**JAVA EFFECTIVE**

1. Tạo và Hủy các đối tượn*g*
   1. Cân nhắc sử dụng static factory method thay vì constructor

* Static factory method là một static method trả về 1 đối tượng của 1 lớp, nó không giống với Factory Method Pattern
* Ưu điểm của Static Factory Method so với Constructor:
  + Không giống constructor, factory method được đặt tên: nếu như constructor chỉ có 1 định dạng duy nhất cho dù có bao nhiêu tham số đầu vào. FM cho phép ta đặt tên để phân biệt các định dạng khác nhau giúp code tường minh hơn và có thể tránh trường hợp dùng sai constructor.
  + FM không yêu cầu tạo mới object trong mỗi lần gọi:
    - Cho phép immutable class có thể thể trả ra các object đã được tạo trước đó và tránh trường hợp tạo ra các object không cần thiết
    - Cho phép duy trì sự quản lý các object đã tồn tại (áp dụng trong các trường hợp như singleton).
  + Có thể trả về bất cứ kiểu dữ liệu con nào của class, cho phép trả về kiểu dữ liệu linh hoạt.
  + Class có thể trả ra các object tùy chỉnh dựa trên các tham số đầu vào.
  + Lớp mà đối tượng được trả ra có thể không cần phải tồn tại khi lơp chứa method được viết. điều này được áp dụng cho *service provider framework*
* Nhược điểm
  + Nhược điểm lớn nhất của SFM là class không có public hoặc protected construct không thể có subclass, có thể khắc phục điều này bằng cách sử dụng tổng hợp( composition) thay vì kế thừa (inheritance), hoặc yêu cầu kiểu không thế sửa đổi (immutable type).
  + Khó để người lập trình có thể tìm được method này
* Một số cách đặt tên phổ biến cho static factory method
  + From
  + Of
  + ValueOf
  + Instance or getInstance
  + Create, newInstance
  + getType
  + newTupe
  + type
  1. Sử dụng Builder trong trường hợp constructor có quá nhiều tham số
* Cả factory method và construct đều có một nhược điểm: khó có thể mở rộng trong trường hợp có nhiều tham số tùy chọn.
* Theo cách truyền thống:
  + Cách 1: Sử dụng *telescoping constructor pattern,* Ta thường tạo nhiều constructor tương ứng với danh sách các tham số được yêu cầu, cách làm này có nhược điểm ta phải viết rất nhiều constructor và gây khó khăn cho việc đọc code
  + Cách 2: sử dụng *JavaBean Pattern:*  Sử dụng constructor không tham số để tạo object sau đó sử dụng setter để gán các tham số được yêu cầu. Phương pháp này loại bỏ hạn chế của cách 1 tuy nhiên vân có các nhược điểm:
    - Có thể xảy ra trường hợp trạng thái không nhất quán (inconsistant state) trong quá trình khởi tạo đối tượng
    - Không thể sử dụng trong trường hợp immutable class.
* Để khắc phục nhược điểm của phương pháp truyền thông ta sử dụng Builder Pattern: để khởi tạo đối tượng ta gọi constructor hoặc static factory để trả về một *builder object*  sau đó sử dụng setter để truyền các tham số, cuối cùng gọi build method để tạo object. Builder là một static member class của class. Lưu ý tên của các Builder pattern giống với tên các tham số được set
* Ưu điểm của Builder so với constructor là có thể sử dụng mềm dẻo và tuy biến dựa trên số lượng các tham số cho việc khơi tạo
* Nhược điểm của builder là vấn đề hiệu năng vì cần tạo builder object vì thế chỉ nên sử dụng trong trường hợp có nhiều tham số trong việc khởi tạo object
  1. Bắt buộc thuộc tính singleton với private constructor hoặc kiểu enum
* Singleton đơn giản là 1 class chỉ có 1 thể hiện (instance), có thể là một đối tượng ko có trạng thái (stateless object) hoặc là 1 thành phần hệ thống về cơ bản là duy nhất
* Có 2 cách để triển khai, đều dựa trên việc giữ constructor private và chứa 1 member static để truy cập vào thể hiện
  + Cách 1: member là 1 trường public final, tuy nhiên phương pháp này vẫn có lỗ hổng, có thể truy cập vào constructor bằng reflection trong JAVA

// Singleton with public final fieldpublic class Elvis {  
 public static final Elvis INSTANCE = new Elvis(); private Elvis() { ... }  
 public void leaveTheBuilding() { ... }  
}

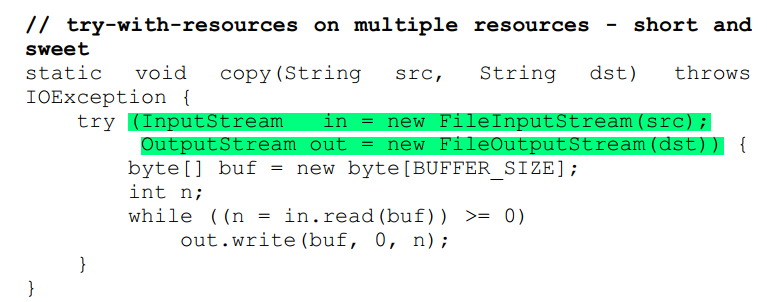
* + Cách 2: public member là 1 static method, ưu điểm của phương pháp này là dễ dàng để nhận ra nó là singleton,
  + // Singleton with static factorypublic class Elvis {  
     private static final Elvis INSTANCE = new Elvis();  
     private Elvis() { ... }  
     public static Elvis getInstance() { return INSTANCE;  
    }  
    public void leaveTheBuilding() { ... }  
    }
  + Cách 3: khai báo một single-element enum, đây là cách tốt nhất để triển khai singleton, nó cung cấp cơ chế tuần tự hóa miễn phí, đảm bảo chỉ có 1 thể hiện, chống lại reflection actack. Tuy nhiên nó lại có nhược điểm , ko thể dùng trong trường hợp nó là superclass

// Enum singleton - the preferred approachpublic enum Elvis {  
 INSTANCE;  
 public void leaveTheBuilding() { ... }  
}

* 1. Bắt buộc các class không cần có thể thiện với private Constructor
* Một số class có các trường và method đều là static (ví dụ các class trong gói util) mà ta không cần 1 instantion nào thì nên để private Constructor (vì nếu không khai báo thì Java sẽ có 1 public constructor mặc định cho các class). Việc này cùng ngăn chặn class có các subclass .
  1. Sử dụng DI (Dependency Injection) trong trường hợp kết nối đến các tài nguyên
* Nhiều class phụ thuộc vào các tài nguyên cơ bản (thông thường sẽ là 1 trường hoặc của class), sử dụng static utility class hoặc singleton không thích hợp trong trường hợp này
* Cách sử lý là truyền rescource vào *constructor* hoặc *static factory method* khi khởi tạo thể hiện của class, đây là 1 dạng của DI : IOC( inversion of control), dictionary sẽ phụ thuộc vào spellCheck và sẽ được *inject* vào SpellCheck khi nó được khởi tạo:

**// Dependency injection provides flexibility and  
testability**public class SpellChecker {  
 private final Lexicon dictionary;  
 **public SpellChecker(Lexicon dictionary)** {  
 this.dictionary =  
 Objects.requireNonNull(dictionary);  
 }

public boolean isValid(String word) { ... }  
 public List<String> suggestions(String typo) { ... }  
}

