương thức public, từ đó sẽ hạn chế được các lỗi khi phát triển chương trình. Tính chất này cũng giống với trong thực tế, bạn không thể thấy được các thuộc tính thực của một người (tính cách, sở thích, các thông tin riêng tư khác, ...), những thứ mà bạn biết đều là thông qua các hành động của người đó. Ví dụ người đó nói cho bạn biết về sở thích, tuổi, ... nhưng các thông tin này chưa chắc đã thực sự là thuộc tính thật của người đó (giống với việc các getter không trả về giá trị thực của thuộc tính mà trả về một giá trị khác).

Việc áp dụng tính đóng gói sẽ đem lại nhiều lợi ích nhưng lợi ích chính vẫn là giúp hạn chế được các lỗi khi phát triển chương trình. Hãy thử tưởng tượng nếu bạn có lớp Student với thuộc tính điểm và thuộc tính này có thể được truy xuất từ bất cứ đâu thì sẽ rất khó để quản lý được giá trị, bạn có thể thoải mái thay đổi giá trị cho thuộc tính điểm thành các giá trị không hợp lệ như -1, 11 ở bất cứ đâu (rõ ràng điểm không thể âm hoặc lớn hơn 10). Thay vào đó, nếu thuộc tính điểm là private và bạn cung cấp hàm setScore(int score) để thay đổi giá trị cho thuộc tính này thì bạn hoàn toàn có thể quản lý được các thay đổi lên thuộc tính điểm. (Trong hàm setScore(int score) bạn có thể bắn ra lỗi hoặc không thay đổi giá trị cho thuộc tính điểm nếu thấy dữ liệu không hợp lệ).

📌Tính kế thừa (Inheritance)

Khi lập trình chắc chắn sẽ có những trường hợp mà các đối tượng có chung một số thuộc tính và phương thức. Ví dụ như khi bạn viết chương trình lưu thông tin về các học sinh và giáo viên. Với học sinh thì cần lưu thông tin về tên, tuổi, điểm và với giáo viên thì cần lưu thông tin về tên, tuổi, tiền lương => lúc này code sẽ bị trùng lặp khá nhiều (từ các thuộc tính cho tới các setter, getter, ...) và nó vi phạm một trong những nguyên tắc cơ bản nhất khi lập trình là DRY (Don't Repeat Yourself - đừng bao giờ lặp lại code). Với kế thừa thì vấn đề này sẽ được giải quyết, kế thừa trong lập trình chính là thừa hưởng lại các thuộc tính và phương thức. Trong trường hợp trên bạn có thể tạo ra lớp cha để lưu các thuộc tính và phương thức chung, sau đó có thể kế thừa và tái sử dụng lại các đoạn code đã được viết ở lớp này. Đó chính là lợi ích của tính kế thừa.

📌Tính đa hình (Polymorphism)

Như bạn đã biết, lập trình hướng đối tượng là phương pháp tư duy và giải quyết bài toán lập trình theo hướng thực tế. Do đó, các tính chất của nó cũng sẽ gắn liền với thực tế nên trước hết bạn cần hiểu về tính đa hình trong thực tế. Đa hình được hiểu là trong từng hoàn cảnh, từng trường hợp khác nhau thì các đối tượng sẽ đóng các vai trò khác nhau. Ví dụ, cùng là một người nhưng khi ở công ty thì có vai trò là nhân viên, khi đi siêu thị thì có vai trò là khách hàng, hay khi ở trường thì lại có vai trò là học sinh, ... => cùng là một người nhưng có nhiều vai trò khác nhau nên đây chính là đa hình trong thực tế.

Trong lập trình thì khi một đối tượng hay một phương thức có nhiều hơn một hình thái thì đó chính là đa hình. Tính đa hình được thể hiện dưới 3 hình thức:

1. Nạp chồng phương thức (overloading): ví dụ phương thức cộng sẽ có các hình thái là cộng 2 số nguyên, cộng 2 số thực, cộng 3 số nguyên, ... => cùng là phương thức cộng nhưng lại có nhiều hình thái nên đây chính là biểu hiện của tính đa hình.

