* **Java** là ngôn ngữ lập trình cấp cao, hướng đối tượng, mạnh mẽ, an toàn, độc lập với nền tảng, hiệu suất cao, đa luồng và ngôn ngữ lập trình di động.
* **Javac** (javac.exe) là trình biên dịch của java nó sẽ biên dịch file code .java thành byte code (file .class).
* **JVM** máy ảo java (java.exe) là trình thông dịch java byte code thành mã máy khác nhau tùy thuộc vào hệ điều hành. Chỉ cần máy tính cài JVM là có thể thông dịch được java byte code - > viết code 1 chỗ và chạy được trên mọi hệ điều hành.
* **JRE:** JRE (là viết tắt của Java Runtime Environment) được sử dụng để cung cấp môi trường runtime. Nó là trình triển khai của JVM. JRE bao gồm tập hợp các thư viện và các file khác mà JVM sử dụng tại runtime. Trình triển khai của JVM cũng được công bố bởi các công ty khác ngoài Sun Micro Systems.
* **JDK** (Java development kit) bộ công cụ dành cho lập trình viên java bao gồm javac, JVM và các thư viện để lập trình java.
* **Phỏng vấn:** cái gì chưa biết trả lời em chỉ biết tới đó, cái gì em chưa biết em mong muốn được trải nghiệm trong dự dán thực tế.
* **Tính đóng gói:** tính chất này nhằm bảo vệ đối tượng, đảm bảo sự bảo mật, toàn vẹn của đối tượng. Trong java tính chất này được biểu hiện bằng các từ khóa acessmodifier Thông thường các thuộc tính của class sẽ được khai báo với từ khóa private. Khi được khai báo với từ khóa này thì chỉ trong bản thân class đó mới có thể sử dụng được thuộc tính đó. Ở bên ngoài muốn truy xuất, sử dụng thuộc tính này phải thông qua các phương thức của lớp. Các phương thức sẽ được khai báo với từ khóa là public

+ Ngoài ra trong java còn cung cấp 1 api là Reflection giúp ta có thể truy xuất được đến thuộc tính private của đối tượng ( có thể nói hoặc không)

* **Tính kế thừa:** cho phép chúng ta cải tiến, nâng cấp chương trình bằng cách kế thừa lại lớp cũ và phát triển những tính năng mới. Lớp con sẽ kế thừa lại tất cả các thành phần của lớp cha mà không cần định nghĩa lại từ đó giúp ta phát triển chương trình 1 cách nhanh chóng và hiệu quả
* **Tính đa hình:** tính đa hình luôn tồn tại song song cùng tính kế thừa. Xảy ra khi có nhiều lớp con kế thừa từ 1 lớp cha nhưng mà có các tính chất , hành động khác nhau (lấy ví dụ)

+về tính chất đa hình này e xin được lấy ví dụ để giải thích cho rõ ràng như sau:

Giải sử e có 1 class là con vật là class astract trong đó có 1 phương thức astract là di chuyển .e có 2 lớp con chó và con chim kế thừa từ lớp con vật này thì nó phải overide lại phương thức di chuyển này. Mỗi con vật sẽ có cách thức di chuyển khác nhau con cho thì chạy, con chim thì bay. Sau đó e tạo 1 mảng các con vật e push các đối tượng con chó và con chim vào . e duyệt mảng và gọi phương thức di chuyển tương ứng với con vật nào thì phương thức đó sẽ được gọi. đây được gọi là đa hình động nó xảy ra trong lúc runtime.

Còn 1 kiểu đa hình nữa là overloading(là các phương thức trùng tên nhưng khác tham số truyền vào) nó xảy ra trong thời điểm complie time gọi là đa hình tĩnh

* **Tính đóng gói:** là kĩ thuật giúp bạn che giấu được những thông tin bên trong đối tượng. Mục đích là để hạn chế được các lỗi khi phát triển chương trình.
* **Tính trìu tượng:** Tính chất này giúp chúng ta trọng tâm hơn vào những tính năng, vấn đề cốt lõi của đối tượng thay vì phải quan tâm tới cách mà nó được thực hiện. Để thực hiện tính trừu tượng trong java bạn có thể sử dụng lớp trừu tượng (abstract class) và interface
* **Lớp trìu tượng(abstract class):** là lớp được khai báo với từ khóa abstract và có 1 số đặc điểm sau:   
  + Không thể khởi tạo đối tượng từ abstract class.

