

## Java

### OOP

#### Encapsulation:

Data and method **Hiding**, access control **modifiers**

as Java Bean (private members and public **getter/setter**)

Encapsulation leads to the desired level of Abstraction.

#### Polymorphism:

The **ability of a class to provide different implementations of a method**

**Overloading:** compile time binding and poly, depending on the **ref** of object

**Overriding:** runtime binding and poly, depending on the **type** of object that the method is called on it. (Inheritance/ Abstraction)

#### Inheritance:

defining a parent-child relationship between classes.

**a subclass inherits** the properties and methods of its superclass.

**code reuse/ creation of more specialized classes**

#### Abstraction:

**hiding the implementing details and showing only functionality**

**Abstract classes – Interfaces**

achieve loose coupling

## Operators

Based on Precedence:

			Parentheses
		<b>Unary Operators</b>	
	!	for boolean      -negative      ++	<b>Arithmetic</b>
	+	*	
<b>instanceof</b>	!=	(کوچکتر مساوی) < (بزرگتر مساوی) >=	<b>Relational</b>
	or^	and&	Bitwise
		shift >>	
		AND (&&), OR (  ) and NOT (!).	<b>Logical</b>
			variable = Expression ? exp1: exp2      conditional operator( <b>Ternary</b> )
		Num += 3	<b>:Assignment</b>

Operators	Notation	Precedence/Priority
Postfix	expr++, expr--	1
Unary	++expr , --expr , +expr -expr , ^ , !	2
Multiplicative	* , / , %	3
Additive	+ , -	4
Shift	<< , >> , >>>	5
Relational	< , > , <= , >= , instanceof	6
Equality	== , !=	7
Bitwise AND	&	8
Bitwise Exclusive OR	^	9
Bitwise Inclusive OR		10
Logical AND	&&	11
Logical OR		12
Ternary	? :	13
Assignment	= , += , -= , *= , /= , %= , &= , ^= ,  = , <<= , >>= , >>>=	14

برای کار با داده های باینری Bitwise

مثال	دسته	نام	عملکرد
$x = y \& z;$	Binary	AND	بیتی &
$x = y   z;$	Binary	OR	بیتی
$x = y \wedge z;$	Binary	XOR	بیتی ^
$x = \neg y;$	Unary	NOT	بیتی ~
$x \&= y;$	Binary	AND Assignment	بیتی =&
$x  = y;$	Binary	OR Assignment	بیتی  =
$x ^= y;$	Binary	XOR Assignment	بیتی ^=

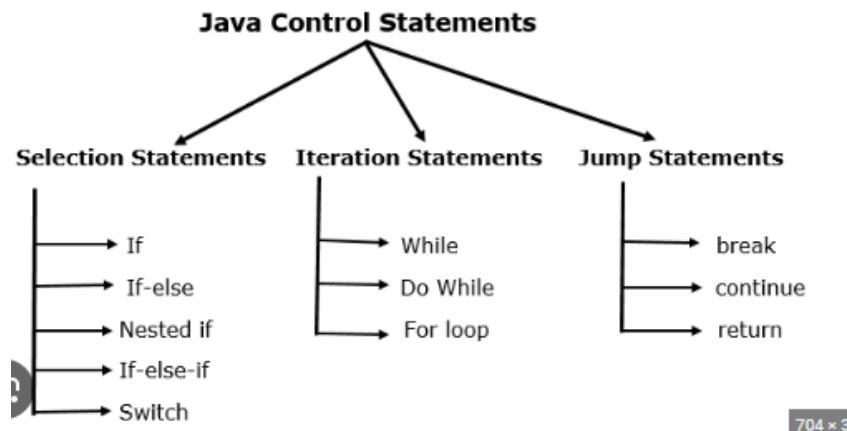
```
int result = 5 & 3;
System.out.println(result);
```

1

اجازه بدھید بیینیم کہ چطور این نتیجہ را بے دست.

```
5: 00000000000000000000000000000000101
3: 00000000000000000000000000000000011
-----
1: 00000000000000000000000000000000001
```

## Java Control Statement



### Switch

سربار زیادی روی رم دارد. چون دستوارت تکراری برای هر بار اجرای سوئیچ switch

با وجود jump حتی break هم نمیشود! خوانده میشود ولی عمل نمیشود.

```
int day = 4;
switch (day) {
    case 6:
        System.out.println("Today is Saturday");
        break;
    case 7:
        System.out.println("Today is Sunday");
        break;
    default:
        System.out.println("Looking forward to the Weekend");
}
// Outputs "Looking forward to the Weekend"
```

### - Iterations

while/ do while/ for/ forEach

ما فقط از **while** استفاده میکنیم و **forEach**

### - ForEach and profational for

عملگر For each با کمک کلاس iterator پیاده سازی شده است

For ( A a: AAA) / .forEach

Array:

```
Student[] arr = new Student[10];
for (Student s : arr) {
    s.setName("Mehrad");
}
```

List:

```
for (String s : list) {
    System.out.println(s);
} (for each)
```

ArrayList:

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

Set:

```
for (String str : set)
    System.out.println(str);
```

Collection:

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

for (Integer integer : s) {
    System.out.println(integer);
}
```

Map:

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

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

mymap.forEach(myFlbicons);
```

متدى بنويسيد که لیستی از رشتهها به عنوان پارامتر بگيرد  
و همه رشتههایی که با Ali شروع می‌شوند را از لیست حذف کند

```
void removeAlis(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.

مراجعه به بخش Fail Fast از فایل 2

For سربار زیادی روی رم دارد.

مشکل دستور **for** : Jvm در زمان تفسیر قطعه **for** ، مدام باید **go to** کند، ولی در ابتدای هر حلقه باید از روی دستور تعریف متغیر بپرد! و این سربار بسیار زیادی را برایش ایجاد میکند.

## Do While

In summary, the main difference between while and do-while loops is that the while loop checks the condition before executing the loop code, while the do-while loop executes the loop code once and then checks the condition.

```
public class DoWhileExample {  
    public static void main(String[] args) {  
        int i=1;  
        do{  
            System.out.println(i);  
            i++;  
        }while(i<=10);  
    }  
}
```

## Comments

### ۱. توضیحات //

از این نوع توضیحات برای توضیحات نک خطي استفاده می شود و جوا آن خط از برنامه را نادیده می گیرد.

### ۲. \*/<sup>\*</sup> توضیحات /\*

از این نوع توضیحات برای توضیحات چند خطی استفاده می شود و جوا خطوطی از برنامه را که در آن قرار می گیرند را نادیده می گیرد.

### ۳. \*/<sup>\*</sup> سند \*/<sup>\*</sup>

از این نوع توضیحات برای توضیحات خاصی استفاده می شود و برای ایجاد آن از کلاس خاصی به نام `javadoc` استفاده می شود. (در مورد این نوع خاص فعلا توضیح داده نمی شود)

## نحوه تعریف javadoc

```
/** This class represents a human.*/
public class Person {
    /** national ID (SSN) */
    private String ID;
    ...
}
```

- کامنتی که با `/**` شروع می‌شود (به جای `/*`) به عنوان جاواداک در نظر گرفته می‌شود
- این جاواداک قبل از تعریف هر چیزی که باشد، همان را توصیف می‌کند
  - مثلاً قبل از یک کلاس، متدها، سازنده، یا ویژگی
- دستور javadoc این توصیفات را به یک مستند HTML تبدیل می‌کند
- امکاناتی برای توصیف بهتر برنامه نیز وجود دارد (فراتر از متن)
  - تگ‌هایی (tags) که توضیح خاصی اضافه می‌کنند (مثل `@author`)
  - امکاناتی برای برقراری ارتباط بین مستندات مختلف (مثل `@see`)

```
/**
 * This method should be called to ask the person run
 * @param speed The speed of running
 * @return returns true if he/she can run with that speed
 */
public boolean run(double speed){...}
```

خروجی: Html

### boolean ir.javacup.practicenotes.Person.run(double speed)

This method should be called to ask the person to run

#### Parameters:

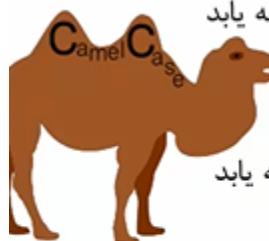
**speed** The speed of running

#### Returns:

returns true if he/she can run at the specified speed

## Naming

این قواعد اجباری نیست



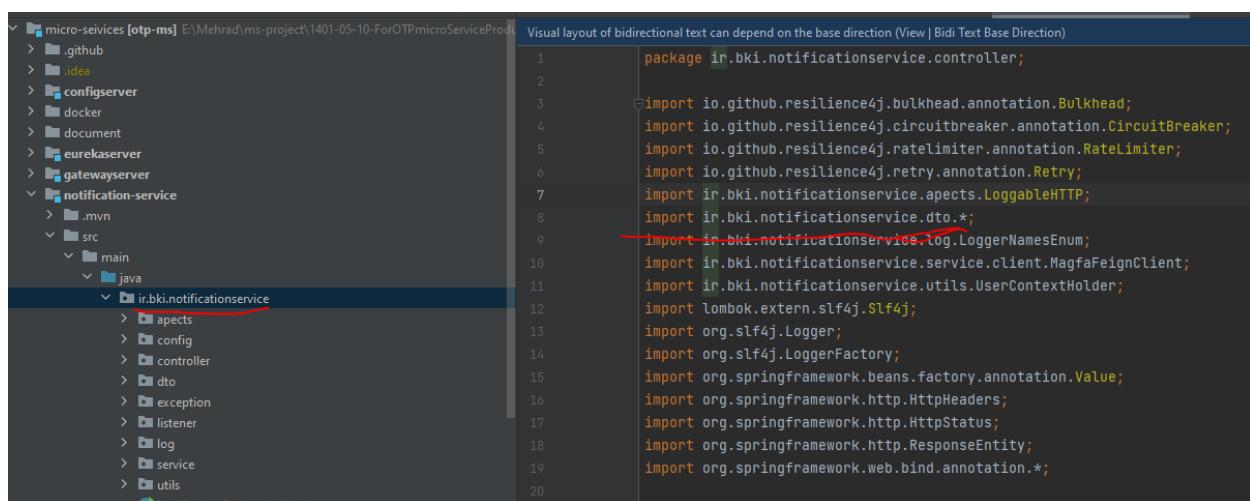
- بسته: همه حروف کوچک. مثل `com.sun.eng`
- کلاس: با حرف بزرگ شروع شود و با الگوی «کوهان شتر» ادامه یابد
  - مثال: `ImageSprite` یا `Raster`
  - از «اسم» برای نام‌گذاری استفاده کنید
- متده: با حرف کوچک شروع شود و با الگوی «کوهان شتر» ادامه یابد
  - مثال: `getBackground` یا `runFast`
  - از «قبل» برای نام‌گذاری استفاده کنید
- متغیرها: شروع با حرف کوچک و ادامه با الگوی «کوهان شتر»
  - مثال: `maxNumber` یا `myWidth`
- ثابت‌ها: همه حروف بزرگ، کلمات مختلف در نام با underscore (`_`) جدا شوند
  - مثال: `MIN_WIDTH`

• نحوه معمول نام‌گذاری بسته‌ها : از کل به جزء

زیرمجموعه . پروژه . دپارتمان . شرکت . دامنه

• مثال:

`ir.javacup`  
`org.apache.commons.io`



The screenshot shows a file explorer on the left and a code editor on the right. The file explorer displays a project structure with several modules like micro-sevices, configserver, document, eurekaserver, gatewayserver, and notification-service. The notification-service module contains sub-directories .mvn, src, main, and java. The java directory contains a package named `ir.bki.notificationservice`. The code editor shows the following Java code:

```
Visual layout of bidirectional text can depend on the base direction (View | Bidirectional Text Base Direction)
1 package ir.bki.notificationservice.controller;
2
3 import io.github.resilience4j.bulkhead.annotation.Bulkhead;
4 import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;
5 import io.github.resilience4j.ratelimiter.annotation.RateLimiter;
6 import io.github.resilience4j.retry.annotation.Retry;
7 import ir.bki.notificationservice.apects.LoggableHTTP;
8 import ir.bki.notificationservice.dto.*;
9 import ir.bki.notificationservice.log.LoggerNamesEnum;
10 import ir.bki.notificationservice.service.client.MafgaFeignClient;
11 import ir.bki.notificationservice.utils.UserContextHolder;
12 import lombok.extern.slf4j.Slf4j;
13 import org.slf4j.Logger;
14 import org.slf4j.LoggerFactory;
15 import org.springframework.beans.factory.annotation.Value;
16 import org.springframework.http.HttpHeaders;
17 import org.springframework.http.HttpStatus;
18 import org.springframework.http.ResponseEntity;
19 import org.springframework.web.bind.annotation.*;
```

برای نام‌گذاری `package` نیز باید تمامی حروف کوچک باشد.

برای نامگذاری متغیر(راه های متفاوتی هست) بهتر است تمامی حروف متغیر کوچک باشد و جدا سازی کلمات بوسیله " \_ " انجام میشود. به عنوان مثال :

Int ageOfAmir

Int age\_of\_amir

: PROPERTY با METHOD تعریف

```
Public Static int a() {  
    Return 120;  
}
```

## Import

- با کمک ستاره (\*) همه کلاس‌های یک بسته قابل استفاده می‌شوند

- نکته: فقط همه کلاس‌های همان بسته، و نه بسته‌های زیرمجموعه آن

```
import static java.lang.Math.*;  
import static ir.javacup.oop.Pride.Length;  
import static ir.javacup.oop.Pride.setLength;
```

- `java.lang`

- `java.lang.String`
- `java.lang.Math`
- `java.util`

- نکته:

- بسته‌ی `java.lang` به طور ضمنی `import` شده است

- نکته: فقط همه کلاس‌های همان بسته، و نه بسته‌های زیرمجموعه آن