2. Ghi đè phương thức (overriding): khi lớp con kế thừa và ghi đè lại các phương thức ở lớp cha thì đó cũng thể hiện tính đa hình (cùng một phương thức nhưng lại được cài đặt khác nhau ở lớp con và lớp cha).

3. Đa hình thông qua các đối tượng đa hình: Nếu bạn đã học qua về OOP thì sẽ biết rằng biến thuộc lớp cha có thể tham chiếu tới đối tượng của các lớp con. Ví dụ như biến thuộc kiểu Animal có thể trỏ tới các đối tượng thuộc lớp Cat, Dog, Duck, ... vậy biến thuộc lớp cha cũng có nhiều hình thái nên đây cũng là đa hình.

Để hiểu rõ hơn thì bạn có thể xem code thể hiện tính đa hình trong bài đầu tiên ở chương đa hình trong khóa học C++ OOP và Java OOP trên hệ thống Codelearn.

📌Tính trừu tượng (Abstraction)

Trừu tượng là tính chất mà đơn giản hóa đi những thông tin bên trong đối tượng, nó cho phép ta giao tiếp với các thành phần của đối tượng mà không cần phải biết về cách mà các thành phần này được xây dựng (chính xác hơn là không cần biết các thành phần này được code như thế nào mà chỉ cần biết các thành phần này được dùng để làm gì). Trước hết, hãy cùng xem một ví dụ thực tế về tính trừu tượng:

Khi bạn đi rút tiền ở cây ATM thì bạn không cần quan tâm tới cách mà cây ATM hoạt động hay các thành phần có trong cây ATM, cái mà bạn quan tâm duy nhất đó là tính năng rút tiền. Trong trường hợp này các thông tin không cần thiết của cây ATM như đếm tiền, trừ tiền trong tài khoản, gửi dữ liệu về máy chủ đã được ẩn đi. Cái mà bạn nhìn thấy về đối tượng cây ATM chính là rút tiền => cây ATM đã ẩn đi những chi tiết không cần thiết và đó chính là tính trừu tượng.

Tương tự trong lập trình cũng vậy, khi gọi tới các phương thức của một đối tượng thì bạn chỉ cần quan tâm tới phương thức đó được dùng để làm gì chứ không cần quan tâm tới phương thức đó được code như thế nào. Tính chất này rất có ích khi làm việc nhóm, bạn chỉ cần quan tâm tới chức năng của các phương thức mà đồng nghiệp code chứ không cần biết nó được cài đặt như thế nào. Để thực hiện tính trừu tượng thì bạn có thể sử dụng các abstract class và interface vì nó chỉ chứa phần khai báo chứ không có phần cài đặt (ở một số ngôn ngữ không có khái niệm về interface nên nếu bạn chưa biết về interface thì có thể hiểu interface chính là abstract class với các phương thức đều là trừu tượng).

Trong thực tế, khi đi làm bạn sẽ sử dụng tới interface rất nhiều, với mỗi lớp bạn thường tạo ra 1 interface riêng để thể hiện các tính năng của lớp đó và sử dụng interface này để giao tiếp với đối tượng. Ví dụ lớp Customer sẽ có interface ICustomer, các đối tượng khác muốn giao tiếp với lớp Customer thì đều phải thông qua interface trên..

📌Kết

Lập trình hướng đối tượng không chỉ gói gọn trong 4 tính chất trên, để viết được một chương trình tốt thì bạn còn phải biết thêm rất nhiều nguyên liệu khác như OOP design, Software Architecture, ... trong bài này mình chỉ tóm tắt về lập trình hướng đối tượng và 4 tính chất chính, nếu muốn học chi tiết hơn thì bạn có thể tham khảo thêm tại khóa học C++ OOP và Java OOP trên hệ thống. Còn về các chủ đề khác trong OOP thì mình sẽ giới thiệu trong các bài viết tiếp theo.

So sánh 2 đối tượng: mỗi đối tượng sẽ được lưu trong một bộ nhớ khác nhau (lấy địa chỉ bộ nhớ để so sánh với nhau)

instanceof (toán tử so sánh kiểu): Toán tử instanceof trong java được sử dụng để kiểm tra một đối tượng có phải là thể hiển của một kiểu dữ liệu cụ thể không (lớp, lớp con, interface).