**+** Lớp trìu tượng có thể chứa các phương thức trừu tượng (khai báo với từ khóa abstract). Phương thức trìu tượng là phương thức chỉ có phần khai báo, không có phần thân. Nếu 1 lớp kế thừa lớp trừu tượng thì phải override lại tất cả các phương thức trìu tượng.

**+** Lớp trừu tượng thường được dùng để làm lớp cha cho các lớp có cùng bản chất. Việc sử dụng lớp trừu tượng và phương thức trừu tượng sẽ giúp bạn ẩn đi được cách cài đặt của phương thức. Tuy nhiên, bạn vẫn có thể khai báo các thuộc tính, các phương thức có phần thân nên việc sử dụng lớp trừu tượng sẽ không được coi là trừu tượng hoàn toàn.

* **interface** được dùng để lưu trữ các phương thức trừu tượng và các biến hằng số. Một số đặc điểm của interface:

**+** Giống với lớp trừu tượng bạn không thể tạo đối tượng của interface mà chỉ tạo đối tượng của lớp được kế thừa từ interface

**+** Tất cả các phương thức trong interface đều được trình biên dịch hiểu là các phương thức trìu tượng và tất cả các biến trong interface đều được hiểu là các hằng số.

**+** 1 lớp implement 1 interface thì phải override lại tất cả các phương thức trong interface đó

**+** Một lớp có thể implement nhiều interface

* **Constructor:** nếu 1 lớp chưa xây dựng 1 hàm tạo thì java sẽ tự thêm vào 1 hàm tạo mặc định không có gì. Nếu mà 1 lớp đã có 1 hàm tạo có đối thì khi khởi tạo đối tượng dùng hàm tạo không đối thì sẽ báo lỗi.
* **Biến static:** là 1 biến được cấp phát 1 vùng nhớ cố định, nó tồn tại suốt thời gian chương trình, và được cấp phát 1 lần duy nhất khi class được load, nó là dùng chung cho tất cả đối tượng không thuộc 1 đối cụ thể nào. Ví dụ khi ta xây dựng 1 class là hóa đơn bán hàng ta cần lưu tổng số hóa đơn -> ta sẽ khai báo 1 thuộc tính tĩnh là tổng số hóa đơn và trong hàm tạo của lớp ta sẽ tăng số hóa đơn lên như vậy ta sẽ lấy được tổng số hóa đơn.
* **Phương thức static:** là phương thức của lớp chứ không phải của riêng đối tượng nào, phương thức được gọi mà không cần tạo ra instance của lớp, phương thức tĩnh chỉ có thể truy xuất đến các thuộc tính tĩnh mà không thể truy xuất đến các thuộc tính riêng của đối tượng
* **Thuộc tính tĩnh, Phương thức static:**  các lớp con vẫn được thừa kế thuộc tính tĩnh và phương thức tĩnh của lớp cha, **không thể override phương thức tĩnh**
* Khi biến của lớp cha tham chiếu tới đối tượng của lớp con thì biến này **chỉ có thể** gọi tới các thuộc tính và phương thức **có ở lớp cha** và nếu lớp con **ghi đè** phương thức của lớp cha thì phương thức của lớp con sẽ được gọi. ( khác với C++ là phải thêm từ khóa virtual vào khai báo của phương thức lớp cha thì mới gọi đc phương thức overide ở lớp con)
* **Static initialization blocks:** là khối lệnh tĩnh, nó có chức năng dùng để khởi tạo giá trị cho các thuộc tính tĩnh, giống như biến tĩnh nó được gọi ngay sau khi class được load -> chương trình đôi khi không phải là bắt đầu từ phương thức main mà có thể bắt đầu từ các khai báo thuộc tính tĩnh ( khi khai báo và khởi tạo) or từ các **Static initialization blocks** trong class chứa phương thức main.
* **Package:** Lợi ích của việc sử dụng package trong java giúp phân loại lớp và interface giúp dễ dàng bảo trì, package cung cấp bảo vệ truy cập, package khắc phục được việc đặt trùng tên
* **Khác nhau giữa CHECKED VÀ UNCHECKED EXCEPTION:** Điểm khác biệt giữa các lớp checked và unchecked expcetion chính là thời điểm xác định được expcetion có thể xảy ra.