```
import java.util.*;
```

- دستور `import` فقط مربوط به کامپایلر است

- کامپایلر جاوا با کمک این دستور، نام دقیق کلاس‌ها را می‌فهمد
- و نام کلاس‌ها را با نام کامل آن‌ها **جایگزین** می‌کند

- دستور `import` در کلاس کامپایل شده دیده نمی‌شود

- فایل `.class` یا `.bytecode`

- دستورهای `import` بلااستفاده هیچ تأثیری در زمان اجرا ندارد

- فقط تأثیر بسیار ناچیزی بر روی کامپایلر دارد
- البته (بیهوده) متن برنامه را طولانی می‌کنند

## Classpath

### (CP) Classpath مفهوم

- یک پارامتر برای کامپایلر جاوا یا JVM است
- مشخص می‌کند در چه محل‌هایی به دنبال کلاس‌ها و بسته‌ها بگردند
- این پارامتر به `javac` یا `java` پاس می‌شود
  - یا به صورت یک متغیر محیطی (Environment Variable) تعریف می‌شود
- البته محل کلاس‌های موجود در زبان جاوا نیازی به معرفی در CP ندارند
  - مثلاً `String`

#### • در ویندوز:

• `java -cp D:\myprogram org.mypackage.HelloWorld`

#### • در لینوکس

• `java -cp /home/user/myprogram org.mypackage.HelloWorld`

#### • استفاده از متغیر محیطی:

```
set CLASSPATH=D:\myprogram  
java org.mypackage.HelloWorld
```

#### • تعیین چند فolder یا JAR در classpath

`java -cp D:\prog;D:\lib\support.jar org.HelloWorld`

#### • برای جذاب کردن بخش‌های مختلف cp در لینوکس و در ویندوز

–classpath – معادل –cp •

#### • مثال: استفاده از چندین فایل JAR و شاخه جاری به عنوان cp :

`java -classpath ':./mylib/*' MyApp`

```
D:\myprogram\  
|  
---> org\  
|  
---> mypackage\  
|  
---> HelloWorld.class  
---> SupportClass.class  
---> UtilClass.class
```

در ویندوز:

```
• java -cp D:\myprogram org.mypackage.HelloWorld
```

• تعیین چند فolder یا JAR در classpath

```
java -cp D:\prog;D:\lib\support.jar org.HelloWorld
```

## Varargs

- امکانی در زبان جاوا تحت عنوان varargs وجود دارد:

متدهای تعریف کنیم که از یک آرگومان، صفر یا چند پارامتر بپذیرند

**void print(String... args){...}** مثال:

- هنگام فراخوانی متدهای print می‌توانیم صفر یا چند رشته به آن پاس کنیم

یعنی همه فراخوانی‌های زیر صحیح هستند:

```
print();
print("Ali");
print("A", "B", "C", "D");
```

- هنگام تعریف متدهای شامل پارامتر varargs می‌شود:

این پارامتر به شکل یک آرایه قابل استفاده است

با توجه به نحوه فراخوانی متدهای (تعداد پارامترها)، این آرایه مشخص می‌شود

```
static void print(String... params) {
    String[] array = params;
    System.out.println(array.length);
    for (String p : params) {
        System.out.println(p);
    }
}
print(); → array.length==0
print("Ali"); → array.length==1
print("Ali", "Taghi"); → array.length==2
```

**String[] array = {"A", "B"};** این دو متدهای چه تفاوتی دارند؟

**void print1(String[] args) { ... }**

**void print2(String... args) { ... }**

متدهای اول فقط به یک شکل قابل فراخوانی است:

```
print1(array);
```

فراخوانی متدهای دوم به همه آشکال زیر صحیح است (دست کاربر باز است)

**print2();**      **print2("Ali", "Taghi");**

**print2("Ali");**    **print2(array);**

## Access Levels modifiers

در اثر **obj** ساختن و ارث بری چه اجزایی در دسترس هستند؟

### - class

کجا میشه از این کلاس **obj** ساخت؟

اگر کلاسی رو **public** تعریف کنیم، همه جا میشه ازش **obj** ساخت فقط کافیه **import** کنیم. (دیده شدن یک کلاس **public** به **import** شدن آن در مقصد هم نیاز دارد.)

کلاسی که file access باشه فقط در فایل جاری میشه ازش **obj** ساخت

One.java :

```
package pack;

public class One {

}

class two{ // file Access
}
```

برای **class** نداریم. فقط برای متدها و متغیرها است

### - Method and props

بعد از ساخت **obj**، اگر **dot** بزنیم، به چه اعضایی از کلاس دسترسی داریم؟  
چه اعضایی قابل ارث داده شدن هستند؟

ابزاریباده سازی **encapsulation**

اعضای **:private**

دسترسی: روی **obj** کلاس جاری فقط در کلاس خودش قابل دسترس است

ارث بری: قابل **inherit** نیستند.

اعضای **private** برای **obj** هایی که در داخل خود هر کلاس تعریف میشوند قابل دسترسند نه حتی کلاس های دیگر داخل همین فایل جاوا.

## اعضای :Public

دسترسی: روی obj کلاس از همه جا قابل دسترس هستند.

ارث بری: قابل inherit هستند. در هر package ای .

## اعضای Package Access و Protected

دسترسی: روی obj کلاس جاری قابل دسترس هستند و در pack خارجی دیده نمیشوند.

ارث بری:

قابل inherit هم در pack داخلی هم در pack خارجی هستند.

قابل inherit فقط در pack داخلی هستند. Package Access

## تفاوت PA و protected

به لحاظ دسترسی مثل هم هستند. (در pack مشترک دیده میشوند و در pack غیر مشترک دیده نمیشوند)

به لحاظ قابلیت به ارث برده شدن:

در pack مشترک، مثل هم هستند. قابل inherit

در pack خارجی: قابل ارث برده شدن است ولی PA قابل ارث برده شدن نیست

نوع فرزند چه فیلد هایی از پدر را میتواند به ارث ببرد؟

در پکیج مشترک: public و protected و package access

در پکیج خارجی: public و protected

پس تنها تفاوت PA و protected در ارث بری در پکیج خارجی است. اگر فرزند خارج از پکیج پدر باشد، PA ها را ارث نمیبرد.

\*\*\*\*

برای package جاری: (public و PA protected) با هیچ فرقی ندارند.)

روی obj sup دیده میشوند.

روی obj sub به ارث برده میشوند.

برای package خارجی:

روی obj sup public فقط دیده میشود، حتی اگر آن کلاس فرزندش باشد، عضو های p یا pa را نمیبیند.

روی protected و public ، obj sub به ارث برده میشوند.

```
package p1;

public class Parent {

    private String pri="pri";
    String package_access="package_access";
    protected String protect="protected";
    1 usage
    public String public="public";

}
```

در کلاس Child که در package دیگری است، میتوانیم به اعضای protected از Parent روی obj پدر،

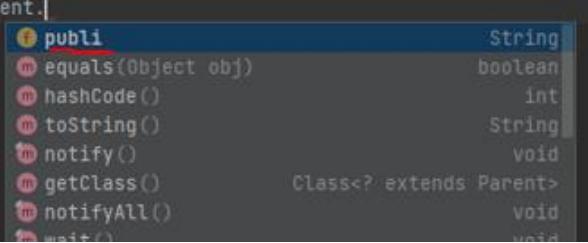
دسترسی داشته باشیم؟

خیر. نمیتواند به اجزای غیر public اش دسترسی داشته باشد حتی با اینکه در کلاس فرزندش است:

```
package p2;

import p1.Parent;

public class Child1 extends Parent {
    public static void main(String[] args) {
        Parent parent = new Parent();
        parent.
    }
}
```



ولی obj نوع فرزند خصوصیات protected پدر را به ارث میرد و از داخل خودش بهش دسترسی دارد:

```
package p2;

import p1.Parent;
// 2 usages
public class Chile1 extends Parent {
    public static void main(String[] args) {
        Parent parent = new Parent();
        // parent.

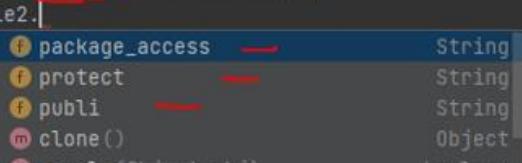
        Chile1 chile1= new Chile1();
        chile1.

    }
}

    ● protect
    ● publi
    ● clone()
    ● equals(Object obj)
    ● hashCode()
    ● toString()
    ● getClass()           Class<? extends Chile1>
    ● notify()
    ● notifyAll()
    ● wait()
    ● wait(long timeoutMillis)
    ● ...
Press Enter to Insert, Tab to replace.
```

اگر `sup` و `sub` در یک pack باشند، هر دو `obj`‌های نوع `subclass` و `superclass` به اعضای `public` و `protected` و حتی `package access` هم دسترسی دارد: (یعنی `subclass` توانسته `field PA` را به ارث ببرد چون داخل یک `P` هستند).

```
package p1;  
  
public class Chile2 extends Parent {  
  
    public static void main(String[] args) {  
  
        Chile2 chile2 = new Chile2() ;  
        chile2.|  
    }  
}
```



The screenshot shows an IntelliJ IDEA code editor with Java code. A code completion dropdown is open at the cursor position, listing several methods:

- f package\_access
- f protect
- f publi
- m clone()
- m equals(Object obj)
- m hashCode()
- m toString()

The method names are color-coded: package\_access, protect, publi, and clone() are in blue; equals(), hashCode(), and toString() are in red.

## UML Class Diagram

- اعضای **protected** : با #
  - اعضای خصوصی : با -
  - اعضای عمومی : با +

در ارث بری متدهای **private** هم در پسر کیو، مشوند ولی قابل **call** کردن نیستند.

## Primitive Data Type/ Wrapper Primitive Classes

S یک ارجاع است. آدرس در آن ذخیره شده است. (obj class).

a یک مقدار در آن ذخیره شده است. (primitive).

- `String s = new String("Ali");`



:Primitives

8 بیت :Byte

short

که برای نگهداری اعداد صحیح استفاده می‌شود. int به 32 بیت

عدد 100 : 000000000000000000000000000000001100100

long: برای نگهداری اعداد بسیار بزرگ

float: که برای نگهداری اعداد اعشاری کوتاه

double: که برای نگهداری اعداد اعشاری بلند

char: که برای نگهداری یک کاراکتر استاندارد

boolean: که برای نگهداری یک حالت منطقی

Primitive type	Wrapper class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

▪ پسندیده مسنتیمیا یک مقدار را نگهداری می‌نمند

▪ متناظر هر نوع اولیه یک کلاس تعریف شده

▪ Primitive Wrapper Classes

▪ هرگاه که یک شیء مورد نیاز باشد

▪ از آنها به جای انواع اولیه استفاده می‌کنیم

▪ امکان مهم: اشیاء این کلاس‌ها، برخلاف انواع داده اولیه، می‌توانند null باشند

حد و حدود مقادیری که میشه ریخت داخل primitive ها و wrapper class ها مثل هم است.  
Primitive Wrapper Class به حساس است. garbage collector  
garbage collector: deleting the objects that are no longer used

### مزایای Wrapper Class ها:

Default of primitives is zero  
Wrapper classes are immutable  
Collection ها را دارد.

Wrapper Class قابلیت نکه داشتن آدرس حافظه در reference را دارد. برای وقتی میخوایم مقدار را در یک collection ها بریزیم بکار می آید.)

### معایب Wrapper Class ها:

Primitives are faster  
Wrapper Class میگیرند  
متلا برای ذخیره یک شی از نوع Long نیاز به یه اشاره گر با اندازه double و همچنین خود مقدار عدد داریم  
اما یه مقدار long فقط شامل ۶۴ بیت میشه که برای ذخیره عدد نیازه.

### تبدیل : primitive به wrapper

```
Double n = new Double(12.2);
double d = n.doubleValue();
int i = n.intValue();
double max = Double.MAX_VALUE;

Integer a = new Integer(12);
int maxint = Integer.MIN_VALUE;
```

اگر برای **age** نوع **int** در نظر میگرفتیم، وقتی **obj** از **Person** ساخته میشود، اگر برايش **age** را **set** نکنیم، مقدار پیشفرض صفر برای آن درنظر گرفته میشود. که از نظر منطقی اشتباه است، پس بهتر است در این موقع از **wrapper class** استفاده کنیم.

Data Type	Default Value (for fields)
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null

برای مقایسه **state** بین دو **wrapper class**

هم میتوانیم از متده **compare To()** استفاده کنیم

. (== , <= , <, >) بجز **relation ops** و هم از

برای مقایسه تساوی **state** دو **Primitive class** فقط میتوانیم از **equals()** استفاده کنیم.

کلاس‌های **String** با **wrapper class** هم **immutable** هستند هم

**Final** هستند برای اینکه توسعه نیابند.

برای اینکه **obj** های آنها تغییری نمیکند.

Ref های آنها میتوانند تغییر کند مگر اینکه وقتی جایی **new** میشوند، آن **feild** هم **final** تعریف شود یا **setter** بدون **private**

## Boxing (auto, un)

هر دو را جاوا پشتیبانی میکند:

### :autoboxing•

- اگر یک مقدار primitive به عنوان یک شیء استفاده شود:
- به صورت خودکار به شیءی از نوع متناظر wrapper تبدیل میشود
- مثال: `Integer i = 2;`

### autoboxing : فرایند برعکس unboxing•

- اگر یک شیء از نوع wrapper به عنوان یک primitive استفاده شود:
- به صورت خودکار به یک مقدار از نوع متناظر primitive تبدیل میشود
- مثال: `int a = new Integer(12);`

۱

Auto boxing برای long به Long نیاز دارد.