Interface: Một interface không phải là một lớp. Viết một interface giống như viết một lớp, nhưng chúng có 2 định nghĩa khác nhau. Một lớp mô tả các thuộc tính và hành vi của một đối tượng. Một interface chứa các hành vi mà một class triển khai.

Java Collection

+ Tập các thư viện class và interface có các thuật toán giúp người dùng làm việc với đối tượng

+ Chỉ làm việc với kiểu dữ liệu đối tượng

2 nhóm chính là Collection và Map (tồn tại độc lập)

* List: danh sách các phần tử có thể trùng nhau
* Set: tập hợp các phần tử khác nhau
* Queue: như xếp hàng, các phần tử được thêm vào cuối hàng
* Map: tập hợp giá trị key/value

List – ArrayList nên dùng cho việc sắp xếp và truy xuất dữ liệu

List – LinkedList phù hợp cho việc thêm sửa xóa dữ liệu

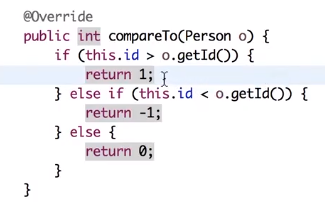
List – Vector

Sắp xếp các phần tử trong Collection

Collections.sort(list muốn sắp xếp);

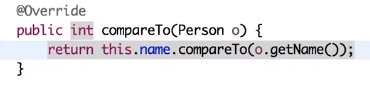
Comparable: implements Comparable<> -> Overrite lại hàm sắp xếp và viết trong hàm sắp xếp đó

+ Sắp xếp theo id tăng dần:



Nếu giảm dần thì đổi 1 thành -1 và -1 thành 1

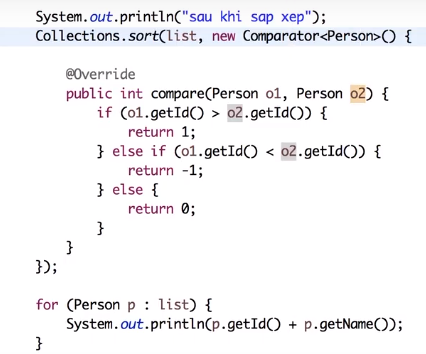
+ Sắp xếp theo bảng chữ cái:



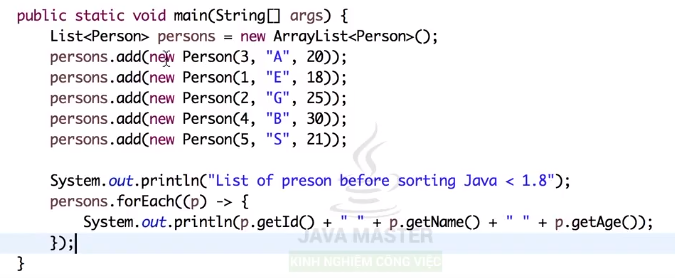
Ngược lại thì thêm – trước this

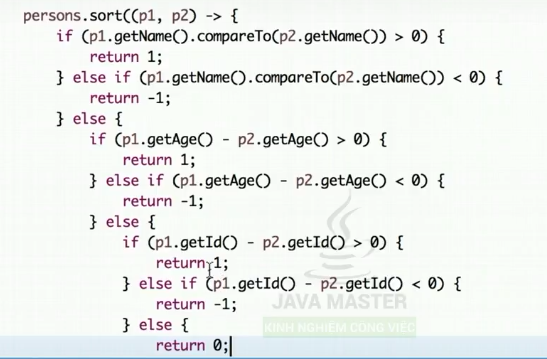
Comparator

So sánh các đối tượng trong List



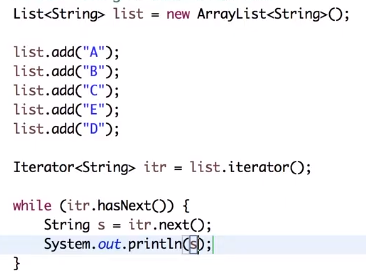
So sánh nhiều tiêu chí (tên, tuổi, địa chỉ)