+ **Đối với checked exception**, việc xác định có thể có exception xảy ra hay không được thực hiện ngay thời điểm compile time, một số IDE sẽ giúp chúng ta bằng cách hiển thị lỗi cú pháp nếu ta gọi một method throw ra bất kỳ checked exception nào mà không được catch. Ví dụ khi ta thao tác với file thì trình biên dịch có thể biết ngay có thể tồn tại trường hợp file không tồn tại vì vậy ide sẽ báo cho ta phải bắt lỗi.

+ **Còn đối với unchecked exception**, việc xác định có thể có exception xảy ra hay không chỉ có thể thực hiện ở thời điểm runtime, và các IDE sẽ không giúp chúng ta xác định được chuyện đó

+ **throws** dùng để khai báo những ngoại lệ được ném ra trong hàm. Và được khai báo sau tên hàm. Vd: int openFile(…) throws FileNotFoundExcpetion. Những hàm mà gọi hàm này thì phải try catch bắt những excpetion này hoặc phải tiếp tục dùng **throws** để khai báo ngoại lệ ném ra trong hàm

+ **throw** dùng trong hàm. Nó dùng để ném ra ngoại lệ trong trường hợp đặc biệt nào đó trong hàm

* **BigDecimal** là kiểu dữ liệu để chứa 1 số thực rất lớn mà kiểu nguyên thủy không chứa nổi, khởi tạo đối tượng **BigDecimal** bằng 1 số thực ở dạng chuỗi, ta có thể thao tác cộng trừ nhân chia so sánh bằng các phương thức của lớp chứ không thể thao tác bằng các toán tử như kiểu dữ liệu nguyên thủy được.
* **ArrayList và LinkedList:** về cách sử dụng thì tương đối giống nhau, nhưng về performance thì lại khác nhau. Performance của **linkedlist** trong **add** và **remove** tốt hơn so với **arraylist** nhưng lại kém hơn khi sử dụng các method **get** và **set**. Vì Arraylist bản chất là 1 mảng động nên nó có thể truy cập trực tiếp đến vùng nhớ chứa phần tử bằng phương thức get và set, còn linkedlist vì là danh sách liên kiết nên phải truy xuất một cách lần lượt. Add và remove linkedlist nhanh hơn vì nó là danh sách liên kết kép nên đã biết trước con trỏ để trỏ đến phần tử đầu tiên và cuối cùng của danh sách.
* **Vector:** không khác vs Arraylist là mấy. chỉ khác là nó có thể sử dụng trong lập trình đa luồng.
* **Set:** chỉ chứa các phần tử duy nhất

**+ HashSet** không duy trì **thứ tự nào**, trong khi **TreeSet** duy trì **thứ tự tăng dần**.

* **Map:** lưu trữ dữ liệu theo cặp key và value.

**+** HashMap duy trì **không có thứ tự**, trong khi TreeMap duy trì **thứ tự tăng dần**.

* **Sự khác nhau giữa HashMap và Hashtable:**

**+ Hashmap chỉ chứa 1 key null và nhiều value null**

**+ Hashtable không thể chứa bất kì khóa null hay giá trị null**

**+ Hashtable là đồng bộ dùng trong lập trình đa luồng**

**+ hashmap là ko đồng bộ**

* **So sánh Array và Array List**: 1 cái là mảng cứng, 1 cái là mảng động, 1 cái chứa được cả dữ liệu nguyên thủy và object, 1 cái chỉ chứa được object, kiểu dữ liệu nguyên thủy được chuyển đổi 1 cách tự động được gọi là auto-boxing trong arraylist. Array thì tốc độ thao tác và lưu trữ nhanh hơn
* **Arrays.sort(s, 0, n, CustomComparator)** là phương thức tĩnh của lớp Arrays dùng để sắp sếp mảng s truyền vào và điều kiện so sánh là **Comparator**. **Comparator** là 1 interface(chứa phương thức compare) để truyền vào hàm sort ta phải tạo ra 1 đối tượng implements interface đó và override lại phương thức compare theo ý muốn.

