

Summary Data Structure

A stack is a last-in first-out (LIFO) data structure while a queue is a first-in first-out (FIFO) data structure.

array

- has index
- a Obj(reference)
- accepting **primitive** and objs
- Less method/ cons: (**size cte**, increase size, remove, ...)
- فقط یک نوع در هر آرایه
- Powered by **Arrays** (order)

I Iterable (method: iterator) → **I Collection** → **I**: Set / List / Queue

Generic Data Types (not primitives): Data structure / container /
Resizable/ searchable/cast able/ algorithms and methods:
(contains, toArray, ...)/ some synchronized collection/

I Collection:

Methods (add, remove, size, **toArray** ...)

Powered by **Collections** (order, search, equals, toString, sort,
reverse)

Interface List

Accept **duplicate member**

Concept **index (orderable)**

Methods (set(i,E) ,)

Class ArrayList

Uses **array** (index)

Pros: **Access/ Navigation** (using index)

Cons:(50% bigger)/ Most **copies** for **add or remove**

Class LinkedList

Uses **double-linked list** (index)

Pros: **add/remove / (Access tail&end)**

Cons: **Access/ Navigation** costs most (using LL)

Interface Set

distinct elems (checking overhead)

can be **orderable**

Class HashSet

Use **hash tech**

Not **orderable**

Class TreeSet

Use **tree**

Orderable (ascending order)

Interface Map

A Table (**Set, Collection**)

no parent

Methods (put, get, size, keyset, values, ...)

algorithms (contains, toArray, ...)

Class HashMap

Set powered by **hash tech.**

Not orderable

allows **null** values

less overhead, best performance, O(1)

Class TreeMap

Set powered by **Red Black Tree.**

sorted order keys

overhead, O(log(n))

Class ConcurrentHashMap

subclass of **HashMap**

Fast

Scalable

atomic operations, thread-safe, lock-free

Not orderable

Set powered by **hash tech.???**

تجزیه پذیری عالی	تجزیه پذیری خوب	تجزیه پذیری ضعیف
ArrayList IntStream.range	HashSet TreeSet	LinkedList Stream.iterate

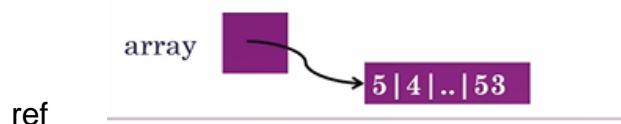
سرعت تجزیه پذیری عالی دارد. - **array**

array

- has index
- a Obj(reference)
- accepting **primitive** and objs
- Less method/ cons: (**size cte**, increase size, remove, ...)
- فقط یک نوع در هر آرایه
- Powered by **Arrays** (order, search>equals,toString)

uses index, with cte size.(accesses, primitive)

```
int[] array = new int[10];
```

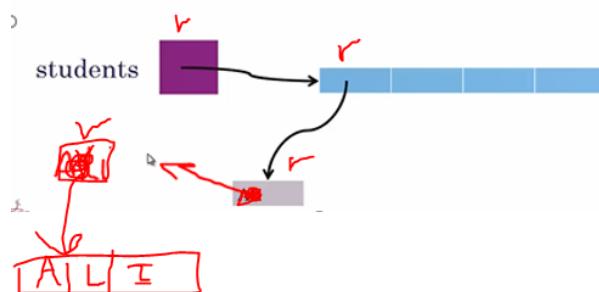


- مثلاً در یک آرایه از جنس String[]، هر مؤلفه یک ارجاع به یک رشته است
- در آرایه‌ای از جنس int[]، هر مؤلفه یک مقدار int است

بعد از تعریف آرایه، مقادیر پیشفرض (null برای obj‌ها و 0 برای int) قرار می‌گیرد.

چهارتا reference داریم:

```
Student[] students;  
students = new Student[4];  
students[0] = new Student();  
students[0].setName("Ali");
```



```
ref students[] /ref Student [0] /ref String name of Student[0] /ref array of chars(for String)
```

این خطای runtime دارد:

```
Student[] arr = new Student[10];  
arr[0].setName("Mehrad");
```

الآن ارجاع arr به آرایه ای اشاره میکند که قرار است شامل ده ارجاع به student ها باشد. هنوز همه ی خانه های آن null هستند چون new Student ای انجام نشده است. پس arr[0] null است. اجرای set() روی آن runt Err میدهد.

اینطوری درست است:

```
Student[] arr = new Student[10];  
arr[0] = new Student();  
arr[0].setName("Mehrad");
```

ولی اگر آرایه از المنشت های primitive ساخته شده بود، مقدار پیشفرض در المنشت ها قرار میگرفت:

```
int[] arr = new int[10];  
Student[] arr1 = new Student[10];  
System.out.println(arr[0]); // 0  
System.out.println(arr1[0]); // null
```

```
Int [] arayeeee = new int [7];
```

```
int [][] coordinate = new int [10][12];
```

محدودیت آرایه‌ها

- می‌دانیم آرایه، امکانی برای ایجاد ظرفی از اشیاء است

- مثال: فرض کنید آرایه‌ای از دانشجویان داریم

```
Student[] students = new Student[size];
```

- اما آرایه‌ها محدودیت‌هایی دارند. مثلاً نیازمندی‌های زیر را در نظر بگیرید:

- اگر طول موردنیاز آرایه (size) را پیش‌اپیش ندانیم، چه کنیم؟

- اگر بخواهیم بعد از ساختن یک آرایه، طول آن را افزایش دهیم چه کنیم؟

- اگر بخواهیم بعضی از عناصر و اعضای آرایه را حذف کنیم، چه کنیم؟

- راه حل ساده‌ای برای موارد فوق در آرایه‌ها وجود ندارد

- مثلاً متوجه شوی که یک خانه از آرایه را حذف کند یا طول آرایه را بیشتر کند

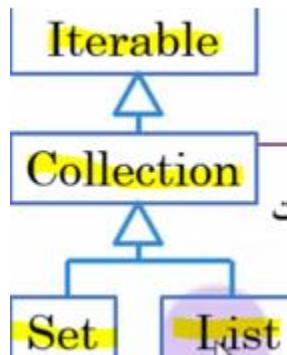
متدهای static کلاس Arrays امکاناتی به array میبخشند:

```
int[] a1 = {1,2,3,4};  
int[] a2 = {1,2,3,4};  
System.out.println(a1==a2); false  
System.out.println(a1.equals(a2)); false  
System.out.println(Arrays.equals(a1, a2)); true  
System.out.println(a1); [I@7852e922  
System.out.println(a1.toString()); [I@7852e922  
System.out.println(Arrays.toString(a1)); [1, 2, 3, 4]
```

The elements of an array can be **arranged in order** by using a static sort() method of the **Arrays** class.

چون از List و array استفاده میکنند index orderable هستند.

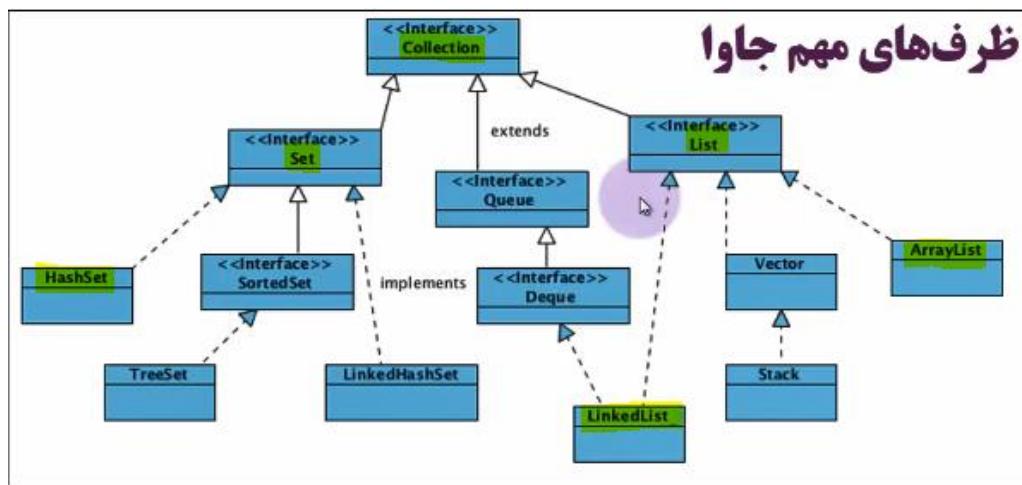
Interface Collection



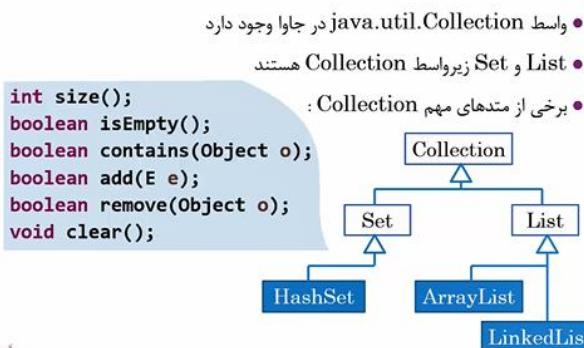
- هر یک از این کلاس‌ها، یک ساختمان داده (Data Structure) است
- هر نمونه ساختمان داده، یک ظرف (container) برای نگهداری اشیاء است
- امکانات و الگوریتم‌هایی بر روی اشیاء داخل ظرف هم پشتیبانی می‌شود
- مانند جستجو، تبدیل به انواع دیگر، مرتب‌سازی و ...

- امکاناتی که جاوا به این منظور ساخته:
 - Java collections framework
 - کتابخانه‌ای از کلاس‌ها و واسطه‌ایی که ساختمان‌های داده مختلف را ایجاد می‌کنند

Powered by **Collections** (order, search, equals, toString, sort, reverse)



Collection



مروار برخی واسطه‌ها و کلاس‌های مهم دیگر

نام	نوع	پدر	توضیح
SortedSet	واسط	Set	یک مجموعه مرتب
TreeSet	کلاس	SortedSet	یک مجموعه مرتب که براساس یک درخت پیاده شده
SortedMap	واسط	Map	یک نگاشت (جدول) که بر اساس کلیدهایش مرتب است
TreeMap	کلاس	SortedMap	نگاشت مرتبی (براساس کلید) که با درخت پیاده شده
Queue	واسط	Collection	یک صف از اشیاء (FIFO)
PriorityQueue	کلاس	Queue	یک صف اولویت‌دار (بر اساس مقایسه و ترتیب اشیاء)
Stack	کلاس	Vector	یک پشته از اشیاء (LIFO)

- برخی از ظرف‌ها synchronized هستند (Synchronized Collections)

- ظرف‌هایی که استفاده از آن‌ها در چند thread همزمان، امن است

- کلاس‌های thread-safe (مراجعه به مبحث thread-safe)

- مثلاً ConcurrentHashMap و Vector

- اگر نیازی به استفاده همزمان از اشیاء کلاس نیست، از اینها استفاده نکنید

- برخی ظرف‌ها غیرقابل تغییر هستند (Unmodifiable Collections)

- فقط می‌توانیم اعضای آن‌ها را پیش‌مایش کنیم (که وریاد کردن اعضا ممکن نیست)

- مثال:

```
List<String> unmod1 = Arrays.asList("A", "B");
List<String> mod1 = new ArrayList<>(unmod1);
Collection<String> unmod2 = Collections.unmodifiableCollection(mod1);
```

- امکان عدم ذکر نوع عام (generic) هنگام ایجاد شیء

- از جواو 7 به بعد

```
List<String> mylist1 = new ArrayList<String>();
List<String> mylist2 = new ArrayList<>();

Set<Integer> set1 = new HashSet<Integer>();
Set<Integer> set2 = new HashSet<>();

Map<String, Integer> m2 = new HashMap<String, Integer>();
Map<String, Integer> m1 = new HashMap<>();
```

Generic types Errors

در زمان استفاده و ساخت **Object** اگر نوع مشخصی را اعلام نکنیم، میتوان ها به شکل **instance** از GDT اسکریپت استفاده کرد.

```
ArrayList list = new ArrayList();
list.add("A");
list.add(new Integer(5));
list.add(new Character('#'));
```

برای کنترل محدودیت نوع ورودی **add** در زمان ساخت **instance** از DS آنرا محدود میکنیم:

- در کد زیر، به ظرف **students** فقط اشیائی از نوع **Student** میتوان اضافه کرد:

```
ArrayList<Student> students = new ArrayList<Student>();
```

- به این تکنیک، اشیاء عام (generics) گفته می‌شود (بعداً در این باره صحبت می‌کنیم)

```
students.add(new Student("Ali Alavi")); ✓
students.add("Taghi Taghavi"); ✗
students.add(new Object()); ✗
```

مثال:

نوع **obj** هایی که این **AL** میتوانست بپذیرد را محدود کردیم. ولی این محدودیت (**erasure**) فقط مربوط به زمان **compile time** است. در **run time** این ظرف در خاص‌ترین حالت که میتواند قرار میگیرد که **Object** است و میتواند **Object** بپذیرد و ممکن است خطای **runtime** بخوریم.

```
1 public class Generics<T> {
2     void add(List<T> l, Object o) {
3         l.add((T) o);
4     }
5     public static void main(String[] args) {
6         Generics<String> g = new Generics<>();
7         List<String> list = new ArrayList<>();
8         list.add("a");
9         g.add(list, new Object());
10        g.add(list, new Integer(1));
11        for (String s : list){           خروجی؟
12            System.out.println(s);      الف) بدون خطا
13        }
14    }
15 }
```

الف) بدون خطا

ب) خطای کامپایل در خط ۹ و ۱۰

ج) خطای کامپایل در خط ۱۱ یا ۱۲

د) خطأ در زمان اجرا در خط ۹

هـ) خطأ در زمان اجرا در خط ۱۱

و) خطأ در خط ۳

راه حل:

- قبل از هر Add به DS نوع آنرا با مثلا instanceof چک کنیم.
- یا ورودی سرویس را check کنیم.
- برای کنترل نوع add در run time ، نوع را در تعریف کلاسی که از آن AL را ارث میبریم محدود کنیم. مثلا فقط numeric باشد.

```
public class CustomList<T extends A> extends ArrayList<T>
```

الان هم در زمان compile و هم در زمان runtime String نمیتوان اضافه کرد به .DS

پس :

برای در compile time محدودیت نوع را در
در زمان ساخت DS آنرا محدود میکنیم

برای جلوگیری از خطای run time محدودیت نوع را
نوع را در تعریف کلاسی که از آن AL را ارث میبریم محدود کنیم.

InstanceOf

چک سرویس

Interface List

Concept **index (orderable)**

Accept **duplicate member**

Methods (set(i,E) ,)

```
public class ArrayList<E>
implements List<E>
```

درباره واسط List

- برخی متدهای مهم کلاس `ArrayList` :
 - طول فهرست `int size()`
 - فهرست خالی است یا خیر `boolean isEmpty()`
 - وجود شیء موردنظر در فهرست `boolean contains(Object o)`
 - یک عضو به فهرست اضافه می‌کند `add(E e)`
 - یک عضو از فهرست حذف می‌کند `remove(Object o)`
 - عضوی با شماره اندیس موردنظر را حذف می‌کند `remove(int index)`
 - همه اعضای فهرست را حذف می‌کند `clear()`
 - عضوی که در اندیس موردنظر است را برمی‌گرداند `E get(int index)`
 - شماره اندیس عضو موردنظر را برمی‌گرداند `int indexOf(Object o)`
- نکته: کلاس `List` واسط `java.util.List` را پیاده‌سازی کرده است
- متدهای فوق همگی در واسط `List` تعیین شده‌اند



```
interface List<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    boolean add(E e);
    boolean remove(Object o);
    void clear();
    E get(int index);
    E set(int index, E element);
    void add(int index, E element);
    E remove(int index);
    int indexOf(Object o);
    int lastIndexOf(Object o);
    List<E> subList(int fromIndex, int toIndex);
}
```

نگاهی به واسط List

Class ArrayList

Uses **array** (index)

Pros: **Access/ Navigation** (using index)

Cons:(50% bigger)/ Most **copies** for **add or remove**

برای **impl** کردن **List(index)** از **array** استفاده میکند.

کلاس **AL** متدهی به **list** اضافه کرده؟ بله:

addAll(Collection<? extends E> c):

remove(int index)

add(int index, E element)

```
ArrayList <String> myArr = new ArrayList <String>();
```

```
myArr.add("hi"); // نوشتن
```

```
myArr.get(3); // خواندن
```

• نمونه کاربرد کلاس **java.util.ArrayList**

• **مانند آرایهای** است که **امکان تغییر اندازه (طول)** آن وجود دارد
(resizable array)

```
ArrayList students = new ArrayList();
students.add(new Student("Ali Alavi"));
students.add(new Student("Taghi Taghavi"));
students.remove(0);
```

- در ابتدا، **ArrayList** خالی است، بهمروز میتوانیم عناصری به این فهرست اضافه یا کم کنیم
- شیء **students** در کد فوق، مانند ظرفی است که اشیاء مختلفی را در خود نگه میدارد
- **اشکال شیء students**: هر شیئی از هر نوعی قابل افزودن به **students** است
- اما معمولاً اعضایی از یک جنس را در یک ظرف قرار میدهیم

مثال‌هایی از ArrayList

```
ArrayList<Student> students = new ArrayList<Student>();  
students.add(new Student("Ali Alavi"));  
students.add(new Student("Taghi Taghavi"));  
students.remove(0);  
students.remove(new Student("Ali Alavi"));  
Student student = students.get(0);  
System.out.println(student);
```

```
List<String> list = new ArrayList<String>();  
Scanner scanner = new Scanner(System.in);  
while(true){  
    String input = scanner.next();  
    if(input.equalsIgnoreCase("exit"))  
        break;  
    list.add(input);  
}  
if(list.isEmpty()) {  
    System.out.println("No string entered");  
}else{  
    System.out.println(list.size());  
    if(list.contains("Ali"))  
        System.out.println("Ali Found!");  
  
    for (String s : list) {  
        System.out.println(s);  
    }  
}
```

مثال

ظرفی از اشیاء است: هر یک از مقادیر داخل آن، یک شیء است

- انواع داده اولیه (primitive types) نمی‌توانند در ArrayList قرار گیرند
- این محدودیت برای سایر انواع ظرفها (مثل Set و LinkedList و ...) هم وجود دارد
- در واقع این محدودیت برای همه انواع عام (generics)، از جمله ظرفها، وجود دارد
- این محدودیت برای آرایه وجود ندارد
- مثلاً `ArrayList<int>` غیرممکن است، ولی `[]` مجاز است

نگاهی به پیاده‌سازی کلاس ArrayList

```
public class ArrayList<E> implements List<E>, ... {
    private Object[] elementData;
    private int size;
    public boolean add(E e) {
        ensureCapacity(size + 1);
        elementData[size++] = e;
        return true;
    }
    public ArrayList(int initialCapacity) {
        ...
        this.elementData = new Object[initialCapacity];
    }
}
```

- برای حذف یک عضو ArrayList، خانه‌های بعدی در خانه قبلی کپی می‌شوند



- هنگام اضافه کردن یک عضو به ArrayList (مثلاً با کمک متده add)

- اگر آرایه‌ای که در دل ArrayList است حافظه کافی نداشته باشد (پر باشد)،

یک آرایه جدید بزرگتر ایجاد می‌شود (معمولًاً ۱.۵ بزرگتر می‌شود)

و همه اعضای آرایه قبلی در این آرایه کپی می‌شوند

- مثلاً اگر list یک ArrayList باشد که هر چهار خانه آرایه داخل آن بر باشد

- با فراخوانی list.add(new Integer(3)) خواهیم داشت:



- حذف و اضافه از ArrayList ممکن است منجر به تعداد زیادی کپی ناخواسته شود

آرایه یا ArrayList؟ مسئله این است...

- گاهی آرایه و گاهی ArrayList بهتر است

- در هر کاربرد، باید انتخاب کنیم: مزایا و معایب هر یک را بررسی کنیم

- مزایای آرایه:

- امکان استفاده از انواع داده اولیه (مثل int و double)

- آرایه می‌تواند کارایی performance (پیشرانه) داشته باشد

- مزایای ArrayList:

- ارائه متدها و امکاناتی که در آرایه نیست

مانند اضافه و کم کردن اعضا به صورت پویا، جستجو در لیست و ...

- یادآوری: کلاس ArrayList با کمک یک آرایه پیاده‌سازی شده است

- در دل هر شیء از جنس ArrayList یک آرایه قرار دارد

Class linkedList

Uses double-linked list (index)

Pros: add/remove / (Access tail&end)

Cons: Access/ Navigation costs most (using LL)

Impl List and Queue

همچنان index دارد چون List را impl کرده است.

برای impl کردن List(index) از Linked List استفاده میکند.

برای add و remove ساختار linked list بهتر است مخصوصا اگر از سر یا تهش بخوان بردارند یا اضافه کنند.

هرچقدر با عناصر میانه اش کار داشته باشد کنتر است و هر چقدر با عناصر کناره کار داشته باشد سریعتر LinkedList است.

کلاس LinkedList

- کلاس java.util.LinkedList در جاوا پیاده‌سازی شده است
- یک لیست پیوندی دوطرفه که هر عضو، ارجاع به بعدی و قبلی دارد
- کلاس LinkedList هم مانند ArrayList واسط List را پیاده‌سازی کرده است
- پس همه متدهای مهم List را دارد، مانند add, get, remove, ...
- بنابراین نحوه کلربرد مشابه LinkedList ArrayList است

```
LinkedList<Double> grades = new LinkedList<Double>();  
grades.add(new Double(18.5));  
grades.add(new Double(19.5));  
grades.add(new Double(17.5));  
for (Double d : grades)  
    System.out.println(d);
```

```
List<String> list = new LinkedList<String>();  
list.add("Ali");  
list.add("Taghi");  
System.out.println(list.get(1));  
list.remove("Taghi");  
for (String string : list) {  
    System.out.println(string);  
}
```



- در کد زیر، متغیر list می‌تواند شیئی از نوع ArrayList یا LinkedList باشد

```

for(int i=0;i<1000000;i++){
    for(int j=0;j<100;j++)
        list.add(0, new Object());
    for(int j=0;j<100;j++)
        list.remove(0);
}

```

- در کدام حالت این کد سریع‌تر اجرا می‌شود؟

```

List<Object> list = new ArrayList<Object>();
List<Object> list = new LinkedList<Object>(); ✓

```

- تعداد زیادی حذف و اضافه در ابتدای ArrayList متوجه به شیفت‌های فراوان می‌شود

؟ LinkedList یا ArrayList بهتر است

- کلاس‌های ArrayList و LinkedList واسط مشابهی را پیاده کرده‌اند (List)

اما پیاده‌سازی متفاوتی دارند: درون هر LinkedList یک آرایه نیست، یک لیست پیوندی است

- در مجموع، کلاس ArrayList پرکاربردتر است

البته در برخی موارد، استفاده از LinkedList کارتر است

- مثال: تعداد زیادی remove و add در لیست ← معمولاً لیست پیوندی بهتر است

گاهی برای افزودن یا حذف، مجبور به کپی تعداد زیادی از عناصر موجود می‌شود

- دسترسی فراوان به عناصر با کمک اندیس ← ArrayList بهتر است

هزینه اجرای get(i) در ArrayList کم است

- ولی در لیست پیوندی i عنصر باید پیمایش شوند تا به عنصری با اندیس i برسیم

- در کد زیر، فهرستی از نوع list است

و شامل تعداد زیادی شیء است

```

Random random = new Random();
Object temp;
for(int i=0;i<100000;i++)
    temp = list.get(random.nextInt(list.size()));

```

- اگر list یک LinkedList باشد کد فوق سریع‌تر اجرا می‌شود یا ArrayList
- پاسخ: •

- کد فوق، به دفعات به سراغ اندیسی تصادفی در میانه لیست می‌رود

Interface Set

distinct elems (checking overhead)
can be orderable

وقتی که نیاز به مجموعه (اجزای متمایز) داشته باشیم.