Iterator

Dùng để lặp các phần tử trong List

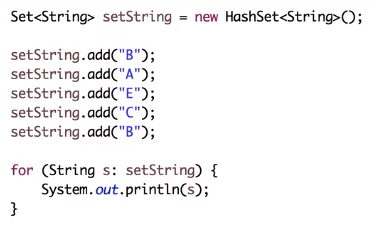


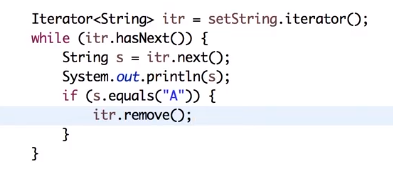
Khác nhau giữa forech và iterator: Iterator là phần duyệt mảng có thể vừa duyệt vừa thêm hay xóa phần tử được. Các phần khác như foreach không thể làm được điều này.

Set – HashSet

Tập hợp các phần tử là duy nhất (không trùng nhau) và tự sắp xếp tăng dần.

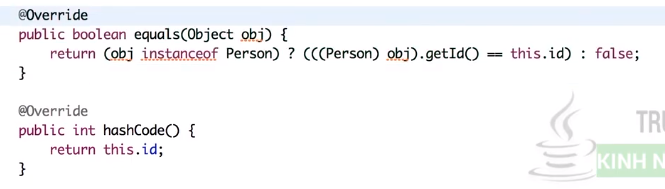
Có thể dùng forech hoặc iterator

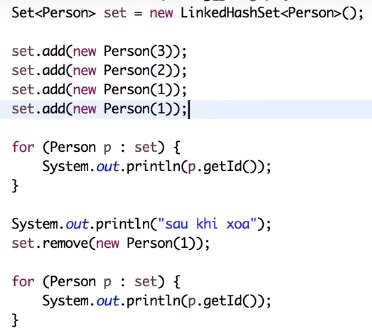




Set – LinkedHashSet

Các phần tử không được sắp xếp như HashSet

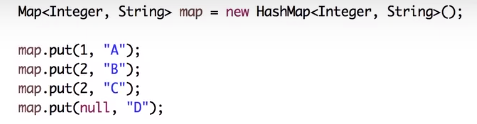




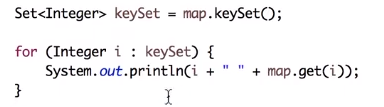
Map - HashMap

Key trong map là duy nhất

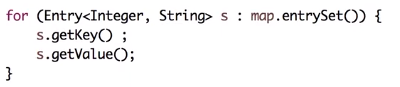
Tạo map:



Lặp các phần tử trong map:



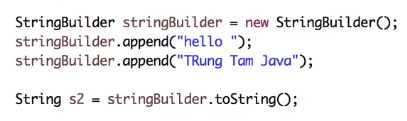
Lặp các phần tử trong map c2:



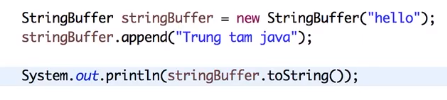
StringBuilder và StringBuffer

Tránh làm tiêu tốn bộ nhớ heap

StringBuilder : không có đồng bộ trong hàm



StringBuffer : tất cả các hàm đều được đồng bộ



JDBC



TRANSACTIONS

Khi thao tác cái JDBC sẽ cấp 1 transactions trong transactions sẽ có các thao tác thêm sửa xóa.

Nếu tất cả thao tác cùng 1 transactions ( thêm sửa xóa ) đều thành công thì sẽ commit. Nhưng không phải lúc nào các thao tác trong 1 transaction đều thành công. Nên trong JDBC sẽ có một cơ chế là rollback. Rollback có nghĩa là không cho tất cả các ( thêm sửa xóa ) thay đổi trong csdl: có nghĩa là nếu 1 cái thất bại thì các cái kia sẽ revert lại không cho đụng chạm gì đến csdl

DAO

Implement

Solid

OOP

3-tier

Static