```
Integer i = new Integer(2);
Integer j = new Integer(2);
i = j; //Reference Assignment
i = 2; //OK. Autoboxing.
Long l = 2; //Syntax Error. Why?
Long l = 2L; //OK
l = i; //Syntax Error. Why?
System.out.println(i==j);
//Prints false. Why?
```

خط آخر: چون به این شکل مقایسه reference ها را مقایسه میکند که هر کدام به شیئ متفاوتی اشاره دارند.

اگر بعد از `i=j` چک کنیم چی؟

```

public class Person {
    private final String name;      I
    private Integer age = null;
    private Double length = null;
    public Person(String name){
        this.name = name;
    }
    public Person(String name, int age){
        this.name = name;
        this.age = age;
    }
    public String toString(){
        return name + "(+" + age + ")";
    }
}

```

دو پر اپریتی برای اینکه بتوانند مقدار primitive default نگیرند (null بگیرند)، بصورت wrapper تعريف شدند.  
autoBoxing داریم. کجاش؟ امضای متدها.

شیی immutable، Person ندارند (چون private هاش prop تعریف شدند و setter ندارند)

تو Integer در نظر بگیر!!!:

```
String name = new String("ali");
```

```
String name = "ali";
```

این دو خط کد دقیقاً یکی هستند. خود جاوا در جریان فرایند autoBoxing، به OBJ String اون یشت میسازه.

پس وقتی مقدار یک String را تغییر میدهیم:

```

String s;
s= "A";
s= "B";

```

در حقیقت دو تا Object String ساخته شده، s اول به اونی که مقدارش A بوده اشاره میکرده، و بعد به اونی که مقدارش B هست.

چون immutable String است، state آن قابل تغییر نیست. فقط میتوان هر بار ارجاع یک obj را عوض کند به obj دیگر که state آن چیز دیگریست.

\*\*\*\*

اگر java wrapper class developer استفاده کند و فقط از فرآیند autoBoxing استفاده کند، خود هندل میکند: obj هایی که یکسان دارند در یک آدرس حافظه نگه میدارد، پس reference آن obj ها یکسان میشود. یعنی دو obj از wrapper class مساوی، ref یکسان دارند.

ولی اگر developer، خودش new کند حتی اگر قبل از خودش یک obj با همان state new کرده باشد و یا با ایجاد شده باشد، باز هم یک آدرس حافظه جدید برایش در نظر میگیرد.

برای مقایسه state بین دو Primitive wrapper class

هم میتوانیم از متدهای compareTo()

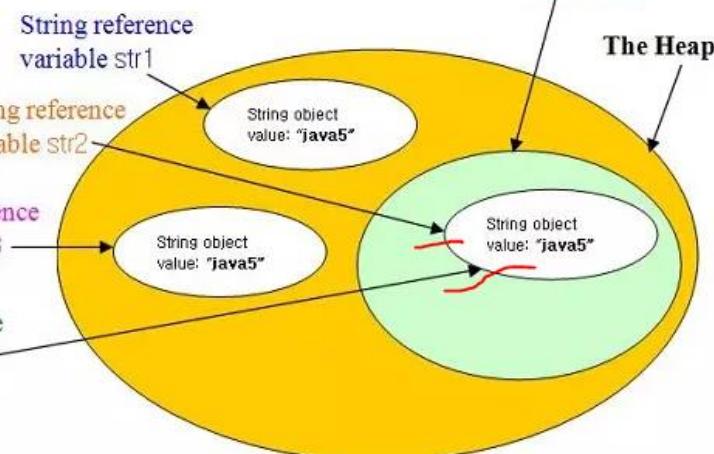
و هم از relation ops ( ==, !=, <=, >=, <, >) میتوانیم از new کند.

برای مقایسه تساوی state دو Primitive wrapper class میتوانیم از equals() استفاده کنیم.

4

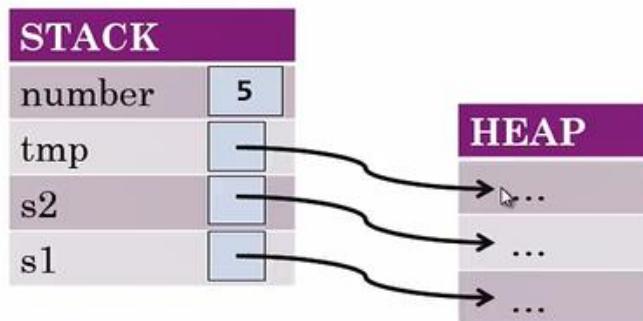
The str4 reference will be pointed to the existing object with the value "java5" within the string constant pool.

```
String str1 = new String("java5");
String str2 = "java5";
String str3 = new String(str2);
String str4 = "java5";
```



## Memory in JAVA

```
public static void swapNames(Student s1, Student s2){  
    String tmp = s1.name;  
    s1.name = s2.name;  
    s2.name = tmp;  
    int number = 5 ;  
}
```



\*\*\*\*

Jdk (platform-dependent)

### Development Tools

writing Java programs

compiler: java Class → byteCode

JRE (executing Java programs)

Reading static block

Jvm

Interpreter (execution engine)

Bytecodes with classLoader → Class Objects in Ram

Management: heap / stack / garbage collector

### Libs

JDBC, JNDI, RMI, ServletContainerInitialization

- جوا با C نوشته شده است Compiler

- برای برنامه‌ی جاوایی ، compiler main بدنبل متند می‌گردد

- برای برنامه‌ی جاوای، interpreter بدنال متد main می‌گردد
- primitive ها در stack قرار می‌گیرند.
- Reference ها stack می‌گرد.
- object ها در داخل heap قرار می‌گیرد.
- اگر برنامه اشیا زیادی را ایجاد می‌کند بهتر است فضای heap را زیاد کنیم.
- اگر برنامه فراخوانی متدهای زیادی دارد بهتر است فضای stack را زیاد کنیم.
- Garbage collector بطور متناوب heap را چک می‌کند.
- کاری با متغیر های primitive stack داخل Garbage
- وقتی اجرای یک scope تمام می‌شود خودکار متغیر های primitive stack را پاک می‌کند.
- instance object ها حساس است هم به Class هم به obj Garbage collector
- اگر obj از روی رم پاک شود همه‌ی data structure هایی که با type آن ساخته شده و همه های آن هم از بین میرود.

شروع اجرای app (جاوای) با :ram JRE روی bytecode

### ■ اجرای static block ■

### ■ interpret (JVM ClassLoader) → create Class Object

این دو مرحله اول برای کلاسی که شامل main است انجام می‌شود و سپس به فراخور فراخوانی، برای بقیه کلاس‌ها انجام می‌شود.

- در اجرای برنامه‌ی Java همه‌ی چی از روی Ram خوانده می‌شود و وابسته به source code (که هیچی) bytecode نیستیم. البته اگر Class Object، garbage classLoader را بکشد باز باید اقدام به ساختن Class Object از روی bytecode کند.
- بر اساس همین مطلب وقتی برنامه‌ی جاوانی را اجرا می‌کنیم نسبت به برنامه‌ی مشابه .net. اجرای اولیه زمان زیادی می‌برد ولی در طول کار با برنامه سرعت آن بهتر از برنامه‌ی مشابه .net. می‌شود. چون .net مدام تکیه دارد به Source code، ولی جاوا فقط برای ساختن Class obj ها به منبع هارد که bytecode ها در آن است نیاز دارد و بقیه‌ی شیئ ها را از روی آن می‌سازد. در جاوا jee در orm حتی db را هم کش می‌کنیم روی رم!!!. هر از چند گاهی با trigger اطلاعات را می‌برد روی db.

## Garbage Collector & Finalize

https://virgool.io/@mohammadehghan/%D8%A2%D9%86%DA%86%D9%87-%DB%8C%DA%A9-%D8%AA%D9%88%D8%B3%D8%B9%D9%87-%D8%AF%D9%87%D9%86%D8%AF%D9%87-%D8%AC%D8%A7%D9%88%D8%A7-%D8%A8%D8%A7%DB%8C%D8%AF-%D8%AF%D8%B1-%D9%85%D9%88%D8%B1%D8%AF-garbage-collector-%D8%A8%D8%AF%D8%A7%D9%86%D8%AF-%D9%82%D8%B3%D9%85%D8%AA-%D8%A7%D9%88%D9%84-k7jmmf5fi7xs

### • می توانیم JVM را تنظیم کنیم

- تا میزان حافظه بیشتر یا کمتری در اختیار برنامه قرار دهد
- با کمک آرگومان هایی که برای jvm ارسال می کنیم

آرگومان	معنی
-Xms	اندازه اولیه Heap
-Xmx	حداکثر اندازه Heap
-Xss	حداکثر اندازه Stack

:مثال

- java Person
- java -Xms512m -Xmx3750m Person
- java -Xss4m Test
- java -Xmx3750m -Xss4m Main

● اگر برنامه‌ای اشیاء فراوانی را new کند، کدام بخش حافظه‌اش پر

OutOfMemoryError: Java heap space می‌شود؟

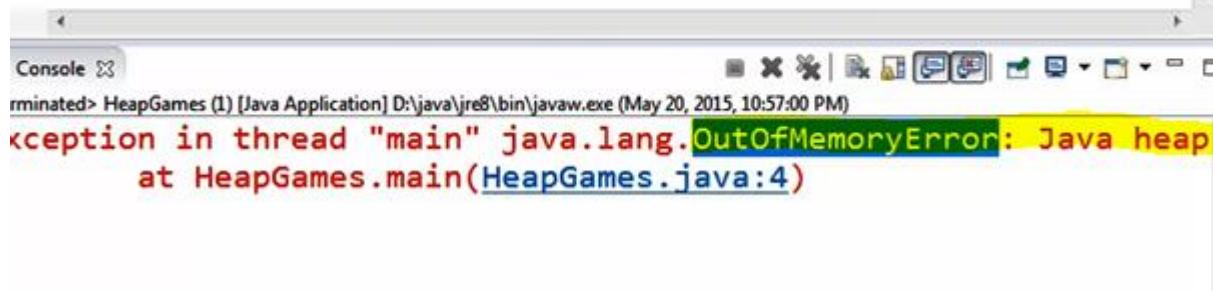
● اگر یک متده را به صورت بازگشتی صدا بزنیم، طوری که هیچ شرط

پایانی نداشته باشد، سرریز می‌شود یا Stack

مثلاً با فراخوانی این متده، چه می‌شود؟

```
int f(int i){  
    return f(i+1);      StackOverflowError  
}  
                                ↓
```

```
2 public class HeapGames {  
3     public static void main(String[] args) {  
4         int[] array = new int[Integer.MAX_VALUE/10];  
5     }  
6 }  
7
```



با اینکه همه obj‌ها null هستند

```
static double f(double d){  
    if(d<30_000){  
        return f(d+1);  
    }  
    return 100;  
}
```



if we get java.lang.OutOfMemoryError we should increase -Xmx to address this issue.

java **-Xmx2048m** -Xms256m

The memory consists of two types of objects: **alive** (still used by the application) and **dead** (not being used anymore).

The garbage collector detects and deletes **unused objects** that are **no longer being referenced**, Eliminates states from heap

JVM implements various versions of garbage collectors with different generations and types



Main steps:

**marking** objects alive, **sweeping** dead objects, and **compacting** the remaining objects.

## *Three main phases to GC*

- 1- Marking objects as Alive
- 2- Sweeping Dead objects
- 3- Compact Remaining objects



لازم است که یه سری GC Root هایی وجود داشته باشن تا به این آبجکت ها هویت بدن زباله جمع کن میداد تونی حافظه های Heap میگرده و آبجکت هایی که کسی باهاشون کاری نداره رو قبل از حذف به قسمتی مجازی انتقال میده که به اصطلاح بهش میگن Old Generations ، بعد حذف می کنه، این فرایند به صورت خودکار در یه ترد کنار برنامه انجام میشه.

## *Generational Garbage Collection*



فیلد های static هست که مستقیم به یه GC Root مرتبط میشن پس سعی کنید هر شی ای رو static نکنید و اگر می کنید بعد اینکه کارتون باهاش تموم شد null کنیدش

## متدها **finalize()**

- هر چند جوا **Destructor** ندارد، برای هر کلاس متدهای با نام **finalize** قابل تعریف است
- هرگاه زبالهروب یک شیء را آزاد کند، متدهای **finalize** از این شیء را فراخوانی می‌کند
- اگر زبالهروب یک شیء را حذف نکند: هرگز متدهای **finalize** برای این شیء فراخوانی نمی‌شود

```
public class Circle {  
    private double radius;  
    public Circle(double r) { radius = r; }  
    public void finalize() {  
        System.out.println("Finalize...");  
    }  
    public static void main(String[] args) {  
        f();  
        System.gc();  
    }  
    private static void f() {  
        Circle c = new Circle(2);  
        System.out.println(c.radius);  
    }  
}
```

```
+ public static void main(String[] args){  
+     test();  
+     System.gc();  
+ }  
+ private static void test(){  
+     Engine e1 = new Engine();  
+     Car car = new Car("Peykan", e1);  
+     System.out.println(car.name);  
+ }
```



## درباره finalize

- چه نیازی به متدهای finalize است؟

- مگر زباله‌روب مرگ شیء را مدیریت نمی‌کند؟

↳ **زباله‌روبی (Garbage Collection)** فقط درباره حافظه است

- گاهی منابعی غیر از حافظه باید آزاد شود

- زباله‌روب این کار را نمی‌کند

## Objects

• یادآوری: هر شیء، حالت، رفتار و هویت دارد

- *state, behavior, identity*



field مقاییر :State

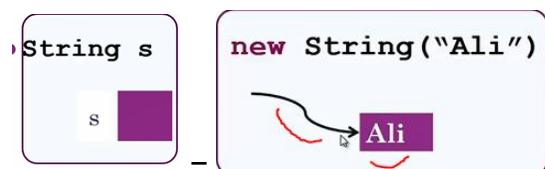
methods :Behavior

reference همان :Identity

یک reference در stack میسازد. (عملگر empty، حتی null هم نیست تا قبل از (=) :Class ref;

عملگر new، آن obj را در heap میسازد. (شامل مقادیر یا ref های دیگر) : new ()

آدرس حافظه آن obj در ref در stack قرار خواهد گرفت. : =



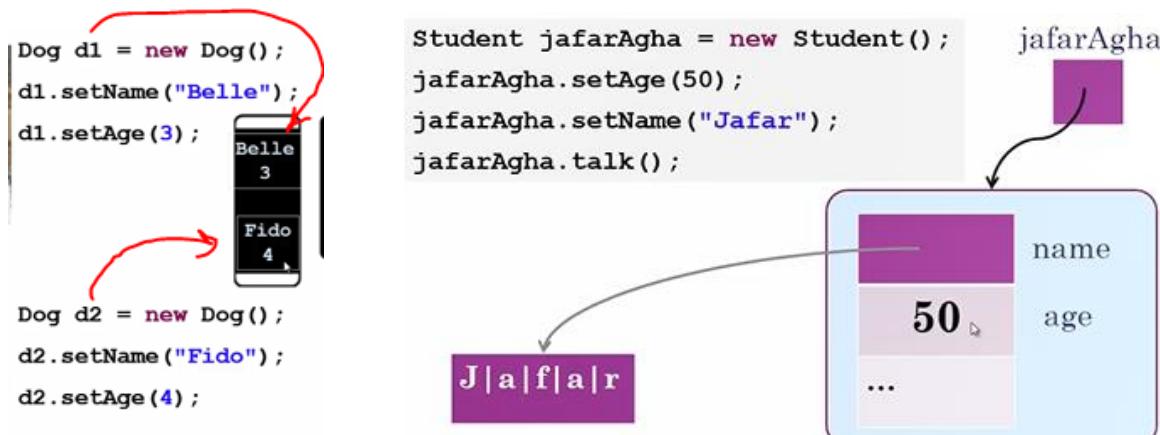
عملگر new

- یک شیء جدید از کلاس مشخص شده میسازد
- ارجاع به شیء ساخته شده را برمی‌گرداند



چیزی که ما بهش میگیم یک obj در حقیقت یک reference به یک obj است.

آدرسی از heap که تجمعی کننده value های مربوط به ref های field است.



در خانه های `ref`, `stack` قرار میگیرند یا `.value`.

مجموعه ای از اینها کنار هم (مستطیل آبی کمرنگ) تحت عنوان `Object` در `heap` به هم متصل و مرتبط میشوند.  
`.(state)`

\*\*\*\*

پس `heap` چیزی نیست جز `bundle` کردن چند خانه از `stack`. آدرس این اتحادیه در `ref obj` قرار میگیرد.  
نهایاتا همه ای مقادیر به شکل `stack primitive` در `stack` قرار میگیرند.

ارجاع هر `obj String`, به قسمتی از `heap` که `array object` است اشاره میکند، این به چند اشاره میکند که در کنار هم مقدار آن رشته را تعیین میکنند.  
`char primitive`

`Str → array → chars`

`Integer → array → chars`

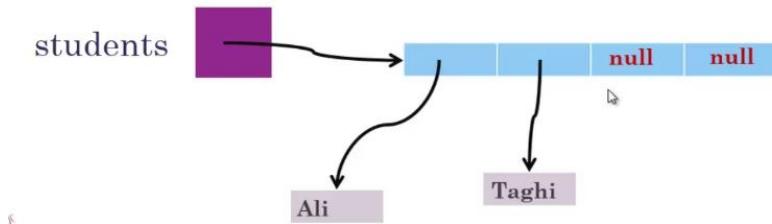
به مجموعه ای از `ref` آرایه ای از `byte` یا یک `int` در `stack` `Ref : Integer`

`:array`