کلاس هایی که Set را impl میکنند به طریقی باید checking را انجام دهند که عضو تکراری نگیرند.

درست است که index ندارد ولی تنها راه orderable شدن tree هم میتوان order داد.

HashSet, TreeSet, and LinkedHashSet

HashMap, TreeMap and LinkedHashMap

<https://www.java67.com/2014/01/when-to-use-linkedhashset-vs-treeset-vs-hashset-java.html>

تفاوت‌های اصلی Set و List

- اشیاء داخل یک Set متمایز هستند، شیء تکراری در Set وجود ندارد
 - اگر شیئی تکراری به مجموعه اضافه شود، شیء قدیمی حذف می‌شود
- اعضای List ترتیب دارند. بین اعضای Set لزوماً ترتیبی وجود ندارد
 - واسط Set هیچ متدهی که با اندیس کار کند، ندارد
 - مثلاً در واسط Set، متدهای get(i)، نداریم، ولی در List داریم
 - متدهای دیگری مثل موارد زیر هم در Set وجود ندارد:
 - set(int index, E element)
 - int indexOf(Object o)
 - int lastIndexOf(Object o)
 - remove(int index)

```

Set<String> set = new HashSet<String>();
set.add("Ali");
set.add("Taghi");
set.add("Taghi");
set.add("Ali");
set.add("Taghi");
System.out.println(set.size()); 2
for (String str : set)
    System.out.println(str); Taghi
set.remove("Ali");
System.out.println(set.contains("Ali")); false
System.out.println(set.contains("Taghi")); true
set.clear();
System.out.println(set.size());

```

مجموعه یا لیست؟ کدام بهتر است؟

- در برخی کاربردها List و در برخی دیگر Set مناسب‌تر است
- دسترسی به اعضا از طریق اندیس را ممکن می‌کند
- اجازه افزودن عضو تکراری به مجموعه را نمی‌دهد
- تکراری بودن عضو جدید را چک می‌کند (سریار محاسباتی)
- می‌تواند از هدر رفتن حافظه جلوگیری کند (کاهش حافظه مصرفی)
- سؤال کلیدی: آیا در فهرست موردنظر، عضو تکراری مجاز است؟
 - اگر بله: List بهتر است، و گرنه Set بهتر است. مثال:
 - فهرست شماره دانشجویی اعضای یک دانشگاه: Set بهتر است
 - فهرست نمرات یک درس: List بهتر است (نمره تکراری ممکن است)

Class HashSet

Use **hash tech** for providing distinct elements (preventing repetitive elems)

Not orderable

Class Type **must override** equals and hashCode methods

شیوه و ذخیره سازی elems در HashSet چطور است؟ مفاهیم برآخت و ... ؟؟؟

ارجاع به بخش پیاده سازی hashCode() و مثال HashSet از فایل **java1**.

Class TreeSet

Use tree

Orderable (ascending order)

Class Type must impl Comparable.

Objects in TreeSet are stored in a sorted and ascending order.

TreeSet doesn't preserve the insertion order but elements are ascending order by keys.

```
class Car implements Comparable<Car> {
    String name;
    Integer price, speed;
    public Car(String name, Integer price, Integer speed) {
        this.name = name;
        this.price = price;
        this.speed = speed;
    }
    public int compareTo(Car o) {
        return this.price.compareTo(o.price);
    }
    Comparator<Car> comp = new Comparator<Car>() {
        public int compare(Car o1, Car o2) {
            return o1.speed.compareTo(o2.speed);
        }
    };
    Set<Car> cars1 = new TreeSet<>(comp);
    Collections.addAll(cars1, new Car("Pride", 20, 200),
                       new Car("Samand", 25, 180));
    Set<Car> cars2 = new TreeSet<>(cars1);
    for (Car car : cars1)
        System.out.println(car.name);
    for (Car car : cars2)
        System.out.println(car.name);
```

خروجی
برنامه
زیر
چیست؟

Samand
Pride
Pride
Samand



Interface Map

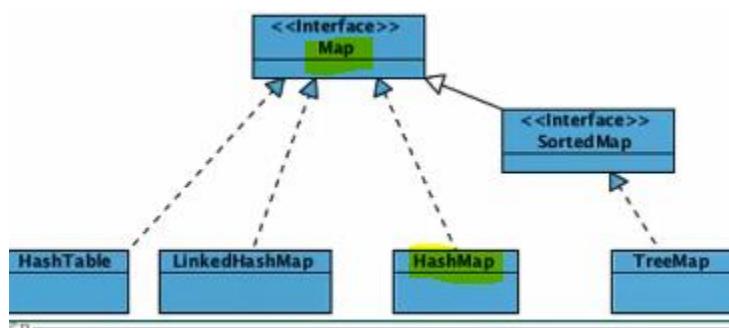
A Table (Set, Collection)

no parent

Methods (put, get, size, keyset, values, ...)

algorithms (contains, toArray, ...)

خودش Interface خودش impl دیگری را نمیکند.



هر MAP یک **MAP** است با دو ستون، ستون اول یک **Set** و ستون دوم یک **Collection** است.

البته میتوانیم آنرا بصورت یک **table** بزرگتر استفاده کنیم مثل پروژه TrackingPrice

Class هایی که Map را impl میکنند باید مشخص کنند با چه روشی میخواهند set را impl کنند.

difference between HashMap, TreeMap and LinkedHashMap comes from their performance for common operations like get(), put(), remove() and containsKey();

both **HashMap** and **LinkedHashMap** provides **O(1)** performance,

while **TreeMap** provides **O(log(n))**, so it's slightly slow compared to other two.

Perhaps most notable difference between them **comes from their orders**

Map.entrySet ???

```

Function<Car, String> Function<Car, Car>
Map<String, Car> map = list.stream()
.collect(Collectors.toMap(car -> car.color, car -> car));

```

یک راه تبدیل یک List به یک Map بود.

Collectors.toMap

نگاشت (Map)

- کلاس ها و واسطه ایی که تا اینجا دیدیم، همه Collection بودند
- Collection, List, ArrayList, LinkedList, Set, HashSet, ...
- واسط دیگری به نام java.util.Map وجود دارد که یک Collection نیست
- یک Map مانند یک جدول یا نگاشت از اشیاء عمل می کند
- همانند جدولی که دو ستون دارد (هر سطر یک زوج مرتب)
- ستون اول را کلید (Key) و ستون دوم را مقدار (Value) می گویند
- اعضای ستون اول (کلیدها) یکتا هستند: کلید تکراری نداریم
- اعضای ستون دوم (مقادیر) ممکن است تکراری باشند

مقادر	کلید	مقدار
۱۸.۵	علی علوی	
۱۹.۵	نقی تقی	
۱۸.۵	نقی نقوی	

- نوع ستون اول و ستون دوم قابل تعیین است
- مثالاً در `Map<String, Double> map;` یک جدول است که:
- کلید آن (ستون اول) رشته‌ها و مقادیر آن (ستون دوم) اعداد حقیقی هستند
(نگاشتی از رشته به عدد حقیقی)
- نگاشتی از عدد صحیح به دانشجو `Map<Integer, Student>`
- هر نوع شیئی به عنوان کلید یا مقدار، قابل استفاده است
- انواع داده اولیه مثل `int` و `double` در هیچ‌یک از ظرفهای جاوا قابل استفاده نیستند

یک واسط است،
یکی از کلاس‌هایی که `Map` را پیاده‌سازی کرده:

```
public interface Map<K,V> {
    V get(Object key);
    V put(K key, V value);
    int size();
    boolean isEmpty();
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    V remove(Object key);
    void putAll(Map m);
    void clear();
    Set<K> keySet();
    Collection<V> values();
}
```

نگاهی به واسط

Class HashMap

Set powered by **hash tech.**

Not orderable

allows **null** values

less overhead, best performance, O(1)

common operations like `get()`, `put()`, `remove()` and `containsKey()`; provides **O(1)** performance

HashMap gives **best performance** because there is **less overhead** than others based upon hash data structure

Set managed by **hash tech.**

Class Type **must override** equals and hashCode methods

HashMap is **not synchronized** and allows **null** values

while Hashtable is synchronized and does not allow null values.

ذخیره سازی در :HashMap

?bucket در کدام hashCode ()

?replace ذخیره یا equals ()

بنابراین برای دو obj برابر:

اگر فقط hashCode () شود: دو obj در یک bucket پیکسان ذخیره میشوند.

اگر فقط equals () شود: دو obj در bucket های متفاوت میتوانند ذخیره شوند.

```
Map<Integer, String> map = new HashMap<Integer, String>();  
map.put(87300876, "Ali Alavi");  
map.put(87234431, "Taghi Taghavi");  
map.put(87300876, "Naghi Naghavi");  
String name = map.get(87300876); Naghi Naghavi  
System.out.println(name);  
System.out.println(map.get(87234431)); Taghi Taghavi
```

• یادآوری:

• در کد فوق به جای int از Integer استفاده شده است

• تبدیل int به صورت خودکار انجام میشود (auto-boxing)

(از جاوا 5 به بعد)

• نوع مورد استفاده در همه کلاس های java collections framework باید شیء باشد

```

Map<Student, Double> map = new HashMap<Student, Double>();
map.put(new Student("Ali Alavi"), new Double(18.76));
map.put(new Student("Taghi Taghavi"), new Double(15.43));
map.put(new Student("Naghi Naghavi"), new Double(17.26));
map.put(new Student("Naghi Naghavi"), new Double(15.26));
map.remove(new Student("Naghi Naghavi"));

Double grade = map.get(new Student("Taghi Taghavi"));
System.out.println("Grade of Taghi=" + grade);

for (Student student : map.keySet())
    System.out.println(student.toString());

Double totalSum = 0.0;
for (Double avg : map.values())
    totalSum += avg;

System.out.println("Average = " + (totalSum / map.size()));

```

این برنامه به شرطی درست
کار می‌کند که متدهای
hashCode و equals
خوبی در کلاس Student
پیاده‌سازی شده باشند

```

Map<String, String> map = new HashMap<String, String>();
map.put("Laptop", "Computers");
map.put("Shahnameh", "Books");
map.put("Tablet", "Books");
map.put("Tablet", "Computers");
System.out.println(map.size());
System.out.println(map.get("Tablet"));
System.out.println(map.get("GOLESTAN"));
System.out.println(map.containsKey("TABLET"));
System.out.println(map.containsValue("Books"));

```

3
Computers
null
false
true

Class TreeMap

Set powered by **Red Black Tree.**

sorted order keys

overhead, O(log(n))

keeps all elements in **sorted** order (natural order of keys: Comparator, Comparable)
for common operations like get(), put(), remove() and containsKey(); TreeMap provides
O(log(n))

the **cost** you need to pay to keep keys in their sorting **order**, every time you add or
remove mapping , it need to **sort the whole map**

LinkedHashMap

Set powered by **Red Black Tree.**

sorted order keys

overhead, O(log(n))

keeps all elements in **sorted** order (insertion order, or access order)

common operations like get(), put(), remove() and containsKey(); both HashMap and
LinkedHashMap provides **O(1)** performance

LinkedHashMap gives **performance** in **between** HashMap and TreeMap.

based upon doubly linked list

LinkedHashMap also provides a great starting point for creating a **Cache** object by
overriding the removeEldestEntry() method. This lets you create a Cache object that
can expire data using some criteria that you define. For example, you can use this
method to create a LRU Cache in Java.

Class ConcurrentHashMap

subclass of HashMap (overriding its methods, doesn't let null)

Fast

Scalable

atomic operations, thread-safe, lock-free

Not orderable

Set powered by **hash tech.???**

برای هر bucket یک lock در نظر میگیرد

بنابرین چند thread write میتوانند همزمان کار کنند.

در زمان پیمایش میشه write کرد!

مناسب برای وقتی که تعداد عملیات write بالاست.

How:

using a special locking mechanism: is called a **segmented lock** (using **ReentrantLock Class**). It works by dividing the data into small segments, and then allowing each thread to lock one segment (lock and unlock individual buckets) at a time. This way, multiple threads can work (writing) on the data concurrently, but they will never be working on the same data at the same time.

In the absence of a lock, the thread will wait.

good for:

where **multiple** threads are **reading** and **writing** to the hash

more efficient to use: perform **frequent reads** and **infrequent writes**

Do not use it:

- if you need **sequential access to one elements**

A concurrent map does not guarantee that an item will be available immediately after another item has been modified

- synchronize access to the **entire map** (which would be inefficient)
(locking each segment individually, I think it is ok! It's better than we have only one writer!)
- there is **no guaranteed ordering** after modifications, if you would need to maintain insertion order of the elements

better performance: we can **don't lock** a bucket for **reading**

without locking the entire Map, only the required element is locked for writing (This allows for better performance because more than one thread can read data at once.)

provides a number of **features**:

Accessing the map from multiple threads efficiently

large amount of data/ **fast lookup and insertion times/ thread-safe**

the concurrency **level** (default value of 16), the number of threads can access its buckets at once

All **operations** on the ConcurrentHashMap are **atomic**, which means you can safely perform concurrent data access.

The ConcurrentHashMap is designed to be **lock-free**, which means that there is no need to acquire a lock in order to read or write data

Scalable, provides several **methods** for doing more advanced operations, such as:

iterating over the key-value pairs in the map, retrieving a set of keys that are associated with a given value.

Class LinkedHashMap

O(1)

Null???

Class Hashtable

synchronized

does not allow null values.

Method Iterator

Interface **Iterable** (method: iterator) → I Collections

عملگر For each با کمک iterator پیاده سازی شده است

مفهوم پیماشگر (Iterator)

- تا قبل از جاوا ۵ ، امکان for each برای پیماش وجود نداشت

- از جاوا ۵ به بعد، برای آرایهها و collection ها ممکن شد

```
int[] array = {1,2,3,7}; List<Integer> list ;  
for (int i : array) ...  
System.out.println(i); for (Integer i : list)  
System.out.println(i);
```

مثال:

- قبل از جاوا ۵ با کمک iterator پیماش روی collection ها انجام می شد

- این امکان همچنان وجود دارد و کاربردهایی نیز دارد

```
List<Integer> list = ...;  
Iterator<Integer> iterator = list.iterator();
```

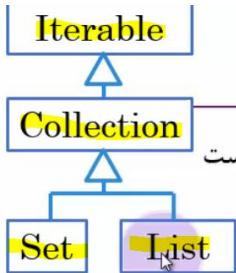
```
while(iterator.hasNext()){  
    Integer i = iterator.next();  
    System.out.println(i);  
}
```

مثال:

چیکار میکنه که for each نمیتونه؟؟؟

چرا هنوز Iterator کاربرد دارد؟

متدهای iterator



• متدهای **iterator** در واسطه **java.lang.Iterable** تعریف شده است

• واسطه **Collection**، از واسطه **Iterable** ارثبری کرده است

• بنابراین **List** ها و **Set** ها همگی **Iterable** هستند

• در واقع همه کلاس هایی که **Iterable** هستند، امکان **for each** دارند

• امکان **iterator** در نسخه های جدید جاوا با کمک **Iterator** پیاده شده است

```
public interface Iterable<T> {  
    Iterator<T> iterator();  
    ...  
}  
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();  
}
```

```
public interface Collection<E> extends Iterable<E> {...}
```

Fail Fast

برای جلوگیری از Concurrent modification برای Iterable ها

وقتی یک Iterable در حال Concurrent modification شدن حتی با **forEach** است، اگر Iterable تغییر یابد، به محض next بعدی همه iterator های گرفته شده از آن به سرعت غیر معتبر میشوند و اکسپشن Concurrent modification Exception پرتاب میشود.

مثال دیگری برای ConcurrentModificationException

```
List<String> list = new ArrayList<String>();
```

```
list.add("A");  
list.add("B");  
list.add("C");
```

```
for (String s : list)  remove  
    if(s.equals("A"))  
        list.remove(s);  ای که با کمک آن حلقه iterator در حال اجراست نامعتبر میشود
```

متدهای بنویسید که لیستی از رشته‌ها به عنوان پارامتر بگیرد

و همه رشته‌هایی که با Ali شروع می‌شوند را از لیست حذف کند

```
void removeAli(List<String> names){...}
```

یک راه حل صحیح دیگر

```
public static void removeAli(List<String> list){  
    for (int i = list.size()-1; i >= 0; i--)  
        if(list.get(i).startsWith("Ali"))  
            list.remove(i);  
}
```

اگر پیمایش For از ابتدا به انتهای لیست می‌بود درست کار نمی‌کرد.

تغییر همزمان (Concurrent Modification)

- فرض کنید: جند بخش برنامه به صورت همزمان در حال استفاده از یک ظرف باشند (مثلاً یک لیست یا مجموعه)
- و در همین حال، یک بخش از برنامه، تغییری در ظرف ایجاد کند
- مثلاً شیئی به آن اضافه یا کم کند
- این تغییر همزمان نباید ممکن باشد
- زیرا یک ظرف توسط یک بخش درحال پیمایش است و در بخش دیگری تغییر می‌کند
- مثلاً شاید در بخشی که پیمایش انجام می‌شود، روی طول ظرف حساب شده باشد
- و یا شیئی که پیمایش و پردازش شده، توسط بخش دیگری از برنامه حذف شود
- جاوا از تغییر همزمان جلوگیری می‌کند

مفهوم شکست سریع (Fail Fast)

- اگر یک ظرف به واسطه یکی از متدهایش تغییر کند، همه iterator هایی که قبلاً روی این ظرف گرفته شده، غیرمعتبر می‌شوند
- هر عملیاتی که از این پس روی این iterator های غیرمعتبر انجام شود، منجر به پرتاب خطای ConcurrentModificationException می‌شود
- به این تکنیک، شکست سریع (Fail Fast) گفته می‌شود
- با تغییر یک ظرف توسط یک iterator، سایر iterator ها غیرقابل استفاده می‌شوند
- این تکنیک، روش جاوا برای جلوگیری از تغییر همزمان است

```
Collection<String> c = new ArrayList<String>();  
Iterator<String> itr = c.iterator();  
c.add("An object"); itr نامعتبر می‌شود  
String s = itr.next(); ConcurrentModificationException  
پرتاب
```

مثال دیگری برای ConcurrentModificationException

```
List<Integer> list = new ArrayList<Integer>();  
  
list.add(1);  
list.add(2);  
list.add(3);  
  
for (Integer integer : list)  
    if(integer.equals(1))  
        list.remove(integer);
```

متدهی بنویسید که لیستی از رشته‌ها به عنوان پارامتر بگیرد و همه رشته‌هایی که با Ali شروع می‌شوند را از لیست حذف کند

```
void removeAli(List<String> names){...}
```

یک راه حل اشتباه

```
void removeAli(List<String> list){  
    for (String string : list)  
        if(string.startsWith("Ali"))  
            list.remove(string);  
}
```

• این راه حل منجر به ConcurrentModificationException می‌شود

یک راه حل صحیح دیگر

```
public static void removeAli(List<String> list){  
    for (int i = list.size()-1; i >= 0; i--)  
        if(list.get(i).startsWith("Ali"))  
            list.remove(i);  
}
```

اگر پیمایش For از ابتدا به انتهای لیست می‌بود درست کار نمی‌کرد.

toArray

Collection to array: list.toArray / که بز نیم / Collections.

collection:

```
Collection<Integer> s = new HashSet<Integer>();
s.add(new Integer(6));
s.add(new Integer(7));
s.add(new Integer(5));

Object[] array = s.toArray();

Integer[] is = s.toArray(new Integer[s.size()]);
```

List:

مثال

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(new Integer(5));
list.add(new Integer(4));
list.add(new Integer(3));

Object[] array = list.toArray();
for (Object object : array) {
    Integer i = (Integer) object;
    System.out.println(i);
}

Integer[] array2 = list.toArray(new Integer[list.size()]);
for (Integer i : array2)
    System.out.println(i);
Integer[] array3 = list.toArray(new Integer[0]);
for (Integer i : array3)
    System.out.println(i);
```

Writing Codes:

تبديل آرایه به ArrayList

- گاهی لازم است یک آرایه را به یک `ArrayList` تبدیل کنیم، یا برعکس : `ArrayList`
- مثال برای تبدیل آرایه به `ArrayList` :

```
String[] strings = {"ali", "taghi"};
ArrayList<String> list = new ArrayList<String>();
for (String str : strings)
    list.add(str);
```

- مثال برای تبدیل `ArrayList` به آرایه :

```
String[] array = new String[list.size()];
for (int i = 0; i < array.length; i++)
    array[i] = list.get(i);
```

- راههای دیگری هم وجود دارد (بعداً می‌بینیم)

array to Collection: `Arrays.asList()`/`Arrays.copyOf()` / کد بزنیم

2.3. Changing the Returned List

Additionally, the list returned by `Arrays.asList()` is **mutable**. That is, we can change the individual elements of the list:

```
List<Integer> list = Arrays.asList(1, 2, 3, 4);
list.set(1, 1000);
assertThat(list.get(1)).isEqualTo(1000);
```



Eventually, this can lead to undesired side effects causing bugs that are difficult to find. When an array is provided as input, the change on the list will also be reflected on the array:

```
Integer[] array = new Integer[]{1, 2, 3};
List<Integer> list = Arrays.asList(array);
list.set(0,1000);
assertThat(array[0]).isEqualTo(1000);
```



Let's see another way to create lists.

3. Using `List.of()`

In contrast to `Arrays.asList()`, Java 9 introduced a more convenient method, `List.of()`. This creates instances of **unmodifiable** `List` objects:

3.1. Differences From `Arrays.asList()`

The main difference from `Arrays.asList()` is that `List.of()` returns an **immutable list that is a copy of the provided input array**. For this reason, changes to the original array aren't reflected on the returned list:

```
String[] array = new String[]{"one", "two", "three"};
List<String> list = List.of(array);
array[0] = "thousand";
assertThat(list.get(0)).isEqualTo("one");
```



Additionally, we cannot modify the elements of the list. If we try to, it will throw `UnsupportedOperationException`:

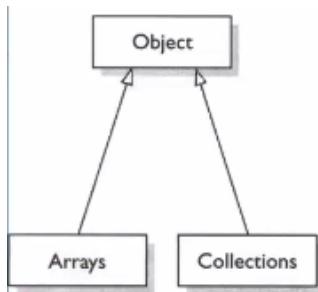
```
List<String> list = List.of("one", "two", "three");
assertThrows(UnsupportedOperationException.class, () -> list.set(1, "four"));
```



```
default Stream<E> stream() {
    return StreamSupport.stream(splitter(), parallel: false);
}
```

???

Class Collections and Arrays



Collections و Arrays کلاس‌های

- جاوا دو کلاس، با متدهای کمکی مفید برای کار با آرایه‌ها و Collection‌ها ارائه کرده است
- کلاس `java.util.Arrays` برای کار با آرایه‌ها
- کلاس `java.util.Collections` برای کار با Collection‌ها
- این کلاس‌ها دارای متدهای استاتیک متنوعی هستند، برای:
 - کپی اشیاء درون آرایه یا ظرف (fill)
 - پر کردن همه اعضا با یک مقدار مشخص (search)
 - جستجو (sort)
 - مرتب‌سازی (و ...)

۶

Collections.SynchronizedMap() ???

میدی sync تحويل میگیری.

وقتی سازگاری مهمه

کار ای بی کمتر از ConcurrentHashMap

مثال برای کاربرد Arrays

```
Random random = new Random();
Long[] array = new Long[100];
Arrays.fill(array, 5L);
Long[] copy = Arrays.copyOf(array, 200);
for (int i = 100; i < copy.length; i++)
    copy[i] = random.nextLong()%10;
//An unmodifiable list:
List<Integer> asList = Arrays.asList(1, 2, 3, 4);
List<Long> asList2 = Arrays.asList(array);
Arrays.sort(array);
int index = Arrays.binarySearch(array, 7L);
int[] a1 = {1,2,3,4};
int[] a2 = {1,2,3,4};           false
System.out.println(a1==a2);     false
System.out.println(a1.equals(a2)); true
System.out.println(Arrays.equals(a1, a2)); true
System.out.println(a1);         [I@7852e922
System.out.println(a1.toString()); [I@7852e922
System.out.println(Arrays.toString(a1)); [1, 2, 3, 4]
```

```
List<String> list = new ArrayList<String>();
Collections.addAll(list, "A", "Book", "Car", "A");
int freq = Collections.frequency(list, "A"); 2
Collections.sort(list); A, A, Book, Car
Comparator<String> comp = new Comparator<String>(){
    public int compare(String o1, String o2) {
        return o1.length() < o2.length() ? -1 :
            (o1.length() == o2.length() ? 0 : +1);
    }
};
Collections.sort(list, comp); A, A, Car, Book
Collections.reverse(list);
String min = Collections.min(list); A
String max = Collections.max(list); Car
String max2 = Collections.max(list, comp); Book
Collections.shuffle(list);
Collections.fill(list, "B");
```

مثال برای کاربرد Collections

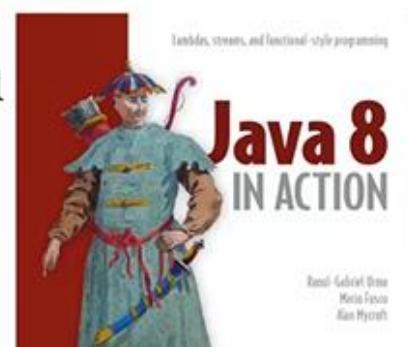
Java 8

جاوا هشت: کاهش خطوط کد و خوانایی؛ اگر خوب کد بزنیم پرفورمنس و پردازش موازی برنامه نویسی فانکشنی، استریم و پارالل ، لامبدا

**** نهایتا کاربرد F1 این است که من با lambda متد abs یک F1 را بدنه میدهم و بعنوان parameter پاس میدهم به متدهی که جاوا مهیا کرده یا خودم نوشتم.

• Java 8 in Action:

Lambdas, Streams, and
functional-style
programming



• عبارت لامبدا (Lambda Expression)

• برنامه‌نویسی تابعی (Functional Programming)

• واسط تابعی (Functional Interface)

• جویبار (Stream)

• جویبارهای موازی (Parallel Streams)

• امکانات جدید و گسترده کتابخانه‌ی جاوا ۸



• برنامه‌نویس جاوا ۸ ، می‌تواند «تابعی» بیاندیشد

چرا جاوا ۸ نسخه مهمی است؟

- Functional Programming
- Thinking Functional

• این تغییر، در عمر جاوا بی‌سابقه است

• برنامه‌نویس جاوا عادت کرده که شیء‌گرا فکر کند

• جاوا ۸ کتابخانه و API زبان را گسترش داده و تقویت کرده است

• نیاز به کتابخانه‌های کمکی (مثل Apache Commons) کمتر می‌شود

• با معرفی جاوا ۸ ، دستخط برنامه‌نویسی جاوا به مرور تغییر خواهد کرد

• اگر دانش جاوا ۸ نداشته باشیم، بسیاری از کدها را نخواهیم فهمید

• جاوا ۸ گسترده‌ترین تغییر در تاریخ «زبان جاوا» است

• حتی گسترده‌تر از جاوا ۵

• که ساختارهای مهمی مانند Generic و Annotation را معرفی کرد

• برنامه‌نویس به جای چگونگی انجام کار، می‌تواند فقط هدف کار را توصیف کند

• «what to do» instead of «how to do»

Lambda Expression

It's for the **creation** of anonymous functions. It **simplifies** the syntax of functional programming in Java.

**** یک قطعه کد، معرف بدنی یک تابع (**anonymous**) یا FI یا ...)

اغلب کاربردش برای **functional Interfaces** کردن **impl** اها است: جهت کاهش کد نویسی و خوانایی

داخل عبارت Lambda میتوانیم متدهای دیگر را **call** کنیم که هر کدام میتوانند عملیات سنگینی را انجام دهند.

متدهای FI utility خود

متدهای Obj ورودی به مت **abs** روی خود obj، یک مت static از یک کلاس utility

توالی اجرای متدهای FI دیگر و یک method ref

عبارت لامبدا (Lambda Expression)

مثال:

$r \rightarrow r^2 * 3.14$
 $(x, y) \rightarrow x + y$

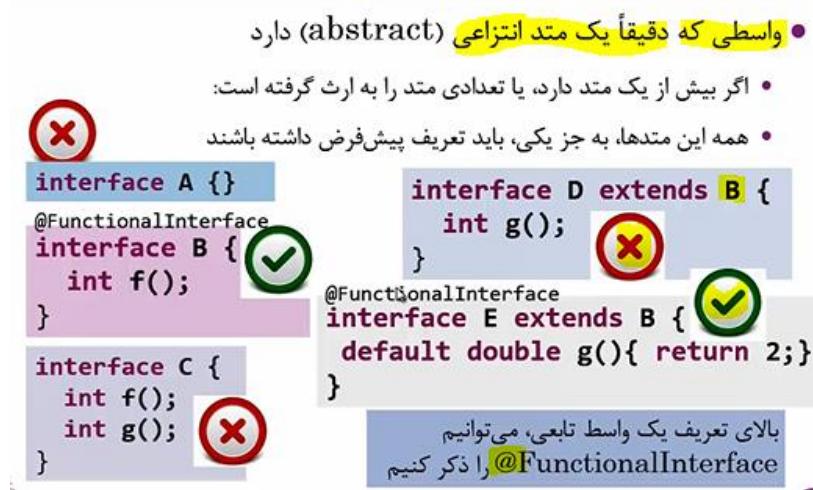
- لامبدا یا لاندا (λ)
- یک عبارت لامبدا:
- قطعه کدی است که بدنی یک تابع را توصیف می کند زیرا هر واسط تابعی، فقط یک متند نامشخص (انتزاعی) دارد
- و به عنوان یک واسط تابعی قابل استفاده است

Functional Interface

ای که فقط یک متد abstract دارد. (اگر بخواهد متدهای دیگری داشته باشد باید default یا static باشند) private

میتوان یک Interface را extend کرد و بجز یک متد، همه متدات آنرا با default بدنده داد و آنرا تبدیل به یک functional Interface کرد.

واسطه تابعی (Functional Interface)



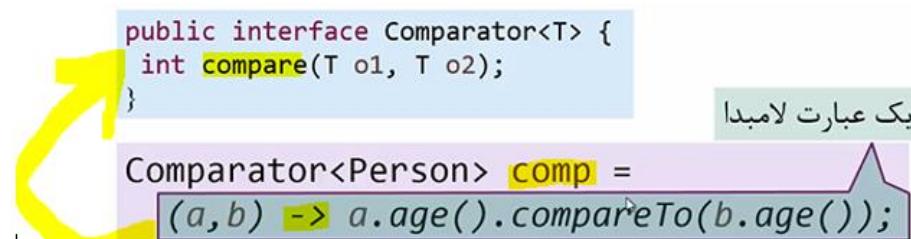
روش های یک impl Functional Interface

روش اول: سنتی

نوشتن یک class جدید / impl کردن آن override / interface abs آن / New کردن یک ref از آن class فرزند و ساخت یک instance

روش دوم: ****

همه چیز در یک FI مشخص است بجز بدنه ای تابع abs، بنابراین با Method Reference یا Lambda آنرا تعیین میکنیم، و در ref از یک FI قرار میدهیم. درست است عجیب است ولی هدف جاوا کاهش خطوط کد بوده است. حالا همه چیز درباره ای آن Interface مشخص است.



:FI استفاده یک ****

حالا میتوانیم از ref ای که از آن Interface داشتیم استفاده کنیم، متدهای غیر abs و abs آن را call کنیم:

```
Comparator<Person> comparator =  
(p1, p2) -> p1.getName().compareTo(p2.getName());  
  
Person p1 = new Person("Ali");  
Person p2 = new Person("Taghi");  
  
comparator.compare(p1, p2); // < 0
```

:FI استفاده دو ****

بسیاری از متدهای پرکاربرد جوا بعنوان ورودی، FI هایی که جوا در خود معرفی کرده میگیرند.

مثل متدهای Stream و Sort و Optional

وقتی برای impl تنها متدهای abs از method ref lambda یا استفاده کنیم باعث کاهش خطوط کد و افزایش خوانایی میشود.
متدهایی که بدن دارند حکم utility دارند.

مثال:

در متدهای sort و ref به یک پیاده سازی از FI comparator میخواهد ، میتوانیم از lambda استفاده کنیم و بدن داری تنها متدهای abs را در ref قرار دهیم و ref را پاس دهیم به sort. حتی بجای ref، خود بدن داری متدهای abs را هم میتوانیم بفرستیم.

```
List<Person> people =  
Arrays.asList(  
new Student("Ali", 1993),  
new Student("Taghi", 1990),  
new Student("Naghi", 1995));  
  
Collections.sort(people,  
(a, b) -> a.age().compareTo(b.age()));  
  
Collections.sort(people,  
new Comparator<Person>() {  
@Override  
اشاره: یک عبارت لامیدا به یک  
کلاس داخلی بینام ترجمه نمی شود
```

Method Reference

ارجاع به متدها (Method Reference)

امکانی جدید در جاوا ۸ که مانند اشاره گر به متدها عمل می‌کند

از :: برای ارجاع به متدها استفاده می‌شود

```
class Str {  
    Character startsWith(String s) {  
        return s.charAt(0);  
    }  
}  
  
@FunctionalInterface  
interface Converter<F, T> {  
    T convert(F from);  
}  
  
Str str = new Str();  
Converter<String, Character> conv = str::startsWith;  
Character converted = conv.convert("Java");
```

ارجاع به متدها

حالت های Method reference

• ارجاع به متدها

```
Converter<String, Character> conv = str::startsWith;
```

• ارجاع به متدهای استاتیک

```
Converter<String, Integer> converter = Integer::valueOf;
```

• ارجاع به سازنده (Constructor)

```
interface Factory<T> {  
    T create();  
}  
Factory<Car> factory1 = Car::new;  
Car car1 = factory1.create();
```

Class Object constructor static را روی Method ref

به یک متدهای instance object به واسطه یک Method ref

روش دوم پیاده سازی یک FI:

همه چیز در یک FI مشخص است بجز بدنی تابع abs، بنابراین با Method Reference یا Lambda آنرا تعیین می‌کنیم، و در ref از یک FI قرار میدهیم. حالا همه چیز درباره آن Interface مشخص است.

Embedded Functional Interfaces

تعدادی **Functional Interface** در بسته‌ی util.function اضافه شده‌اند که میتوانیم بعنوان قرارداد ازشون استفاده کنیم، هر کدام از این FI‌ها متدهای پیاده سازی شده کمکی زیادی هم دارند.

**** پس خلاصه:

میتوانیم FI بنویسیم، یا از FI‌های قراردادی جواوا استفاده کنیم

با lambda یا ref method آن را کنیم

در هر جای منطق کد خودمان از آن استفاده کنیم، مثل چکینگ null بودن یک متغیر

یا ورودی بدھیم به متدهایی که جواوا مهیا کرده، مثل Sort

FI Predicate

```
boolean test (T var1);
```

یک FI است، متدهای abs و GT میگیرد و یک Boolean بر میگرداند.

Predicate

- یک واسط تابعی است: یک پارامتر می‌گیرد و boolean بر می‌گرداند

- این واسط، متدهای پیشفرض مختلفی دارد

- مانند برای ارزیابی و negate و or و and برای ترکیب Predictae ها

```
String st = "ok";
boolean b;
Predicate<String> notEmpty = (x) -> x.length() > 0;
b = notEmpty.test(st); // true
b = notEmpty.negate().test(st); // false

Predicate<String> notNull = x -> x != null;
b = notNull.and(notEmpty).test(st); // true

Predicate<String> isEmpty = String::isEmpty;
Predicate<String> isNotEmpty = isEmpty.negate();
```

FI Comparator

```
int compare(T o1, T o2);
```

متدهای abs آن دو ورودی obj میگیرد و یک int(1,0,-1) برمیگرداند.

```
Collections.sort(apples, (a1,a2)->a1.size-a2.size);
```

FI Function

```
R apply (T var1);
```

Function

- تابعی است که یک پارامتر میگیرد و یک خروجی تولید میکند
- با متدهای `apply` این تابع اجرا میشود
- متدهای پیشفرضی مانند `andThen` برای ترکیب زنجیروار تابعها

```
Function<String, Integer> toInteger =
    Integer::valueOf;

Function<String, String> backToString =
    toInteger.andThen(String::valueOf);

backToString.apply("123"); // "123"
```

FI Supplier

```
T get();
```

Supplier

- تابعی که یک شیء تولید (تأمین) می‌کند

- (برخلاف Function) هیچ پارامتری نمی‌گیرد

- اجرای تابع: با کمک متدهای **get**

```
Supplier<Person> personSupplier = Person::new;
personSupplier.get(); // new Person
```

```
@Test
public void whenOrElseGetWorks_thenCorrect() {
    String nullName = null;
    String name = Optional.ofNullable(nullName).orElseGet(() -> "john");
    assertEquals("john", name);
}
```

FI Consumer

```
void accept(T var1);
```

Consumer

- تابعی که فقط یک پارامتر می‌گیرد و خروجی ندارد

- یک شیء (پارامتر) را مصرف می‌کند

- اجرای تابع (مصرف شیء) با متدهای **accept** انجام می‌شود

```
Consumer<Person> greeter =
    p -> System.out.println(p.getFirstName());
greeter.accept(new Person("Ali"));
```

مروری بر واسطه‌های تابعی

واسطه تابعی	توضیح
Predicate<T>	یک پارامتر از جنس T می‌گیرد و boolean برمی‌گرداند
Consumer<T>	یک پارامتر از جنس T می‌گیرد و آن را پردازش می‌کند ولی چیزی برنمی‌گرداند (یک شیء از جنس T را مصرف می‌کند)
Supplier<T>	یک شیء از جنس T تولید می‌کند (پارامتر نمی‌گیرد)
Function<T, U>	نوع T را به نوع U تبدیل می‌کند. پارامتری از جنس T می‌گیرد، آن را پردازش می‌کند و یک شیء از جنس U برمی‌گرداند.
BiFunction<T, U, V>	دو پارامتر به ترتیب از جنس T و U می‌گیرد و شیءی از نوع V برمی‌گرداند
UnaryOperator<T>	یک عملگر تکی که یک شیء از جنس T می‌گیرد و شیءی از همین جنس برمی‌گرداند
BinaryOperator<T>	دو پارامتر از جنس T می‌گیرد و شیءی از همین جنس برمی‌گرداند

ورودی و خروجی تک متده abs را برای هر Fl

به همین ترتیب، واسطه‌های تابعی دیگری هم تعریف شده‌اند:

BiConsumer <T , U>
BiPredicate <T , U>

```
BiFunction<Integer, Integer, String> biFunction =
    (num1, num2) -> "Result:" +(num1 + num2);
System.out.println(biFunction.apply(20,25));
```

```
BiPredicate<Integer, String> condition =
    (i, s) -> i.toString().equals(s);
System.out.println(condition.test(10, "10"));//true
System.out.println(condition.test(30, "40"));//false
```

FI as method input

نهایتاً کاربرد FI این است که من با abs متد lambda یک FI را بدهم و بعنوان parameter پاس میدهم به متندی که جاوا مهیا کرده یا خودم نوشتم.

my filter

```
public class Exercise1 {  
    public static void main(String[] args) {  
        List<Apple> apples = Arrays.asList(  
            new Apple(1),  
            new Apple(2),  
            new Apple(2),  
            new Apple(3),  
            new Apple(5));  
  
        List<Apple> filtered = filter(apples, a-> a.size>2);  
    }  
  
    static List<Apple>  
    filter(List<Apple> list, Predicate<Apple> condition){  
        List<Apple> result = new ArrayList<>();  
        for (Apple apple : list) {  
            if(condition.test(apple))  
                result.add(apple);  
        }  
    }  
}
```

Collection.forEach

وقتی ورودی یک متدهیم ref از یک فیلتر فارم (فیلتر فارم) میدهیم (جاوا هم در کلاس های خود اینکار را میکند)، در داخل آن متدهیم abs روی ref ای که در ورودی گرفته call خواهد شد. مثلا در کلاس HashMap، ورودی متدهیم forEach است. متدهیم accept را روی ref ای که ما از برایش فرستادیم call خواهد کرد.

```
public void forEach(BiConsumer<? super K, ? super V> action) {
    if (action == null) {
        throw new NullPointerException();
    } else {
        Node[] tab;
        if (this.size > 0 && (tab = this.table) != null) {
            int mc = this.modCount;
            Node[] var4 = tab;
            int var5 = tab.length;

            for(int var6 = 0; var6 < var5; ++var6) {
                for(Node<K, V> e = var4[var6]; e != null; e = e.next) {
                    action.accept(e.key, e.value);
                }
            }

            if (this.modCount != mc) {
                throw new ConcurrentModificationException();
            }
        }
    }
}
```

روی یک عملگر forEach HashMap را کال می کنیم و Consumer را به آن پاس میدهیم:

```
Map<Integer, String> mymap= new HashMap();
mymap.put(1,"Mehrad");
mymap.put(2,"Sareh");
mymap.put(3,"Baba");

BiConsumer<Integer, String> myFibicons= (k,v)-> System.out.println("".concat(String.valueOf(k)).concat(":").concat(v)) ;
mymap.forEach(myFibicons);
```

Collections.sort

راه اول: وقتی که Class Type آنرا باشد ، فقط کافی است **comparable** ای از آنرا بدهیم به .Collections

```
Collections.sort(apples)
```

راه دوم: یک class بسازیم و **comparator** را برای مقایسه به sort بدهیم:

راه سوم: ساده کردن راه دوم

```
Collections.sort(apples, new Comparator<Apple>() {  
    @Override  
    public int compare(Apple o1, Apple o2) {  
        return o1.size-o2.size;  
    }  
});
```

راه چهارم: چون FI است میتوانیم فقط متد abs آن را بذنه بدهیم. دو ورودی و یک int خروجی.

```
Collections.sort(apples, [(a1,a2)->a1.size-a2.size]);
```

داخل پیاده سازی متد sort از کلاس Collections گرفته، متد آن به نام compare صدا زده میشود. این متد دو obj میگیرد و یک int برمیگرداند.

```
Collections.sort(fruitList,
```

```
    Collections.reverseOrder(anotherRefOfCompraratorFI));
```

Optional methods

ورودی بعضی از متدهای `FI` ، `Optional` است.

```
@Test
public void givenOptional_whenIfPresentWorks_thenCorrect() {
    Optional<String> opt = Optional.of("baeldung");
    opt.ifPresent(name -> System.out.println(name.length()));
}
```

consumer

```
@Test
public void whenOrElseGetWorks_thenCorrect() {
    String nullName = null;
    String name = Optional.ofNullable(nullName).orElseGet(() -> "john");
    assertEquals("john", name);
}
```

Supplier

```
@Test(expected = IllegalArgumentException.class)
public void whenOrElseThrowWorks_thenCorrect() {
    String nullName = null;
    String name = Optional.ofNullable(nullName).orElseThrow(
        IllegalArgumentException::new);
}
```

Supplier

Stream methods

مثالهاش از روی فهرست بخش خودش هم مشخص هست

Class Optional

خروجی متدها را Optional تعریف میکنیم که obj مورد نظر داخل آن قرار میگیرد.

حوالمند باشد اگر null بود exception نخوریم.

که با متدهایی که Optional در اختیارمان میدهد چک کردن obj null بودن obj و راه های جایگزین (ساخت obj و یا پرتاب exception) را آسانتر و با خطوط کد کمتری باشد.

ورودی بعضی از متدهای F1 ، Optional است.

Optional مفهوم

- یک کلاس جدید: تلاشی برای جلوگیری از NullPointerException
- هر Optional یک شیء در دل خود دارد

- خروجی (مقدار برگشتی) بسیاری از متدهای جدید، از جنس Optional است
- این متدها به جای null، یک Optional بر می‌گردانند که شیء درون آن null است

```
Optional<String> opt = Optional.of("salam");
boolean b = opt.isPresent(); // true
String s = opt.get(); // "salam"
String t = opt.orElse("bye"); // "salam"

// prints "s"
opt.ifPresent((s) -> System.out.println(s.charAt(0)));
```

- کمک می‌کند که برنامه‌نویس حواسش را جمع کند
- وجود مقدار را (قبل از استفاده از آن) چک کند (متلاً با متدهای توسعه داده شوند)
- اشتباه برنامه‌نویس و رخداد NullPointerException کمتر رخ می‌دهد
- کدهای تولیدشده هم تمیزتر و خواناتر و موجزتر می‌شوند

```
public Car findCar(String color) {...}
System.out.println( findCar ("Black").getColor() );
public Optional<Car> findCar(String color) {...}
findCar("Black").
ifPresent(car->System.out.println(car.getColor()));
```

Optional of ()

خوب نیست: وقتی `of()` خوب است که قبلش چک کرده باشیم داخلش `null` نیست.