**Ví dụ 1:**

class Checker **implements** Comparator<Player> {

@Override

**public** **int** compare(Player a, Player b) {

**if**(b.score == a. score) {

**return** a.name.compareTo(b.name);

}**else** {

**return** b.score - a.score;

}

}

}

Player[] player = new Player[n];

       Checker checker = new Checker();

 Arrays.sort(player, checker);

**Ví dụ 2:** so sánh 2 số thực rất lớn từ 2 string chuyền vào(dùng cách khai báo tắt)

Comparator<String> CustomComparator = **new** Comparator <String> () {

@Override

**public** **int** compare(String s1, String s2) {

BigDecimal a = **new** BigDecimal(s1);

BigDecimal b = **new** BigDecimal(s2);

**return** b.compareTo(a);

}

};

  Arrays.sort(Strings, CustomComparator);

**Ví dụ 3:** dùng cách khai báo và khởi tạo tắt

Collections.sort(listStudents, **new** Comparator<Student>() {

            @Override

**public** **int** compare(Student o1, Student o2) {

**return** o1.getAge() > o2.getAge() ? 1 : -1;

            }

        });

* **Class Lồng nhau:** Trong trường hợp các class lồng nhau, muốn tạo instance của lớp bên trong bạn phải tạo instance của lớp bên ngoài và thông qua instance bên ngoài .new lớp bên trong. Vì sao lại như vậy vì lớp bên trong có thể truy cập tất cả các thuộc tính của lớp ngoài nên nó cần phải gắn vs 1 đối tượng nhất định.
* **Singleton pattern:** giúp giới hạn chỉ có duy nhất 1 instance của 1 class đó được tạo ra ở bất kì thời điểm nào. Trường hợp nào cần áp dụng pattern này: Những tài nguyên được chia sẻ như DB, file vật lý được sử dụng chung. Những tài nguyên này nên chỉ có 1 instance tồn tại để sử dụng và theo dõi trạng thái của nó trong suốt quá trình sử dụng.

**+ Cách triển khai:** khi tạo class tạo 1 biến static có kiểu của class. Tạo Contructor không đối có phạm vi truy cập là private (để chỉ trong class mới có thể truy cập được -> không có đối tượng nào được tạo ngoài class) tiếp tục tạo 1 phương thức tĩnh để tạo đối tượng, trong phương thức gọi contructor để tạo đối tượng.

**+ Code demo:**

**class** Singleton{

**public** **static** Singleton *instance*;

**public** String str;

**private** Singleton() {}

**public** **static** Singleton getSingleInstance () {

**if**(*instance* == **null**) {

//Đảm bảo trong 1 thời điểm chỉ có 1 thread chạy đoạn code này

**synchronized** (Singleton.**class**) {

**if**(*instance* == **null**) {

*instance* = **new** Singleton();

}

}

}

**return** *instance*;

}

}

* **Factory Design Pattern** được sử dụng để tạo ra các object, khi chúng ta có một superclass có nhiều class con kế thừa cần phải trả về obj thuộc cclass con dựa trên một input nhất định. Design Pattern này giúp chúng ta khởi tạo đối tượng từ factory class thay vì khởi tạo trên chương trình bên client.

**+ Mục đích sử dụng:** che dấu logic của việc khởi tạo bằng cách đưa toàn bộ logic của việc khởi tạo vào trong Factory, Giảm tính phụ thuộc nhằm tăng tính mở rộng.

**+ Trường hợp sử dụng:** thường được sử dụng trong các thư viện (người sử dụng đạt được mục đích tạo ra object mới mà không cần quan tâm đến cách mà nó được tạo ra)

**+ Code demo:**

**interface** Food {

**public** String getType();

}

**class** Pizza **implements** Food {

**public** String getType() {

**return** "Someone ordered a Fast Food!";

}

}

**class** Cake **implements** Food {

**public** String getType() {

**return** "Someone ordered a Dessert!";

}

}

**class** FoodFactory {

**public** Food getFood(String order) {

**if** (order.equals("cake")) {

**return** **new** Cake();

}

**if** (order.equals("pizza")) {

**return** **new** Pizza();

}

**return** **null**;

}// End of getFood method

}// End of factory class

* **Cookies** dùng để lưu trữ dữ liệu tạm thời là 1 file text lưu ở phía browsers của client
* + Cookie lưu thông tin dưới dạng key – value