* Hạn chế của DI là khi project lớn và có quá nhiều đối tượng phụ thuộc sẽ rất khó quản lý, tuy nhiên đã có rất nhiều framwork hỗ trợ như Sping…
  1. Tránh việc tạo các object không cần thiết
* Việc sử dụng lại các object thay vì tạo mới thường được đánh gia cao đặc biệt là trong trường hợp việc khởi tạo đối tượng mới là “tốn kém”,
* Immutable object luôn luôn có thể tái sử dụng.
* Để tránh tạo các object không cần thiết ta nên sử dụng *factory method* thay vì *constructor.*
* Autoboxing là cơ chế cho phép chuyển đổi giữa kiểu nguyên thủy (*primitive*) và kiểu đối tượng (boxed primitive) cũng tạo ra cac object không cần thiết mỗi lần chuyển đổi vì thế ta nên ưu tiên sử dụng kiểu nguyên thủy .
* Tuy nhiên việc tái sử dụng các object đôi khi là không cần thiết bởi các lý do: làm phức tạp thuật toán, tăng dung lượng bộ nhớ, ảnh hưởng đến hiệu năng. Mô hình JVM đã hỗ trợ nhiều trong việc khởi tạo và thu gom cac đối tượng ko được sử dụng => Hạn chế tái sử dụng object, chỉ sử dụng trong trường hợp các đối tượng này khởi tạo rất tốn kém.
  1. Hạn chế sử dụng đối tượng tham chiếu đã lỗi thời
* Không giống như C, C++, ngôn ngữa JAVA có hỗ trợ cơ chế thu gom các đối tượng không còn được tham chiếu trên bộ nhớ (garbage-collection) mộ cách tự động
* Tuy nhiên có 1 số kiểu dữ liệu lại làm hạn chế tính năng này. Ví dụ như *STACK,*  đối tượng tuy đã được lấy ra khỏi stack nhưng stack vẫn giữ 1 tham chiếu lỗi thời (obsolete reference: là các đối tượng không bao giờ được hủy tham chiếu đến) làm cho cơ chế GC không thể thu gom các đối tượng này.
* Nếu một đối tượng tham chiếu được giữa lại một cách không chủ ý (*unintentional object retentions*) thì không chỉ các đối tượng này không được GC xử lý mà các đối tượng được nó tham chiếu đến cũng thế.
* Để xử lý trường hợp này, ta có thể khai báo = null khi không còn được tham chiếu đến, tuy nhiên cách làm này chỉ nên áp dụng với các trường hợp collection (chứa các thành phần là các tham chiếu đến các đối tượng)
* Một trường hợp khác của memory leak là các đối tượng được sử dụng như một bộ nhớ đệm (cache),có thể sử dụng *WeakHashMap*  để khắc phục.
* Một trường hợp khác của memory leak là listeners và callbacks, với trường hợp này chúng ta chỉ nên lưu một tham chiếu yếu (weak reference) tới nó, cách tốt nhất là dùng *WeakHashMap*
  1. Tránh sử dụng finalizers and cleaners
* Khi sử dụng các tài nguyên liên kết với object nên sử dụng *try-with-resource* hoặc sử dụng *try-finally*
  1. Nên sử dụng try-with-resource hơn là try-finally
* Trong các thư viện của java nhiều tài nguyên cần được đóng 1 cách thủ công bằng cách sử dụng method close, và cách tốt nhất là sử dụng try-finally trong đó sử dụng method close trong finallly, tuy nhiên cách này có thể ko hiệu quả nếu có nhiều hơn 1 resource
* Java7 đã giới thiệu cấu trúc *try-*with-resource, đê sử dụng nó thì resource cần được impliment ***Autocloseable*** inteface:
  + try (BufferedReader br = new BufferedReader(  
     new FileReader(path))) {  
     return br.readLine();  
     }
  + hoặc
  + 
* Try with ressourec làm code ngắn và dễ đọc hơn và trường hợp có exception có thể hữu ích cho việc trace hơn

1. Phương thức chung cho tất cả các đối tượng
2. ***Tuân theo các nguyên tắc chung khi ghi đè (override) phương thức equal()***

* Thông thường, để tránh các sai sót ta không nên ghi đè lại method equals
* Trong 1 số trường hợp, ta muốn so sánh 2 đối tượng xem có tương đương về mặt logic (không phải trường hợp tham chiếu đến cùng 1 object) thì cần phải khai báo phương thức *equals(),*  khi đó ta phải tuân theo các nguyên tắc sau:
  + ***Ánh xạ (reflexve):*** x.equals(x) luông trả ra true với x!= null
  + ***Đối xứng (symmetric):*** x.equals(y)= true khi và chỉ khi y.equals(x) với x,y!=null
  + ***Tính bắc cầu (transitive):*** nếu x.equals(y)=true và y.equals(z)=true thì x.equals(z)=true với x,y,z!=null
  + ***Tính nhất quán (consitency):*** x.equals(y) luôn phải trả ra cùng 1 giá trị true or false trong mọi trường hợp
  + ***x.***equals(null)=false trong mọi trường hợp
* Một số công thức (recipe) để thực hiện method equals():
  + Sử dụng toán tử == để kiểm tra đối số cùng tham chiếu đến 1 đối tượng.
  + Sử dụng toán tử *instance of*  để kiểm tra đối số có thuộc kiểu dữ liệu đúng
  + Cast đối số về đúng định kiểu dữ liệu (thực hiện sau khi kiểm tra instance of).

1. Classes và Inteface
2. Giảm thiểu phạm vi truy cập đối với classer và thành viên

* Nhân tố quan trọng để phân biệt một thiết kế tốt và thiết kế tối là khả năng giấu dữ liệu và chi tiết triển khai giữa các component, và chúng chỉ được giao tiếp với nhau qua API
* Việc ẩn dữ liệu này đem lại các lợi ích:
  + Tạo liên kết lỏng (decouple) giữa các thành phần trong hệ thống cho phép việc dev, test, tối ưu… trở nên độc lập
  + Dễ dàng trong quá trình vận hành: khi có lỗi xảy ra có thể dẽ dàng phát hiện và thay thế các componenr
  + Tái sử dụng code
* Làm cho các classer và member có phạm vi truy cập nhỏ nhất có thể:
  + Với classer level cap nhất chỉ có 2 mức độ truy cập: public (được phép truy cập trong toàn ứng dụng) private-packgae
  + Với thành viên của class (các trường, hàm, nested class) thì có 4 phạm vi truy cập:
    - Private: chỉ được truy cập trong pham vi class được khai báo
    - Default: chỉ được truy cập trong phạm vị package của class
    - Protected: Chỉ được truy cập trong phạm vị class và các class con
    - Public: được truy cập trên toàn ứng dụng

1. Trong Public class, sử dụng phươn thức truy cập, ko để public các trường
2. Giảm thiểu sự biến đổi

* Immutable class là các class mà các thể hiện của nó không thể bị sửa đổi, việc sử dụng immutable class đem lại lợi ích như dễ thiết kế, triển khai và sử dụng hơn là mutable class
* Để tạo ra immutable class cần tuân theo 5 luật sau:
  + Không cung cấp bất cứ hàm nào để sửa đổi trạng thái object
  + Đảm bảo các class không thể được kế thừa (ko có các subclass)
  + Các trường đều phải được khai báo final
  + Các trường đều phải được khái báo private
  + Loại trừ sự truy cập tới bất cứ thành phần có thể sử đổi của class
* Ưu điểm của immuable object
  + Immutable object là thread-safe, nó không yêu cầu sự đồng bộ
  + Immutable class có thể được sử dụng để cung cấp factory method chứa thể hiện thường xuyên được truy cập mà không cần phải tạo mới, điều này làm giảm mức chiếm dụng bộ nhớ và hỗ trợ GC làm việc
  + Ta không chỉ chi sẻ immutable object, ta có thể chia dẽ dữ liệu bên trong chúng
  + Immutable object giúp xấy dụng khối object ưu điểm hơn trong việc bảo trì
* Nhược điểm của immutable object:
  + Nhược điểm lớn nhất là ta phải tạo đối tượng mới khi thay đổi dữ liệu của đối tượng, việc này có thể làm giảm hiệu năng của hệ thống nếu như các đối tượng khởi tạo phức tạp
* Cách thay thế để tạo immutable class: thay vì phải để *final,*  ta có thể khai báo tất cả constructor là private và cung cấp *factory method:*