```
@Test(expected = NullPointerException.class)
public void givenNull_whenThrowsErrorOnCreate_thenCorrect() {
    String name = null;
    Optional.of(name);
}
```

But in case we expect some `null` values, we can use the `ofNullable()` method:

Optional ofNullable ()

بهتر: `get` و `check` را باهم خودش انجام میدهد:

```
@Test
public void givenNonNull_whenCreatesNullable_thenCorrect() {
    String name = "baeldung";
    Optional<String> opt = Optional.ofNullable(name);
    assertTrue(opt.isPresent());
}
```

By doing this, if we pass in a `null` reference, it doesn't throw an exception but rather returns an empty `Optional` object:

```
@Test
public void givenNull_whenCreatesNullable_thenCorrect() {
    String name = null;
    Optional<String> opt = Optional.ofNullable(name);
    assertFalse(opt.isPresent());
}
```

Void ifPresent ()

است که یک **Optional** میگیرد و چک میکند اگر **null** نبود از **obj** استفاده میکند. با **ifPresent** چک کردن **null** با تعداد خطوط کد کمتری انجام میشود.

```
public void ifPresent(Consumer<? super T> action) {  
    if (value != null) {  
        action.accept(value);  
    }  
}
```

```
@Test  
public void givenOptional_whenIfPresentWorks_thenCorrect() {  
    Optional<String> opt = Optional.of("baeldung");  
    opt.ifPresent(name -> System.out.println(name.length()));  
}
```

Object orElseGet ()

همچنین متد **orElseGet** ، یک میگیرد ، چک میکند اگر null بود با کمک **supplier** ، obj مورد نیاز را میسازد:

```
@Test
public void whenOrElseGetWorks_thenCorrect() {
    String nullName = null;
    String name = Optional.ofNullable(nullName).orElseGet(() -> "john");
    assertEquals("john", name);
}
```

```
String defaultText = Optional.ofNullable(text).orElseGet(this::getMyDefault);
assertEquals("Default Value", defaultText);
```

Exception orElseThrow ()

متد **orElseThrow** ، یک میگیرد ، چک میکند اگر null بود با کمک **supplier** ، obj Excep مورد نیاز را میسازد:

```
@Test(expected = IllegalArgumentException.class)
public void whenOrElseThrowWorks_thenCorrect() {
    String nullName = null;
    String name = Optional.ofNullable(nullName).orElseThrow(
        IllegalArgumentException::new);
}
```

Class Stream (methods)

یک stream از روی **Collection** ساخته میشود.

برای عملیات بیمایش و پردازش

:Stream فایده‌ی

تعداد خطوط کمتر میشود ، خواناتر / پردازش موازی / خطای دولوپ احتمال کمتر میشود

متدهای کلاس Stream (filter, sorted, intermediate, terminal) در جاوا تعییه شده اند (مثل ... مثل ...) که پارامتر به شکل FI دریافت میکنند. مثلاً متدهای abs و ref از FI predicate یک filter میگیرد.

طبیعی است که متدهای مهیا شده در Stream در متن خود، متدهای abs و ref از FI که در آنها عنوان پارامتر ورودی گرفتند را صدا میزنند.

رفع سوءتفاهم درباره جویبار

- آن را با مفهوم جویبار در مباحث فایل و I/O اشتباه نگیرید
- هیچ ربطی به OutputStream و InputStream ندارد



مفهوم جویبار (Stream)

- جویبار یک دنباله از اشیاء است

- بر روی اعضای این دنباله، یک یا چند عمل انجام می‌شود

```
interface java.util.stream.Stream<T>
```

- بر روی هر collection می‌توانیم یک جویبار ایجاد کنیم

- برای پردازش اعضای این مجموعه با کمک جویبار

```
List<String> list = ...  
Stream<String> stream = list.stream();  
stream.forEach(System.out::println);
```

Program “**what to do**” instead of “**how to do**”

:

Terminal operations return **Optional** type

- جویبار بر روی یک منبع (source) ایجاد می‌شود
- منبع، معمولاً یک collection است (مثلاً یک List یا Set)
- (ممکن است منبع یک جویبار چیز دیگری مانند آرایه، یک کانال IO و یا یک تابع مولد باشد)
- جویبار، بر اعضای یک مجموعه از اشیاء پیمایش (عبور) می‌کند
- و یک یا چند عمل (operation) بر روی این اعضاء انجام می‌دهد
- دو نوع عمل بر روی جویبار ممکن است:
 - عمل پایانی (terminal)
 - داده‌ای از یک نوع خاص برمی‌گرداند (Void, String, Integer مثلاً)
 - عمل میانی (intermediate)
 - همان جویبار را برمی‌گرداند: می‌توانیم عمل‌های دیگری را به زنجیره عمل‌ها اضافه کنیم



عملیات جویبار

- در ادامه، امکانات جویبارها را مرور می‌کنیم
- و عمل‌های مختلف (Stream Operations) بر روی یک جویبار را می‌بینیم
- ... map, filter, forEach

در ادامه فرض می‌کنیم:

```
class Car{  
    int price;  
    String color;  
    // getters & setters  
    public Car(String color, int price) {  
        this.color = color;  
        this.price = price;  
    }  
    public String toString() {  
        return "Car[color="+color+",price="+price+"]";  
    }  
}
```

```
List<Car> list =  
    Arrays.asList(  
        new Car("Black", 30),  
        new Car("Black", 40),  
        new Car("Black", 50),  
        new Car("White", 20),  
        new Car("Yellow", 60)  
    );
```

Creating Stream

یک: بهترین راه: با `stream()` تبدیل کنیم و با `List` یا `Set` درست کنیم؛ Collections

```
list.stream()
```

دو: برای primitive ها

بازه‌ای از اعداد

- واسطه‌ای مانند `LongStream` و `IntStream` وجود دارند
- که می‌توانند بازه (range) اعداد ایجاد کنند
- با انواع اولیه کار می‌کنند
- `Long` به جای `long`، `Integer` به جای `int`

```
IntStream oneTo19 = IntStream.range(1, 20);
IntStream oneTo20 = IntStream.rangeClosed(1, 20);
```

```
oneTo19.forEach(System.out::println);
oneTo20.forEach(System.out::println);
```

- با کمک امکان **iterate** می‌توانیم دنباله‌ای از مقادیر را توصیف کنیم

```
Stream.iterate(0, x -> x + 2)
    .limit(10)
    .forEach(System.out::println);
```

- پارامتر اول **iterate**: اولین عضو دنباله

- پارامتر دوم: یک **UnaryOperator**

- که نحوه ایجاد هر عضو بعدی از عضو قبلی را مشخص می‌کند

- نکته: متدهای **iterate** یک جواب بینهایت می‌سازد

- با کمک **limit** تعداد اعضای دنباله که قرار است پردازش شوند، محدود شود

چهار: **Files.lines()**

- جواب از مقادیر

```
Stream<String> stream = Stream.of("A", "B", "C");
```

- ساخت جواب بر روی آرایه

```
String[] array = {"Ali", "Taghi"};
```

```
Arrays.stream(array);
```

- جواب بر روی فایل

```
Stream<String> lines =
    Files.lines(Paths.get("data.txt"));
```



filter (FIPredicate)

Filter

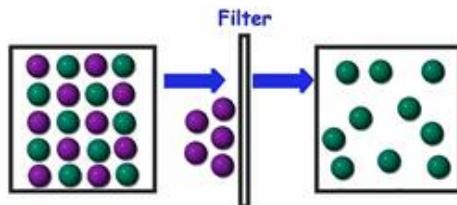


- برخی از اعضای جویبار را حفظ می‌کند

- سایر اعضا که شرط موردنظر را ندارند نادیده گرفته می‌شوند

```
list.stream()
  .filter(a->"Green".equals(a.color));
  .forEach(System.out::println);
```

از اعضای لیست، آنها بیکه رنگشان سبز است
چاپ شوند



- یک عملیات پایانی است یا میانی؟

- میانی (intermediate)

- یعنی همان جویبار را بر می‌گرداند

- پارامترش از چه نوعی است؟

- Predicate (شیء را بررسی می‌کند و boolean برمی‌گرداند)

sorted (FIComparator)



- یک عمل میانی (intermediate operation)

- جویبار را مرتب (sort) می‌کند

- اگر اعضای جویبار Comparable باشند:

- عملیات sorted بدون پارامتر کار می‌کند

- عملیات sorted امکان دریافت یک پارامتر از نوع Comparator را دارد

- که نحوه مقایسه اعضای جویبار را مشخص می‌کند

- مثل list که اعضای آن (Car) از جنس Comparable نیستند

- یا در مواقعي که می‌خواهیم روش خاصی را برای مقایسه استفاده کنیم

```
list.stream().sorted((a,b)->a.price-b.price)
```

- نکته: عملیاتی مانند filter و sorted تغییری در منبع جویبار ایجاد نمی‌کنند

- مثال لیستی که جویبار بر روی آن ساخته شده، با فراخوانی sorted مرتب نمی‌شود

```
Stream<T> sorted(Comparator<? super T> comparator);
```

stream.sorted() هستند، هر سه Collections.sort() و Collection.sort() میگیرند.

limit () & skip ()

Limit

- تعداد اعضای جویبار که پردازش می‌شوند را محدود می‌کند
- مثلاً limit(2) یعنی دو عضو ابتدای جویبار در نظر گرفته شوند

عملیات موردنظر روی سایر اعضا انجام نمی‌شود

skip ← limit نقطه مقابل

- list.stream().limit(2).forEach(System.out::println);
دو عضو ابتدای لیست را چاپ می‌کند
- list.stream().skip(2).forEach(System.out::println);
همه اعضا لیست به جز دو عضو اول را چاپ می‌کند

Void forEach (F1Consumer)

ForEach

- یک عملیات را بر روی هر عضو جویبار انجام می‌دهد. مثال:

```
list.stream().forEach(System.out::println);
```

```
list.stream().forEach(a-> System.out.println(a.color));
```

- یک عملیات پایانی (terminal) است یا میانی (intermediate) است

پایانی، یعنی بعد از فراخوانی آن عمل دیگری بر روی جویبار قابل انجام نیست



چه چیزی برمی گرداند؟

(void) هیچ

پارامترش از چه نوعی است؟

(System.out::println) مثلاً Consumer

مثلاً مثل یک ارجاع به متده است (مثال اول فوق) یا یک عبارت لامبدا (مثال دوم)

map (Function)

Map

- یک عملیات میانی: هر یک از اعضای جویبار را به شیء دیگری تبدیل می‌کند

```
list.stream()
    .map(car->car.color)
```

- نحوه تبدیل را به صورت پارامتر دریافت می‌کند
- پارامترش یک Function است
- `java.util.function.Function<T, R>`

- جویباری از ماشین‌ها را به جویباری از رشتلهای تبدیل می‌کند

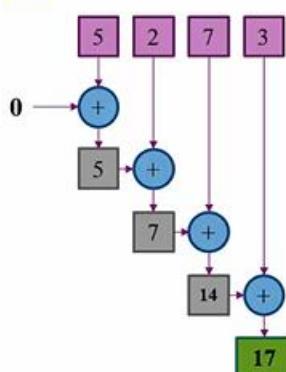
- `Stream<Car> → Stream<String>`

- نکته: این عملیات map ربطی به `HashMap` (مثلاً `java.util.Map`) ندارد

```
list.stream()
    .map(car->car.color)
    .filter(color ->
        color.startsWith("B"))
    .forEach(System.out::println);
```

Optional reduce (BinaryOperator)

Reduce



- یک عمل پایانی
- یک مقدار تجمعی از مجموعه اعضای جویبار استخراج می‌کند
- مثال: از همه ماشین‌ها، مجموع قیمت‌شان را محاسبه کن
- پارامتر `reduce`: دو مقدار می‌گیرد و آن دو را ترکیب می‌کند
- از این دو مقدار، یک مقدار حاصل می‌کند
- پارامتر `reduce` از چه نوعی است?
- `BinaryOperator`
- خروجی آن از جنس `Optional` است
- چرا؟

```
Optional<Integer> sumOfPrices = list.stream()
    .map(car->car.price)
    .reduce((price1, price2)->price1+price2);
```

```
sumOfPrices.ifPresent(System.out::println);
```

Long count ()

Count

- یک عملیات پایانی (terminal)

```
long lowPrices =  
list.stream()  
.filter((a) -> a.price<40)  
.count();
```

تعداد اعضای جویبار را برمی‌گرداند

برمی‌گرداند long

مثال:

- تعداد اعضای لیست که قیمتی کمتر از ۴۰ دارند



Collection collect (Collector)

Collect

- یک عملیات پایانی

• یک مجموعه از اشیاء (مثلًا یک Set یا List) از جویبار استخراج می‌کند

• یک Collector به عنوان پارامتر می‌گیرد

مثال:

```
List<Car> newList = list.stream().filter((a) -> a.price<40)  
.collect(Collectors.toList());
```

```
Set<Car> set = list.stream().filter((a) -> a.price<40)  
.collect(Collectors.toSet());
```

Function<Car, String>

Function<Car, Car>

```
Map<String, Car> map = list.stream()  
.collect(Collectors.toMap(car->car.color, car-> car));
```

یک راه تبدیل یک List به یک Map بود.

Boolean match (FIPredicate)

Match

```
boolean anyBlack =  
list.stream()  
.anyMatch(car -> car.color.equals("Black"));  
  
boolean allBlack =  
list.stream()  
.allMatch(car -> car.color.equals("Black"));  
  
boolean noneBlack =  
list.stream()  
.noneMatch(car -> car.color.equals("Black"));  
.
```

distinct ()

```
Stream<T> distinct();
```

Returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream.

Other methods

- `stream.distinct()`
 - `Optional<T> stream.findAny()`
 - `Optional<T> stream.findFirst()`
 - `stream.max()`
 - `stream.min()`
 - `stream.toArray()`
 - ...
-

practicing

کوییز

قطعه برنامه‌ای بنویسید که:

- از مجموعه کارمندان شرکت (List<Employee>) از میان کسانی که متاهل هستند و حقوقشان کمتر از ۱۰۰۰ است نام ۱۰ نفر اول را به ترتیب حقوق (از کم به زیاد) چاپ کند

```
List<Employee> list = ...
list.stream()
.filter(e->e.isMarried)
.filter(e->e.salary<1000)
.sorted((a,b)->a.salary-b.salary)
.limit(10)
.forEach(e->System.out.println(e.name));
```

- می‌خواهیم یک لیست از خودروها را پردازش کنیم
- یک جویبار بر روی این لیست ایجاد می‌کنیم؛ stream()

از بین مواردی که رنگشان مشکلی است

یک فیلتر بر روی این جویبار ایجاد می‌کنیم
که فقط مشکل‌ها را بپذیرد: filter()

خودروها را براساس قیمت مرتب کنیم
sorted() جویبار را مرتب می‌کنیم؛

دو مورد با کمترین قیمت را انتخاب کنیم
limit() اعضای موردپردازش را محدود کنیم؛

و اطلاعات این دو خودرو را چاپ کنیم
forEach()

هر یک از اعضای جویبار فوق را چاپ کنیم؛

```
list.stream()
.parallel()
.filter((a)->a.price<40)
.forEach(System.out::println);
```

حتی می‌توانیم فرایند اجرای جویبار را موازی کنیم

(Multi-Thread)

به سادگی و با فراخوانی یک متدها parallel

Map reduce

```
public class Exercise2 {  
    public static void main(String[] args) {  
        Set<String> set = new HashSet<>();  
        set.add("log>Hello World");  
        set.add("log>Ok");  
        set.add("log>Warning");  
        set.add("log>Fatal Error");  
        set.add("Salam");  
        set.add("Java8");  
        set.add("Streams are great!");  
  
        Optional<Integer> sum =  
            set.stream()  
                .parallel()  
                .map(s->s.length())  
                .reduce((a,b)->a+b);  
  
        sum.ifPresent(System.out::println);  
    }  
}
```

```
public static void main(String[] args) {  
    Set<String> set = new HashSet<>();  
    set.add("log>Hello World");  
    set.add("log>Ok");  
    set.add("log>Warning");  
    set.add("log>Fatal Error");  
    set.add("Salam");  
    set.add("Java8");  
    set.add("Streams are great!");  
  
    Optional<Integer> sum =  
        set.stream()  
            .parallel()  
            .filter(s->s.startsWith("log"))  
            .map(s->s.length())  
            .sorted()  
            .limit(2000)  
            .reduce((a,b)->a+b);  
  
    sum.ifPresent(System.out::println);  
}
```

● این برنامه چه می‌کند؟

```
IntStream.range(2, 100)
    .filter(  
        a -> IntStream.range(2, a - 1)
            .noneMatch(x -> a % x == 0)
    )
    .forEach(System.out::println);
```

● اعداد اول بازه دو تا ۱۰۰ را چاپ می‌کند

Parallel Stream

معیارهای استفاده از `parallel()`

از منبعی که تجزیه پذیری خوب است ساخته شده باشد

اندازه ای `stream` زیاد باشد (تعداد اعضا)

حجم پردازش بالا باشد

رفتار هسته های `CPU`

عملیات ابتدا ظرف `stream` را تجزیه میکند و سپس بصورت موازی پردازش را انجام میدهد.

هر نوع `Data Structure` ای که تجزیه پذیری راحت تری دارد، احتمالاً با `parallel` کردن `stream` آن، سرعت پردازش زیاد خواهد شد.

- سرعت تجزیه پذیری عالی دارد.

- گاهی مثلاً تعداد عناصر در `ArrayList` آنقدر کم است که `parallel` کردن آن سرعت پردازش را کاهش میدهد.

- گاهی در `LinkedList` آنقدر عمل پردازش سنگین است که با وجود تجزیه پذیری ضعیف باز هم میصرفه که پردازش `stream` را بصورت `parallel` انجام دهیم.

جویبارهای موازی (Parallel Streams)

• با کمک متدهای `parallel`

• عملیات مختلف به صورت موازی بر روی جویبار اجرا می‌شوند

```
list.stream().parallel()  
.sorted().forEach(System.out::println);
```

• به این ترتیب: نیازی به ایجاد دستی `thread` ها نیست

• هم عملیات `sort` و هم `forEach` به صورت موازی انجام می‌شود
(multi-thread)

• این که متدهای `parallel` در کجا فراخوانی شده مهم نیست

- دنباله اعضا را به چند بخش تقسیم می کند
- و هر بخش در یک thread مجزا پردازش می شود
- به صورت پیش فرض، به تعداد هسته های پردازشی thread می سازد



• نکته:

- امکانات جدید جاوا از نسخه ۵ به بعد
- برای تسهیل و کم خطر شدن برنامه نویسی همروند
- Java 5: Thread Pools, Concurrent Collections
- Java 7: Fork/Join Framework

. با کمک Parallel () Fork/Join Framework پیاده سازی شده است.

.parallel()
.sequential()
.parallel()

مهم اینه آخرین بار کدام صدا زده شود.

- استفاده از جویبار موازی، بسیار ساده و وسوسه برانگیز است
- اما استفاده نامناسب از جویبار موازی ممکن است:
 - موجب کاهش کارایی شود
 - نتایج اشتباه به بار آورد
- در استفاده از parallel() برای جویبارها باید دقت کنیم

مثال: کاهش کارایی جویبار موازی

- کد زیر، مجموع اعداد یک تا n را به صورت موازی محاسبه می‌کند
- اگر parallel را فراخوانی نکنیم:

```
Stream.iterate(1L, i -> i+1).  
    .limit(n)
```

```
.parallel()
```

```
.reduce((a,b)->a+b);
```

- چندین برابر سریع‌تر می‌شود!
- فکر می‌کنید چرا؟

- زیرا iterate ماهیتی متوالی دارد

- تقسیم دنباله به چند بخش موازی، هزینه‌بر و کند است

- متدهای موازی‌سازی مناسب نیست

- (iterate نیست parallel-friendly)

- دیدیم موازی‌سازی iterate پرهزینه است

- زیرا تقسیم (تجزیه) اعضای این دنباله به چند بخش سخت است

- هرگاه تجزیه‌پذیری سخت باشد، موازی‌سازی کند می‌شود

- چند مثال دیگر

تجزیه‌پذیری عالی	تجزیه‌پذیری خوب	تجزیه‌پذیری ضعیف
ArrayList	HashSet	LinkedList
IntStream.range	TreeSet	Stream.iterate

- کارایی کدام بیشتر است؟

- LongStream.range(1, n+1).parallel() ✓

- Stream.iterate(1L, i -> i+1).limit(n).parallel()

اشتباه در نتایج با موازی سازی

```
class Accumulator {
    long total = 0;
    public void add(long value) {
        total += value;
    }
} long sideEffectParallelSum(long n) {
    Accumulator accumulator = new Accumulator();
    LongStream.rangeClosed(1, n)
        .parallel().forEach(accumulator::add);
    return accumulator.total;
}
```

```
sideEffectParallelSum(200_000)
```

نتیجه باید 20_100_000 باشد

ولی نتیجه عدد دیگری (مثل 159_159_899_923_19_899) خواهد بود



با خاطر دسترسی همزمان تردهای مختلف به متغیر total باعث این مشکل میشود.