```

Student[] students;
students = new Student[4];
students[0] = new Student();
students[0].setName("Ali");
students[1] = new Student();
students[1].setName("Taghi");

```



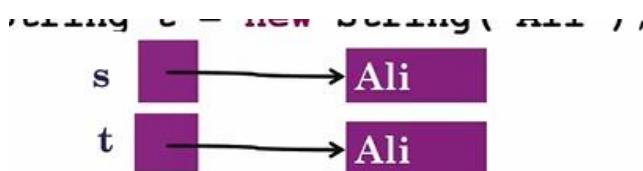
= assignment

وقتی این عملگر جلوی یک ارجاع به `obj` قرار میگیرد، آدرس حافظه ای از `heap` مورد بحث است.

با این عملگر میتوان یک ارجاع به شیء را از شیء ای که به آن اشاره میکرده به شیء ای دیگر تغییر داد.

وقتی `ref` تغییر کند، طبیعتا `state` هم تغییر خواهد کرد. با این عملگر میتوان فقط `obj state` یک `obj` را تغییر داد.

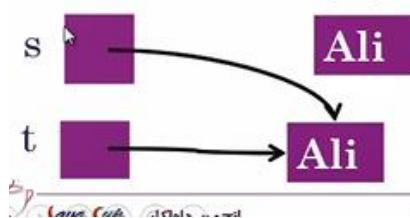
برای اینکه فقط `state` تغییر کند، نیاز است که کلاش `immutable` نباشد.



- متغیرهای `s` و `t` به دو شیء متفاوت (متمايز) اشاره میکنند

- هويت اين دو شیء متفاوت است (identity)

- هرچند حالت اين دو شیء يكسان است (محتو، وضعیت یا state)

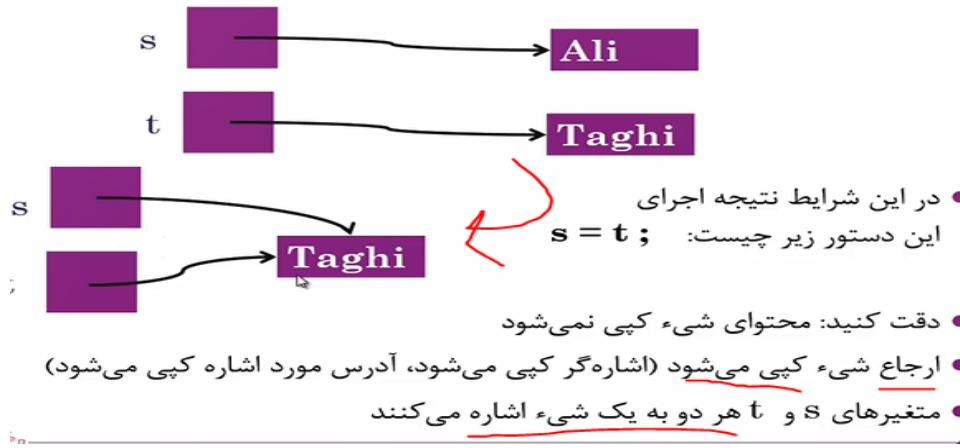


با اين دستور چه میشود: `s=t`;

• هويت `s` با هويت `t` يكی میشود

بنجا که `State` هر دو `obj` از قبل هم بود، اگر هم نبود با عملگر `=`، هم `reference` و هم `state` برای هر دو `s` و `t` يكسان میشود. هر دو شیء `identity` و `state` یكی میشون.

پس از مدتی اون `state` بالای رو `C garbage` از بين خواهد برد.



اگر دو obj دارای reference یکسانی باشند، حتما state یکسانی خواهند داشت.

اگر دو obj یکسانی داشته باشند الزاما identity یکسانی ندارند

## composition

ساخت ارجاع به واسطه composition

وقتی کلاس A دارای prop از کلاس دیگر باشد، وقتی کلاس A new شد خود compiler از کلاس های prop هم میسازد و در متغیر reference آنها null قرار میدهد.

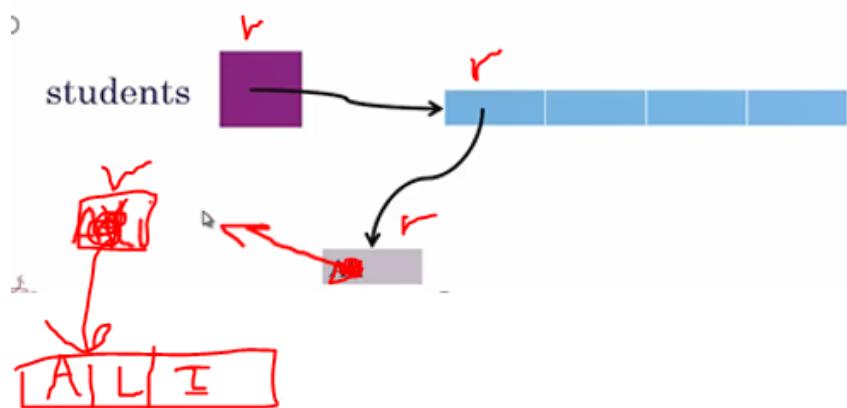
```
Two two = new Two();
System.out.println("two.one = " + two.one); //null, ref its empty, toString(): null
System.out.println("two.long = " + two.aLong); //null

two.one=new One();
two.aLong=10L;
System.out.println("two.one = " + two.one); // pack.One@4eec7777 (ref)
System.out.println("two.long = " + two.aLong); // 10
```

\*\*\*\*

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)

## Call

مثال برای **Call by value** : اگر پارامتری که ارسال میکنیم **primitive** باشد .

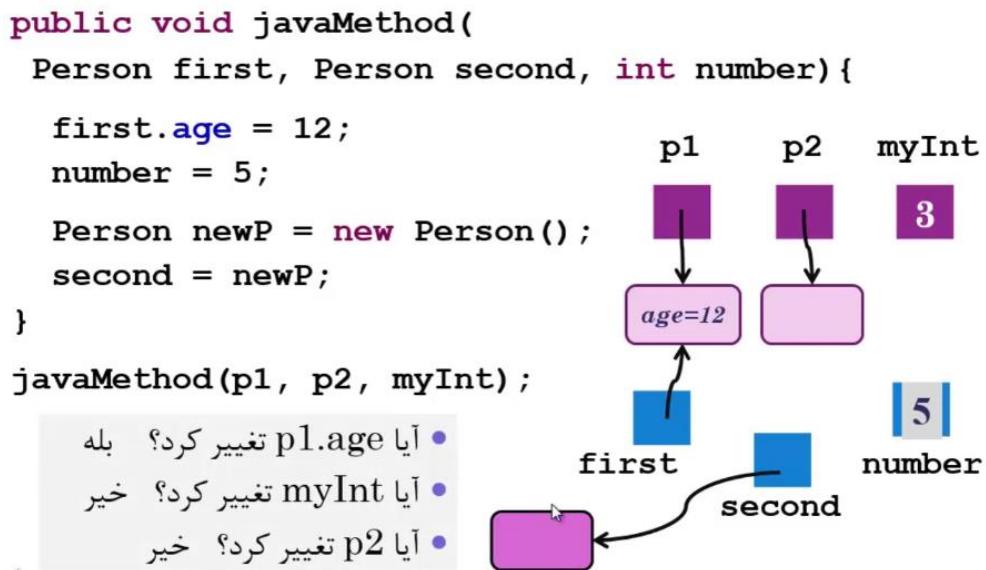
آرگمن از **value** استفاده میکند. یک خانه ی جديد در **stack** ایجاد میشود.

وقتی متده آنرا دریافت میکند و تغییر میدهد، در مبدأ مقدار تغییر نمیکند.

مثال برای **Call by reference** : اگر پارامتری که ارسال میکنیم **Object** باشد .

آرگمن یک کپی از **obj reference** استفاده دریافت میکند.

وقتی متده مقصد **ref** را دریافت میکند و **state** **obj** را تغییر میدهد هر تغییری در **state** بدهد، چون هر دو reference سمت مبدأ و مقصد به یک **obj** اشاره میکنند در هر دو سمت تغییر دیده میشود.



وقتی برای فیلدی از کلاس **getter** مینویسیم یک کپی از ارجاع به ما میدهد.

وقتی **Setter** را برای **obj** ای **call** میکنیم مینتوانیم یک ارجاع بهش بدم که بره بذاره جای ارجاع قبلی آن فیلد

\*\*\*\*

تغییر **ref** مبدا: چون کپی **ref** می‌آید هیچ دسترسی به **ref** اصلی نداریم و نمیتوانیم آنرا تغییر دهیم.

تغییر **state** مبدا: ممکن است؛

اگر نخواهیم **تغییر state** روی **obj** مبدا اعمال شود ، دو راهکار داریم:

اولاً اگر آن کلاس ، **state** باشد، **immutable** را نمیتوانه تغییر بده.

دوماً اگر **immutable** هم نباشه

در مقصود یک **obj** جدید می‌سازیم که یک **reference** جدید خواهیم داشت و **state** را از آن **obj** آرگمان **get** می‌کنیم و برای **obj** جدید **set** می‌کنیم.

راه بهتر استفاده از **serialize** و استفاده از متدهای **write** و **read** آن

راه بهتر تر **clone** کردن است.

## بعد از فرآخوانی badSwap مقدار a و b چه خواهد بود؟

```
static void badSwap(String var1, String var2) {  
    String temp = var1;  
    var1 = var2;  
    var2 = temp;  
}  
  
public static void main(String[] args) {  
    String a = "5";  
    String b = "4";  
    badSwap(a, b);  
}
```

این متدها را نمیتوانند انجام دهد.

قادر به تغییر **ref** نیست : با جابجا کردن **ref** ها راه بجایی نمیرد: مقدار **a** و **b** در مبدا تغییر خواهند کرد، **ref** که پاس داده میشود در داخل متدها مقصود ارجاعات برای **var1** و **var2** جابجا میشود.

قادر به تغییر **state** نیست: اگر در مقصود **state** را تغییر میداد سمت مبدا هم تغییر میکرد ولی چون **String** است نمیتواند **state** را تغییر دهد.

```

public static void swapNames(Student s1, Student s2){
    String tmp = s1.name;
    s1.name = s2.name;
    s2.name = tmp;
}

Student a = new Student();
Student b = new Student();
a.setName("Ali");
b.setName("Taghi");
swapNames(a, b);
System.out.println(a.name);
System.out.println(b.name);

```

دو obj سمت مبدا تغییر میکند. چون student یک کلاس immutable نیست این امکان وجود دارد.

```

public class ParameterPassing {
    public static void main(String[] args) {
        int[] array = new int[4];
        for (int i = 0; i < array.length; i++) {
            array[i] = i;
        }
        f(array);
        System.out.println(array[2]);
    }

    private static void f(int[] a) {
        a[2] = 0;
        for (int i = 0; i < a.length; i++) {
            int x = a[i];
            x= 5;
        }
        a = new int[10];
        a[2] = 1;
    }
}

```

انجمن جاوای ایران aliakbarv@asta.ir

0 . چون arr هم یک ref به obj arr میفرستند، متده میتواند state را تغییر دهد.

کلاس‌های wrapper نسبت به primitive‌ها برای استفاده در چارچوب‌های برگشتی مناسب نیستند، چرا؟  
چون این کلاس‌ها، immutable هستند، یا مقدارشان تغییر نمیکند. بلکه فقط میتوان یک obj جدید ساخت و به reference رو به اون تغییر داد. این کار سریع و حافظه‌ی زیادی میخواهد. ولی اگر primitive باشد روی همان مقدار قبلی، مقدار جدید بروزرسانی میشود.

حتی برای عملگرهای انتسابی مثل ++ هر بار یک Obj جدید میسازد.