**// Immutable class with static factories instead of  
constructors**public class Complex {  
private final double re;  
private final double im;  
**private** Complex(double re, double im) {  
this.re = re;  
this.im = im;  
} **p  
ublic static Complex valueOf(double re, double im)**{  
return new Complex(re, im);

* Tóm tắt:
  + Classes nên là immutable trừ khi có lý do cần thiết để nó là immutable
  + Nếu class không thể là immutable, giói hạn các thành phần của nó immutable nhất có thể,
  + Khai báo các trường là private final khi có thể
  + Constructor nên khởi tại giá trị của object đầy đủ nhất có thể

1. Nên sử dụng các thành phần hơn kế thừa

* Kế thừa là một phương pháp mạnh mẽ để tái sử dụng code, tuy nhiên sự kế thừa chỉ thực sự an toàn nếu class cùng nằm trong package
* Không giống method invocation, sự kế thừa vi phạm quy tặc đóng gói, khi class cha thay đổi, nó buộc các class cũng phải thay đổi theo
* Sự kế thừa chỉ nên được dùng chỉ khi subclass thực sự là 1 *subtype*  của superclass
* Để khắc phục ta có thể sử dụng cách thiết kế được gọi là *composition*, class cần tham chiếu đến sẽ là 1 trường của class mới và đươc gọi là lớp bao trùm lên class cũ (wrapper class), đó cũng là nguyên lý thiết kế của *Decorator Pattern*
* Nhược điểm của wrapper là không thích hợp cho callback framework

1. Thiết kế và viết tài liệu co sự kế thừa hoặc cấm chúng

* Các class cần được viết tài liệu việc nó tự overrrider(ghi đè) các phương thức, bất cứ trường hợp nào nó triệu hồi 1 hàm được overrider
* Với các trường hợp class impl *Cloneable*  hoặc *Serializable*  không bao giờ ghi đè method *clone* và *readObject:*
  + Với trường hợp *readObject*: method được ghi đè sẽ chạy trước khi trạng thái của đối tượng được deserialized
  + Với trường hợp *clone:* phươn thức ghi đè sẽ chạy trước khi hàm clone của subclass có cơ hội cố định trạng thái của clone
* Khi implement Serializable mà class có thể được kế thừa và có các hàm *readResolve*
* vì nó sẽ được bỏ qua bởi subclass
* Vì vậy, các tốt nhất là cấm kế thừa class con đối với các class không được thiêt kế và tài liệu cho việc tạo các sub class an toàn. Có 2 cách để thực hiện:
  + Khai báo class là final
  + Khai báo các constructor là private và cung cấp 1 factory method
* Với trường hợp class impl các kiểu dữ liệu Map, Set, List nên sử dụng wrapper class hơn là kế thừa

1. Ưu tiên Interface hơn là abtract Class

* Các class đã tồn tại có thể được trang bị thêm một cách dễ dàng bằng việc impl thêm interface vì interface hỗ trợ đa kế thừa
* Interface lý tưởng cho việc tạo kiểu dữ liệu hỗn hợp
* Interface cho phép framework tạo ra kiểu cấu trúc không cần thứ bậc Ví dụ như ta có 2 interface *Single* và *SongWrite,*  trong một số trường hợp 1 người có thể vừa là single vừa là songwriter, ta có thể tạo ta một interface khác *SinglerSongWriter implement Single, Songwriter*
* Interface cho phép cải tiến các hàm an toàn và mạnh mẽ thông qua wrapper class
* Tuy nhiên interface có hạn chế là không thể khai báo trường hoặc method với phạm vi truy cập private (từ java8 đã có thêm *default method*), để khắc phục, ta có thể khai báo interface để định nghĩa kiểu dữ liệu và class xương sống để chứa các hàm không nguyên thủy

1. Thiết kế interface cho sử dụng lâu dài

* Phiên bản Java8 bổ sung thêm default method cho khép khai báo và định nghĩa method trong interface tuy nhiên có một điểm lưu ý là method này sẽ được “inject” trong tất các các triển khai mà có thể không có được hiểu biết hoặc sự đồng ý với các method này
* Sự hiện diện của default method có thể không vấn đề tại thời điểm compiler nhưng có thể lỗi khi chạy

=> *Hết sức cẩn thận khi thiết kế default method*

1. Chỉ sử dụng interface để định nghĩa kiểu dữ liệu

* Khi 1 class implement 1 interface, interface cung cấp 1 kiểu dữ liệu được sử dụng để tham chiếu đến bất kì thể hiện nào của class, interface không thực sự thích hợp cho bất kì mục đích nào khác ngoài định nghĩa kiểu dữ liệu
* Một kiểu sử dụng interface là constant interface (không chứa các method, chỉ chứa trường static final ) đây là một cách sử dụng interface một các nghèo nàn: nếu 1 class impl 1 constant interface, các subclass của nó sẽ bị nhầm lẫn hoặc dư thừa *namespace* của interface => nên sử dụng *enum type* hoặc *class utilities* để thay thế constant interface
* Nên sử dụng dấu “\_” ngăn cách các từ trong constant (MAX\_VALUE, MIN\_VLAUE) và trong các số (123\_456\_789) để dễ đọc hơn

1. Ưu tiên sự phân cấp class hơn là class được gắn

* Tagged class (class được gắn thẻ) thường dài dòng, dễ lỗi và không hiệu quả:
* Thay vào đó, các ngôn ngữ hướng đối tượng cung cấp 1 sự thay thế tốt hơn cho các đối tượng có nhiều kiểu thể hiện: kiểu dữ liệu con (subtype)
* Để chuyển đổi từ tagged class sang class hierarchy:
  + đầu tiên ta định nghĩa 1 abtract class chưa các abtract method mà cách xử lý phụ thuộc vào dữ liệu đầu vào
  + Định nghĩa các class con cụ thể (conrete class) mà tagged class có thể tạo ra
* Tóm lại, ưu điểm của sự phân cấp class so với tagged class là có thể tham chiếu đến bất cứ kiểu dữ liệu hoặc dữ liệu con nào, cho phép tăng sự linh hoạt và hỗ trợ tốt hơn cho việc kiểm tra dữ liệu tài thời điểm compiler

1. Ưu tiền các static member hơn là nonstatic trong class

* Khi khai báo 1 member class mà không yêu cầu truy cập đến instance kèm theo thì luôn luôn sử dụng static khi khai báo: vì mỗi instance của inner class sẽ được tham chiếu bởi 1 instance kèm theo của outer class, việc lưu các tham chiếu này tốn thời gian và bộ nhớ và có thể dẫn tới memory leak

1. Giới hạn file nguồn ở một lớp cao nhất

* Rủi ro khi định nghĩa nhiều top-level class cho file nguồn(file .java) là có thể cung cấp nhiều định nghĩa cho class
* Giả sử ta có file A.java chứa 2 class là tên X và tên Y, tuy nhiên file B.class cũng chứa class tên X và class tên Y(khác với X và Y trong file A.java), khi compiler sẽ tùy thuộc vào thứ tự (A.java trước, B.java sau hoặc ngược lại) để xác định được đối tượng của class X hay class Y trong file A hay B

1. **GENERICS**
2. ***Đừng sử dụng kiểu dữ liệu thô***

* Một class hoặc interface mà sự khai báo có 1 hoặc nhiều kiểu dữ liệu tham số là các generic class hoặc interface. Ví dụ: *List<E>, Set<E>…*
* Mỗi kiểu generic sẽ định nghĩa một tập các kiểu tham số.
* Tuy sử dụng các kiểu thô (kiểu generic không chứa các kiễu dữ liệu tham số), tuy nhiên ta cũng không nên sử dụng vì không thể hỗ trợ việc kiểm tra lỗi trong quá trình biên dịch.
* Một nguyên tắc về kiểu dữ liệu phụ cho generic*: List<String>* là kiểu dư liệu con của kiểu thô *List* nhưng không phải là kiểu dữ liệu con của *List<Object>*
* Trong trường hợp bạn muốn sử dụng kiểu generic nhưng không muốn quan tâm đến kiểu dữ liệu tham số, ta có thể sử dụng *unbounded wildcard types.* Ví dụ List<E>, Set<E> hoặc List<?>, Set<?>, … Kiểu khai báo này cũng ngăn chặn việc ta có thể push bất cứ kiểu dữ liệu gì vào trong Collection
* Một số trường hợp cần phải sử dụng kiểu dữ liệu thô:
  + Trong các *class literals.* Ví dụ : List.class, Set.class
  + Sử dụng toán tử *instance of.* Ví dụ: s *intance of* Set