```
5 public class Exercise3 {
6  public static void main(String[] args) {
7      IntStream.rangeClosed(1, 4_000)
8          .filter(a->a%1000==0)
9          .parallel()
10         .forEach(System.out::println);
```

The screenshot shows an IDE interface with the following details:

- Toolbar: Javadoc, Search, Console.
- Status Bar: <terminated> Exercise3 [Java Application] D:\java\jre8\bin\javaw.exe (May 14, 2015, 12:39:17 PM)
- Output Window:

```
3000
4000
2000
1000
```

Multi-threading programming

، shared data میخواهد روی یک concurrent (concurrent) داریم که بصورت multi-threads میخواهد روی یک access داشته باشد.

اجرای نوبتی th ها: Concurrent

Simultaneous: به OS مربوط میشود

وقتی os، multi-processing است، در app ما multi threads میتوانند بصورت concurrent اجرا شوند (یه ذره 1 یه ذره 2). (شاید در سطح os وابسته به تعداد cpu ها، parallel هم اجرا شوند (simultaneous)، این از دید app پنهان است).

(involves shared-data) critical area: دسترسی بهش توسط چند Concurrent th باید مدیریت شود تا (whole a Class, a method, a block of code) thread safety باشد.

مثالاً متدهایی که برای خواندن و نوشتمن یک فایل مشترک هستند. یا متدهای واریز و برداشت یک کلاس ممکن است روی یک obj موجودی حساب کار کند.

Synchronized کردن: تدابیری برای اجرای th ها بصورت concurrent. (نوبتی کردن دسترسی th ها به CA ها) thread safety: اجرای th بدون تناقض با اجرای th ها دیگر.

کلاس های immu یا یک DS thread safe هستند، یعنی th ها میتوانند بدون مشکل با instance های آنها بعنوان shared data کار کنند.

atomic commands: دستور یا دستورات که، که توسط یک thread بصورت یکجا در یک نوبت روی cpu اجرا شود (اگر در یک دستور انجام شود که ایده آل است)

در Java های app ، (main, GC, ...) thread default برنامه بیش از یک دارد (Thread or Runnable) بصورت concurrent اجرا میشوند، برای تولید بیشتر thread باید کد بزنیم (Thread or Runnable) وقتی app جاوای روی application server قرار میگیرد؛ thread های بیشتری هم تولید میشوند که توسط shared data هندل میشود و concurrent بصورت multi-threads اجرا میشوند. اگر synchronized میشوند. بین آنها ممکن است ایجاد شود باید synchronized شوند.

های thread را در برنامه اش میتواند مدیریت هم بکند (join, sleep, wait, notify, ...). بر اساس priority میتوانیم به thread ها تقدیم اجرایی بدهیم.

(در بردارنده ای آجکتها) بین thread ها مشترک است، ولی هر stack خودش را دارد.

هایی که که shared data ندارند، critical area نداریم. multi-thread ها بدون نیاز به همگام سازی thread هایی که که thread safety (concurrent) ، آن code ها را اجرا میشوند و synchronized (کردن) هستند، یعنی بصورت

shared resource باشد، چون در Heap نیست بین thread های مختلف دیده نمیشود. در تناقض با مثال درس

ولی اگر shared-data داشته باشیم، چی؟ ممکن است یک shared resource داشته باشیم مثل یک file یا DB باشد؛ و یا دو thread بخواهند یک instance object state از کلاسی را تغییر دهند، مثلاً آجکت موجودی حساب بانکی یک فرد، توسط دو thread متفاوت در یک زمان بخواهد با واریز یا برداشت state تغییر کند. (در هر دو حالت، چه زمانی که thread ها توسط app server روی چند session ایجاد شده باشند چه زمانی که توسط خود developer ایجاد میشوند).

راهها synchronized را در دسترسی concurrent thread های shared Resource به

atomic Classes -A

اگر دستوراتی که روی shared resource توسط هر thread اجرا میشود، ذاتا atomic commands باشند. (دستور یا دستورات کد، که توسط یک thread بصورت یکجا در یک نوبت روی cpu اجرا شود (اگر در یک دستور انجام شود که ایده آل است))، وقتی یک thread، آن atomic commands را روی یک shared obj اجرا میکند، مطمین هستیم در میانه ای آن، thread دیگری cpu را نمیگیرد تا تغییر resource نصفه کاره بماند. (cpu هم باید بتواند پشتیبانی کند).

برای این منظور در جاوا Atomic Class ها ایجاد شده اند، و تضمین شده است (اگر cpu پشتیبانی کند) برخی متدهای آن کلاس، روی obj instance هایش، بصورت atomic اجرا شوند، پس آن متدها هستند.

AtomicInteger, AtomicLong, AtomicBoolean, and AtomicReference

عنی اگر shared data را که باید `shared` روش کار کنند رو از این کلاس بسازیم، و بواسطه متدهای خودش باهاش کار کنیم، `thread` ها میتوانن `Concurrent` و `safe` باهم کار کنند.

در داخل این کلاس ها؛ از مکانیزم `mutex` (`synch, lock`) استفاده نشده، کار با `HW` را طوری انجام میدهدن که `atomic` باشد. (بخاطر همین `cpu` باید بتواند پشتبانی کند)

این کلاس ها `mutable` هستند (پس برای بعضی نیازمندی ها از primitive Wrapper ها بهتر هستند،
increase دارد)

`Integer`: Immutable/ Thread-safe

`AtomicInteger`: Atomic/ mutable/ Thread-safe

Volatile???

Immutable Classes -B

اگر `shared resource` یک `immutable class` instance object باشد، قابل `get` شدن در `thread` های مختلف هستند، پس بدون نگرانی میتوانیم از آنها بعنوان `readable shared resource` استفاده کنیم. ولی غیرقابل تغییر هستند (String, primitive Wrapper Classes, own immu. class)

(synchronizing Mutex/ Mutual Exclusion mechanism) -C

از سمت کد ما، یا کد لایبریری های جاوا، ایجاد مکانیزمی که هر لحظه فقط یک `thread` بتواند یکی از CA های مرتبط به یک `shared data` را اجرا کند.

The **synchronized keyword** is used to **provide thread safety** by ensuring that **only one thread can access a block of code at a time**.

این مکانیزم **دستورات یک `thread`** که الان نوبتش را `atomic` های `block` میکند، **thread** های دیگر `block` میشوند.
Th ها میخواهند دستوراتی که با `shared data` کار میکند را اجرا کنند)

■ مکانیزم `Mutex` توسط جاوا: درون برخی کلاس های آماده شده توسط جاوا انجام شده که بخوبی `thread` safety شده اند:

String: immutable, thread safety

StringBuffer: mutable, thread safety (Mutex)

StringBuilder: mutable

concurrent Collection: vector, I BlockingQueue

- ArrayBlockingQueue
- ConcurrentHashMap
- CopyOnWriteArrayList

Servlet (@service method)

inner synchronized (locks): (increasing BigO)

این کلاس ها thread safety ارایه شده اند. بعضی هاشون کل متدهاش `th safe` است، بعضی هاشون تعدادیش.

Mutex مثل ConcurrentHashMap که در دخشنان کارهای لازم برای thread safe شدن کل کلاس با مکانیزم synch ساخت که در داخل خودش متدها را انجام شده است. میتوان یک shared obj با ConcurrentHashMap ساخت که در داخل خودش متدها را کرده. البته o Big o متدهای آن بالاتر میرود. در ConcurrentHashMap به نحوی دسترسی simultaneous را به بخش‌های مختلف از shared data فراهم میکند. ولی به هر بخش، دسترسی ها جداگانه synchronized شده است.

کل کلاس thread safe ، servlet thread safety نیست یعنی اگر shared data در آن قرار دهیم باید خودمان آنرا کنیم تا برنامه درست کار کند؛ ولی متدهای service آن بصورت thread safety ارایه شده است.

■ مکانیزم Mutex توسط developer انجام میشود:

مثلا میتوان یک shared obj با HashMap ساخت و با synch. Blocks دسترسی thread های مختلف به آن را هندل کرد.

Mutex کردن کدام: synchronized کردن (نوبتی کردن دسترسی thread ها به CA (ها

- تعیین `lock` بعنوان `shared data`
- تعیین قطعات کد بعنوان CA با کلمه کلیدی `synchronized`
- بر اساس ، `race condition`

را به یک thread میدهد و اجازه میدهد وارد یک CA شود
از ورود thread های دیگر به همه CA ها جلوگیری میکند (block میکند) ؛ تا زمانی که lock آزاد شود.

شرایطی که Race condition برای تعیین نوع lock تعیین میکند، مشخص میشود کدام thread میتواند Lock را بگیرد و وارد CA شود. (به کمک شبی کلاس Monitor)

- حالتی که lock و Shared data یک instance object است.

For each mutable state variable (instance object) that may be accessed by more than one thread, all accesses to that variable must be performed with the same lock held.

مثل در کلاس instance of balance ؛ bank account (موجودی حساب بانکی آقای بحیرابی) یک lock است.
متدهای withdraw و deposit میشوند چون ممکن است state برای CA میتواند توسط دو thread همزمان تغییر کند.

پس هر thread یک lock instance object of property میشود که وقتی یک آنرا میگیرد، (lock of this) میتواند وارد یکی از CA ها شود، و بقیه thread ها اگر میخواهند روی آن obj به CA ای دسترسی پیدا کنند تا زمان آزاد شدن آن lock (یعنی احتمالاً اتمام کار یا مهلت) دارنده ای lock در CA ای که وارد شده باید صبر کنند (block شوند)

- حالتی که یک static property Shared data یک Class های instance object است، lock همه CA های static property است.

وقتی یک static property را میکنیم بین همه object های class مشترک است. اگر قابل تغییر باشد، آن هم static خواهد بود، این static property قابلیت یک shared resource بین thread های همزمان روی همه obj ای از کلاس را دارد. پس متدها یا خطوطی که میخواهند روی آن دسترسی داشته باشند باید synch شوند. در این حالت متدهای static setter CA synchronized را تعریف میکنیم و هر object با هر thread باشد وارد CA ای شود، دیگر هیچ روشی برای obj دیگری از آن Class نمیتواند وارد CA ای شود و block میشود.

در static property ما یک obj برای کلاسی تعریف میکنیم و فقط یک obj از آن کلاس میسازیم و به همه thread ها همان یک obj را بدهیم، پس آن property obj بین همه thread ها است و خطوط کد مرتبط به آن باید CA شوند.؟؟؟ مطالعه بیشتر

- حالتی که Shared data یا lock یک object که در دسترس چندین thread ها میتوانند روی obj instance های متفاوتی ساخته شوند، مثل یک file یا MapData.

ها را با synchronized block CA با ذکر نام lock که همان shared data Object است مشخص میکنیم. وقتی یک lock CA میشود، thread آن object را میگیرد و دیگر هیچ thread ای نمیتواند وارد CA ای بشود که آن این object است.

قسمت های **** رو از توی عکسها بخون.

Join ربطی به shared data res نداره، ولی wait و notify روی یک thread هستند ساخته میشوند باهم در ارتباطند

When we have simultaneous access by multi-threads to a Critical area (includes shared obj):

We should manage:

Mutex and Race (synchronizing in java (defining CA and lock, blocking))

+ (without error and excep. (time out))

→ thread safety CA (but is it without deadlock & starvation too?)

- این امکانات در بسته java.util.concurrent قرار دارند
- معمولاً کارایی بهتری به نسبت امکانات قدیمی دارند
- از پردازنده های چند هسته ای امروزی به خوبی بهره میبرند
- در بسیاری از کاربردها، برنامه نویسی را ساده تر می کنند
- برای کاربردهای متنوع، امکانات متفاوت و متنوعی ایجاد شده است

مکانیزم mutex را پیاده سازی میکنیم ولی ناکارامد:

Bad thread safety:

- Increasing Big O

- Deadlock:

هر یک **thread** shared resource **lock** را گرفته و منتظر دیگری است که در دست دیگری است، که آن **thread** هم منتظر shared resource **lock** ای است که در اختیار **thread** قبلى است. مثلا فرض کن دو نفر بخواهند به حساب بانکی هم واریز کنند.

به تدریج Thread ها توسط task ها پر میشوند و memory پر میشود و error و افتادن برنامه

- Starvation

چند **thread** همیشه **wait** خواهند بود چون منتظر هم هستند (تا قفلی را به هم بدنهند یا تمام شوند).

بعضی **thread** ها هیچ وقت نوبت بهشون نرسد و همیشه **wait** بمانند.

احتمال رخدادش کمتر از deadlock هست، این توسط الگوریتم هایی در os های جدید و application server ها هندل میشود. اگر **developer** هم thread های خجالتی و بی پروا در کنار هم درست کند هم باعثش میشود.

- Livelock ???

ابزارهای کمکی ایجاد (java 5: java.util.concurrent) :**thread safety**

Interface Callable: (instead for I Runnable)

It has returned a Generic value for call ()

Interface Executor: (helps in creating and terminating threads)

we need one Obj of it, and it has **thread pool**, executer instance obj will manage and allocate threads to tasks.

های application server از این استفاده میکنند.

The Executor represents a thread pool that can be shared between Connectors in Tomcat.

Synchronizers objects:

better of wait and notify: waiting and runabling threads/ controlling shared res.

semaphore, CountDownLatch, Exchanger, phaser,

Interface locks

با استفاده از lock (shared obj) گرفته و آزاده میشود، ولی با کمک wait, notify)synch. به صورت گرفته و آزاد میشوند.

Class Thread Local: ساخت متغیر های محلی برای یک ترد

volatile

متغیر های volatile میتوانند هم بر روی داده های ابتدایی (**primitive**) و هم بر روی اشیاء (**objects**) استفاده شوند.

البته، در صورت استفاده از volatile بر روی اشیاء، باید مطمئن شوید که عملیات خواندن و نوشتمن به آن اشیاء از سوی نخ ها درست همگام سازی شده است، زیرا volatile متغیرها بر روی خود متغیر اثرگذار است و عملیات داخلی اشیاء را نمیتوان کنترل کرد

این ویژگی به صورت معمول برای متغیر هایی استفاده میشود که به طور همزمان توسط چندین نخ خوانده میشوند و تغییر میکنند.

به JVM می‌گوید که هرگز از مقدار کش شده (**cached**) متغیر استفاده نکند و همیشه مقدار جدید را مستقیم از حافظه بخواند.

این ویژگی تضمین می‌کند که هرگاه یک نخ مقدار متغیر volatile را تغییر دهد، این تغییر فوراً در حافظه نگهداری می‌شود و تمامی نخ‌های دیگر آخرین مقدار آن را ببینند.

استفاده از متغیرهای volatile در برنامه‌های چندنخی می‌تواند به مشکلات زیر منجر شود:

- **کارایی کاهش یافته:** استفاده از متغیرهای volatile ممکن است باعث کاهش عملکرد برنامه شود. زیرا هر بار که متغیر volatile خوانده یا نوشته می‌شود، دسترسی به حافظه اصلی و بروزرسانی‌های مربوطه انجام می‌شود.
- **عدم حفاظت از ترتیب عملیات:** استفاده از متغیرهای volatile تنتها تضمین می‌کند که همه نخ‌ها آخرین مقدار متغیر را ببینند، اما این ویژگی ترتیب انجام عملیات‌ها را حفظ نمی‌کند.
- نمی‌تواند جایگزین همه مشکلات همگامسازی باشد: استفاده از volatile عموماً برای رفع مشکلات ساده همگامسازی مورد استفاده قرار می‌گیرد. اگر برنامه شما درگیر مشکلات پیچیده‌تری از جمله ترتیب عملیات‌ها و تراکم پیکربندی‌ها باشد، volatile نمی‌تواند به تنها یی را حل مناسبی باشد.
- کارایی در برخی معماهای سخت افزاری را کاهش میدهد: استفاده از متغیرهای volatile در برخی معماهای مانند سیستم‌های با حافظه پردازنده مشترک (Shared-memory systems) ممکن است عملکرد سیستم را کاهش دهد.

در پروژه‌ی tracking price، فقط یک object می‌خواهم از کلاس MapModelTrackingPrice داشته باشم که فقط یک producer object هم خواهم داشت و این در اختیار همه ی thread‌ها (درخواست‌ها) و spring framework (scope: default customer) قرار خواهد گرفت، چطور؟ یا با singleton یا با . singleton

اگر singleton نکنیم، یعنی object‌های متفاوتی از MapModelTrackingPrice ساخته شود، نوشتن synchronized block با قفل mapData کار بی فایده ای خواهد بود، چون هر thread روی obj متغیری (یعنی shared lock متفاوتی) می‌خواهد synchronized block را اجرا کند و در واقع بدون shared lock ای انجام میدهد چون data بین کل thread‌ها مشترک نیست.

یک راه حل خوب برای TrackingPrice

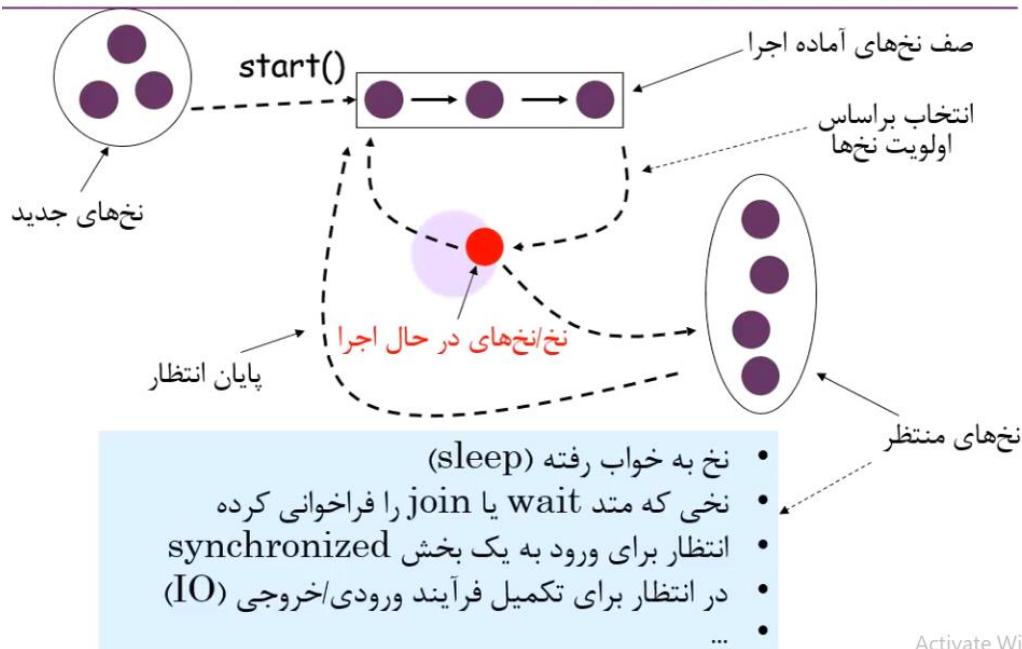
هر بار که یک `thread` خواست روی یکی از `obj` های مثلاً یکی از `LinkedHashMap` های یک `instrument` ، `read` یا `write` کند، میپرسد که آیا `thread` فعال (عمل مقابل، چون `w` که `CA` است و ممکن نیست، ۲ همزمان هم مشکلی ندارد) وجود دارد روی این `obj`? اگر وجود داشت، `join` میزنند تا آن `thread` فعال کارش تمام شود، بعد بتوانند کارش را انجام بدهد.

(با `wait` و `join` راهی به پیدا نکردم چون این دو در `CA` میتوانند استفاده شوند، یعنی کلید را دارند و کسی دیگری ندارد که بخواهد عمل همزمانی انجام دهد. `Wait` تازه میخواهد `lock` را آزاد کند. این دو به درد این نمیخورند که چک کنیم آیا `thread` فعالی روی این `obj` وجود دارد یا نه، چون `lock` دست خودشه و مسلماً `thread` فعال دیگری وجود ندارد. ولی `join` برای استفاده ازش نیاز نیست در `CA` باشیم. پس میتواند همزمان `thread` فعال دیگری روی آن `obj` داشته باشد.)

جدا از این `write` کردن باید در `CA` قرار بگیرد.

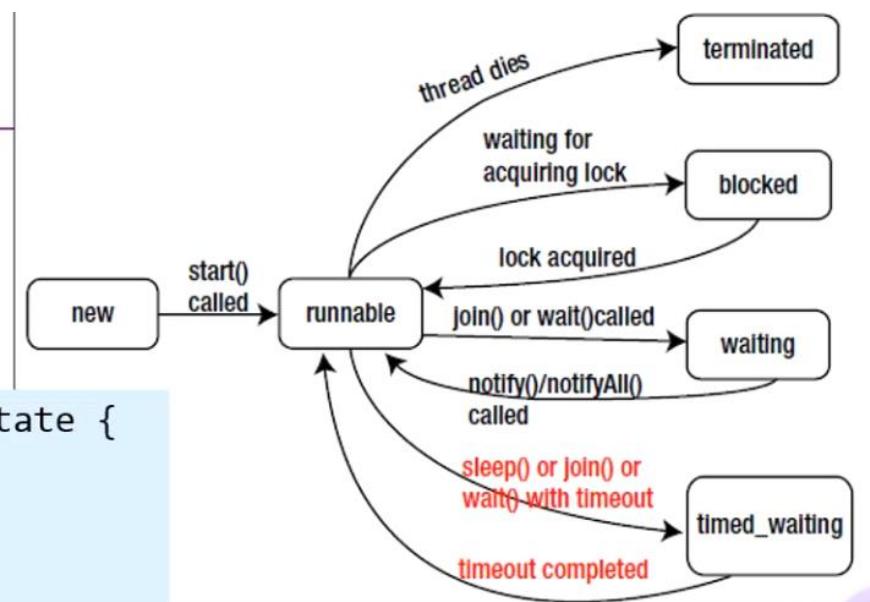
new Thread(...)

داستان زندگی یک نخ



حالت‌های نخ