```
counter++ --> counter = counter + 1
```

### Creating instance obj- Priority in state Initialize

\*\*\*

را داریم، در JRE هستیم و میخواهیم app جاوایی را اجرا کنیم:

JRE ، خارج از JVM، کلاسی که متدهای main را دارد را پیدا میکند:

### First Load in JRE:

- 1- Read **static fields** and method
- 2- Run **static block**

### Run in JVM:

- 3- JVM: For first **usage** of each class:

**ClassLoader creates Class Object**

+ لیست اجزای غیر استاتیک + (شامل متغیرها و متدهای استاتیکی که قبلتر خوانده)

- 4- JVM: for each **new**:

- a. Default-Constructor makes a reference to Class Object for each new  
**instance and create an instance**
- b. Initialize instance object (Inline initiation + Initiation block+ Constructor)

5- اگر inheritance وجود داشته باشد، و این اولین استفاده از child باشد

اول برای parent مراحل بالا انجام میشود و بعد برای child.

a. اول برای parent Class Object ساخته میشود و بعد برای child.a

b. با new، بعد از ساخت Class Object reference به first، اول child برای parent object انجام میشود و بعد برای second child.

### • یک بار برای هر کلاس

- مقداردهی در خط متغیرهای استاتیک در ابرکلاس
  - بلوک استاتیک در ابرکلاس
  - مقداردهی در خط متغیرهای استاتیک در زیرکلاس
  - بلوک استاتیک در زیرکلاس



خلاصه روند  
مقداردهی اولیه

- مقداردهی در خط و بُزگی‌های ابرکلاس
  - بلوک مقداردهی اولیه در ابرکلاس
  - سازنده‌ی ابرکلاس
  - مقداردهی در خط و بُزگی‌های زیرکلاس
  - بلوک مقداردهی اولیه در زیرکلاس
  - سازنده‌ی زیرکلاس

```
public class Parent {  
    static int a = A();  
    static{  
        a=B();  
    }  
    int b = E();  
    {  
        b = F();  
    }  
    public Parent(){  
        b = G();  
    }  
}  
  
class Child  
extends Parent{  
  
    static int c = C();  
    static{  
        c=D();  
    }  
    int b = H();  
    {  
        b = I();  
    }  
    public Child(){  
        b = J();  
    }  
}
```

1234 in jre

Before 5: first usage is **new**, so:

ClassLoader makes Class Objects

Default constructor makes references to Class objs for instances

اگر یکبار new Parent() بود: تا آخر 7 انجام میشد

```
public class Person {  
    public static int MAX_AGE ;  
    private static double PI = 3.14; ①  
    static{  
        MAX_AGE = 150; ②  
    }  
    private String nation = "Iran"; ③ ⑥  
    private int age;  
    private String name;  
    {  
        name = "Ali"; ④ ⑦  
    }  
    public Person(){  
        ⑤ age = 10;  
    }  
    public Person(int a, String n){  
        ⑧ age = a; ⑨ public static void main(String[] args) {  
            name = n;  Person p1 =new Person();  
        }  Person p2 =new Person(20, "Taghi");  
    }  }
```

در برنامه زیر به ترتیب کدام مقاداردهی‌ها انجام می‌شود؟

ممکن بود بین دو خط متدهای static مقدار فیلدهای PI برای همه instance obj تغییر کند.

\*\*\*

## مسئله اول

- چگونه کلاسی بنویسیم که:

تعداد اشیاء زنده که از این کلاس ساخته شده را نگهداری کند

- شیء زنده: شیءی که ایجاد شده و هنوز توسط زبالهروب حذف نشده است

- هدف: می خواهیم متدهای بنویسیم که تعداد اشیاء زنده این کلاس را برگرداند

- به جزئیات دقت کنید

- چه روشی برای مقداردهی اولیه مناسب است؟

- چگونه به ازای ایجاد هر شیء تعداد را افزایش دهیم

- کدام بخش‌ها public باشند و کدام بخش‌ها نباشند؟

- کدام متغیرها و متدهای استاتیک باشند و کدامها نباشند؟

```
public class LiveObjects {  
    private static int liveInstances = 0;  
    {  
        liveInstances++;  
    }  
    public static int getLiveInstances() {  
        return liveInstances;  
    }  
    public void finalize() {  
        liveInstances--;  
    }  
}
```

```

LiveObjects lives = new LiveObjects();
new LiveObjects();
new LiveObjects();
new LiveObjects();
new LiveObjects();
new LiveObjects();

System.out.println(LiveObjects.getLiveInstances());
// prints 6

System.gc();
Thread.sleep(1000);
System.out.println(LiveObjects.getLiveInstances());
// prints 1

```

## مسئله دوم

- می خواهیم کلاسی بنویسیم که ساختن اشیاء جدید از این کلاس غیرممکن باشد!
- فقط یک شیء از این کلاس ایجاد شود
- هر کس به شیءی از آن کلاس نیاز دارد، از همان شیء استفاده کند
  - و شیء جدیدی نسازد (اصلاً تواند شیء جدیدی بسازد)
- در این کلاس، متدهی تعریف شود که همان شیء را برگرداند
  - به جزئیات دقیق تر کنید
    - چه روشی برای مقداردهی اولیه مناسب است؟
    - چگونه ایجاد شیء جدید را غیرممکن کنیم؟
    - کدام بخش‌ها public باشند و کدام بخش‌ها نباشند؟
    - کدام متغیرها و متدها استاتیک باشند و کدامها نباشند؟

## Singleton

```

public class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton(){}
    public static Singleton getInstance() {
        return instance;
    }
}

```

Singleton s = new Singleton();

Singleton s = Singleton.getInstance();

چگونه برنامه فوق را تغییر دهیم که اولین  
بار که متدهای `getInstance()` فراخوانی  
شده، همان زمان شیء ساخته شود؟

```

class Singleton
{
    private static Singleton obj;
    private Singleton(){}
    public static Singleton getInstance()
    {
        if (obj==null)
            obj = new Singleton();
        return obj;
    }
}

```

- چرا `Constructor` در کلاس `Math` به صورت `private` است؟!
- چه مزایایی برای تعریف `getter` و `setter` وجود دارد؟
- کلاس‌هایی مانند `String` و `Integer` اصطلاحاً `immutable` هستند.  
○ یعنی چه؟ چرا؟
- چگونه برنامه‌ای که شامل بسته‌های مختلف است را کامپایل و اجرا کنیم؟

## **Static field**

**JRE:** Read **static fields** and run **static block**

Read static fields and methods ( خارج jvm ) (by JRE)

run static block ( خارج jvm ) (by JRE)

توسط JRE و خارج از JVM:

Static field ها خوانده شده و در Ram قرار میگیرند.

- یک متغیر استاتیک در واقع یک ویژگی برای کلاس است
  - نه اشیاء
- مثل `Pride.length`
- یک متغیر استاتیک، در بین تمام اشیاء آن کلاس مشترک است
- یک متغیر استاتیک، فقط یک خانه در حافظه دارد
  - هر شیء، احتیاج به حافظه مستقلی برای این ویژگی ندارد
  - بدون ساختن هیچ شیءی می‌توانیم از متغیرهای استاتیک استفاده کنیم
  - با کمک اسم کلاس
- مثلاً `Pride.length = 393;`
- یک ویژگی غیراستاتیک، به ازای هر شیء یک خانه در حافظه ایجاد می‌کند

■ کاربرد: مقدار این متغیر `static` برای همه اشیائی که از آن کلاس ساخته می‌شود یکسان خواهد بود و با تغییر یکی بقیه نیز تغییر می‌کنند. مثلاً برای نمایش تعداد افراد حاضر در اتاق گفتگو مناسب است.

■ متغیرهایی که در سطح کلاس می‌سازیم، یا `static` اند یا غیر `static`:

- اونهایی که `static` نیستند، برای همه نمونه‌ها به طور پیشفرض مقدارش همانیست که در سطح کلاس تعریف شده و اگر نمونه‌ای بخواهد می‌تواند آنرا برای خودش تغییر دهد. هیچ نمونه‌ای و به هیچ طریقی امکان ندارد مقدار پیشفرض را برای `Class Object` تغییر دهد. (شاید با `rop` بشود)
- اونهایی که `static` اند، برای همه نمونه‌ها به طور پیشفرض مقدارش همانیست که در سطح کلاس تعریف شده و اگر نمونه‌ای بخواهد مقدار آنرا تغییر دهد برای همه `obj`‌های قبلی و خود کلاس اصلی و نمونه‌هایی بعد از این ساخته خواند شد، مقدار جدید در نظر گرفته خواهد شد. وجه تلپاتی.

\*\*\*

تفاوت مقدار دهی اولیه `field`‌های `static` با `field` معمولی:

هر دو مقدار دهی اولیه هستند و هر `ins obj` که ایجاد می‌شود دارای همین `state`‌هایی خواهد بود که یا از این `class obj` یا مقدار دهی اولیه شده و لی تفاوت در `ref` برای فیلهای است، اگر مقدار دهی اولیه فیلد از روی `Object` رفرنس گرفته باشد، `ref` اش هم بین `instance`‌ها یکسان خواهد ماند. ولی با مقدار دهی اولیه، `ref`‌های متفاوتی خواهیم داشت.

های static را نمیتوان با constructor یا initialization block برای field غیر static بود مقدار دهی اولیه کرد. آنها زود تر مقدار دهی اولیه شده اند (در static block). البته می توان بعدا با هر obj ای مقدار آنها را تغییر داد که برای همه اعمال خواهد شد

## روش های مقدار دهی اولیه استاتیک

- دو روش برای مقدار دهی اولیه متغیرهای استاتیک:

- 1- مقدار دهی در خط

```
public static int MAX_AGE = 150;  
private static double PI = 3.14;  
  
static String defaultName = theDefaultName();  
private static String theDefaultName() {  
    return "Ali Alavi";  
}
```

2- بلوک استاتیک (Static Block)

↓

??static class or Interface

A static class is a class that is created **inside a class**, is called a **static nested class** in Java. It cannot access non-static data members and methods. It can be accessed by outer class name. It can access static data members of the outer class, including private

## Static Method

متدهایی که فراخوانیشان، روی Class obj قابل انجام است و حتما نیازی نیست برای صدا زدنش یک obj ساخته شود. مثل static field instance.

کاربرد:

متدهایی را static تعریف میکنیم که برای obj call کردنش نیاز به obj خاصی از آن کلاس نباشد. مثل متدهای کلاس Math. روی هر obj class یا obj instance اگر صدایش کنیم انتظار یک عملکرد رو ازش داشته باشیم. داخل این متدها فقط میتوان به متغیر های static دسترسی داشت.

متدهای static رو میشه از سطح OS هم فراخانی کرد. چون خارج از jvm خوانده میشوند.

پروپرتی ها و متدهای util را با این امکان میسازند:

• (کلاس Math با تغییرات جزئی نمایش داده شده است)

```
public class Math {  
    public static double PI = 3.1415926;  
  
    public static double pow(double a, double b) {...}  
    public static int round(float a) {...}  
    public static int abs(int a) {...}  
    public static double max(double a, double b) {...}  
    public static double sqrt(double a) {...}  
}
```

• کدام متدها باید استاتیک باشند؟

• امانت داده شدن

✓ • دریافت فهرست همه نویسندها

توسط JER همان زمانی که static field ها خوانده میشوند، static method هم خوانده میشود و در آن زمان کدها اجرا نمیشوند فقط خطوط کدها کپی میشوند به ram. سپس static block اجرا میشود که میتواند static field ها را مقدار دهی کند. سپس در اولین استفاده با class obj ساختار JVM class loader ساخته میشود.

آیا متدهای static instance obj توسط آنها قابل صدا زدن هستند؟

بله.

متدهای static داخلشان فقط میتوانند از متغیرها و متدهای static استفاده کنند.

به this هم دسترسی ندارند.

(البته بصورت غیر مستقیم میتوان به اجزای غیر static هم دسترسی پیدا کرد.)

متدهای غیر static که کلا آزادند.

\*\*\*\*

آیا راه هایی هست که بتوان در متدهای static به اجزای غیر static دسترسی پیدا کرد؟

دریافت آرگمن آن یک static temp obj instance obj و یا ساخت static instance obj یا static در دل متدهای static ، سپس روی آن اجزای غیر استاتیک را داخل static method میگیریم.

```

public class Rectangle {
    private int length;
    private int width;
    private static int DIMS = 2;

    public void setLength(int length){
        this.length = length;
        System.out.println(DIMS);
        f(this);
    }
    private static void f(Rectangle rectangle) {
        rectangle.width = 2;
    }
    public int getLength() {
        return length;
    }
}

```

\*\*\*\*

```
public class CallClass {
```

```

public void method() { }
public static void staticMehod() { }

```

```
public void callMethod () {
```

```

CallClass c = new CallClass(); //instance obj
c.method();
c.staticMehod();

```

```

new CallClass().staticMehod(); // obj movaghat
new CallClass().method();

```

```

CallClass.staticMehod(); // Class obj
CallClass.method(); //error

```

زمانی که **Class obj** در **ram** ساخته شده فقط اجزای **static** برایش شناخته شده اند

this.staticMehod(); //ye objei in method (callMethod)ra call mikonad

```

this.method(); // ye objei in method (callMethod)ra call mikonad

staticMehod();

method();      // baraye yek obj in method call khahad shod va ejra mishavad

}

```

**public static void callMethodStatic(String[] args) {**

**staticMehod();**

**method(); //error**

چون در زمان قرار گرفتن این متد روی رم و class obj، هنوز () method وجود ندارد

CallClass f = **new** CallClass();

f.method();

f.**staticMehod();**

**new** CallClass().**staticMehod();**

**new** CallClass().method();

CallClass.**staticMehod();**

CallClass.method(); // **error**

زمانی که در ram ساخته شده فقط اجزای static برایش شناخته شده اند.

داخل متد **this** نمیشه static را صدا کرد :

**this.staticMehod(); //error** , chon this eshare be obj developeri darad va method static aval mire roye ram ke hanoz obj developeri sakhte nashode

**this.method(); //error** , chon this eshare be obj developeri darad va method static aval mire roye ram ke hanoz obj developeri sakhte nashode

```
}
```

```
}
```

## static block

برای مقدار دهی فیلد های استاتیک

Class Object ای برای ساخت constructor

A static block is a block of code **that is executed when a class is loaded into memory**. It is used for initialization and cannot be called directly by developers.

دو کار برای همه کلاس های برنامه (به فراخور اجرای برنامه به محض اولین نیاز) اجرا میشود.

ها قبل از static block خوانده و در Ram قرار میگیرند.

حالا **Class Obj** را هم داریم.

اگر ما بلوک استاتیک را ننویسیم جواوا خودش را بهینشو در نظر میگیره.

هر کلاس میتواند یک بلوک استاتیک داشته باشد.

بلوک استاتیک قابل فراخانی توسط برنامه نویس نیست.

استفاده از بلوک استاتیک خطرناک است، یک سری از `manage` های آبجکت را خودمون را باید بکنیم (یعنی کلاس آبجکت رو ارث نبردیم) مثل کلون کردن آبجکت

بهتر است چه کدهایی در `static block` بنویسیم؟ مقدار دهی اولیه متغیرهای `static`

اگر بخواهیم `c` بزنیم، بخواهیم لایبرری های آنرا بزنیم (`pointh`) ، مثلاً بخواهیم در متدهای `main` دستور `c` بزنیم، لازم است که قبل از اجرای آن، لایبرریهای `pointh` را در `static block` بدهیم.