1. ***Hạn chế unchecked warning***

* Nếu ta hạn chế các unchecked warning khi sử dụng generic, ta có thể đảm bảo code là typesafe, sẽ không gặp *ClassCastException*
* Nếu ta không thể hạn chế các lỗi này nhưng có thể đảm bảo lỗi sẽ không xảy ra ta có thể sử dụng annotation: *@SuppressWarning(“unchecked”)* , tuy nhiên nên sử dụng trong phạm vi nhỏ nhất có thể. Nên thêm các comment để giải thích vì sao sử dụng nó lại an toàn

1. ***Nên sử dụng List thay vì Array***

* Có 2 điểm khác biệt giữa List và Array:
  + *Array* là đồng biến, điều này có nghĩa nếu *Sub* là kiểu dữ liệu con của *Super* thì *Sub[]* cũng là kiểu con của *Supper[].* Còn List là bất biến: nếu Sub là kiểu dữ liệu con của Super thì *List<Sub>* không phải là kiểu dữ liệu con của *List<Super>*
  + Array là thống nhất (reified) có nghĩa là array biết và thi hành kiểu dữ liệu của nó tại thời điểm runtime, còn Genaric đã đươc tẩy bỏ (earsure) nó chỉ thực thi kiểu dữ liệu tương ứng tại thời điểm compiler
* Vì thế không thể tạo các mảng của 1 kiểu generic hoặc kiểu tham số. Ví dụ: List<E> [], new E[] là không hợp lệ.

1. ***Nên sử dụng các kiểu Generic***

* Các bước để thực hiện kiểu khái quát hóa (*genarify*) :
  + Khái quát hóa class bằng cách thêm một hoặc nhiều kiểu tham số trong khai báo của nó.
  + Thay thế toàn bộ các chỗ sử dụng kiểu Object bằng kiểu tham số thích hợp.
* Tuy nhiên ta không thể khởi tạo 1 màng các kiểu non-reifiable(các kiểu ta không thể có được toàn bộ thông tin của nó tại thời điểm runtime) Ví dụ: **new E[DEFAULT\_INITIAL\_CAPACITY]**;
  + Cách xử lý thứ nhất là tạo 1 mảng object và cast sang kiểu dữ liệu tương ứng: **(E[]) new Object[DEFAULT\_INITIAL\_CAPACITY]**. Tuy nhiên ta phải chứng minh rằng không cần phải lo lắng về *unchecked cast.*
  + Cách thứ 2 để hạn chế các lỗi tạo mảng generic là chuyển đổi các trường element từ E[] thành Object[]: E result = **(E)** elements[--size];
* Vì E là kiểu non-reifiable nên không các cách nào để compiler có thể kiểm tra việc cast tại thời điểm runtime.
* => ***kiểu generic thì an toàn và dễ sử dụng hơn các kiểu dữ liệu yêu cầu cast trong client code. Khi thiết kế một kiểu dữ liệu đảm bảo rằng nó có thể sử dụng mà không cần phải cast (thiết kế các kiểu generic)***

1. ***Nên sử dụng các hàm generic***

* Cũng giống như các kiểu dữ liệu, các hàm cũng có thể thiết kế kiểu generic. Các hàm tiện tích static mà hoạt động dựa trên parameter type đều là generic.
* Danh sách các kiểu dữ liệu tham số (khai báo các kiểu tham số ) sẽ nằm từ khi khai báo phạm vi truy cập cho đến kiểu trả về cho hàm.
* Một vài trường hợp, ta cần các các immutable object nhưng có thể chấp nhận nhiều kiểu dữ liệu khác nhau, ta có thể tạo ra một single Object mỗi khi có yêu cầu một kiểu tham số hóa. Khi đó ta sẽ sử dụng *generic singleton factory.*

1. ***Sử dụng các kí tự đại điện giới hạn (bounded wildcard) để tăng tính linh hoạt cho API***

* Kiểu dữ liệu tham số là bất biến (A là kiểu con của B nhưng List<A> không phải là kiểu con của List<B>)
* Java cung cấp một dạng đặc biệt của kiểu dữ liệu tham số được gọi là *bounded wildcard type*>. Ví dụ *Iterable<? extends E>* hoặc *Collection<? super E>*
* Để tối đa hóa sự linh hoạt, sử dụng wildcard type trong các tham số đầu vào biểu diễn producer hoặc consumer theo nguyên tắc: PECS (producer-extend, consumer-super)
* Không sử dụng bounded wildcard type với các kiểu return

1. ***Kết hợp generic và vararg một cách khôn ngoan***

* Vì gần như tất cả các kiểu generic và tham số hóa đều là non-reifiable (kiểu dữ liệu mà thời điểm runtime sẽ có ít thông tin hơn là thời điểm compiler) nên khi khai báo kiểu generic là tham số đầu vào của các hàm vargar có thể gây ***heap pollution*** ( xảy ra khi một biến của một kiểu tham số tham chiếu đến một object không đúng kiểu của nó)
* Một phương pháp đước sử dung là thay thế tất cả tham số vargar bằng tham số List. Ưu điểm là compiler có thể đảm bảo hàm có kiểu dữ liệu an toàn. Nhược điểm là làm giảm hiệu năng

1. ***Cân nhắc container của kiểu dữ liệu an toàn không đồng nhất (typesafe heterogenerous container)***

* Một số kiểu dữ liệu container như collection bị giới hạn số lượng kiểu tham số truyền vào, ý tưởng để khắc phục là tham số hóa key thay vì container và dùng chúng để push hoặc get giá trị

1. **METHOD**
2. ***Kiểm tra hiệu lực của các tham số***

* Hầu hết các method đều có một vài hạn chế cho các tham số đầu vào của nó, ta nên kiêm tra các điều kiện này ngày tại đầu của method để phát hiện sớm và hạn chế nhất các lỗi có thể xảy ra
* Trường hợp các tham số vị phạm ta nên ném ra 1 ngoại lệ (Exception): *IllegalArgumentException, IndexOutOfBoundsException, or NullPointerException*
* Java7 cung cấp hàm *Objects.requireNonNull* là 1 hàm check null rất linh hoạt và tiện lợi, vì thế ta nên sử dụng hàm này
* Một trường hợp quan trọng khác là check các tham số không được sử dụng bởi method nhưng được dùng để chứa kết quả sau này
* Một ngoại lệ trong việc kiểm tra tham số đầu vào trước khi thực hiện việc tính toán là khi việc kiểm tra này tốn kém và gây ảnh hưởng ví dụ: Collections.sort(List) mỗi phần tử trong list cần được so sánh với các phần tử còn lại