+ Mỗi lần Request gửi lên sẽ gửi kèm Cookies trong header của request

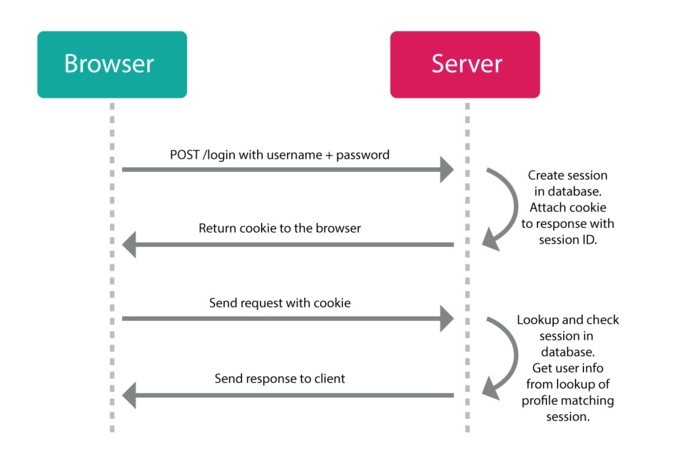
+ Java servlet hỗi trợ HTTP cookie

+ cookie có thời gian sống xác định server set thời gian sống của cookie là 1s hay 30s thì sau thời gian đó thì trình duyệt sẽ tự động xóa đi

* **Session** là 1phiên làm việc nó sẽ bắt đầu khi client gửi request đến server và tồn tại xuyên suốt từ trang này đến trang khác trong ứng dụng và chỉ kết thúc khi hết thời gian or đóng trình duyệt. Nó dùng để lưu trữ dữ liệu tạm thời, là 1 file được lưu trên server, khi Session được tạo server sẽ tạo ra SessionID gửi về trình duyệt dưới dạng cookie từ đó mỗi lần trình duyệt gửi sessionID lên server, server sẽ nhận dạng ra là client nào đang gửi đến. Session giúp duy trì trạng thái của dữ liệu người dùng

**+** Session cũng lưu thông tin dạng key/value

**+** Servlet quản lí session bằng Session Management



* Đối tượng Servlet context là đối tượng dùng chung cho tất cả các servlet, ta có thể setAttribute(key, value) tại 1 servlet và getAttribuet ở 1 servlet khác.
* **String, StringBuilder, StringBuffer:**

+ **String** là không thể thay đổi (immutable), khi thực hiện các toán tử + hay các phương thức của String thực chất là ta đang tạo ra 1 string mới chứ không hề thao tác tới String ban đầu. Có 2 cách để tạo 1 String là dùng **string literal hoặc toán tử new. Dùng string literal sẽ giúp tốc độ nhanh, và tiết kiệm bộ nhớ hơn dùng toán tử new vì** Các **string literal** được chứa trong một bể chứa (common pool), nếu ta khai báo 2 **string literal**  có nội dung giống nhau sẽ sử dụng chung 1 vùng nhớ trên stack giúp tiết kiệm bộ nhớ và nâng cao hiệu năng, nếu dùng toán tử new để tạo 1 String thì mỗi String sẽ có 1 vùng nhớ riêng biệt

* + **StringBuffer**, **StringBuilder**có thể thay đổi (*mutable*) chỉ khác nhau là nếu xử lý văn bản sử dụng nhiều luồng (Thread) bạn nên sử dụng **StringBuffer**để tránh tranh chấp giữa các luồng. Nếu xử lý văn bản sử dụng 1 luồng (Thread) nên sử dụng **StringBuilder**

**+** Nếu so sánh về tốc độ xử lý **StringBuilder**là tốt nhất, sau đó **StringBuffer**và cuối cùng mới là **String** vì **StringBuilder thể thay đổi nên khi thao tác với nó ta không cần phải tạo ra một đối tượng mới sẽ tiết kiệm bộ nhớ và tăng hiệu suất.**

**Access Modiflier:** java có 4 kiểu access modifier là: private, default, protected, public.

**Private:** thì chỉ trong class đó mới được phép truy cập, tuy nhiên nếu dùng reflection thì vẫn có thể truy cập được Field[] declaredFiels = oClass.getDeclaredFields(), field.setAccessible(true), field.set(o, ten)

**Default:** trong class or cùng package mới được phép truy cập

**Protected:** trong class or cùng package or được truy cập bởi class con ngoài package

**Public:** truy cập rộng nhất

* **Class Object:** là class cha của mọi class, mọi class đều kế thừa từ nó.
* **Java Reflection:** là 1 api của java cho phép truy cập thông tin của đối tượng như tên class, các field, các method và có thể chỉnh sửa các field của đối tượng kể cả private trogn quá trình runtime

+ ta có thể áp dụng java reflection trong những trường hợp không biết object được xử lí là gì (tên class gì, ở package nào, có những field nào, có những method nào…)

+ ví dụ: ta cần viết 1 hàm copy 2 đối tượng có thể dùng cho mọi đối tượng khác nhau.