```
public class Person {  
    public static int MAX_AGE ;  
    private static double PI ;  
    static String defaultName ;  
    private static String theDefaultName() {  
        return "Ali Alavi";  
    }  
    static{  
        MAX_AGE = 150;  
        PI = 3.14;  
        String s = theDefaultName();  
        if(s != null)  
            defaultName = theDefaultName();  
    }  
}
```

ها وقتی میخواهند `SW` بنویسند از `static block` استفاده میکنند.

## Class Object

در هنگام اولین استفاده از کلاس، یکبار:

Jvm: `ClassLoader` → create Class object (شامل فیلدها و متدهای استاتیک)

create Class object by `ClassLoader` (شامل فیلدها و متدهای استاتیک) jvm

\*\*\*\*

اولین استفاده از یک کلاس چیست؟

ساخت `ref` باعث ساخت `class object` نمیشود.

مثلاً عملگر `import` / `call static field or method` / `ioc` / `new` کردن!

`new Amir().age;`

impl جدا از اجزای static و لیست متدها (پارامترها) و متغیرها از چه کلاس هایی Class object کرده و ... را هم دارد. وقتی اطلاعات را از Class Object بگیریم و call annotation کنیم.

Class object پیشنهادی برای ساخت instance object است. هر ref به Class Object خود دارد. کپی در کار نیست. مثلا برای اجزای static در همه‌ی instance هایها یک ref به آنها هست که در ابتدا در قرار گرفتند.

برای Class Obj ها و Interface هم abs annotation داریم؟؟؟

### :Class Object Call

```
Mehrad.staticFeild;        Mehrad.staticMethod;  
Mehrad.method; // error
```

\*\*\*\*

ها طبق Class.java از پکیج lang ساخته می‌شوند.

```
Class<?> aClass = Class.forName("com.javaland.proxy.Test");  
Object testObject = aClass.newInstance();
```

Class Class را نمیتوانیم داخل Object Class قرار دهیم، بلکه باید در ref از Class Object قرار دهیم. چرا که طبق type این کلاس ClassLoader اقدام به ساخت Class Object می‌کند.

در مثال بالا مشخص است میتوانیم از Class Object instance ، Class Object بسازیم ، مثل new() (مربوط به کلاسی هست که آدرس مشخص شده) و آنرا در ref ای Object Class قرار دهیم.

```
Mehrad.class.newInstance();
```

راه های گرفتن :Class Object

Class.forName("zer.Mehrad")

Mehrad.class

mehrad1.getClass().

توضیح بیشتر:  
Reflection در مبحث

## شیء کلاس و متدها

- اولین بار که از یک کلاس استفاده می‌کنیم، این کلاس در حافظه بارگذاری می‌شود
- Dynamic Loading
- اطلاعات مربوط به این کلاس، در شیئی با نام «شیء کلاس» در حافظه جای می‌گیرد
- Class Object
  - مثلاً یک شیء در حافظه اطلاعات کلاس String و شیء دیگری، اطلاعاتی درباره کلاس Person را نگهداری می‌کند
  - هر شیئی، یک ارجاع به «شیء کلاس» خودش دارد
  - این ارجاع با کمک متدهای `getClass()` یا `getName()` برآورده می‌گردد
  - متدهای `final` در `getClass()` پیاده‌سازی شده و است

```
Animal a = new Dog("Fido");
String s = a.getClass().getSimpleName();
s = a.getName();
```

: Class Object را با تغییر ویژگی های static

Mehrad.integer\_static=2;

. Class object ، اشاره می‌کند به

این نوع obj فقط میتواند اعضای static کلاس را دسترسی داشته باشد.

مقادیر field های غیر استاتیک را نمیتوان از Class obj خواند.

Class Obj به Class Obj هم حساس است(هم به سایر instance ها). اگر یک Obj را پاک کرده باشد و در طول اجرای app به محض اولین ارجاعی به آن کلاس ، دوباره Class Obj آنرا میسازد و از روی آن instance میسازد. پس ممکن است برنامه‌ی جاوایی هم بعد از run شدن باز هم به byte code روی هارد نیاز داشته باشد.

## بارگذاری پویا (Dynamic Loading)

- یک برنامه جاوا، از کلاس‌های مختلفی استفاده می‌کند
- اما همه این کلاس‌ها، در ابتدای اجرای برنامه در حافظه بارگذاری نمی‌شوند
- هر زمان که به یک کلاس نیاز شود، این کلاس در حافظه بارگذاری می‌شود
- در واقع در اولین استفاده از یک کلاس، آن کلاس بارگذاری می‌شود
- به این امکان در جاوا، بارگذاری پویا (Dynamic loading) می‌گویند
- سؤال: به ازای هر کلاس، دقیقاً چه چیزی در حافظه بارگذاری می‌شود؟

## درباره بارگذاری پویا

- چه زمانی کلاس موردنظر بارگذاری می‌شود؟
  - در اولین استفاده، مثلاً:
- هنگامی که اولین بار یک نمونه از آن ایجاد شود (با عملگر new)
- و یا اولین بار که یک متادستاتیک از آن فراخوانی شود
- هنگام بارگذاری یک کلاس چه اتفاقاتی می‌افتد؟
  - یک شیء کلاس (Class Object) برای کلاس ایجاد و در حافظه بارگذاری می‌شود
  - فرایند مقداردهی اولیه متغیرهای استاتیک (static initialization)
  - چه بخشی مسؤول بارگذاری کلاس جدید است؟
  - بخشی با نام بارگذار کلاس (Class Loader)
  - بارگذار کلاس مسؤول پیدا کردن کلاس موردنظر و بارگذاری آن در حافظه است

```
class Example {  
    static int s1 = f();  
    static {  
        System.out.println("static block");  
        s1 *= 2;  
    }  
    public static void g() {}  
    private static int f() {  
        System.out.println("inline static init");  
        return 5;  
    }  
}  
  
public class Statics {  
    public static void main(String[] args) {  
        Example e;  
        System.out.println("After Declaration");  
        Example.g();  
        e = new Example();  
        e = new Example();  
    }  
}
```

مثال

After Declaration  
inline static init  
static block

در خط زرد ، class object ساخته می‌شود

## انواع بارگذار کلاس (Class Loader)

### Bootstrap class loader

- بخشی از JVM که به صورت سطح پایین (native) پیاده‌سازی شده است
- هسته اصلی جاوا را (از شاخه <JAVA\_HOME>/jre/lib) بارگذاری می‌کند

### Extensions class loader

- کلاس‌های موجود در شاخه <JAVA\_HOME>/jre/lib/ext را بارگذاری می‌کند

### System class loader

- کلاس‌های موجود در CLASS-PATH را می‌یابد و بارگذاری می‌کند

### User-defined class loaders

- برنامه‌نویس می‌تواند یک بارگذار (Class Loader) جدید معرفی کند
- (مثال برای دریافت اطلاعات کلاس از پایگاه داده یا از طریق شبکه)

## شیء کلاس و متدهای آن

- اولین بار که از یک کلاس استفاده می‌کنیم، این کلاس در حافظه بارگذاری می‌شود
- اطلاعات مربوط به این کلاس، در شیئی با نام «شیء کلاس» در حافظه جای می‌گیرد
- مثال یک شیء در حافظه اطلاعات کلاس String
- و شیء دیگری، اطلاعاتی درباره کلاس Person را نگهداری می‌کند
- هر شیء، یک ارجاع به «شیء کلاس» (Class Object) مربوط به کلاس خودش دارد
- این ارجاع با کمک متدهای **getClass()** بر می‌گردد
- متدهای **final** در **Object** برای **getClass()** معرفی شده و است

```
Animal a = new Dog("Fido");
String s = a.getClass().getSimpleName();
s = a.getName();
```

## امکانات کلاس

```
public final class Class<T> implements Serializable, ...
```

- اطلاعاتی درباره متدهای کلاس موردنظر
- فهرست متدها
- دربیافت یکی از متدها با کمک نام و مشخصات پارامترها
- فیلدهای کلاس موردنظر
- فهرست فیلدها، دربیافت یکی از فیلدها، ...
- سازنده‌ها (Constructor)
- اطلاعاتی درباره حاشیه‌نویسی‌ها (Annotation)
- ...

## رواههای رسیدن به شیء کلاس

۱- استفاده از دستور `class`. بعد از نام کلاس

• مثال: `Class c = Person.class;`

۲- استفاده از متدهای استاتیک `Class.forName`

`Class c = Class.forName("ir.javacup.Person");`

۳- فراخوانی متدهای `getClass` بر روی یک شیء

`Object o = new Person();  
Class c = o.getClass();`

## تفاوت ماهیت عملگر instanceof و شیء کلاس

`if(c instanceof Person)...`

• این دو دستور چه تفاوتی دارند؟

`if(c.getClass().equals(Person.class))...`

• دستور اول (عملگر instanceof) :

• اگر `c` از نوع `Person` یا یکی از زیرکلاس‌های `Person` باشد، `true` برمی‌گرداند

• رابطه is-a را بررسی می‌کند

• دستور دوم (استفاده از شیء کلاس) :

• اگر `c` دقیقاً از نوع `Person` باشد، `true` برمی‌گرداند

• نکته: عملگر instanceof همانند متدهای `isInstance` در `Class` است

`if(Person.class.isInstance(c))...`

## ROP

وقتی XML میخوانیم و در آن مشخص شده چه کلاس‌ها و متدهایی باید اجرا شوند باید با `rop` اجرایش کنیم.

<<Java Class>>

 Method

java.lang.reflect

- getDeclaringClass():Class<?>
- getName():String
- getModifiers():int
- getTypeParameters():TypeVariable<Method>
- getReturnType():Class<?>
- getGenericReturnType():Type
- getParameterTypes():Class<?>
- getParameterCount():int
- getGenericParameterTypes():Type[]
- getExceptionTypes():Class<?>
- getGenericExceptionTypes():Type[]
- equals(Object):boolean
- hashCode():int
- toString():String
- toGenericString():String
- invoke(Object, Object[]):Object
- isBridge():boolean
- isVarArgs():boolean
- isSynthetic():boolean
- isDefault():boolean
- getDefaultValue():Object
- getAnnotation(Class<T>):T
- getDeclaredAnnotations():Annotation[]
- getParameterAnnotations():Annotation[][]
- getAnnotatedReturnType():AnnotatedType

```
Class<?> aClass = Class.forName("com.javaland.proxy.Test");
Object testObject = aClass.newInstance();
Object plus = aClass.getMethod( name: "plus", int.class, int.class )
    .invoke(testObject, ...args: 12, 10);
System.out.println(plus);
```

```

Method[ ] m2= Mehrad.class.getMethods() ;

Method[ ] m3= Class.forName("zer.Mehrad").getMethods() ;

Mehrad mehrad1 = new Mehrad() ;

Method[ ] m1= mehrad1.getClass().getMethods() ;

```

## Constructor - new

متدهای بدون نوع برگشتی، هم نام کلاس

میتوانند public یا private(singleton) تعریف شود

وقتی جایی **new** میشود، یا حتی IOC میخواهد obj بسازد؛

**1- Default-Constructor** (JVM): creates an instance and for that ref, with reference to Class Object (not copy).

**2- Inline initialization و Initialization block**

**3- initialization by constructor + constructor**

برای بارگذاری اشیاء جدید در سطح RAM استفاده می‌شود، بنابراین در این متدهای دستورات حجیم استفاده نکنید زیرا باعث کندی عمل شی‌سازی در سطح RAM می‌شود.

## Default constructor

اگر برنامه نویس constructor ننویسد، compiler توسط default constructor استفاده می‌شود.

```
Person person = new Person();
```

آجکت person هنوز null است. / مقدار ref، person یک آدرس حافظه است.

بصورت public و بدون parameter Default constructor است. بدنه ندارد ولی کارهایی میکند:

کارهایی که constructor میکند :

یک obj instance میسازد ، یک رفرنس به class obj اش میدهد

برایش در heap یک آدرس تخصیص داده میشود

میتوانیم مقداردهی اولیه یا کارهای دیگری هم انجام دهیم، که باید برایش default constructor را overload کنیم.

وقتی هر نوع constructor ای بنویسیم، default constructor را دیگر نداریم:

```
public class Circle {  
    private double radius;  
    public double getArea() {  
        return radius * radius * 3.14;  
    }  
    public Circle(double r) {  
        radius = r;   
    }  
    public static void main(String[] args) {  
        Circle c;  
        c = new Circle(12);   
         c = new Circle();   
    }  
}
```

وقتی (singleton) overload را میکنیم میتوانیم آنرا private کنیم.

برنامه نویس میتواند default constructor را overload کند:

```

public class Circle {
    private double radius; Overloading Constructors
    public double getArea(){
        return radius*radius*3.14;
    }
    public Circle(double r) {
        radius = r;
    }
    public Circle() {
}

```

Circle c;  
c = new Circle();  
c = new Circle(12);

اگر سازنده مقدار یک ویزگی (Property) را مشخص نکند، چه می‌شود؟

در این صورت، هر ویزگی مقدار پیش‌فرض نوع داده خودش را می‌گیرد

مثلاً یک ویزگی از نوع int، مقدار صفر می‌گیرد

مقادیر پیش‌فرض انواع مختلف داده:

- مقدار پیش‌فرض boolean : false
- مقدار پیش‌فرض بقیه انواع داده اولیه (مثل int, long, char, boolean) : صفر
- مقدار پیش‌فرض متغیرهای ارجاعی (اشیاء) : null