1. Ngăn chặn việc sao chép các đối tượng khi cần thiết

* Với ngôn ngữ an toàn như JAVA có thể tạo các class và biết được chúng là bất biến trong bất kỳ hoàn cảnh nào. Tuy nhiên khi lập trình bạn cần đặt giả thiết client có thể cố gắng phá hủy sự bất biến của class, có thể sao chép các thể hiện của 1 class và truyền vào dữ liệu không mong muốn
* Kiểu dữ liệu *Date* đã lỗi thời và không nên sử dụng thay vào đó ta nên sử dụng Instant (hoặc Local-DateTime hoặc Zone-DateTime),
* Ta nên ngăn chặn việc sao chép các tham số có thể thay đổi ngay tại constructor và chú ý việc ngăn chặn sao chép được thực hiện trước khi kiểm tra hiệu lực của các tham số đầu vào. Việc này sẽ ngăn chặn hành động thay đổi dữ liệu giữa khoảng thời gian check và thời gian copy
* Không sử dụng *clone()*  để ngăn chặn việc sao chép tham số từ các kiểu dữ liệu là class con từ các nguồn ko tin cậy
* Một trường hợp khác: các tham số truyền vào hàm hoặc constructor có phải đang tham chiếu đến một đối tượng chứa trong cấu trúc dữ liệu nào đó ko? Việc thay đổi này có thể dẫn đến lỗi ảnh hưởng đến sự bất biến của Set hoặc Map trong trường hợp nó bị sửa đổi sau khi đã insert vào
* Ta cũng cần ngăn chặn sao chép trong thời gian trả về kết quả cho client

1. Thiết kế ký hiệu method cẩn thận

* Chọn tên cho method cẩn thận: Tên của method nên tuân theo quy ước, mục đích chính trong việc chọn tên là dễ hiểu và nhất quán với tên khác trong cùng package, muc đích thứ 2 là chọn tên với sự đồng thuần rộng rãi, tránh các đặt tên quá dài
* Đừng quá lan man trong việc cung cấp cá method tiện ích: việc có quá nhiều method làm cho class khó trong việc học, sử dụng, test, bảo trì…Với mỗi hành động được hỗ trợ bởi class thì hãy cung cấp method với đầy đủ chức năng, chỉ nên cung cấp các method “shorthand” nếu chúng thường xuyên được sử dụng
* Tránh việc quá nhiều tham số đầu vào, chỉ nên có 4 tham số hoặc ít hơn. Việc có quá nhiều tham số đầu vào sẽ gây khó nhớ, có thể gây nhầm lẫn. Có 3 kỹ thuật để giải quyết vấn đề này:
  + Chia thành các method nhỏ hơn
  + Tạo các helper class để chứa tập tham số
  + Kết hợp 2 phương pháp trên để tạo ra Builder pattern, sử dụng khi có rất nhiều tham số đầu vào tuy nhiên có nhiều tham số là tùy chọn
* Với kiểu dữ liệu tham số đầu vào nên sử dụng inteface hơn là class
* Nên sử dụng kiểu enum với 2 thành phần hơn là sử dụng boolean

1. Sử dụng nạp chồng một cách khôn ngoan

* Giả sử class có nhiều phương thức được overloaded, việc sử dụng method nào được quyết định tại thời điểm *biên dịch.*Lựa chọn phương thức nạp chồng nào là tĩnh còn việc lựa chọn phương phức ghi đè nào là động (tại thời điểm runtime)
* Không thiết kế các phương thức nạp chồng có cùng số lượng tham số chỉ có thể sử dụng nếu một trong các tham số tương ứng khác nhau về kiểu (không thể cast cho nhau) hoặc ta có thể thay thế bằng phương thức có tên khác trong trường hợp này

1. Sử dụng đối số linh động (varargs) theo cách khôn ngoan

* *Varargs method*  được biết đến như là các hàm có tính biến đổi, chấp nhận đầu vào có thể là không có hoặc nhiều đối số của cùng một kiểu
* Hoạt động như sau: tạo 1 array với kích thước bằng với số lượng đối số được truyền vào, đẩy các đối số này vào mảng và cuối cùng pass đến hàm
* Trường hợp sử dụng varargs method nhưng bắt buộc có ít nhất 1 tham số truyền vào nhưng client lại ko truyền vào, để khắc phục trường hợp này, ta có thể sử dụng như sau:

static int min(int firstArg, int... remainingArgs)

* Tuy nhiên sử dụng hàm có đối số linh động sẽ gây ảnh hưởng đến hiệu năng vì cần phải tạo và phân bổ 1 mảng

1. **Trả về empty collections hoặc array, không trả về null**

* Khi trả về null thay vì collections hoặc array, người dùng sẽ phải thêm bước kiểm tra giá trị trả về có phải là null hay không
* Một số lập luận cho rằng trả về empty collection hoặc array thay vị *null*  sẽ làm giảm hiệu năng của hệ thống vì cần phải phân bố vùng nhớ cho container tuy nhiên điều này không hoàn toàn chính xác. Ta có thể tối ưu trường hợp trả về empty collections bằng cách trả về một immutable object được chia sẽ hoặc sử dụng các hàm được cấp sẵn như *Collections.emptyList,* *Collections.emptySet*
* Với trường hợp array: không bao giờ được trả về null thay vì mảng rỗng (mảng có độ dài bằng không Array[0]). Zero-length array is immutable

1. **Trả về optionals một cách khôn ngoan**

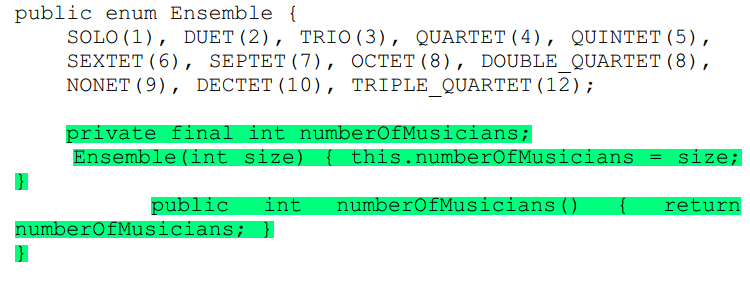
* Trước Java8 có 2 cách viết khi một hàm không thể trả về một giá trị trong trường hợp cụ thể và cả 2 cách này đều không hoản hảo:
  + Ném ra một ngoại lệ: chỉ nên sử dụng trong điều kiện cụ thể và cách này cũng khá “đắt đỏ” vì toàn bộ stack trace được chụp khi ngoại lệ được tạo
  + Trả về *null:* điều này làm client phải viết thêm trường hợp khi trả về giá trị *null*
* Với Java8 có 1 cách tiếp cận khác khi hàm không thể trả về 1 giá trị : *Optional<T>* là một immutable container có thê chứa 1 tham chiếu của kiểu T hoặc không có gì (được gọi là *empty*)
* Sử dụng Optionals thì linh hoạt và dễ sử dụng hơn throw exception và hạn chế lỗi hơn trả về null
* Nếu method sử dụng Optional<T> thì không bao giờ được trả về giá trị null
* Các kiểu container như Map, Set, List, Stream, Array không nên được bọc trong optionals, thay vì trả về *Optional<List<T>>* thì ta nên trả về *List<T>* rỗng
* **Khi nào nên sử dụng kiểu trả về *Optional<T>* thay vì T: nếu hàm không thể trả ra dữ liệu và client phải xử lý đặc biệt trong trường hợp không trả ra dữ liệu đó.**
* Trường hợp trả về optional có chứa kiểu dữ liệu boxed primitive thì phức tạp và tốn kém hơn do optional cần phải có 2 level boxed thay vì giá trị 0, do đó một số thư viện đã thiết kế các kiểu dữ liệu OptionalInt, OptionalLong, OptionalDouble => ***không bao giờ trả ra Optional của các kiểu dữ liệu boxed primitive***
* **Không sử dụng optional để đóng vai trò key, value hoặc một element trong collection hoặc array**

1. **Viết tài liệu cho tất cả các thành phần API**
2. **Enum and Anotation**
3. **Sử dụng enum thay vì các hằng số kiểu int**