Thì hàm đó phải đối số là 1 kiểu object trong java. Để viết được hàm này ta cần phải biết 2 đối tượng có cùng kiểu hay không, có những field nào để copy dữ liệu của từng field cho nhau.

+ Ngoài ra đối với những field, method có modifier là private thì ta không thể truy cập bên ngoài class đó. Trong những trường hợp bắt buộc phải gọi, truy cập các field, method ở bên ngoài class ta thì Java Reflection là 1 giải pháp.

* **Hạn chế của Reflection:**

**+** hiệu năng thấp vì nó phải quét class path để tìm class.

**+** các vấn đề bảo mật việc chính sửa class/object trong quá trình runtime có thể ảnh hưởng đến các thread … khiến ứng dụng bị fail

**+** khó bảo trì: vì nó khá khó cho người mới và không dễ để debug, nên sẽ khó tìm ra lỗi. Ngoài ra chúng ta không thể check được 1 số lỗi trong quá trình compile (không tìm thấy class, không tìm thấy field)

* **Anotation:** Nó được hiểu là 1 dạng chú thích hoặc là siêu dữ liệu của class, thuộc tính, method trong java. Chúng ta có thể sử dụng anotation ở thời điểm biên dịch để chỉ dẫn cho trình biên dịch. Ví dụ như anotation @overrire dùng để chỉ cho trình biên dịch biết đây là 1 method được override từ 1 method của lớp cha. Bạn cũng có thể truy cập vào anotation của 1 class, 1 field, 1 method tại thời điểm runtime bằng phương thức getAnnotations() .
* **Anotation:** có thể hiều là những thẻ @xxx được thêm vào source code để thực hiện một số chức năng gì đó. Ví dụ như anotation @overrire được java cung cấp sẵn, được thêm vào trên định nghĩa method dùng để chỉ cho trình biên dịch biết đây là 1 method được override từ 1 method của lớp cha. Và thông thường anotation dùng Reflection API để xử lí
* **Quy trình thực tế khi tạo anotation là:** tạo anotation, thêm nó vào source code, và viết hàm xử lí cho anotation đó (sử dụng reflection)
* **Tạo Anotation:**

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

public @interface ChuHoa {

boolean VietHoa();

}

* **Đa luồng** **trong java:** đầu tiên ta phải biết luồng là gì đã luồng thực chất là 1 sub process ,1 chương trình như excel là 1 process, trong 1 chương trình java ngoài luồng chính nó còn cho phép chạy các luồng con 1 cách đồng thời giúp nâng cao hiệu suất và hiệu quả chương trình
* **Ưu điểm đa luồng:** thực hiện được nhiều công việc 1 lúc giúp tiết kiệm thời gian, mỗi luồng có thể dùng chung hoặc chia sẻ tài nguyên trong quá trình chạy nhưng có thể thực hiện 1 cách độc lập, luồng là độc lập nên nếu 1 luồng xảy ra ngoại lệ thì không ảnh hưởng đến luồng khác.
* **Nhược điểm của đa luồng:** càng nhiều luồng càng phức tạp, xử lí vấn đề tranh chấp bộ nhớ, đồng bộ dữ liệu khá phức tạp, cần phát hiện để tránh các luồng chết (dead lock), luồng chạy mà không làm gì trong ứng dụng.
* **Trạng thái của 1 luồng:**
  + **New**: đây là trạng thái mà luồng vừa được khởi tạo từ phương thức khởi tạo cả lớp thread nhưng chưa được start();
  + **Runable**: là trạng thái sau khi gọi phương thức start() thì luồng đã được CPU cấp phát tài nguyên và các lịch điều phối CPU cũng bắt đầu có hiệu lực. Ở đây chúng ta dùng trạng thái runable chứ không phải Running vì luồng không thực luôn chạy mà tùy thuộc vào hệ thống và sự điều phối của CPU
  + **Waiting**: thread chờ không giới hạn cho đến khi một luồng khác đánh thức nó.
  + **Timed\_wating**: thread chờ trong 1 thời gian nhất định, hoặc có 1 luồng khác đánh thức nó
  + **Blocked**: đây là 1 dạng của trạng thái “Not Runable” là trạng thái khi thread vẫn còn sống, nhưng hiện tại không được chọn để chạy.
  + **Terminated**: một thread ở trạng thái terminated hoặc dead khi phương thức run của nó bị thoát.
* Cách tạo luồng trong java: có 2 cách tạo luồng trong java là extends từ class thread trong hoặc implements interface Runable trong java. Cách implement interface Runable thường được sử dụng vì nó không bắt tạo 1 lớp kế thừa từ lớp thread.
* Constructor không được kế thừa
* Không thể override phương thức tĩnh vì nó là 1 phần của lớp và bị ràng buộc với lớp
* **Servlet:** là 1 api dùng tạo ra ứng dụng web trong java, servlet chạy trên các web server để tiếp nhận request, xử lí và phản hồi lại cho client
* **Vòng đời Servlet:**