```
public enum State {
    NEW,
    RUNNABLE,
    BLOCKED,
    WAITING,
    TIMED_WAITING,
    TERMINATED;
}
```



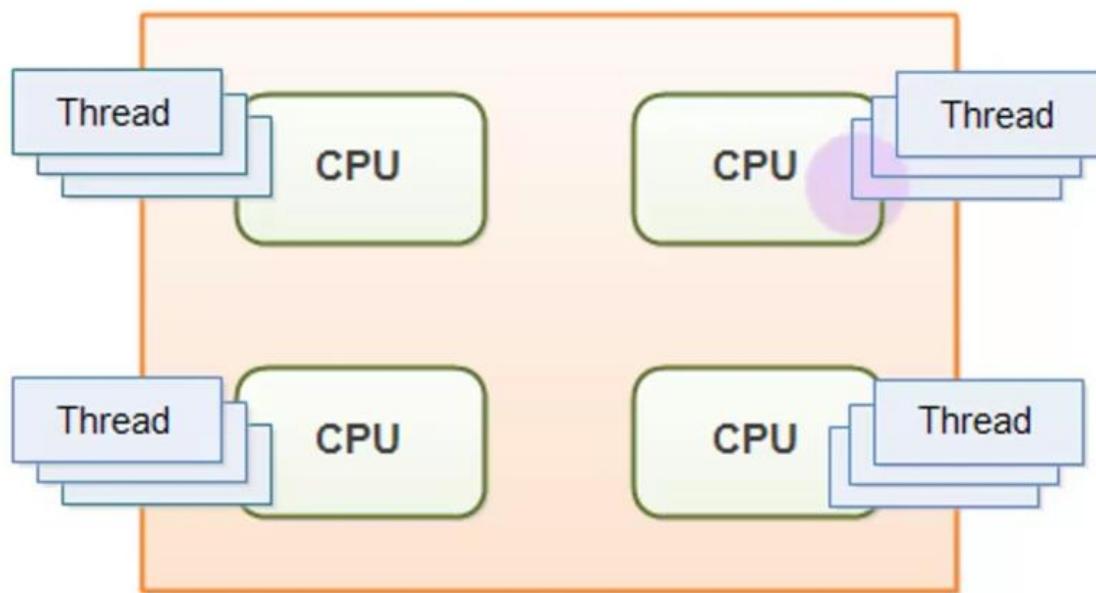
- متد getState() برای هر شیء از نوع Thread وضعیت آن نخ را برمی‌گرداند

هایی که runnable هستند، لزوماً در حال Run نیستند، این به قدرت و امکانات CPU وابسته است، از دید application ما قابلیت اجرا دارند.

چندپردازشی، چندنخی

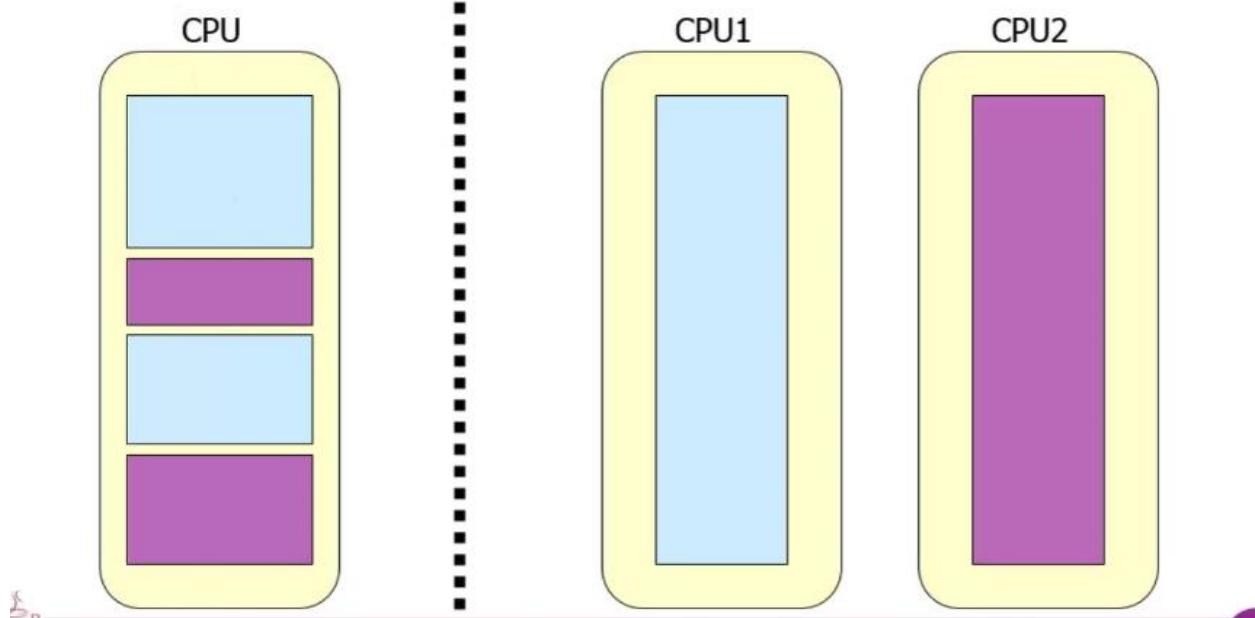
- مفهوم چند پردازشی (multi-processing) یا چندنخی (multi-tasking)
 - یعنی سیستم عامل بتواند چند «برنامه» را همزمان اجرا کند
 - سیستم عامل‌های مهم و معمولی این امکان را دارند (ویندوز، لینوکس و ...)
 - مثلاً در ویندوز همزمان با Eclipse می‌توانیم Chrome را هم اجرا کنیم
- مفهوم چندنخی (multi-thread)
 - یعنی یک برنامه بتواند چند بخش را به صورت همزمان اجرا کند
 - هر جریان اجرایی در یک برنامه: یک نخ اجرایی (thread of execution)
 - مثلاً همزمان با یک متده، متده دیگر را در اجرا داشته باشد
 - به همزمانی در اجرای چند بخش، همرونده (concurrency) می‌گویند
- مفهوم اجرای موازی (Parallel)
 - یعنی دو دستور واقعاً همزمان با هم در حال اجرا باشند
 - مثلاً همزمان که یک پردازنده (CPU) یک متده را اجرا می‌کند، یک پردازنده دیگر متده دیگر را اجرا کند
- اجرای همرونده (concurrency)
 - یعنی ظاهراً چند بخش همزمان با هم در حال اجرا باشند
 - چند بخش همزمان در حال پیشرفت هستند
 - ولی لزوماً به صورت موازی اجرا نمی‌شوند
 - شاید در هر لحظه، یکی از این کارها در حال اجرا باشد

اجرای موازی، اجرای همروند



همروند

همروند و موازی

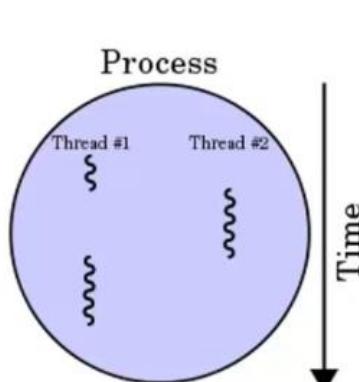


- همروندي در يك برنامه چه فوایدی دارد؟
 - افزایش کارایی
 - مثلاً اگر چند پردازنده و یا چند هسته پردازشی داشته باشیم
 - کامپیوترهای چندپردازنده‌ای و پردازنده‌های چندهسته‌ای (مثل i7)
 - پیشرفت همزمان چند کار (مثلاً ذخیره و پردازش اطلاعات)
 - برنامه‌های پاسخگو (همزمان با پردازش، تعامل با کاربر ممکن است)
 - حتی بدون امکان اجرای موازی، امکان همروندي برنامه مفید است

هر thread به هر شکلی ایجاد شود (توسط app server یا developer) ، یک نمونه از کلاس Thread برایش باید ساخته شود.

مفهوم نخ (Thread) در برنامه‌نویسی

- وقتی یک برنامه جاوا را اجرا می‌کنیم:
- یک نخ (thread) ایجاد می‌شود که متدهای main() را اجرا می‌کند
- برنامه می‌تواند نخ‌های جدیدی ایجاد کند و سپس آنها را اجرا کند



- نخ‌های مختلف به صورت همرونگ اجرا می‌شوند
- شاید به صورت موازی

ایجاد نخ

- دو راه اولیه برای تعریف رفتار یک نخ جدید در برنامه وجود دارد
 - در هر دو راه، کلاس جدیدی می‌سازیم
 - کلاس جدید زیرکلاس `java.lang.Thread` باشد
 - کلاس جدید واسط `java.lang.Runnable` را پیاده‌سازی کند
 - متده `run` را در کلاس جدید پیاده‌سازی می‌کنیم
 - این متده، دستورات نخ (`thread`) جدید را توصیف می‌کند

```

class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("Hello");
        System.out.println("Bye");
    }
}

public class ThreadExample{
    public static void main(String[] args) {
        System.out.println("Salam");
        MyThread t = new MyThread();
        t.start();
        System.out.println("Khodahafez");
    }
}
  
```

راه اول

خروجی محتمل:

Salam
Khodahafez
Hello
Bye

Salam
Hello
Khodahafez
Bye

- راه اول: ایجاد زیرکلاس `Thread`

- برای ایجاد نخ جدید: یک شیء از این کلاس بسازیم و متده `start` آن را فراخوانی کنیم
 - برنامه فوق دو نخ (جریان اجرایی همروند) دارد
 - یکی `Salam` و `Khodahafez` را چاپ می‌کند و دیگری `Hello` و `Bye`

راه دوم: پیاده‌سازی واسط Runnable

```
class MyRunnable implements Runnable{  
    @Override  
    public void run() {  
        System.out.println("Hello");  
        System.out.println("Bye");  
    }  
}
```

```
Thread t = new Thread(new MyRunnable());  
t.start();
```

- برای ایجاد نخ جدید: یک شیء (مثلاً با نام r) از این کلاس جدید بسازیم
- یک شیء از کلاس Thread بسازیم (مثلاً با نام t) و در سازنده آن r را پاس کنیم
- متده است start را روی t فراخوانی کنیم



- راه اول (زیرکلاس Thread) بهتر است یا راه دوم (پیاده‌سازی واسط Runnable)؟

- هرچند راه اول پیاده‌سازی ساده‌تری دارد

- در راه دوم دست طراح بازتر است تا کلاس موردنظر از کلاسی دلخواه ارثبری کند

- اگر کلاس ما زیرکلاس Thread باشد نمی‌تواند از کلاس دیگری ارثبری کند

- معمولًاً واسط Runnable پیاده‌سازی می‌شود

- چرا متده run را پیاده‌سازی می‌کنیم ولی متده start را فراخوانی می‌کنیم؟

- متده start یک متده خاص در کلاس Thread است

- که یک فرایند سطح پایین و سیستمی (ایجاد نخ جدید) را اجرا می‌کند

- و در نخ جدید، متده run را صدا می‌زند

- فراخوانی متده run فراخوانی تابعی معمولی است که به ایجاد نخ جدید منجر نمی‌شود



هایی که توسط Thread app server ساخته می‌شوند را خودش هندل می‌کند.

ما میتوانیم `Thread` هایی که میسازیم را هندل کنیم

متدهای Thread

- برای هر نخی که اجرا می‌شود، یک شیء از کلاس `Thread` ساخته شده است
- متدهای شیء `Thread` امکاناتی برای نخ متناظر ارائه می‌کنند
- متدهای کلاس `Thread`
- `run`, `start`, `getId`, `setPriority`, `setDaemon`, ...
 - متداستاتیک `currentThread` : نخ جاری را برمی‌گرداند
- متداستاتیک `sleep` : نخ جاری مدتی به خواب می‌رود
 - (اجرای آن به اندازه مشخصی متوقف می‌شود و سپس ادامه می‌یابد)
 - نحوه فراخوانی: `sleep(m, n)` یا `sleep(m)`
- اجرای این نخ به مدت `m` میلی ثانیه و `n` نانوثانیه متوقف می‌شود

**** ربطی به `lock` ندارد. متدهای از `Class Thread` است.

متدهای join

- گاهی لازم است کار یک نخ تمام شود، تا اجرای یک بخش از کد ادامه یابد
- مثلاً نخ «ارسال پیام» متوقف شود تا کار نخ «جستجوی ویروس» تمام شود
- یک نخ می‌تواند تا اتمام یک نخ دیگر منتظر بماند (موقتاً متوقف شود)
- این کار با کمک متدهای `join` انجام می‌شود

```
Thread virusScan = new VirusScanThread();
virusScan.start();
prepareEmail();
virusScan.join();
sendEmail();
```

نکته:

متدهای `join`, `sleep` و `InterruptedException` ممکن است خطای `IOException` پرتاب کنند

اولویت نخ

- اولویت (priority) یک نخ قابل تنظیم است
- اولویت نخ، با کمک متده است setPriority تغییر می‌کند
- اولویت یک عدد بین ۱ تا ۱۰ است که میزان اهمیت نخ را نشان می‌دهد
- سیستم عامل سعی می‌کند نخ‌های بالا اولویت بالا را بیشتر اجرا کند
- زمان بیشتری از CPU به نخ‌های بالا اولویت تخصیص می‌یابد

```
MyThread th = new MyThread();
th.setPriority(Thread.MAX_PRIORITY);
th.start();
```

```
MIN_PRIORITY = 1;
NORM_PRIORITY = 5;
MAX_PRIORITY = 10;
```

نخ‌های شبح (Daemon Threads)

- نوع خاصی از نخ‌ها هستند که در پس زمینه اجرا می‌شوند
- معمولاً خدماتی به سایر نخ‌ها ارائه می‌کنند و مستقلانه و به تنها بی معنا ندارند
- مثلاً زباله‌روب (garbage collector) یک daemon thread است
- از آنجا که اجرای مستقل و تنها آن‌ها بی معنی است:
- اگر فقط نخ‌های شبح در یک برنامه زنده باشند و نخ‌های معمولی پایان یافته باشند، JVM نخ‌های شبح را هم خاتمه می‌دهد و برنامه پایان می‌پذیرد
- با استفاده از متده است setDaemon() : نخ به صورت شبح یا معمولی تغییر می‌کند

```
MyThread th = new MyThread();
th.setDaemon(true);
th.start();
```

• مثال:

کوییز

```
class T extends Thread {  
    public void run() {  
        for (int i = 1; i <= 100; i++)  
            System.out.println(i);  
    }  
}  
class R implements Runnable{  
    public void run() {  
        for (char c = 'A'; c < 'Z'; c++)  
            System.out.println(c);  
    }  
}  
public class Threading{  
    public static void main(String[] args) {  
        new Thread(new R()).start();  
        new T().start();  
        new Thread(new R()).start();  
        new T().start();  
        for (char c = 'a'; c < 'z'; c++)  
            System.out.println(c);  
    }  
}
```

- این برنامه چند نخ دارد؟

پنج نخ و ۴ نخ همروند جدید (main)

- خروجی؟ چاپ موارد زیر:

دو بار از ۱ تا ۱۰۰

دو بار از A تا Z

یک بار از a تا z

- اما ترتیب چاپ قابل پیش‌بینی نیست

مثالاً شاید بعد از A عدد ۱ و سپس a چاپ شود

The screenshot shows an IDE interface with two panes. The left pane displays the Java code for `ThreadPractice.java`. The right pane shows the console output.

Code (ThreadPractice.java):

```
13     }  
14 }  
15 }  
16  
17 public class ThreadPractice {  
18     public static void main(String[] args) {  
19         new Thread(new Printer()).start();  
20         new Thread(new Printer()).start();  
21         for(char c='A';c<='Z';c++)  
22             try{  
23                 Thread.sleep(10);  
24             }catch(InterruptedException e){  
25                 e.printStackTrace();  
26             }  
27             System.out.println(c);  
28     }  
29 }  
30 }  
31 }
```

Console Output:

```
<terminated> ThreadPractice [Java Application] D:\java\jre8\bin\javaw.exe  
J  
9  
K  
10  
L  
11  
M  
12  
N  
13  
O  
14  
P
```

```
1 package ir.javacup.threa
2
3 class PrintThread implements Runnable {
4     @Override
5     public void run() {
6         for (int i = 0; i < 100; i++) {
7             try {
8                 Thread.sleep(10);
9             } catch (InterruptedException e) {
10                 e.printStackTrace();
11             }
12             System.out.println(i);
13         }
14     }
15     Thread currentThread = Thread.currentThread();
16     System.out.println(currentThread.getId());
17     System.out.println(currentThread.getName());
18 }
19 }
```

```
18 }
19 }
20
21 public class ThreadPractice {
22     public static void main(String[] args) {
23         new Thread(new PrintThread()).start();
24
25         for(char c='A';c<='Z';c++) {
26             try {
27                 Thread.sleep(10);
28             } catch (InterruptedException e) {
29                 e.printStackTrace();
30             }
31             System.out.println(c);
32         }
33         Thread currentThread = Thread.currentThread();
34         System.out.println("Main:"+currentThread.getId());
35         System.out.println("Main:"+currentThread.getName());
36 }
```

Debug

ThreadPractice [Java Application]

- ir.javacup.threading.ThreadPractice at localhost:61006
- Thread [main] (Suspended (breakpoint at line 31 in ThreadPractice))
 - ThreadPractice.main(String[]) line: 31
- Thread [Thread-0] (Suspended (breakpoint at line 13 in PrintThread))
 - PrintThread.run() line: 13
 - Thread.run() line not available

D:\java\jre8\bin\javaw.exe (Mar 13, 2016, 2:58:07 PM)

Variables Breakpoints Expressions

PrintThread [line: 13] - run()
ThreadPractice [line: 31] - main(String[])

Hit count: Suspend thread Suspend VM
Conditional Suspend when 'true' Suspend when value changes
<Choose a previously entered condition>

ThreadPractice.java Thread.class

```
29         e.printStackTrace();
30     }
31     System.out.println(c);
32 }
33 Thread currentThread = Thread.currentThread();
34 System.out.println("Main:"+currentThread.getId());
35 System.out.println("Main:"+currentThread.getName());
36
37 }
```

Debug

ThreadPractice [Java Application]

- ir.javacup.threading.ThreadPractice at localhost:61006
- Thread [main] (Suspended)
- Thread [Thread-0] (Suspended)
 - PrintThread.run() line: 9
 - Thread.run() line: not available

D:\java\jre8\bin\javaw.exe (Mar 13, 2016, 2:58:07 PM)

Variables Breakpoints

PrintThread [line: 13]
ThreadPractice [line]

Hit count: Suspend when 'true'
Conditional Suspend when value changes
<Choose a previously entered condition>

Debug

ThreadPractice [Java Application]

- ir.javacup.threading.ThreadPractice at localhost:61006
- Thread [main] (Suspended)
- Thread [Thread-0] (Suspended)
 - PrintThread.run() line: 9
 - Thread.run() line: not available

D:\java\jre8\bin\javaw.exe (Mar 13, 2016, 2:58:07 PM)

Variables Breakpoints

PrintThread [line: 13]
ThreadPractice [line]

Hit count: Suspend when 'true'
Conditional Suspend when value changes
<Choose a previously entered condition>

ThreadPractice.java Thread.class

```
4 @Override
5 public void run() {
6     for (int i = 0; i < 100; i++){
7         try{
8             Thread.sleep(10);
9         }catch(InterruptedException e){
10             e.printStackTrace();
11     }
12 }
```

- قبل‌اً دیده بودیم که:
- در حافظه هر برنامه، بخش‌هایی مثل پشته (stack) و Heap وجود دارد
- متغیرهای محلی در پشته و اشیاء در Heap نگهداری می‌شوند
- در واقع هر نخ، یک پشته مخصوص خودش دارد
- مثلاً اگر دو نخ مختلف، یک متده‌یکسان را فراخوانی کنند، هر نخ، حافظه مجزایی برای متغیرهای محلی آن متده است، در پشته خودشان خواهند داشت
- ولی همه نخها از حافظه Heap به‌طور مشترک استفاده می‌کنند
- دو نخ مختلف، می‌توانند از یک شیء مشترک استفاده کنند

بخش‌های بحرانی (Critical Section)

- دو نخ مختلف، می‌توانند همزمان از یک شیء مشترک استفاده کنند
- این وضعیت ممکن است مشکلاتی را ایجاد کند. مثال:
- همزمان که یک نخ در حال تغییر یک شیء است، یک نخ دیگر همان شیء را تغییر دهد
- در زمانی که یک نخ مشغول کار با یک فایل است، یک نخ دیگر آن فایل را ببندد
- بخش‌های بحرانی (critical section) :
- بخش‌هایی از برنامه که نمی‌خواهیم همزمان توسط چند نخ اجرا شوند
- اگر یک نخ وارد بخش بحرانی شد، نباید نخ دیگری وارد آن شود
- اجرای نخ دوم باید متوقف شود، تا زمانی که اجرای بخش بحرانی در نخ اول خاتمه یابد

چند اصطلاح

- منبع مشترک (shared resource)
- یک موجود (متغیر، شیء، فایل، دستگاه، ...) که هم زمان در چند نخ، مورد استفاده است
- شرایط مسابقه (race condition)
- شرایطی که در آن چند نخ، هم زمان به یک منبع مشترک دسترسی می‌یابند
- وحداقل یکی از نخها سعی در تغییر منبع مشترک دارد
- بخش بحرانی (critical section)
- بخشی از برنامه‌ی هر نخ، که در آن وارد شرایط مسابقه می‌شود
- انحصار متقابل (Mutex Mutual Exclusion)
- چند نخ نباید هم زمان بخش بحرانی را اجرا کنند
- با ورود یک نخ به بخش بحرانی، باید از ورود نخ‌های دیگر به بخش بحرانی جلوگیری شود

معنای synchronized

- هر شیئی در جاوا می‌تواند به عنوان مجوز ورود به بخش بحرانی استفاده شود
- هرگاه یک متدهynchronized روی یک شیء فراخوانی شود، قبل از ورود به این متده، سعی می‌کند قفل همان شیء را بگیرد
 - یعنی قفل this را بگیرد
 - به ازای هر شیء، یک قفل وجود دارد
- وقتی یک متدهynchronized در حال اجراست:
- هم زمان هیچ متدهynchronized دیگری روی همان شیء آغاز نمی‌شود
- چون تا پایان این متده، متده دیگری نمی‌تواند قفل this را بگیرد

- برنامهنویس باید بخش‌های بحرانی برنامه و شرایط ورود به آن‌ها را مشخص کند
- هر نخ، هنگام ورود به یک بخش بحرانی یک قفل (lock) را در اختیار می‌گیرد
- اگر همین قفل را قبلاً یک نخ دیگر گرفته باشد، نمی‌تواند وارد بخش بحرانی شود
- و تا زمان آزاد شدن قفل منتظر می‌ماند
- هنگام خروج از بخش بحرانی، قفلی که گرفته را آزاد می‌کند
- برنامهنویس مشخص می‌کند که برای ورود به هر بخش، چه قفلی لازم است

لطفاً از این مطلب پنهان نگیرید

```
public class BankAccount {
    private float balance;
    public synchronized void deposit(float amount) {
        balance += amount;
    }
    public synchronized void withdraw(float amount) {
        balance -= amount;
    }
}
```

مثال

- در این کلاس:

متدهای withdraw و deposit (واریز و برداشت) بخش‌های بحرانی هستند

- اگر یک نخ مشغول تغییر موجودی یک حساب (balance) است،

نباید یک نخ دیگر سعی در تغییر موجودی بدهد

- باید صبر کند تا کار نخ قبلی تمام شود

- بخش بحرانی با کلیدواژه **synchronized** مشخص می‌شود

سؤال: کدام گزینه‌ها صحیح هستند؟

```
public class BankAccount {  
    private float balance;  
    public synchronized void deposit(float amount) {  
        balance += amount;  
    }  
    public synchronized void withdraw(float amount) {  
        balance -= amount;  
    }  
}
```

۱- هیچ گاه دو نخ مختلف نمی‌توانند متدهای deposit را همزمان اجرا کنند

۲- اگر یک نخ در حال اجرای deposit نمی‌تواند اجرای withdraw را آغاز کند

۳- اگر یک نخ روی شیء X متدهای deposit را اجرا می‌کند،
نخ دیگری نمی‌تواند اجرای deposit روی همان شیء (X) را آغاز کند

۴- اگر یک نخ روی شیء X متدهای withdraw روی همان شیء (X) را آغاز کند
نخ دیگری نمی‌تواند اجرای withdraw روی همان شیء (X) را آغاز کند

• در موارد فوق منظور از همزمان، هموارند است (یعنی قبل از پایان یکی، دیگری شروع شود)

بلوک synchronized

- دیدیم یک متدهای synchronized باشد
- یعنی هر نخ باید قبل از ورود به متدهای synchronized قفل this را بدست آورد و در انتهای آزاد کند
- امکان ایجاد بخش بحرانی با کمک قفلی به جزء this هم وجود دارد
- این کار با ایجاد بلوک synchronized و ذکر یک شیء ممکن است

```
List<String> names; • مثال:  
...  
synchronized(names){  
    names.add("ali");  
}
```

blوک synchronized

- یعنی دو نخ مختلف به شرطی می‌توانند همزمان وارد این بلوک شوند
که شیء names در آن دو نخ متفاوت باشد

- این دو تعریف برای متد g تقریباً هم معنی هستند:

```
void g() {  
    synchronized(this) {  
        h();  
    }  
}
```

```
synchronized void g() {  
    h();  
}
```

- اگر یک متد استاتیک synchronized شود:

یعنی هر نخ برای ورود به متد باید قفل کلاس را بگیرد (به جای قفل یک شیء)

- یعنی هیچ دو نخی همزمان نمی‌توانند این متد را اجرا کنند

- یک متد غیراستاتیک synchronized را ممکن است دو نخ همزمان اجرا کنند،
به شرطی که روی دو شیء مختلف فراخوانی شوند

lock کار می‌کنند و از Class Object با **** هستند.

متدهای notify و wait

- گاهی لازم است دو نخ با هم تعامل داشته باشند

- گاهی یک نخ، صبر کند (wait) تا نخی دیگر به آن خبر دهد (notify)

- مثلاً فرض کنید دو نخ داریم: ۱- نمایش‌دهنده ویروس‌ها ۲- جستجوگر ویروس‌ها

- نخ اول متوقف می‌شود، هرگاه نخ دوم ویروسی پیدا کند، به نخ اول خبر می‌دهد هربار نخ اول باخبر می‌شود، به اجرا (نمایش ویروس) ادامه می‌دهد و سپس دوباره متوقف می‌شود

- متدهای wait و notify برای برقراری تعامل بین نخ‌ها استفاده می‌شوند

- این متدها در کلاس Object تعریف شده‌اند، final هستند

- از پیاده‌سازی سطح پایین (native) استفاده می‌کنند

- وقتی یک نخ متد wait را روی یک شیء دلخواه فراخوانی می‌کند، متوقف می‌شود تا این که یک نخ دیگر، روی همان شیء متد notify را فراخوانی کند

- روی هر شیء، تعدادی نخ wait کرده‌اند
- هر شیء، فهرستی از نخ‌های منتظر دارد
- با هر فراخوانی notify روی یک شیء، یکی از این نخ‌ها بیدار می‌شود
- یکی از نخ‌هایی که روی آن شیء منتظر هستند، اجرایش را ادامه می‌دهد
- متدهای notifyAll همه‌ی نخ‌های منتظر روی آن شیء را بیدار می‌کند
- نکته: متدهای wait می‌توانند حداقل مهلت انتظار را مشخص کنند
- مثلاً; wait(100) یعنی بعد از ۱۰۰ میلی‌ثانیه از انتظار خارج شود
(حتی اگر در این مدت، متدهای notify توسط نخ دیگری روی این شیء فراخوانی نشود)

- متدهای notify و wait فقط در صورتی روی شیء X قابل فراخوانی هستند که در یک بلوک synchronized(X) قرار گرفته باشد
- یک نخ برای فراخوانی wait یا notify روی یک شیء باید قفل آن شیء را گرفته باشد
- وگرنه خطای IllegalMonitorStateException پرتاب می‌شود
- البته با فراخوانی X.wait، بلافاصله قفل X آزاد می‌شود
- تا نخ‌های دیگر بتوانند وارد بلوک synchronized(X) شوند
- و X.notify را صدا بزنند تا این نخ از حالت انتظار (waiting) خارج شود

```
synchronized (obj) {
    obj.notify();
}
```

```
synchronized void f() {
    wait();
}
```

• مثال:

متدهای interrupt

- گاهی یک نخ منتظر است و اجرای آن متوقف شده است
- مثلاً به خاطر فراخوانی sleep یا join یا wait به حالت انتظار رفته است
- در این حالت اگر متدهای interrupt روی شیء این نخ فراخوانی شود:

 - نخ منتظر، از حالت انتظار خارج می‌شود
 - دریافت می‌کند InterruptedException و یک

- به همین دلیل است که متدهای wait و join و sleep این خطا را پرتاب می‌کنند

```

public class Interrupting extends Thread {
    public static void main(String[] a)
        throws InterruptedException{
        Interrupting t = new Interrupting();
        t.start();
        Thread.sleep(1000);
        t.interrupt();
    }
    @Override
    public synchronized void run() {
        try {
            wait();
            System.out.println("After wait");
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println("Resume");
    }
}

```

مثال

Interrupted
Resume

مثال

```
System.out.println("Main Starts.");
Scan scan = new Scan();
Object obj = scan.obj = new Object();
scan.start();
synchronized (obj) {    obj.wait();    }
System.out.println("Main other jobs");
```

نتیجه:

تا زمانی که Scan چاپ نشود، Main other jobs خواهد شد

```
class Scan extends Thread {
    public Object obj;
    public void run() {
        try { Thread.sleep(1000); } catch (InterruptedException e) {} ...
        System.out.println("Scan");
        synchronized (obj) {    obj.notify();    }
        System.out.println("Scan other jobs");
    }
}
```

سؤال

• تفاوت فراخوانی sleep و wait و join چیست؟

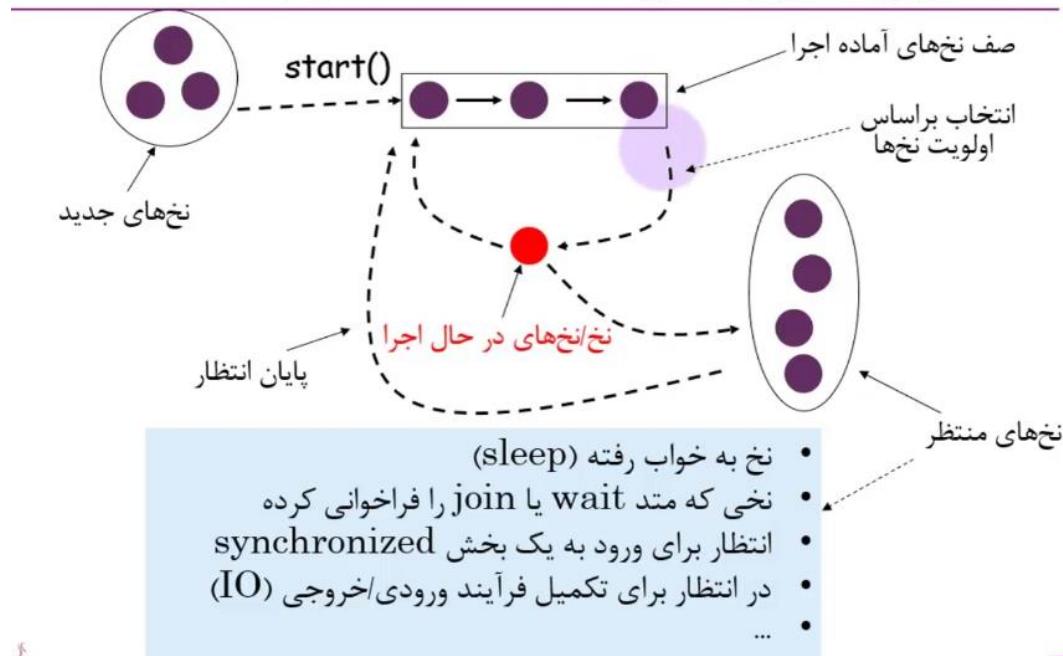
• پاسخ:

sleep : برای مدت مشخصی متوقف می‌شود و سپس به اجرا ادامه می‌دهد

wait : متوقف می‌شود تا یک نخ دیگر آن را باخبر (notify) کند

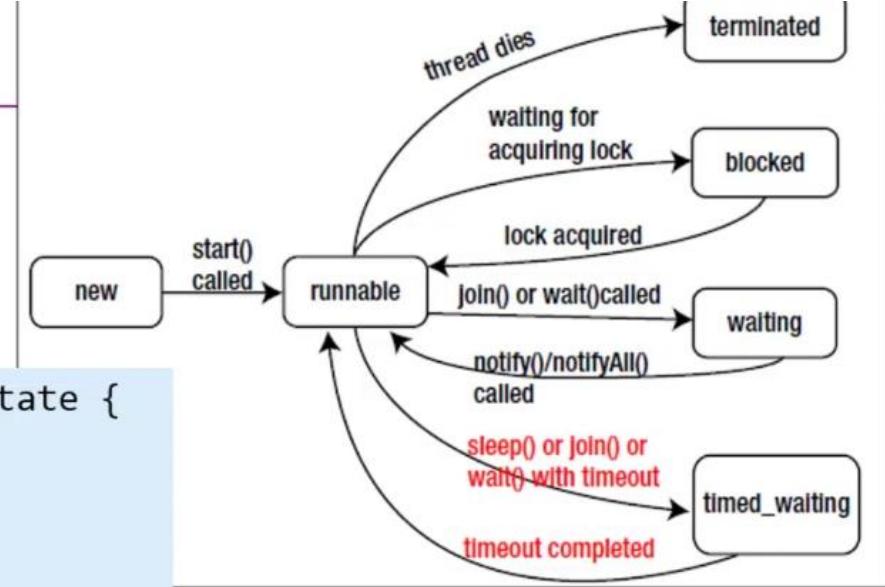
join : متوقف می‌شود تا یک نخ دیگر پایان یابد

داستان زندگی یک نخ



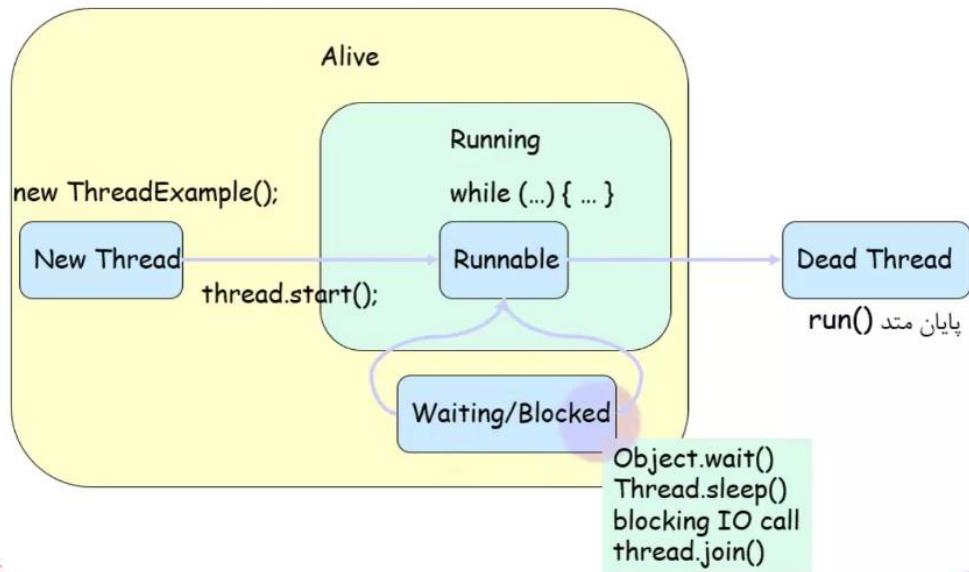
حالت‌های نخ

```
public enum State {
    NEW,
    RUNNABLE,
    BLOCKED,
    WAITING,
    TIMED_WAITING,
    TERMINATED;
}
```



- متد getState() برای هر شیء از نوع Thread وضعیت آن نخ را برمی‌گرداند

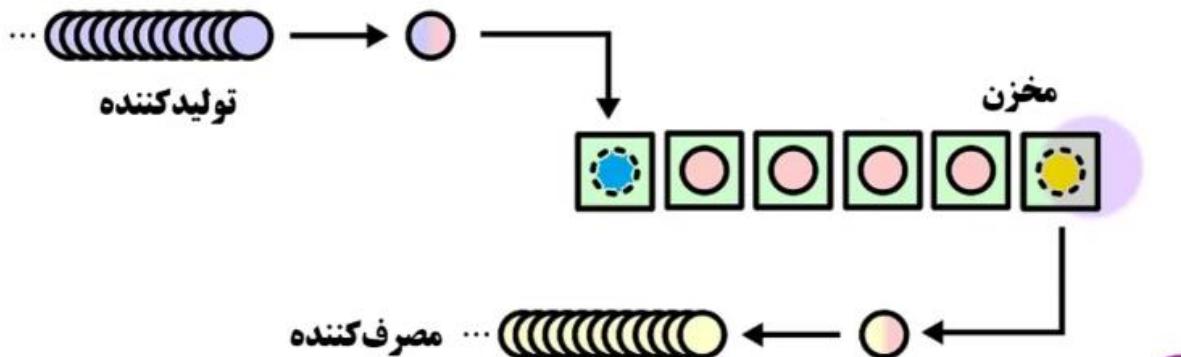
نگاهی دیگر به حالت‌های نخ



چند نکته درباره مسئله تولیدکننده/صرف‌کننده

- دو نخ مختلف همزمان نباید با مخزن کار کنند
- اگر یکی مشغول خواندن یا نوشتمن از مخزن است، نخ دیگری وارد نشود
- اگر مخزن خالی است، نخ مصرف‌کننده باید منتظر بماند تا یک تولیدکننده، داده تولید کند
- در صورتی که اندازه مخزن محدود است:

اگر مخزن پراست، نخ تولیدکننده باید منتظر بماند تا یک مصرف‌کننده، داده مصرف کند



```
class Producer extends Thread {  
    List<Integer> list;  
    Producer(List<Integer> l) {  
        list = l;  
    }  
    public void run() {  
        for (Integer i = 0; i < 100; i++) {  
            synchronized (list) {  
                Integer num = (int) (Math.random() * 1000);  
                System.out.println("Added:" + num);  
                list.add(num);  
                list.notify();  
            }  
            try {  
                Thread.sleep((long) (Math.random() * 10));  
            } catch (InterruptedException e) {}  
        }  
    }  
}
```

```
class Consumer extends Thread {  
    List<Integer> list;  
    Consumer(List<Integer> l) {  
        list = l;  
    }  
    public void run() {  
        for (Integer i = 0; i < 100; i++) {  
            synchronized (list) {  
                while (list.size() == 0)  
                    try {  
                        list.wait();  
                    } catch (InterruptedException e) {}  
                Integer fetch = list.remove(0);  
                System.out.println("Fetched:" + fetch);  
            }  
        }  
    }  
}
```

```
class ProducerConsumer1 {  
    public static void main(String args[])  
        throws InterruptedException {  
  
        List<Integer> list = new LinkedList<>();  
  
        Thread[] threads = {  
            new Producer(list),new Producer(list),  
            new Consumer(list),new Consumer(list) };  
  
        for (Thread thread : threads)  
            thread.start();  
  
        for (Thread thread : threads)  
            thread.join();  
  
        System.out.println("Finished:" + list.size());  
    }  
}
```

```
Added:867  
Fetched:867  
Added:461  
Fetched:461  
Added:385  
Fetched:385  
Added:495  
Fetched:495  
Added:558  
Fetched:558  
Added:961  
Fetched:961  
Added:591  
Added:27  
Added:797  
Added:188  
Fetched:591  
Fetched:27  
Fetched:797  
Fetched:188  
Added:233  
Added:902  
Fetched:233  
Fetched:902  
Added:955  
Added:556
```

- امکانات سیستم‌عامل‌ها برای برنامه‌نویسی چندنخی
- نحوه مدیریت و زمان‌بندی نخ‌ها توسط سیستم‌عامل
- مزایا و معایب برنامه‌نویسی چندنخی
- در چه مواردی، چندنخی باعث افت کارایی برنامه می‌شود

امکانات سطح بالا برای همروندی

- امکاناتی برای مدیریت دسترسی همزمان به اشیاء مشترک دیدیم:
 - synchronized, wait, notify, ...
 - این موارد، امکاناتی سطح پایین هستند
 - از نسخه ۵ جوا، برخی امکانات سطح بالا و جدید برای مدیریت همروندی اضافه شد
 - High-level concurrency APIs
 - این امکانات در بسته‌ی **java.util.concurrent** قرار دارند
 - معمولاً کارایی بهتری به نسبت امکانات قدیمی دارند
 - از پردازنده‌های چندهسته‌ای امروزی به خوبی بهره می‌برند
 - در بسیاری از کاربردها، برنامه‌نویسی را ساده‌تر می‌کنند
 - برای کاربردهای متنوع، امکانات متفاوت و متنوعی ایجاد شده است

Thread-safe مفهوم

- برخی از کلاس‌ها، thread-safe هستند: ایمن در همروندي
- استفاده از اشیاء این کلاس‌ها به صورت همرونده، ایمن است
- از اشیاء این کلاس‌ها می‌توانیم به طور مشترک در چند نخ استفاده کنیم
- برای استفاده از این اشیاء در چند نخ همزمان، نیازی به قفل یا synchronized نیست
- تمهیدات لازم در داخل همان کلاس پیاده‌سازی شده است
- مثال:

نایمن در همروندي	ایمن در همروندي (Thread-safe)
ArrayList	Vector
HashMap	ConcurrentHashMap
StringBuilder	StringBuffer

- کلاس‌های معمولی بهترند یا معادل thread-safe آن‌ها؟
 - مثلاً بهتر نیست همیشه به جای ArrayList از Vector استفاده کنیم؟
 - خیر
 - اگر نیاز به استفاده مشترک از یک شیء در چند نخ نداریم، کلاس‌های معمولی کارترند
 - تمهیداتی که برای thread-safety پیاده شده (مثل synchronized) اجرا را کنترل می‌کند
- اشیاء تغییرناپذیر (immutable) همواره thread-safe هستند
 - ویژگی‌های اشیاء تغییرناپذیر بعد از ساخت این اشیاء قابل تغییر نیست
 - مثلاً setter ندارند. مانند: Integer و String
 - امکان تغییر وضعیت آن‌ها وجود ندارد: استفاده از آن‌ها در چند نخ همزمان ایمن است

- کلاس‌های `StringBuilder` و `StringBuffer` هر دو برای نگهداری رشته هستند
- برخلاف کلاس `String` اشیاء این کلاس‌ها تغییرپذیر (`mutable`) هستند
- مثلاً برای اضافه کردن یک مقدار به انتهای رشته، متدهای `append` دارند

```
StringBuffer buffer = new StringBuffer("12");
buffer.append("345");
String s = buffer.toString();    12345
```

- با کلاس `StringBuilder` هم دقیقاً به همین شکل می‌توان کار کرد

- اما متدهای تغییردهنده در `StringBuffer` به صورت `synchronized` هستند

```
public synchronized StringBuffer append(String str) {...}
```

- `StringBuilder` یک کلاس `thread-safe` است، ولی `StringBuffer` نیست



عملیات اتمیک (Atomic)

- گاهی یک عملیات ظاهرآ ساده ممکن است به چند دستور سطح پایین ترجمه شود
- مثلاً `i++` به `i=i+1` ترجمه شود (خواندن `i`، عملیات جمع و تغییر مقدار `i`)
- نباید در میانه اجرای این عملیات، نخ دیگری از این متغیر استفاده کند
- گاهی چند عملیات ظاهرآ مستقل، به یک دستور سطح پایین قابل ترجمه هستند
- **عملیات اتمیک:** همه عملیات، یکجا اجرا شود (در میان اجرای آن، نخ دیگری وارد نشود)
- کل عملیات به صورت یک دستور سطح پایین اجرا شود (در صورت پشتیبانی پردازنده)
- یا با گرفتن قفل پیاده‌سازی شود: برای نخهای دیگر به توقف (`blocking`) منجر شود
- گاهی برای انواع ساده (مثل عدد و آرایه) به عملیات اتمیک نیاز داریم
- مانند افزایش یک متغیر عددی، و یا «خواندن و تغییر» مقدار یک متغیر



کلاس‌های اتمیک (Atomic Class)

- برخی عملیات بر روی اشیاء این کلاس‌ها به صورت اتمیک ممکن شده است
- اجرای متغیر اتمیک کاراتر از پیاده‌سازی با قفل و synchronized و ... خواهد بود
- کلاس‌های اتمیک جاوا در بسته‌ی `java.util.concurrent.atomic` هستند
- مانند `AtomicLongArray` ، `AtomicLong` ، `AtomicInteger` و ...
- متغیرهای کلاس‌های اتمیک
- به صورت امن در چند نخ قابل اشتراک هستند `thread-safe`
- برای استفاده در چند نخ نیازی به قفل و synchronized و ... ندارند `lock-free`

```
AtomicInteger()
AtomicInteger(int initialValue)
int get()
void set(int newVal)
int getAndSet(int newValue)
int getAndIncrement()
int getAndDecrement()
boolean compareAndSet (int expect, int update)
```

```
AtomicInteger at = new AtomicInteger(12);
int a_12 = at.get();
at.set(20); //at=20
int a_21 = at.incrementAndGet(); //at=21
int b_21 = at.getAndIncrement(); //at=22
int a_27 = at.addAndGet(5); //at=27
boolean is9_false = at.compareAndSet(9, 3); //at=27
boolean is27_true = at.compareAndSet(27, 30); //at=30
```

AtomicInteger : مثال

- برخی متدهای این کلاس:

• مثال:

- متغیرهای اتمیک، `thread-safe` هستند

- بدون نیاز به قفل و synchronized و ... از آن‌ها در چند نخ استفاده می‌کنیم

- از بین کلاس‌های زیر،
- ١- چه کلاس‌هایی تغییرناپذیر (Immutable) هستند؟
- ٢- چه کلاس‌هایی thread-safe هستند؟
- ٣- اشیاء چه کلاس‌هایی را بدون نیاز به قفل و synchronized می‌توانیم بین چند نخ به اشتراک بگذاریم؟

- String ١ ٣و٢
- Integer ١ ٣و٢
- AtomicLong ٣و٢
- ArrayList
- HashMap
- ConcurrentHashMap ٣و٢

نکته:
 - سوال ٢ و ٣ یکی هستند
 - هر چه immutable باشد، thread-safe هم هست
 (ونه لزوماً برعکس)

ظرف‌های همروند (Concurrent Collections)

- راهی ساده برای این‌که یک کلاس thread-safe شود:
- همه متدها را synchronized کنیم! (اما این راه کارایی مناسبی ندارد)
- از نسخه ۵ (JDK 1.5) بسته java.util.concurrent به جاوا اضافه شد
- این بسته شامل کلاس‌های جدید همروند است
- این کلاس‌ها، نه تنها thread-safe هستند، بلکه کارایی مناسبی در برنامه‌های همروند دارند
- قفل‌ها به صورت بهینه گرفته و آزاد می‌شوند

ConcurrentHashMap مثلاً یک map است که به اشتراک گذاشتن اشیاء آن بین چند نخ، امن است

- مانند:
- ArrayBlockingQueue
- ConcurrentHashMap
- CopyOnWriteArrayList

مثال: واسط BlockingQueue

- یکی از زیرواسطهای Queue که thread-safe است
- معرفی متدهای put برای اضافه کردن و متدهای take برای حذف از صف
- هنگام استفاده از اشیائی از این نوع، در صورت لزوم هنگام خواندن و نوشتمن، نخ در حال اجرا معطل می‌شود
- اگر صف خالی باشد، هنگام خواندن متوقف می‌شود تا عضوی به صف اضافه شود
- اگر ظرفیت صف پر باشد، هنگام نوشتمن متوقف می‌شود تا عضوی از صف خارج شود
- مشابه مفهوم تولیدکننده/صرفکننده (producer/consumer)
- **ArrayBlockingQueue**: پیاده سازی این واسط مبتنی بر آرایه با طول ثابت
- **LinkedBlockingQueue**: پیاده سازی مبتنی بر لیست پیوندی

اشیاء هماهنگ کننده (Synchronizer)

- یک شیء هماهنگ کننده (synchronizer) برای ایجاد هماهنگی بین چند نخ استفاده می‌شود
 - چنین شیئی، یک وضعیت (حالت درونی) دارد و با توجه به این وضعیت، به نخهای همکار، اجازه اجرا یا توقف می‌دهد
 - کلاس‌های متفاوتی برای کاربردهای مختلف ایجاد شده است. مانند:
 - Semaphore
 - CountDownLatch
 - Exchanger
 - CyclicBarrier
- در این زمینه، قبلاً امکانات سطح پایین‌تری مانند *wait* و *notify* را دیده بودیم

سمافور (Semaphore)

- دسترسی به منابع مشترک را کنترل می‌کند
- یک عدد برای تعیین تعداد «استفاده‌کننده‌های همزمان» نگهداری می‌کند
- این عدد حالت سمافور را مشخص می‌کند
(حداکثر تعداد نخ‌هایی که همزمان از منبع مشترک استفاده می‌کنند)
- متدهای اصلی سمافور: `release()` و `acquire()`
- هر نخ قبل از استفاده از شیء مشترک، باید متدهای `acquire()` را فراخوانی کند
 - اگر حالت سمافور صفر باشد، این متدهای بلاک می‌شود (اجرای نخ متوقف می‌شود)
(تعداد نخ‌هایی که همزمان وارد شده‌اند، از عدد اولیه تعیین‌شده بیشتر شده است)
- هر نخ در پایان استفاده از شیء مشترک، باید متدهای `release()` را فراخوانی کند
 - موجب آزاد شدن یک نخ (که منتظر `acquire()` است) می‌شود

مثال: پیاده‌سازی تولید‌کننده/صرف‌کننده با سمافور