* Enum là một kiểu dữ liệu bao gồm các giá trị hợp lệ của một bộ các hằng số,
* Trước đó người ta sử dụng int enum pattern: khai báo một bộ các hằng số kiểu *int,*  hoặc một vài biến thể như *String enum pattern.* Tuy nhiên phương pháp này không hiệu quả trong việc hiển thị và có thể làm giảm hiệu năng do phụ thuộc vào so sánh string.
* Ý tưởng của enum là tạo ra một instance cho mỗi hằng số enum thông qua một trường static final và không thể truy cập constructor.
* Enum là một dạng của Singleton Pattern
* Có thể sử dụng toán tử “==” để so sánh giá trị của 2 kiểu enum khác nhau
* Enum cung cấp sự bổ sung chất lượng cao các hàm của Objects , bổ sung *Comparable* và Serial
* Để kết hợp dữ liệu với hằng số enum , ta có thể khai bao trường dữ liệu và viết constructor lấy data và chưa trong trường dữ liệu đó. Vì enum là singleton ta nên khai báo các trường là final
* Một nhược điểm của các hàm triển khai hằng số cụ thể là làm khó khăn trong việc chia sẽ code giữa các hằng số enum.
* Switch-case nên sử dụng enum

1. **Sử dụng các thể hiện của trường thay vì thứ tự**

* Enum cung cấp hàm *ordinal()* giúp trả về thứ tự của hằng số enum trong kiểu dư liệu, tuy nhiên ko nên sử dụng bởi nó gây khó khăn trong việc bảo trì vì khi thay đổi dữ liệu của kiểu enum thì thứ tự cũng sẽ bị thay đổi theo
* Thay vào đó ta nên thiết lập giá trị được chứa trong các trường thể hiện. ví dụ



1. **Sử dụng enum thay cho trường bit**
2. **Sử dụng EnumMap thay vì chỉ số thứ tự**
3. **Mô phỏng các enum có thể mở rộng với các interface**
4. **Sử dụng annotations cho các mẫu đặt tên**

* Trong lịch sử, naming pattern thường được sử dụng để đánh dấu một phần của các chương trình sẽ được xử lý đặc biệt bởi các tool hoặc framework, việc này có một số nhược điểm:
  + Có thể gây sai sót nêu tên không được chính xác
  + Không có cách nào đảm bảo rằng chúng chỉ được sử dụng trong các thành phần thích hợp của chương trình .
  + Không thể cung cấp một phương pháp tốt để liên kết giá trị của các tham số với các thành phần của chương trình
* Annotation có thể giải quyết được các nhược điểm này

1. **Nhất quán trong việc sử dụng Override Annotation**
2. **Sử dụng các interface được đánh dấu cho định nghĩa kiểu dữ liệu**

* Các lợi ích của việc sử dụng các marker interface so với annotation:
  + Marker interface định nghĩa kiểu dữ liệu được triển khai bởi các instance của các marker class.
  + Giúp được nhắm tới một cách chính xác hơn

1. **Lambdas and Stream**
2. ***Nên sử dụng lambdas cho các anonymous class***

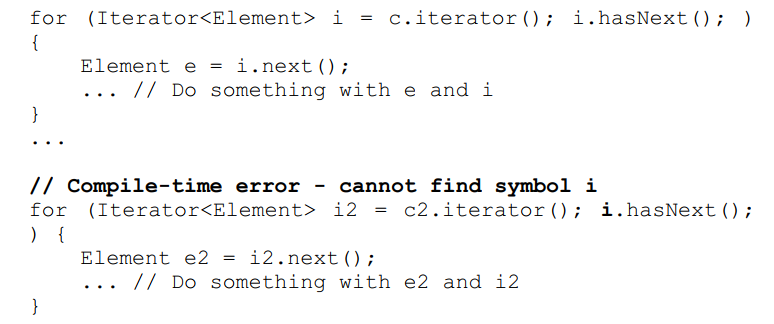
* Lambdas tương tự như các hàm trong anomynous class
* Tuy nhiên khi sử dụng lambdas compiler không thể xác định được kiểu dữ liệu và bạn phải xác định chúng, tuy nhiên chúng ta có thể bỏ qua tất cả các tham số của lambdas trừ khi biểu diễn chúng làm chương trình trở nên sáng hơn
* Các đoạn code cho lambdas chỉ nên có 1 dòng và tối đa là 3 dòng
* Lambdas có nhược điểm so với anomynous class là nó bị giới hạn trong hàm chức năng inteface, nếu muốn tạo 1 instance của abstract class thì ta chỉ có thể dùng với anomynous class
* Không nên sử dụng lambdas cũng như anomynous class trong trường hợp serializable

1. ***Sử dụng hàm tham chiếu thay cho lambdas***
2. ***Ưu tiên sử dụng functional inteface tiêu chuẩn***
3. ***Sử dụngStream một cách khôn ngoan***

* Stream API đưa ra 2 khái niệm:
  + *Stream*: Biễu diễn một chuỗi hữu hạn hoặc vô hạn thành phần dữ liệu
  + Stream pipeline: biễu diễn một sự tính toán nhiều tầng trên các thành phần
* Các thành phần dữ liệu có thể là các object tham chiếu hoặc kiểu dữ liệu nguyên thủy (chỉ hỗ trợ *int, long* và *double*)

1. **General Programming**
2. ***Giới hạn phạm vi của các biến cục bộ***

* Cách tốt nhất để giới hạn phạm vi của các biến cục bộ là chỉ nên khai báo khi dùng nó trong lần đầu tiên. Khai báo biến quá sớm không chỉ gây ra phạm vi bắt đầu quá sớm mà có thể gây ra phạm vi kết thúc muộn
* Mọi biến cục bộ được khai báo nên có một giá trị khởi tạo. Tuy nhiên try-catch là một trường hợp ngoại lệ: nếu việc khởi tạo biến từ một hàm có thể ném ra một ngoại lệ thì ta nên khởi tạo biến trong phạm vi try-catch
* Nên sử dụng vòng lặp for thay vì while vì for loop cho phép khai báo các biến cục bộ và giới hạn chúng trong vùng mà ta cần: ví dụ đây là lỗi copy-paste và đoạn code này sẽ không được biên dịch



1. ***Nến* sử dụng for-each thay cho vòng lặp for truyền thống**

* Lợi thế của việc sử dụng for-each là dấu việc duyệt vòng lặp và các biến index làm cho code đỡ lộn xộn và hạn chế lỗi
* Một số trường hợp không thể sử dụng for-each:
  + Phá hủy các bộ lọc(***Destructive filtering***): trong trường hợp duyệt một collection và remove thành phần vừa được duyệt, ta cần phải duyệt và gọi hàm remove, nên sử dụng hàm *removeIf()* (được thêm vào từ Java8)
  + Biến đổi(***Transforming***): trong trường hợp duyệt 1 List hoặc Array và muốn thay thế một hoặc nhiều phần tử của mảng đó
  + Duyệt song song (***Parallel iteration***):
* Nếu bạn viết một kiểu dữ liệu thể hiện là một tập các thành phần thì nên cân nhăc việc implement *Iterable*

1. ***Hiểu và sử dụng các thư viện***

* Lợi ích đầu tiên: Bằng việc sử dụng các thư viện chuẩn ta có được nhiều lợi ích từ sự hiểu biết của các chuyên gia
* Lợi ích thứ 2: Ta không cần phải tốn thời gian nghĩ ra các giải pháp cho các vấn đề đã được giải quyết mà chỉ cần tập trung vào nghiệp vụ
* Lợi ích thứ 3: hiệu năng của các giải pháp này được cải thiện theo thời gian qua các phiên bản khác nhau
* Lợi ích thứ 4: Có được nhiều chức năng hơn qua các phiên bản theo thời gian
* Lợi ích cuối cùng là: Code của bạn chỉ trên luồng chính. Điều này làm cho code dễ đọc, dễ bảo trì và tái sử dụng hiệu quả

1. ***Tránh dùng float và double nếu yêu cầu kết quả chính xác***

* Không nên dùng ***float*** và ***double*** trong các trường hợp tính toán liên quan đến tiền tệ, thay vào đó nên sử dung ***BigDecimal, int, long***
* Hai nhược điểm của ***BigDecimal*** so với ***int*** và ***long***: ít tiện lợi hơn và có một chút chậm hơn
* Lưu ý: nếu số lượng không vượt quá 9 chữ số thì nên sử dụng ***int***, không vượt quá 18 chữ số thì sử dụng ***long*** và nếu vượt quá 18 chữ số thì sử dụng ***BigDecimal***