```

public class ConstructorQuiz {
    private int number;
    private double real;
    private boolean condition;
    private String name;
    private Circle circle;
    public ConstructorQuiz(int num, String title) {
        number = num;
        name = title;
    }
    public static void main(String[] args) {
        ConstructorQuiz q = new ConstructorQuiz(5, "Ali");
        System.out.println(q.number);
        System.out.println(q.real);
        System.out.println(q.condition);
        System.out.println(q.name);
        System.out.println(q.circle);
    }
}

```

پاسخ صحیح

5  
0.0  
false  
Ali  
null

• یک سازنده، با کمک کلیدوازه this می‌تواند سازنده دیگری را فراخوانی کند

• در صورت وجود، این فراخوانی باید حتماً اولین دستور سازنده باشد

## Builder

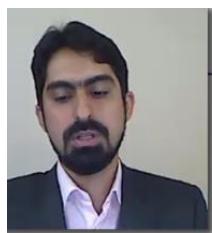
File architecture

File spring

## Inline Initialize & initialize block

وقتی **new** برای یک کلاس صدا زده شود ، اول یک رفرنس جدید برای **instance** جدید میسازد و سپس هر سه مرحله زیر برای مقدار دهی اولیه به Obj جدید طی میشه:

- Inline initialization
- Initialization block
- Constructor



### ترتیب مقداردهی اولیه

- ۱- همه مقداردهی های در خط اجرا می شوند
  - ۲- همه بلوک های مقداردهی اولیه اجرا می شوند
- معمول نیست که یک کلاس چند بلوک مقداردهی اولیه داشته باشد
  - (البته ممکن است)

## \*\*\* تفاوت constructor با init block

- برای هر ویژگی، به مقداردهی اولیه لازم دقت کنید
- و روش (یا روش‌های) مقداردهی اولیه مناسب را انتخاب کنید
- اگر مقداردهی، ساده و در حد یک مقدار مشخص است
  - از مقداردهی در خط (inline initialization) استفاده کنید
  - اگر یک مجموعه کد برای آماده‌سازی اولیه، قرار است در همه سازنده‌ها تکرار شود و نیاز به پارامتر خاصی ندارد
  - از بلوک مقداردهی اولیه (initialization block) استفاده کنید
- اگر مقداردهی اولیه، به پارامتر نیاز دارد: از سازنده استفاده کنید

```
public class Quiz {  
    public int number = f();  
    private int f() {  
        System.out.println("Inline Initialization"); return 1;  
    }  
    {System.out.println("Initialization Block"); number = 2;}  
    public Quiz() {  
        System.out.println("NO-arg constructor"); number = 3;  
    }  
    public Quiz(int num) {  
        System.out.println("ONE-arg constructor"); number = num;  
    }  
  
    Quiz q = new Quiz();  
    System.out.println(q.number);  
    q = new Quiz();  
    System.out.println(q.number);  
    q = new Quiz(7);  
    System.out.println(q.number);
```

خروجی قطعه برنامه زیر چیست؟

Inline Initialization  
Initialization Block  
NO-arg constructor  
3  
Inline Initialization  
Initialization Block  
NO-arg constructor  
3  
Inline Initialization  
Initialization Block  
ONE-arg constructor  
7

وقتی از کلاس فرزند آجکت ساخته می‌شود ترتیب اجرای سازنده‌ها: پدر سپس فرزند- اجباری.

## Inline Initialize

مقادیر اولیه فیلدهای primitive و state اولیه فیلد های obj class هستند. ( آن فیلد ها متفاوت میشود ) بنابرین برای همه instance obj ها مقادیر اولیه فیلدها یکسان هستند.

```
public class Car {  
    private Engine engine = new Engine();  
    private int numberofTyres = 4;  
    private Tyre[] tyres = new Tyre[numberofTyres];  
  
    public Car() {  
        for (int i = 0; i < tyres.length; i++) {  
            tyres[i] = new Tyre();  
        }  
    }  
}
```

## initialize block

## بلوک مقداردهی اولیه (Initialization Block)

- یک (یا چند) بلوک بدون نام که در میان تعریف کلاس قرار می‌گیرد

```
public class Car {  
    private int numberOfType = 4;  
    private Tyre[] tyres;  
  
    {  
        tyres = new Tyre[numberOfType];  
        for (int i = 0; i < tyres.length; i++) {  
            tyres[i] = new Tyre();  
        }  
    }  
}
```

Initialization Block

- هر گاه یک شیء جدید ایجاد شود، بلوک مقداردهی اولیه اجرا می‌شود

## Types of Object

### Class object -

قسمت قبلی توضیح داده شد

```
CallClass.method(); //error
```

زمانی که در ram ساخته شده فقط اجزای static برایش شناخته شده اند

### instance obj -

Obj هایی که در طول اجرای برنامه new می‌شوند و به class obj رفرنس دارند.

راه دیگر بجای new کردن:

```
Mehrad.class.newInstance();
```

## Temporary Object -

```
new Amir().age
```

این آبجکت دقیقا مثل instance obj ، و مقدار دهی های غیر static بعدش انجام میشود؛ ولی آدرس حافظه آن در reference ای ذخیره نمیشود.

```
class Amir
```

```
{
```

```
    int age=20;
```

```
}
```

```
class Fact
```

```
{
```

```
    Public static void main (String [] arg)
```

```
{
```

```
        Amir amir2 = new Amir();
```

```
        amir2.age=30;
```

```
        System.out.print(amir2.age); // 30
```

```
        System.out.print(new Amir().age); // 20 temp
```

```
        System.out.print(Amir.age); // خط static Class Object
```

```
}
```

}

- آبجکتی که در دستور اول ساخته میشود یک آبجکت instance obj است. آبجکتی که دلولپر میسازد، از روی Class obj ساخته میشود و تکمیل میشود برای هر new جدگانه.
- آبجکتی که در دستور دوم ساخته میشود یک آبجکت temp است.
- آبجکتی که در دستور سوم ساخته میشود یک آبجکت static است. که فقط میتواند متدها یا متغیرهای static را کال کند که در Class object وجود دارد. دستور آخر خطای دارد و باید انتظار داشته باشیم خروجی 30 بدهد!!!! چرا؟؟ چون اگر قرار بود این اتفاق بیفتد باید age را static تعریف میکردیم . چون پرپریتی ای رو میشه اینطوری با شیئ Amir static فراخانی کرد که خودش static باشد.

## This

\*\*\*

برابر است با temp obj در یک This scope.

- یک سازنده، با کمک کلیدواژه this میتواند سازنده دیگری را فراخوانی کند
- در صورت وجود، این فراخوانی باید حتماً اولین دستور سازنده باشد

در متدهای static قابل استفاده نیست. (چون زمانی که این متده میخواهد بره روی ram، آبجکت instance This نمیتواند وجود داشته باشد).

با this میتوان متدهای static element و غیر static را call static کرد. (هم مقدار متغیرهای static را تغییر داد هم غیر static. هم میتوان متدهای static را کال کرد هم غیر (. static).

چرا در متدهای static قابل استفاده نیست ولی میشود آبجکت new کرد یا بعنوان آرگمنان گرفت؟  
چون دستور ساخت آبجکت یعنی وقتی خطوط این متدهای اجرا شد یک obj بساز. (temp instance یا  
ولی this یعنی obj‌ای که الان وجود داره؛ که در زمان رفتن متدهای static روی ram، چنین obj‌ای نمیتوانه  
وجود داشته باشد! هنوز خود class obj هم ساخته نشده.

\*\*\*\*

وقتی کلمه‌ی this استفاده شود و در نتیجه‌ی آن مقدار متغیری تغییر یابد:  
اگر آن متغیر static باشد که تکلیف معلوم است، برای همه‌ی اشیاء مقدار آن تغییر کرده است.  
اگر آن متغیر static نبود مقدار آن متغیر فقط برای obj‌ای که آن متدها را Call کرده تغییر میکند. حتی  
برای temp obj داخل یک Scop . مثال پایین.

```
class Amir{  
    int age = 10;  
  
    public void showAge(){  
        int age = 20;  
        this.age=30;  
  
        System.out.println(age); // 20  
        System.out.println(this.age); // 30  
        Amir amir = new Amir();  
        System.out.println(amir.age); // 10  
    }  
}
```

در سطح یک متده میگوید: در حال حاضر (الان) در این متده مقدار متغیری در سطح کلاس مثل `age` ، 30 است.

با دستور: `this.age=30;` فقط میتوان گفت مقدار `age` در حال حاضر در این متده این است(متنه برای حلقه ها) نه برای ساختن اشیائی که در این متده ساخته میشوند. نباید انتظار داشته باشیم خروجی سوم 30 بده! چرا؟! چون اگر قرار بود این اتفاق بیفتد باید `age` را `static` تعریف میکردیم تا مقدارش همه جا یکسان میبود.

## کاربرد `this` برای سازندها

- گاهی لازم است که یک سازنده، سازنده دیگری را فراخوانی کند (Code reuse)
- به خصوص از منظر استفاده مجدد از کد
- تا کدی که در یک سازنده نوشته شده، در سازنده دیگر تکرار (کپی) نشود
- یک سازنده، با کمک کلیدواژه `this` میتواند سازنده دیگری را فراخوانی کند
- در صورت وجود، این فراخوانی باید حتماً اولین دستور سازنده باشد
- مشخص میکنیم که دقیقاً کدام سازنده دیگر باید فراخوانی شود
- با کمک پارامترهای `this` کلیدواژه `this` کاربردهای دیگری هم دارد که بعداً خواهیم دید

6

## Object Class

## کلاس Object در جاوا

- کلاسی در جاوا با نام **Object** وجود دارد
- هر کلاسی در جاوا، زیرکلاس (فرزنده) **Object** است
- همه کلاس‌ها از **Object** ارثبری می‌کنند
- کلاس‌هایی که هنگام تعریف از کلیدواژه **extends** استفاده نمی‌کنند:
  - به صورت **ضمی** از **Object** ارثبری می‌کنند
  - مثلاً در تعریف **{...}** **class Parent{...}** انگار که نوشته‌ایم:
- **class Parent extends Object{...}**
- کلاس‌هایی که هنگام تعریف از کلیدواژه **extends** استفاده می‌کنند:
  - به صورت **غیرمستقیم** از **Object** ارثبری می‌کنند

### • کلاس Object متدهای آشنایی دارد:

- **toString** , **finalize** , **equals** , **hashCode** , ...
- هر کلاس جدیدی که ایجاد می‌کنیم:
  - امکانات **Object** را گسترش می‌دهد
  - و برخی از رفتارهای **Object** را تغییر می‌دهد
  - مثلاً برای **equals** یا **toString** تعریف جدیدی ایجاد می‌کنیم
- تعریف این متدها در کلاس **Object** (که معمولاً ناکارامد هستند) را تغییر می‌دهیم
  - یعنی **Override** می‌کنیم

## toString ()

متodo `toString` را بدين شکل پیاده سازی کرده که اگر برای `obj` یک `reference` شود `call` شود خروجی آدرس `reference` را بدهد.

`className@MemoryAddress`

ما متodo `toString` را بدين شکل `override` کرده اند که خروجی `Value` بدهد. میدانیم اگر `value` نگیرند، پیشفرض `null` میگیرد.

اجراي متodo `toString` روی حالت های مختلف `:reference`

```
Student s0;  
System.out.println(s0); // Syntax Error
```

روی ارجاع بدون `new` ، `Error` بر میگردد.

چون `new` باعث میشود `Class Obj reference` به `default constructor` ایجاد کند و `instance` میخواهیم آدرس کی رو بدمی؟! بسازد، پس حالا که `instance` نیست، میخواهیم آدرس کی رو بدمی؟!

```
Student s1 = new Student();  
System.out.println(s1); // reference : one.Student@eed1f14  
System.out.println(s1.age); // int : 0  
System.out.println(s1.id); // Long : null  
System.out.println(s1.name); // String : null
```

```
Two two = new Two();  
System.out.println("two.one = " + two.one); //null (composition)  
System.out.println("two.long = " + two.aLong); //null  
  
two.one=new One();  
two.aLong=10L;  
System.out.println("two.one = " + two.one); // pack.One@4eec7777  
System.out.println("two.long = " + two.aLong); // 10
```

اگر کلاس Student را override نکرده باشد:

```
Student[] arr1 = new Student[10];
System.out.println(arr1); // [Lone.Student;@eed1f14
System.out.println(arr1[0]); // null (composition)
arr1[0]=student1;
System.out.println(arr1[0]); // Student;@eeddsf9
```

```
int[] arr = new int[10];
System.out.println(arr[0]); // default: 0
```

بهتر است خودمان متدهای toString() را برای کلاسها override کنیم:

```
public String toString() {
    return "Circle [radius=" + radius + "]";
}
```

```
Circle [radius=2.0]
```

```
enum Status{ SENT, DELIVERED, PENDING }
public class SMS {
    private Status status;
    private final String msg;
    private final String from, to;
    public SMS(String msg, String from, String to) {
        this.msg = msg;
        this.from = from;
        this.to = to;
    }
    public void setStatus(Status status) {
        this.status = status;
    }
    public String toString() {
        return String.format("%s=>%s:%s(%s)", from, to, msg, status);
    }
}
```

۱- آیا شیء sms تغییرناپذیر است؟ خیر

۲- خروجی این قطعه برنامه چیست؟ 0912=>0935:Salam! (DELIVERED)

```
SMS sms = new SMS("Salam!", "0912", "0935");
sms.setStatus(Status.DELIVERED);
System.out.println(sms);
```

وقتی یک object با یک String + call toString کردن میشود :

## Equality check == op, ref

== مناسب برای مقایسه تساوی:

(ها wrapper Class نه) primitive value  
ها Object reference بین

:value مناسب برای مقایسه <=, >=, <, >

ها primitive

مربوط به State wrapper class (میدانیم String ها جزئیان نیست).

برای مقایسه state بین دو wrapper class ، هم میتوانیم از متدهای compareTo() استفاده کنیم و هم از . (==, <=, >=, <, >) بجز relation ops

برای مقایسه تساوی state دو Primitive class فقط میتوانیم از equals() استفاده کنیم.

## Equality equals (), state

برای مقایسه تساوی **state** دو object کاربرد دارد.

حتی برای Primitive Wrapper Classes ها باید از این استفاده کنیم و == رفنسها را مقایسه میکند.

ولی برای اینکار هر کلاسی باید خودش بر اساس مفهومی که از **state** دارد آنرا **override** کند.

متدهای زیادی از متدهای **Object Class** برای **equal** بصورت **default** پیاده سازی شده. ولی حتماً باید آنرا برای کلاس های

دلوپری override کنیم. چون، **state** را بررسی نمیکند، چون جوا نمیداند که مفهوم **state** در کلاس

هایی که ما میسازیم چگونه خواهد بود و بسیار متنوع است، پس بصورت دیفالت **reference** آنها یعنی آدرس

حافظه شیئه ها را بین دو obj (خروجی) برای هر کدام مقایسه میکند. همان کاری که عملگر

== میکرد.

متدهای زیادی از متدهای **equals()** برای پیاده سازی خود استفاده میکنند:

contains/ indexOf/ remove/ add in HashSet /

هایی که میتوانند **Object** را با **Object** مقایسه کنند.