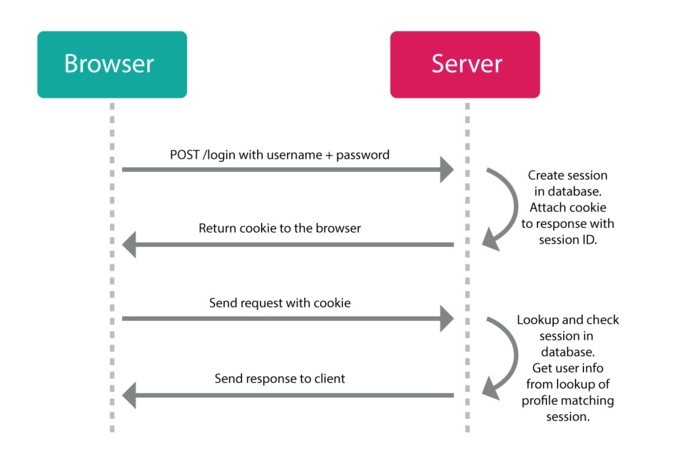
**+ init():** khi chúng ta tạo 1 servlet mới thì phương thức init() được gọi để khởi tạo các giá trị cho servlet đó. Servlet được tạo khi có request đầu tiên đến(truy cập vào URL tương ứng với servlet). Hoặc bạn có thể cấu hình servlet được tải khi máy chủ được khởi động. Servlet được khởi tạo 1 lần duy nhất

**+ Service():** Sau khi servlet được tạo thì phương thức service() được gọi để đón nhận các request từ client, xử lí các request và trả về respon cho client. Hàm Service sẽ gọi các phương thức dopost(), doget()… tùy thuộc vào yêu cầu của client

**+ Destroy():** hàm destroy được gọi khi servlet bị xóa đi trong bộ nhớ heap

* **Cookies:** là đoạn text nhỏ chứa dữ liệu được lưu phía client, để cho server biết người dùng là ai
* **Session:** là 1vùng nhớ lưu trữ dữ liệu tạm thời được lưu trên server và có thể mất đi.

**Ví dụ:** khi ta đăng nhập hay truy cập vào 1 trang web nếu là lần đầu thì server sẽ tạo 1 session để lưu thông tin người dùng và lưu trên server. Đồng thời nó sẽ trả về cho client session ID và Trình duyệt sẽ lưu sessionID vào trong cookie, lần sau mỗi lần request lên server trình duyệt sẽ gửi kèm cookie, server sẽ kiểm tra sessionID trong cookie để biết đó là ai và lấy ra thông tin phù hợp mà không cần phải đăng nhập lại hay truy cập database -> session giúp lưu trạng thái người dùng. Nếu người dùng ko tương tác nữa thì server sẽ xóa session



* MVC(Model, View, Controller): là 1 mô hình được sử dụng trong hầu hết các dự án phát triển web. Nó giúp phân bố source thành 3 phần, mỗi phần có 1 nhiệm vụ chức năng riêng biệt và độc lập với nhau từ đó giúp ta dễ dàng phát triển, quản lí, bảo trì, và nâng cấp.