1. ***Nên sử dụng kiểu nguyên thủy (primitive type) hơn là sử dụng kiểu boxed primitive***

* Mỗi kiểu nguyên thủy như int, long, double đều có kiểu tham chiếu tương ứng và cơ chế autoboxing và auto-unboxing cho phép chuyển đổi giữa 2 kiểu này.
* Có 3 điểm khác biệt lớn nhất giữa 2 kiểu :
  + Kiểu nguyên thủy chỉ chứa giá trị của nó trong khi kiểu boxed còn chứa cả thông tin để phân biệt giữa các giá trị của chúng()
  + Kiểu boxed có 1 giá trị phi chức năng (nonfunctional value) là null, còn lại các giá tri chức năng đều có giá trị tương ứng với kiểu nguyên thủy
  + Kiểu nguyên thủy có hiệu năng cao hơn (về thời gian và mức chiếm dụng bộ nhớ) so với kiểu boxed
* Sử dụng toán tử “==” để so sánh 2 đối tượng boxed luôn luôn là sai cách
* Khi sử dụng hỗn hợp kiểu nguyên thủy và kiểu boxed trong cùng một toán tử thì ***kiểu boxed sẽ được auto-unboxed***
* Khi nào nên sử dụng kiểu boxed: sử dụng để push vào các Collection
* Nên sử dụng kiểu nguyên thủy thay vì boxed vì chúng đơn giản và có hiệu năng cao hơn

1. ***Tránh sử dụng string khi có những kiểu dữ liệu khác thích hợp***

* String là một sự thay thế nghèo nàn cho các kiểu dữ liệu khác: thông thương dữ liệu đầu vào là string nhưng chúng ta nên biến đổi chúng về các kiểu dữ liệu thích hợp.
* Nên sử dụng kiểu enum thay vì string
* String là một sự thay thế nghèo nàn cho các kiểu dữ liệu tổng hợp, thay vào đó ta nên viết các class để thể hiện cho các kiểu dữ liệu này

1. ***Hiểu được hiệu năng của sự nối chuỗi string***

* Sử dụng toán tử nối chuỗi để nối n chuỗi với nhau sẽ tốn O(n2) vì string là kiểu immutable, khi nối 2 chuỗi với nhau thì nội dung của 2 chuỗi sẽ được copy
* Sử dụng StringBuilder cho việc nối chuỗi sẽ cho hiệu năng cao hơn

1. ***Tham chiếu đến object bằng inteface của chúng***

* Nên sử dụng interface để sử dụng cho các kiểu dữ liệu trong các trường hợp các tham số , trả về giá trị , các biến và các trường, chỉ thực sự cần tham chiếu đến class khi sử dụng trong constructor
* Sử dụng interface đem lại lợi ích là chương trình sẽ linh hoạt hơn

1. ***Sử dụng interface cho reflection***

* Reflection cho phép xấy dựng các instance, gọi hàm và truy cập vào các trường của class . tuy nhiên có một số nhược điểm của việc sử dụng reflection:
  + Mất đi các lợi ích của việc kiểm tra kiểu dữ liệu tại thời điểm biên dịch.
  + Làm cho code vụng về và dài dòng.
  + Ảnh hưởng đến hiệu năng

1. ***Sử dụng hàm native một cách khôn ngoan.***
2. ***Tối ưu một cách khôn ngoan***

* Việc tối ưu nên được thực hiện sau khi các nghiệp vụ đã thực hiện hoàn chỉnh, không nên cố thiết lập việc tối ưu hiệu năng ngày từ đầu: ***Cố gắng viết một chương trình tốt hơn là một chương trình nhanh.***
* Một chương trình tốt là chương trình tuân theo quy tắc che dấu thông tin: việc thay đổi một hoặc một số thành phần sẽ không ảnh hưởng đến phần còn lại của hệ thống
* Cố gắng tránh việc lựa chọn các thiết kế làm giới hạn hiệu năng: Thành phần khó thay đổi nhất trong hệ thống là các thành phần phụ trách việc giao tiếp với các hệ thống bên ngoài (API, giao thức liên kết, truy cập dữ liệu…)
* Cân nhắc hệ quả của hiệu năng trong việc lựa chọn thiết kế API, không nên làm *cong vênh*  API chỉ để nâng cao hiệu năng vì điều này sẽ gây khó khắn trong việc nâng cấp API trong tương lai

1. ***Tuân thủ các quy ước đặt tên***

* ***Quy ước đặt tên theo hình thể:*** 
  + Tên của package và module chỉ nên bao gồm các chữa cái in thường và hiếm khi chứa các chữ số. Phần còn lại của package nên mô tả chức năng của nó và có thể chấp nhận từ viết tắt
  + Tên của class, inteface, enum và anotation nên bao gồm một hoặc nhiều từ và viết hoa chữ cái đầu tiên của mỗi , nên tránh sử dụng từ viết tắt trừ khi từ viết tắt mang ý nghĩa rõ ràng ví dụ: min, max
  + Các hàm và các trường cũng nên tuân theo quy tắc như trên ngoại trừ từ đầu tiên sẽ được viết thường
  + Với trường hợp các hằng số thì sẽ viết hoa tất cả và các từ ngăn cách nhau bằng dấu gạch dưới ví dụ MAX\_INTEGER.
  + Với tên các biến cũng nên tuân theo nguyên tắc như tên các hàm và các trường, có thể chấp nhận các từ viết tắt
  + Các kiểu tham số với chỉ 1 chữ cái:
    - T cho kiểu dữ liệu bất kỳ.
    - E cho kiểu thành phần của collection
    - K và V cho kiểu key và value trong Map
    - X cho Exception
    - R cho kiểu return
* Với quy ước đặt tên theo ngữ pháp:
  + Tên các class thông thường là danh từ số ít, các class tiện ích không có instance thì là danh từ số nhiều
  + Interface cũng có quy tắc giống như class hoặc là đặt tên như một tính từ ví dụ: Runable, Accessiable…
  + Các hàm thông thường sẽ là các động từ ,
    - các hàm trả về kiểu boolean nên được bắt đầu bằng *is, has.*
    - Các hàm dùng để biến đổi kiểu dữ liệu nên bắt đầu bằng *to: toType, toString..*
    - Các hàm trả về 1 view nên được bắt đầu bằng *as: asList*

1. **EXCEPTION**
2. ***Sử dụng exception chỉ cho các điều kiện ngoại lệ***

* Trong một số trường hơp, ta không nên sử dụng exception-base vì nó không chỉ làm giảm hiệu năng mà còn chưa đảm bảo sẽ hoạt động
* Exception chỉ nên sử dụng cho các trường hợp ngoại lệ không nên sử dụng trong các flow bình thường
* Một API được thiết kế tốt sẽ không ép người dùng của nó sử dụng ngoại lệ cho các flow thông thường

1. ***Sử dụng checked exception cho các điều kiện có thể khôi phục được và runtime exception cho các lỗi chương trình***

* Java cung cấp 3 loại throwable: checked exception, runtime exception và error sử dụng trong các trường hợp khác nhau:
* Sử dụng checked exception trong các điều kiện người gọi có thể mong đợi phục hồi được dữ liệu một cách hợp lý. Điều này yêu cầu người dùng cần phải xử lý các ngoại lệ này trong *catch* hoặc ném nó ra ngoài.
* Có 2 loại unchecked throwable: runtime exception và error. Cả 2 loại này đều không cần và cũng không nên được bắt. Bời vì khi xảy ra thì ta không thể khôi phục được và nếu tiếp tục thực thi có thể gây ra hậu quả xấu.
* Sử dụng runtime exception trong để chỉ thị các lỗi lập trình đặc biệt là precondition violation (nó đơn giản là các lỗi gay ra bởi client do vi phạm các quy tắc của API)
* Tất cả các unchecked throwable mà ta sử dụng đều nên là subclass của *RuntimeException.* Ta không nên tự định nghĩa các subclass của *Error* mà với trường hợp exceptin của *AssertionError*  ta cũng không nên ném nó ra