```
semaphore.acquire();
synchronized (list) {
    obj = list.remove(0);
}
```

صرف‌کننده

```
synchronized(list){
    list.add(obj);
}
semaphore.release();
```

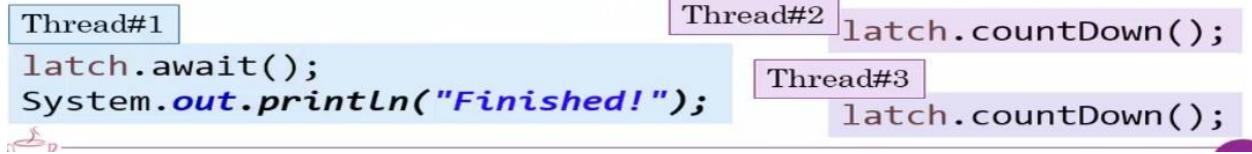
تولید‌کننده

- اشیاء `semaphore` و `list` بین چند نخ به اشتراک گذاشته می‌شود
- شیء سمافور مشترک (`sem`) به صورت زیر ایجاد شده است:
`Semaphore sem = new Semaphore(0);`
- دو دغدغه:
 - دو نخ، همزمان از لیست مشترک استفاده نکنند \leftarrow synchronized
 - اگر لیست خالی است، نخ مصرف‌کننده متوقف شود \leftarrow سمافور

CountDownLatch هماهنگ‌کننده

- یک هماهنگ‌کننده (Synchronizer) دیگر
- به چند نخ اجازه می‌دهد تا پایان یک شمارش معکوس متوقف شوند
- کاربرد: در نخ‌های مختلف تعداد مشخصی عملیات باید رخ دهنند، تا امکان ادامه برخی نخ‌ها فراهم شود
- متدهای اصلی این کلاس:
 - متод `await`: منتظر پایان شمارش معکوس می‌شود
 - متود `countDown`: شمارش معکوس را یک واحد پیش می‌برد

```
CountDownLatch latch = new CountDownLatch(2);
```



Lock واسط

- بسته‌ی `java.util.concurrent.locks` از جاوا ۵ اضافه شد
- شامل واسطه‌ها و کلاس‌های جدید برای گرفتن قفل در برنامه‌های همرون
- مثل واسطه‌های `Condition` و `ReadWriteLock` ، `Lock`
- با کمک `lock`: دسترسی همزمان چند نخ به یک بخش حیاتی را محدود می‌کنیم
 - تا در هر لحظه حداقل یکی از نخ‌ها در حال اجرای بخش حیاتی باشد
 - امکانی مشابه `synchronized` ، اما پیچیده‌تر و انعطاف‌پذیرتر
 - `lock` ← `Synchronized` قفل ضمنی و قفل صریح
 - هدف هر دو ساختار یکی است:
- در هر لحظه تنها یک نخ به منبع مشترک (بخش بحرانی) دسترسی دارد

اشیاء Lock

- با کمک شیء Lock ، محل گرفتن و آزادسازی قفل توسط برنامهنویس مشخص می شود
• متدهای مهم واسط Lock :

- (synchronized) متدهای lock و unlock برای گرفتن و آزادسازی قفل (مثل بلوک Lock)
- متدهای lock قفل همان شیء Lock را می گیرد و unlock قفل را آزاد می کند
- متدهای lock مانند tryLock عمل می کند، ولی اگر قفل آزاد نبود، متوقف نمی شود

```
Lock l = new ReentrantLock();
l.lock();
try {
    ... // critical section
}finally {
    l.unlock();
}
```

مثال

non-blocking

• اگر نتواند قفل را بگیرد:
برمی گرداند false

- یکی از کلاس هایی که واسط Lock را پیاده سازی کرده است: ReentrantLock

واسط WriteLock

- یک بخش بحرانی از یک برنامه را در نظر بگیرید که در آن:
 - بسیاری از نخ ها متغیر مشترک را می خوانند
 - بعضی از نخ ها متغیر مشترک را تغییر می دهند
- اگر همروندي را با روش های معمولی مثل synchronized کنترل کنیم:
 - حتی اگر دو نخ بخواهند متغیر مشترک را بخوانند، یکی باید منتظر پایان دومی بماند
 - این وضعیت کارا نیست
- واسط ReadWriteLock دو قفل مجزا در نظر می گیرد:
یکی برای نویسنده ها و یکی برای خواننده ها!
- اجازه می دهد چند نخ که فقط می خواهند متغیر مشترک را بخوانند، همزمان اجرا شوند
- متدهای readLock برای قفل خواندن و writeLock برای نوشتن

مثال: کلاس ReentrantReadWriteLock

- واسط ReadWriteLock را پیاده‌سازی کرده است. مثال:

```
List<Double> list = new LinkedList<>();  
ReadWriteLock lock = new ReentrantReadWriteLock();
```

```
class Reader extends Thread{  
    public void run() {  
        lock.readLock().lock();  
        System.out.println(list.get(0));  
        lock.readLock().unlock();  
    }  
}
```

```
class Writer extends Thread{  
    public void run() {  
        lock.writeLock().lock();  
        list.add(0, Math.random());  
        lock.writeLock().unlock();  
    }  
}
```

انحصار، حافظه اکانت Java Cafe



مدیریت ایجاد نخ

- برای ایجاد یک نخ اجرایی جدید، یک راه دیدیم:
 - ایجاد شیء از Thread و فراخوانی متدهای start()

```
Thread t = new MyThread();  
t.start();
```

```
Thread t = new Thread(runnable);  
t.start();
```

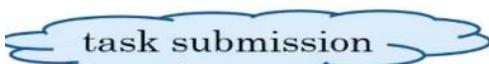
- ولی این راه معمولاً مناسب نیست!

برنامه‌های بزرگ معمولاً این‌گونه هستند:

- در بخشی از برنامه، نیاز به اجرای فعالیتی مشخص (task) تعیین می‌شود

- در بخشی مجزا از برنامه، نحوه ایجاد نخها، زمان‌بندی و ... مدیریت می‌شود

- برنامه‌نویسی که یک وظیفه را پیاده‌سازی می‌کند، نحوه مدیریت نخها را تعیین نمی‌کند



انحصار، حافظه اکانت Java Cafe

چارچوب‌های اجراگر نخ‌ها (Executors)

- فرایند ایجاد و اتمام یک نخ جدید پیچیده و پرهزینه است
- برای مدیریت این کار الگوریتم‌های مناسبی پیاده‌سازی شده است
- به اشیائی که ایجاد و اجرای نخ‌ها را مدیریت می‌کنند، اجراگر (Executor) می‌گویند

```
package java.util.concurrent;  
interface Executor {  
    void execute(Runnable command);  
}
```

:Executor واسط

- متد execute یک شیء Runnable را در یک نخ اجرا می‌کند

◦ شاید از یکی از نخ‌هایی که قبلاً ایجاد شده، استفاده کند

◦ شاید یک نخ جدید برای اجرای آن ایجاد کند

◦ شاید در همین نخ جاری اجرا کند

نحوه ایجاد و مدیریت نخ به نوع Executor بستگی دارد

خزانه نخ (Thread Pool)

- معمولاً به شیئی از نوع Executor برای مدیریت ایجاد و اجرای نخ‌ها نیاز داریم
- کلاس‌های مختلفی به عنوان Executor طراحی شده‌اند
- این اشیاء معمولاً یک خزانه نخ (thread pool) دارند
- خزانه نخ (thread pool) :
- تعداد m کار به طور همزمان در n نخ اجرا می‌شوند
- تعداد کارها ممکن است از تعداد نخ‌ها بیشتر شود
- از ایجاد و خاتمه مکرر نخ‌ها (که حافظه و زمان را مصرف می‌کند) جلوگیری می‌شود
- برای هر کار جدید، حتی‌الامکان یکی از نخ‌های بیکار thread pool به کار می‌رود
- الگویی برای کاهش تعداد نخ‌های ایجادشده در برنامه

کلاس کمکی Executors

- یک کلاس کمکی در بسته `java.util.concurrent` است.
- متدهای استاتیک ساده‌ای برای برگرداندن شیء از انواع `Executor` دارد. مثال:
 - متدهای `newSingleThreadExecutor`: خزانه‌ای با یک نخ کارها. پشت سر هم در همان یک نخ اجرا می‌شوند (برای آغاز کار ۱ باید تمام شود).
 - متدهای `newFixedThreadPool`: خزانه‌ای با تعداد مشخصی نخ کارها، در تعداد مشخصی نخ اجرا می‌شوند.
 - اگر تعداد کارها بیشتر از تعداد نخ‌ها باشد، اجرای برخی کارها بعد از اتمام کارهای قبلی متدهای `newCachedThreadPool`: برای هر کار جدید یک نخ ایجاد می‌کند.
 - با پایان کار یک نخ، آن را نگه می‌دارد و برای اجرای کارهای بعدی بازاستفاده می‌کند.

```
Executor e = Executors.newFixedThreadPool(2);
Runnable runnable = new Runnable(){
    public void run() {
        for (int i = 0; i < 4; i++)
            System.out.println(Thread.currentThread().getId()+":"+i);
    }
};
for (int i = 0; i < 3; i++)
e.execute(runnable);
```

شروع کار اول
شروع کار دوم
9:0
10:0
10:1
10:2
9:1
10:3
9:2
10:0
9:3
10:1
10:2
10:3

شروع کار سوم در یکی از نخ‌های قبلی



Java Code - Java Code

www.java.com/en

www.java.com/en

```

Executor e = Executors.newSingleThreadExecutor();
Runnable runnable = new Runnable(){
    public void run() {
        for (int i = 0; i < 4; i++)
            System.out.println(Thread.currentThread().getId()+"："+i);
    }
};
for (int i = 0; i < 3; i++)
    e.execute(runnable);

```

شروع کار اول ←
9:0
9:1
9:2
9:3
9:0 ← شروع کار دوم
9:1
9:2
9:3
9:0
9:1
9:2
9:3

شروع کار سوم →

 Java Code Editor      

```

Executor e = Executors.newCachedThreadPool();
Runnable runnable = new Runnable(){
    public void run() {
        for (int i = 0; i < 4; i++)
            System.out.println(Thread.currentThread().getId()+"："+i);
    }
};
for (int i = 0; i < 3; i++)
    e.execute(runnable);

Thread.sleep(1000);
e.execute(runnable);

```

مثال

شروع کار اول ←
9:0
10:0 ← شروع کار دوم ←
11:0 ← شروع کار سوم ←
9:1
10:1
10:2
9:2
9:3
11:1
10:3
11:2
11:3

شروع کار بعدی در یکی از نخهای قبلی →
10:0
10:1
10:2
10:3

 Java Code Editor      

واسط **java.util.concurrent.Callable**

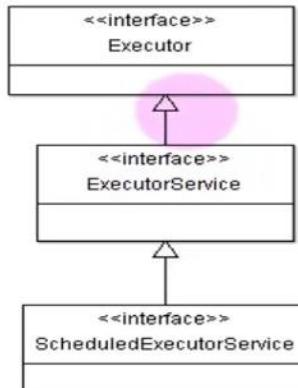
```
interface Callable<V> {  
    V call() throws Exception;  
}
```

- مشکل واسط :Runnable
- متدهای run مقدار برگشتی ندارد (void است)
- اما گاهی یک عملیات باید یک مقدار محاسبه یا تولید کند
- با کمک run این خروجی باید در یک منبع مشترک نوشته شود
- نخی که خروجی عملیات را لازم دارد، باید منتظر پایان آن نخ بماند
- همروندی و دسترسی مشترک نخها هم باید کنترل شود
- واسطی مشابه Callable که یک عملیات را توصیف می‌کند
- به جای متدهای run ، متدهای call دارد
- برخلاف run مقداری برمی‌گرداند و ممکن است خطا پرتاب کند

واسط **java.util.concurrent.Future**

- این واسط خروجی یک عملیات را نشان می‌دهد (که احتمالاً در نخی دیگر اجرا شده)
- متدهایی دارد برای:
 - بررسی این که آیا عملیات تمام شده است (isDone و isCancelled)
 - انتظار برای پایان عملیات و دریافت نتیجه عملیات (get)
 - قطع عملیات (cancel)
- بسیاری از کلاس‌های موجود Executor ، متدهای با نام submit دارند:

```
interface ExecutorService extends Executor {  
    <T> Future<T> submit(Callable<T> task);  
    ...  
}
```



مرواری بر واسطه های Executor

Executor واسطه •

دارای متدها execute(Runnable task) •

ExecutorService واسطه •

دارای متدها submit(Callable<T> task) •

ScheduledExecutorService واسطه •

دارای متدها schedule برای زمان بندی اجرای یک کار با تأخیر:

```
public <V> ScheduledFuture<V> schedule(
    Callable<V> callable, long delay, TimeUnit unit);
```

```

class WordLengthCallable implements Callable<Integer> {
    private String word;
    public WordLengthCallable(String word) {
        this.word = word;
    }
    public Integer call() {
        return word.length();
    }
}
ExecutorService pool = Executors.newCachedThreadPool();
Set<Future<Integer>> set = new HashSet<>();
String[] words = { "Ali", "Taghi", "Naghi" };
for (String word : words) {
    Callable<Integer> callable = new WordLengthCallable(word);
    Future<Integer> future = pool.submit(callable);
    set.add(future);
}
int sum = 0;
for (Future<Integer> future : set)
    sum += future.get();
System.out.println("The sum of Lengths is " + sum);
  
```

مثال

می خواهیم مجموع طول چند رشته را به صورت چند نخی محاسبه کنیم

ThreadLocal مفهوم

• کلاس ThreadLocal

- اشیائی که از این کلاس ایجاد می‌شوند، فقط داخل همان نخ قابل استفاده خواهد بود
- حتی اگر دو نخ، یک کد یکسان را اجرا کنند:
- این دو نمی‌توانند متغیرهای ThreadLocal یکدیگر را بخوانند یا تغییر دهند
- وقتی چنین متغیری new می‌شود، عملیات new در هر نخ فقط یک بار اجرا می‌شود
- متغیرهای ThreadLocal فقط در محدوده یک نخ استفاده می‌شوند
- کاربرد:
- بهاشتراک‌گذاری یک داده با بخشی از برنامه که در همین نخ اجرا خواهد شد
- به این ترتیب نیازی به کنترل دسترسی همزمان به این متغیر (با قفل و ...) نیست

```
class Task implements Runnable{  
    ThreadLocal<Integer> tl = new ThreadLocal<>();  
    public void run() {  
        tl.set(tl.get() == null ? 1 : tl.get() + 1);  
        long thrID = Thread.currentThread().getId();  
        System.out.println(thrID + ":" + tl.get());  
    }  
}
```

9:1
9:2
9:3
9:4
9:5

```
Executor e = { Executors.newSingleThreadExecutor(); }
```

```
Task task = new Task();  
for (int i = 0; i < 5; i++)  
    e.execute(task);
```

```

class Task implements Runnable{
    ThreadLocal<Integer> tl = new ThreadLocal<>();
    public void run() {
        tl.set(tl.get() == null ? 1 : tl.get() + 1);
        long thrID = Thread.currentThread().getId();
        System.out.println(thrID + ":" + tl.get());
    }
}

```

```

Executor e = {
    Executors.newFixedThreadPool(10); 9:1
    11:1
    13:1
    10:1
    12:1
}
Task task = new Task();
for (int i = 0; i < 5; i++)
    e.execute(task);

```

```

class SumTask implements Callable<Integer> {
    private int num = 0;
    public SumTask(int num) {
        this.num = num;
    }
    @Override
    public Integer call() throws Exception {
        int result = 0;
        for (int i = 1; i <= num; i++)
            result += i;
        return result;
    }
}

```

کویز: خروجی برنامه زیر؟

```

ExecutorService service =
    Executors.newSingleThreadExecutor();
SumTask sumTask = new SumTask(10);
Future<Integer> future = service.submit(sumTask);
Integer result = future.get();
System.out.println(result);

```

55

۱) معضلات برنامه‌های همروند

- دسترسی و تغییر همزمان متغیر مشترک، یکی از معضلات برنامه‌های همروند است
- شرایط مسابقه (Race condition)
 - که درباره این معضل و راههای کنترل و جلوگیری از آن صحبت کردیم
- اما به واسطه استفاده نابجا یا ناکارامد از امکاناتی مثل lock و synchronized
- نه تنها ممکن است کارایی و سرعت برنامه به شدت افت کند
- بلکه ممکن است اشکالات دیگری ایجاد شود، مانند:
 - گرسنگی (Starvation)
 - بنبست (Deadlock)

تشخیص امکان و جلوگیری از
گرسنگی و بنبست در برنامه‌های
بزرگ بسیار مشکل است

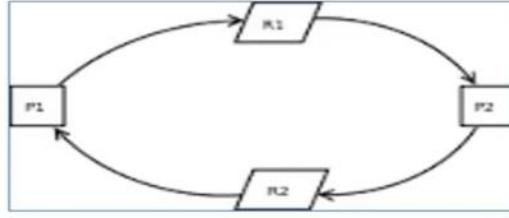
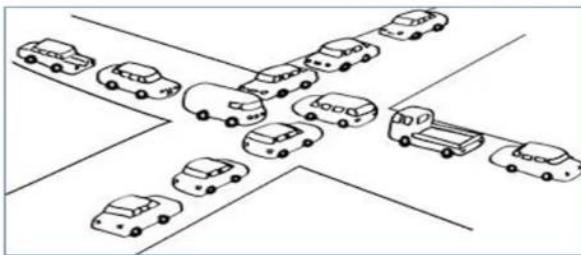
• طراحی یک برنامه همروند باید به گونه‌ای باشد که:

۱- امکان دسترسی نامناسب به منابع مشترک را ندهد

۲- کارایی مناسبی ارائه کند. به ویژه از گرسنگی و بنبست جلوگیری کند.

بنبست (deadlock)

- شرایطی که در آن چند نخ برای همیشه متوقف می‌مانند زیرا منتظر یکدیگرند
- چند بخش همروند وجود داشته باشد که هر یک منتظر پایان دیگری باشد
- مثلاً نخ ۱ قفل الف را گرفته ولی برای ادامه اجرا منتظر آزاد شدن قفل ب است
- همزمان نخ ۲ قفل ب را گرفته و منتظر آزاد شدن قفل الف است



گرسنگی (starvation)

- برخی از نخها همواره منتظر باشند و هیچ وقت نوبت اجرای آنها نشود
- مثلاً یک نخ همواره منتظر گرفتن قفل برای ورود به بخش بحرانی بماند
- زیرا نخهای دیگری همواره زودتر قفل را می‌گیرند
- مسئله گرسنگی کمتر از بنبست به وجود می‌آید ولی کشف و رفع آن هم پیچیده‌تر است
- معمولاً به خاطر الگوریتم‌های ساده (ناکارامد) زمان‌بندی و اولویت‌بندی ناشی می‌شود
- سیستم‌های عامل جدید از الگوریتم‌های مناسبی برای زمان‌بندی نخها استفاده می‌کنند
- یک برنامه به خاطر الگوریتم بدوى زمان‌بندی بین نخها ممکن است ایجاد گرسنگی کند
- مثال:
نخهای خجالتی: تا وقتی منبع مشترک قفل است ۱۰ ثانیه صبر کن، سپس یک کار یک‌دقیقه‌ای
نخهای بی‌پروا: تا وقتی منبع مشترک قفل است ۱۰ میلی‌ثانیه صبر کن، سپس یک کار یک‌ ساعت

- در جاوا ۸ چه امکانات مهم و مفیدی برای برنامه‌نویسی چندنخی اضافه شده است؟

- Parallel Streams

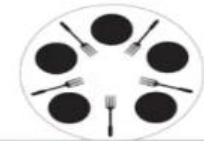
Condition • واسط

ForkJoinPool (از جاوا ۷) • کلاس جدید

volatile (کلیدواژه) • متغیرهای

Dining Philosophers Problem (مشكله غذاخوردن فلاسفه) • مسئله

livelock (مفهوم) • مفهوم و تفاوت آن با گرسنگی و بنبست



- برخی از ظرف‌ها **synchronized** هستند (Synchronized Collections)
 - ظرف‌هایی که استفاده از آن‌ها در چند thread همزمان، امن است
 - کلاس‌های thread-safe (مراجعه به مبحث ConcurrentHashMap و Vector)
 - اگر نیازی به استفاده همزمان از اشیاء کلاس نیست، از اینها استفاده نکنید
- برخی ظرف‌ها غیرقابل تغییر هستند (Unmodifiable Collections)
 - فقط می‌توانیم اعضای آن‌ها را پیشمایش کنیم (کم‌وزیاد کردن اعضا ممکن نیست)
 - مثال:

```
List<String> unmod1 = Arrays.asList("A", "B");
List<String> mod1 = new ArrayList<>(unmod1);
Collection<String> unmod2 = Collections.unmodifiableCollection(mod1);
```

- دنباله اعضا را به چند بخش تقسیم می‌کند
- و هر بخش در یک thread مجزا پردازش می‌شود
- به صورت پیش‌فرض، به تعداد هسته‌های پردازشی thread می‌سازد



• نکته:

- امکانات جدید جاوا از نسخه ۵ به بعد
- برای تسهیل و کم خطر شدن برنامه‌نویسی هموارند
- Java 5: Thread Pools, Concurrent Collections
- Java 7: Fork/Join Framework