+ View: chính là giao diện, giúp chúng ta hiển thị dữ liệu tương tác với người dùng

+ Controller: là nơi nhận request xử lí request và trả về respon cho người dùng

+ Model: là nơi chứa các hàm truy cập vào cơ sở để lấy dữ liệu

* **Các bước để kết nối tới DB sử dụng jdbc:**

**+ Khai báo driver class:** để java tải các driver cần thiết cho việc kết nối CSDL

Class.*forName*("com.mysql.jdbc.Driver");

**+ Get Connection:**

String url = "jdbc:mysql://localhost:3306/new\_servlet";

String user = "root";

String password = "Lamgiaukhongkho123#";

connection = DriverManager.*getConnection*(url, user, password);

**+ Tạo đối tượng** **Statement or PreparedStatement**

statement = connection.prepareStatement(sql);

**+ Execute querry trả về 1 đối tượng là ResultSet**

resultSet = statement.executeQuery();

Nếu là câu insert, update, delete, thì cần phải thêm câu lệnh connection.commit() thì dữ liệu mới đc thêm vào database. Nếu trong quá trình insert, update, delete mà gặp lỗi j ta có thể rollback về thời điểm trước khi update bằng câu lệnh connection.rollback()

**+ Đóng Connection**

connection.close()

* **Khi nào thì nên dùng Abtract class, khi nào nên dùng interface:** ta sẽ lấy 1 ví dụ giả sử chúng ta cần xây dựng các lớp con chó, mèo, chim, cò, cá, tôm. Những con vật này thì con nào cũng có tên, con nào cũng có 1 phương thức là ăn vì vậy ta sẽ tạo 1 class trìu tượng là con vật có thuộc tính là tên và 1 phương thức trìu tượng là ăn() và cho con chó, mèo, chim cò kế thừa class này. Nhưng 1 số con vật lại có khả năng bay, bơi chẳng hạn thì chúng ta không thể thêm các phương thức bay, bơi vào trong class con vật này được. Lúc này ta sẽ tạo ra interface là có thể bơi, có thể bay chứa các phương thức bơi và bay để cho các con vật có khả năng này implement lại.
* **Nói tóm lại:** theo em hiểu là **abstract class**  là 1 cái trìu tượng, chung chung nhất cho những đối tượng cụ thể vì là trìu tượng nên nó không có thật -> không thể tạo đối tượng cụ thể cho lớp trìu tượng. Còn **interface** giống như các khả năng của 1 nhóm đối tượng có thể có được
* **Các cách năng cao hiệu suất, hiệu năng chương trình java:**

+ Sử dụng thuật toán có độ phức tạp thấp.

+ sử dụng những cấu trúc dữ liệu phù hợp ví dụ muốn lưu mã sinh viên vào trong bảng, ta biết mã sinh viên có kiểu dữ liệu là 4 kí tự, ta nên sử dụng char(4) thay cho varchar(4) vì char(4) sẽ có tốc độ nhanh hơn

+ Ta có thể sử dụng lại đối tượng (object pool)

+ Ta có thể hạn chế việc đọc ghi file liên tục, hạn chế việc thao tác với database bằng cách sử dụng bộ nhớ cache

+ dùng Stringbuider cho việc thao tác với chuỗi

+ Nên sử dụng String literal để tạo String chứ ko nên dùng toán tử new

* **Các tính năng mới của java 8**:

+ Java 8 cho phép bạn thêm một method không trừu tượng vào interface

+ Biểu thức lamda giúp bạn code nhanh hơn

* **Dependency Injection** là việc các Object nên phụ thuộc vào các Abstract Class và thể hiện chi tiết của nó sẽ được **Inject** vào đối tượng lúc runtime.
* *1. Các module cấp cao không nên phụ thuộc vào các modules cấp thấp. Cả 2 nên phụ thuộc vào abstraction.*
* *2. Interface (abstraction) không nên phụ thuộc vào chi tiết, mà ngược lại. ( Các class giao tiếp với nhau thông qua interface, không phải thông qua implementation.)*
* Nói một cách đơn giản dễ hiểu về DI:
* *Ta****không gọi toán tử new****để khởi tạo instance, mà instance đó sẽ được****truyền từ ngoài vào****(Truyền manual, hoặc nhờ DI Container).*
* **Dependency Injection** là một dạng design pattern được thiết kế nhằm ngăn chặn sự phụ thuộc lẫn nhau giữa các class, các class sẽ không phụ thuộc trực tiếp lẫn nhau mà thay vào đó chúng sẽ liên kết với nhau thông qua 1 interface, việc tạo các đối tượng sẽ do các interface quản lí thay vì class phụ thuộc nó.
* **Ví du:** có 2 class A và B, class B cần dùng đối tượng của class A. thay vì phải khai báo đối tượng của A và khởi tạo trong class B thì ta chỉ việc khai báo 1 interface mà A implements trong B. Cơ chế Inversion of control sẽ giúp ta tạo tạo ra các object thích hợp
* **Class digram:** thằng không có mũi tên sử dụng thằng có mũi tên