1. ***Tránh sử dụng checked exception không cần thiết***

* Checked exception bắt buộc người lập trình xử lý trong mệnh đề catch hoặc ném nó ra để hàm triệu hồi nó xử lý
* Một điểm lưu ý là khi một hàm có unchecked exception và nó ném ra một ngoại lệ thì các hàm gọi nó cần phải được đặt trong try-catch và không thể dùng được trong stream
* Cách dễ dạng nhất để hạn chế sử dụng checked exception là trả về một optional của kiểu kết quả thích hợp thay vì ném ra một checked exception
* Cách thứ 2 là chuyển checked exeption thành unchecked exception
* Tóm lại nếu caller không thể khôi phục trong trường hợp lỗi thì nên ném ra một unchecked exception, còn nếu có thể khôi phục được thì nên cân nhắc trả về optional trước

1. ***Nên sử dụng các exception tiêu chuẩn***

* Việc sử dụng các exception tiêu chuẩn đem lại nhiều lợi ích:
  + Làm cho API dễ dàng đọc và sử dụng bì nó dựa trên các quy ước đã quen thuộc với các lập trình viên
  + Dễ dàng để đọc
  + Tốn ít bộ nhớ và ít thời gian để load hơn

1. ***Ném ra ngoại lệ thích hợp cho sự trìu tượng***

* Các lớp cao hơn nên bắt các exception ở lớp thấp hơn và tại đó có thể ném ra một ngoại lệ tiếp để giải thích cho ngoại lệ này nó được gọi là exception translation
* Tuy nhiên ta không nên lạm dụng cách này mà thay vào đó cố gắng đảm bảo hàm lower-layer thành công
* Nếu không thể tránh các ngoại lệ từ các lớp thấp hơn thì cách tốt nhất là các lớp cao hơn nên âm thầm xử lý chúng

1. ***Document tất cả các exeception ném ra bởi các method.***
2. ***Bao gồm các thông tin để nắm bắt lỗi trong các thông điệp chi tiết***

* Thông thường chúng ta cần đến các thông tin chi tiết về exception để trace được nguyên nhân
* Để chụp một lỗi, các thông điệp chi tiết của mỗi exception cần bao gồm giá trị của tất cả các tham số và các trường đóng góp trong ngọi lệ đó
* Không nên hiển thị các thông tin nhạy cảm : password, các key giải mã ..

1. ***Phấn đấu cho failure atomic***

* Nếu một hàm được triệu hồi lỗi thì nó nên để các object có trạng thái như trước khi được triệu hồi:
  + Các đơn giản nhất là xây dựng các imputable object. Trong trường hợp imutable object ,cách phổ biến nhất để đạt được là kiểm tra hiệu lực của các tham số trước khi thực hiện, nó sẽ gây ra exception trước khi các tham số được chỉnh sửa
  + Cách thứ hai là sắp xếp các bước sao cho có thể thực hiện các bước khác trước các bước có thực hiện chỉnh sửa object
  + Cách thứ 3 là trước khi thực hiện chỉnh sửa thì copy nội dung ra một object tạm thời và thực hiện chỉnh sửa object tạm thời sau khi thực hiện xong
  + Cách cuối cùng và ít được áp dụng nhất là thực hiện một đoạn code để khôi phục trạng thái của object

1. ***Đừng bỏ qua exception***

* Các exception dẽ dàng bị bỏ quả nếu sử dụng try-catch, và một khối catch trống sẽ phá vỡ mụch đích của các ngoại lệ

1. **CONCURRENCY**
2. **Đồng bộ việc truy cập với các dữ liệu được chia sẻ có thể thay đổi được**

* Sử dụng *synchronize* đảm bảo chỉ duy nhất một thread có thể thực thi một hàm hoặc một khối tại một thời điểm.Thực tế việc đồng bộ không chỉ đảm bảo tất cả các thread có thể quan sát được trạng thái nhất quán của đối tượng mà còn đảm bảo bất cứ thread nào truy cập vào khối đồng bộ sẽ nhìn thấy được hiệu ứng của tất cả sự thay đôi trước đó được đảm bảo trong cùng một khóa
* Sự đồng bộ yêu cầu sự giao tiếp một cách tin cậy giữa các thread hoặc sự loại trừ lần nhau vì thế lời khuyên phân chia sự đồng bộ khi đọc hoặc viết để nâng cao hiệu năng là sai lầm
* Memory model in JAVA
* Để dừng 1 thread ta nên có một cờ boolean mà có thể thay đổi giá trị thành true bởi thread thứ 2 ngụ ý rằng thread thứ nhât đã dừng, vì kiểu boolean mà atomic trong việc đọc và ghi nên ta có thể bỏ qua việc đồng bộ khi truy cập vào biến này
* Việc đồng bộ chỉ đảm bảo chính xác nếu cả read và write được đồng bộ

1. **Tránh việc đồng bộ một cách quá mức**

* Việc đồng bộ quá mức có thể gây ra các vấn đề như giảm hiệu năng, deadlock và các vấn đề không thể kiểm soát được
* Không bao giờ được nhường việc kiểm soát cho client trong một khối hoặc một hàm được đồng bộ hóa
* Thực hiện ít việc nhất có thể trong khối được đồng bộ hóa
* Việc đồng bộ hóa một các quá mức sẽ làm giảm hiệu năng kể cả trong các hệ thống CPU nhiều core vì nó làm giảm khả năng chạy song song (các core phải quan sát được sự nhất quán của các đối tượng)

1. ***Ưu tiên sử dụng executor, taks và stream thay cho thread***

* Thay vì sử dụng thread ta nên sử dụng executor framework với nhiều tiện ích hơn và dễ dàng kiểm soát các thread hơn

1. ***Ưu tiên sử dụng hàm tiện ích concurrency cho các hàm wait và notify***

* Các tiện ích trong gói concurrency có thể chia làm 3 loại: Executor framework, concurrency collection, synchronizers

1. ***Sử dụng lazy initialization một cách khôn ngoan***

* Lazy initialization là hoãn lại việc khởi tạo của một trường cho đến khi giá trị của nó được dùng tới
* Trong hầu hết các trường hợp, việc khởi tạo thông thường sẽ được ưu tiên hơn việc khởi tạo chậm, nếu muốn sử dụng đê tăng hiệu năng trong một trường static thì ta nên sử dụng lazy initialization holder class:
  + private static class FieldHolder {  
    static final FieldType field = computeFieldValue();  
    } p  
    rivate static FieldType getField() { return  
    FieldHolder.field; }

1. ***Đừng phụ thuộc vào thread schedule***

* Thread schedule sẽ quyết định 1 thread được chạy vào lúc nào và chạy trong bao lâu (trong trường hợp có nhiều thread cùng chạy trên 1 core) và không nên phụ thuộc vào việc lập lịch này để quyết định các kết quả giữa các thread
* Cách tốt nhất để tránh vấn đề này là đảm bảo số lượng thread đang chạy (không phải là số lượng các thread) không lớn hơn đáng kể so với số core của CPU

1. **SERIALIZATION**
2. **Sử dụng các phương pháp thay thế cho tuần tự hóa trong JAVA**

* Việc sử dụng deserialization từ các nguồn không tin được tin tưởng có thể gây ra các vấn đề về lỗi hệ thống như DOS, RCE (remote code execution). Một ví dụ về DOS là khi deserialization một chuỗi ngắn nhưng lại tốn nhiêu thời gian để deserialize
* Các tốt nhất để tránh các lỗi này là không deserialization bất cứ thứ gì
* Chúng ta có thể sử dụng cơ chế cross-platform structured-data representiation:cấu trúc dữ liệu tiêu biểu cho cơ chế này là JSON và protocol Buffers