```
String str1 = new String("Ali");
String str2 = new String("Ali");
String str3 = "Ali";
String str4 = "Ali";
    همه این اشیاء با هم equal هستند
    str1 == str2 ✗
    str2 == str3 ✗
str3 == str4 ✓
```

```
Integer int1 = new Integer(2);
Integer int2 = new Integer(2);
Integer int3 = 2; autoboxing
Integer int4 = 2;
    همه این اشیاء با هم equal هستند
    int1 == int2 ✗
    int2 == int3 ✗
int3 == int4 ✓
```

اگر wrapper class developer ها را new نکند و فقط از فرآیند autoBoxing استفاده کند، خود هندل میکند: obj هایی که state یکسان دارند در یک آدرس حافظه نگه میدارد، پس reference آن obj ها یکسان میشود. یعنی دو obj از state wrapper class مساوی، ref یکسان دارند.

ولی اگر new developer کند حتی اگر قبل خودش یک obj با همان state new کرده باشد و یا با ایجاد شده باشد، باز هم یک آدرس حافظه جدید برایش در نظر میگیرد. پس == برای مقایسه تساوی primitive wrapper class ها مناسب نیست و باید از equals() استفاده کنیم.

برای primitive wrapper class object هایی: اگر از autoBoxing استفاده کنیم خود جاوا پیاده سازی جاوا مساوی را در یک ref قرار میدهد.

```
Integer integer1 = 12;
Integer integer2 = 12;

System.out.println("+" + (integer1.equals(integer2))); // true
System.out.println("+" + (integer1 == integer2)); // (their hashCode) true

Set<Integer> mySet = new HashSet<Integer>();
mySet.add(integer1);
System.out.println("+" + mySet.contains(integer2)); // true
```

هنگام overload کردن متدهای equal() اگر ورودی متدهای Object را تعريف نکنیم در حقیقت آنرا کردیم!

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    MenuComponent that = (MenuComponent) o;
    return Objects.equals(url, that.url) &&
           Objects.equals(name, that.name);
}

@Override
public int hashCode() {
    return Objects.hash(url, name);
}
```

in Java 7:

## مثال

```
class Student {  
    private String name;  
    public Student(String name) {  
        this.name = name;  
    }  
}  
List<Student> list = new ArrayList<Student>();  
list.add(new Student("Ali"));  
System.out.println(list.contains(new Student("ALi")));  
راه حل: باید برای کلاس Student مناسبی پیاده کنیم  
public boolean equals(Object obj) {  
    Student other = (Student) obj;  
    if (!name.equals(other.name))  
        return false;  
    return true;  
}  
• البته متد equals فوق کامل و دقیق نیست  
• جزئیاتی مثل null بودن پارامتر را بررسی نمی‌کند  
• مثلاً:
```

## اهمیت تعریف متد equals در ساختمان داده‌های جاوا

- بسیاری از ساختمان داده‌های جاوا تساوی اعضای فهرست را بررسی می‌کنند
- مثلاً: متد contains به دنبال یک شیء مساوی شیء موردنظر می‌گردد
  - این کار با کمک متد equals انجام می‌شود
  - متد equals روی اشیاء فهرست فراخوانی می‌شود و شیء موردنظر به آن پاس می‌شود
  - متدی مانند remove(Object o) و indexOf(Object o) نیز equals را صدا می‌کنند
  - در مجموعه‌ها (مثل HashSet) تکراری بودن عضو جدید با کمک equals بررسی می‌شود
- بنابراین اگر بخواهیم ظرفی از جنس یک کلاس دلخواه داشته باشیم، باید متد equals مناسبی برای کلاس موردنظر پیاده‌سازی شده باشد

## IComparable and I Comparator, orderable class

اگر بخواهیم به یک کلاس معنای ترتیب بین instance هایش بدھیم ازین دو API میتوانیم استفاده کنیم.

\*\*\*\*

== مناسب برای مقایسه تساوی:

( wrapper Class ) نه primitive value  
Object reference بین

:value مناسب برای مقایسه <= , >=, <, >

primitive value ها

String ( میدانیم wrapper class جزئیان نیست ).

equals ()

مقایسه تساوی state دو Obj حتی primitive wrapper class، دو مثل Integer ها

## Comparable / Comparator های Interface

Obj دو state مقایسه

تعیین ترتیب برای obj هایی از یک نوع کلاس و مرتب سازی obj ها

جستجوی سریع بین obj ها

Primitive Wrapper Class و String interface Comparable را impl کرده اند.

Class → I Comparable → compareTo()

مشخص میکند **Default** به چه شکلی مقایسه و مرتب شوند؟ instance

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

## Comparable واسط

- اگر کلاسی **Comparable** را پیاده کند، یعنی برای اشیاء این شیء ترتیب معنا دارد
- متده است **compareTo**: شیء جاری را با پارامتر متده مقایسه میکند و یک عدد برمیگرداند
- منفی، صفر و مثبت بودن این عدد به ترتیب یعنی این شیء کوچکتر، مساوی یا بزرگتر از

```
class java.util.Date implements Comparable<Date>{  
    public int compareTo(Date anotherDate) {
```

مثال

```
        long thisTime = getMillisOf(this);  
        long anotherTime = getMillisOf(anotherDate);  
        return (thisTime < anotherTime) ? -1 :  
            (thisTime == anotherTime) ? 0 : 1);  
    }
```

برای هر کلاس جدید که ترتیب اشیاء در آن معنی و اهمیت دارد، Comparable کلاس را فرزند **compareTo** کنید و متده **compareTo** برای آن پیادهسازی کنید

```
//Deprecated Constructors:  
Date d1 = new Date(2015, 10, 21);  
Date d2 = new Date(2013, 7, 26);  
Date d3 = new Date(2013, 7, 26);  
System.out.println(d1.compareTo(d2));  
System.out.println(d2.compareTo(d1));  
System.out.println(d2.compareTo(d3));
```

1  
-1  
0

۸۰

ClassComparator → I Comparator → compare()

نوشتن ترتیب های جدید برای **Obj** های یک کلاس

## Comparator واسط

- گاهی می خواهیم اشیاء را با ترتیبی غیر از آن چه خودشان تعریف کرده اند مقایسه کنیم
- مثلاً کلاس دانشجو واسط Comparable را پیاده سازی کرده و متده **compareTo** را بر اساس معدل دانشجو تعریف کرده ولی ما می خواهیم فهرست دانشجویان را بر اساس «سن» مرتب کنیم (ترتیب بر اساس سن)
- گاهی نیز می خواهیم اشیائی را مقایسه کنیم که کلاسشن Comparable نیست
- در این موارد واسط **Comparator** را برای مقایسه این اشیاء پیاده سازی می کنیم

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

## مثال برای Comparator

```
class Student implements Comparable<Student> {
    int age;
    double grade;
    public int compareTo(Student s) {
        return (this.grade < s.grade) ? -1 :
            (this.grade == s.grade) ? 0 : +1);
    }
    public Student(int age, double grade) {
        this.age = age;
        this.grade = grade;
    }
} class StudentComparator implements Comparator<Student>{
    public int compare(Student s1, Student s2) {
        return s1.age < s2.age ? -1 : (s1.age == s2.age) ? 0 : +1;
    }
}
```

```
StudentComparator comparator = new StudentComparator();
Student s1 = new Student(21, 17.5);
Student s2 = new Student(20, 18.5);
System.out.println(s1.compareTo(s2));
System.out.println(comparator.compare(s1, s2));
```

-1  
1

ظرفها و مستعارهای داده

۸۲

```
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);
```

```

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



4004

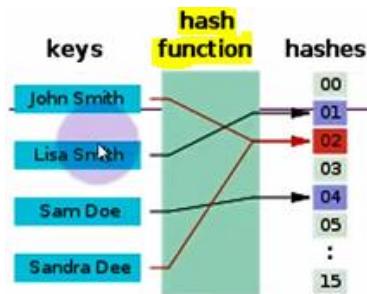
10-5,30

۱۳

## hashCode ()

کاربرد: محل ذخیره سازی هر obj در ram با خروجی متد hashCode مشخص میشود و در reference فرار میگیرد.

دارای متد hashCode() است. متد ( hashCode() ) از تکنیک hash استفاده میکند.



## ساختهای داده مبتنی بر Hash

- برخی از ساختهای داده‌های جاوا مبتنی بر تکنیک Hash هستند (HashMap و HashSet)

### تکنیک Hash

- از هر شیء که قرار است ذخیره شود، یک عدد صحیح استخراج شود
- این عدد صحیح (hash)، مبتنی بر ویژگی‌های داخل شیء محاسبه شود
- از hash برای محاسبه محل ذخیره شیء استفاده می‌شود
- ممکن است دو شیء مقدار hash مساوی داشته باشند
- ولی تابع hash مناسب اعدادی حتی الامکان متفاوت برای اشیاء متفاوت برمی‌گرداند
- دو شیء با ویژگی‌های مساوی، باید مقدار hash مساوی برگردانند (مقدار hash تصادفی نیست)

متدهای مناسب: hashCode

برای دو obj با state مقادیر (reference) همما (equals= true) یکسان تولید نماید.

اگر بارها برای obj که state ثابت مانده، مقدار hash خواستیم، همان خروجی قبل را بدهد.

برای دو obj متفاوت حتی الامکان مقادیر (reference) متفاوت تولید نماید.

متد **override** اگر نکرده باشیم (دیفالت Object Class) محل ذخیره obj را از متد **hashCode()** میپرسد و نشان میدهد.

متد **equals** اگر **override** اش نکرده باشیم (دیفالت Object Class) ، خروجی تابع hashCode() محل ذخیره سازی آنها (reference) را با هم مقایسه میکند نه state آنها را .

متد **hashCode()** در Object Class کارش را غلط انجام نمیدهد، هرچند هیچ دو obj را برابر فرض نمیکند که بخواهد مقادیر مساوی hash برایشان تولید کند مگر در موارد خاص autoBoxing و این باعث استفاده زیاد از رم میشود چون obj‌ها عموماً مساوی نیستند که یک ref بگیرند.

\*\*\*\*

متد **equal** را باید override کنیم تا دو obj با state یکسان را مساوی اعلام کند؛ قانون **hashCode** این است که برای دو obj مساوی مقدار hash یکسان بدهد، بنابر این باید **hashCode** override کنیم. اگر نکنیم در حالت عادی تعریف **hashCode** نقض کردیم، همچنان تعداد obj‌هایی که تولید میشوند زیاد است، و اگر از **data structure** استفاده کنیم درست کار نمیکند.

اگر فقط **equal** را override کنیم ولی همچنان **hashCode** بصورت default باقی بماند:

برای obj‌هایی با state مساوی، مقادیر متفاوت میدهد. مشکل بزرگی پیش نمیاید؛ جز مصرف حافظه

و اینکه قانون تابع hash نقض میشود  
ولی وقیکه که از **data structure** استفاده کنیم مشکل ساز میشود.

مثال اگر override equals() را نکنیم، حتماً به مشکل بر میخوریم:

```
15 public class Practice4 {  
16     public static void main(String[] args) {  
17         Rectangle r1 = new Rectangle(10, 5);  
18         Rectangle r2 = new Rectangle(10, 5);  
19  
20         System.out.println(r1.equals(r2));  
21  
22         List<Rectangle> list = new ArrayList<Rectangle>();  
23         list.add(r1);  
24         System.out.println(list.contains(r2));  
25     }  
26 }
```

Console >  
<terminated> Practice4 [Java Application] D:\java\jre8\bin\javaw.exe (Feb 5, 2016, 8:57:31 AM)  
false  
false

پس از تعریف مناسب فقط متد equal:

```
41 public class Practice4 {  
42     public static void main(String[] args) {  
43         Rectangle r1 = new Rectangle(10, 5);  
44         Rectangle r2 = new Rectangle(10, 5);  
45  
46         System.out.println(r1.equals(r2));  
47  
48         List<Rectangle> list = new ArrayList<Rectangle>();  
49         list.add(r1);  
50         System.out.println(list.contains(r2));  
51     }  
52 }
```

Console >  
<terminated> Practice4 [Java Application] D:\java\jre8\bin\javaw.exe (Feb 5, 2016, 9:00:11 AM)  
true  
true

چون متد contains در ArrayList فقط equal را چک میکند.

اگر از **data structure** های مبتنی بر **hash** استفاده کنیم وقتی فقط equals() پیاده سازی شده است:

```
43 public class Practice4 {  
44     public static void main(String[] args) {  
45         Rectangle r1 = new Rectangle(10, 5);  
46         Rectangle r2 = new Rectangle(10, 5);  
47  
48         System.out.println(r1.equals(r2));  
49  
50         Set<Rectangle> set = new HashSet<Rectangle>();  
51         set.add(r1);  
52         System.out.println(set.contains(r2));  
53     }  
54 }
```

Console >  
<terminated> Practice4 [Java Application] D:\java\jre8\bin\javaw.exe (Feb 5, 2016, 9:00:34 AM)  
true  
false

r2 اگر به set اضافه شود عضو تکراری محسوب نمیشود و جاداگانه در آن ذخیره میشود. حالا که نشده با وجود equal بودنش با r1، متد contains در HashSet؛ چون hashCode و equal را چک میکند میبیند چنین ای را ذخیره نکرده، پس میگوید ندارمش.

مشکل: چون `r1` و `r2` هستند باید `hash` آنها هم یکی میشد و وقتی یکیشون میرفت داخل `set` انگار هر دو رفته اند چون تکراری اند. اگر `hashCode` درست نمیشد مشکل ایجاد نمیشد.

نتیجه میگیریم برای ذخیره سازی اشیا در ساختمان داده های مبتنی بر `hash` باید هر دو را پیاده کنیم تا درست کار کند. دیدم برای `ArrayList` که مبتنی بر هش نبود مشکلی پیش نیامد.

اگر (`hashCode()` هم `override` کنیم :

```
43 public class Practice4 {  
44     public static void main(String[] args) {  
45         Rectangle r1 = new Rectangle(10, 5);  
46         Rectangle r2 = new Rectangle(10, 5);  
47  
48         System.out.println(r1.equals(r2));  
49  
50         Set<Rectangle> set = new HashSet<Rectangle>();  
51         set.add(r1);  
52         System.out.println(set.contains(r2));  
53     }  
54 }
```

Console >>  
<terminated> Practice4 [Java Application] D:\java\jre8\bin\javaw.exe (Feb 5, 2016, 9:01:42 AM)  
true  
true

اگر برای دو شیی خروجی متدهای `equal` و `hashCode` هم یکی باشند آن یعنی آن دو شی در حقیقت یکی هستند و یک جا در حافظه باید داشته باشند. آن یک جا از قبل با متدهای `equal` و `hashCode` مشخص شده. پس حتما باید برای آن دو `obj` مقدار خروجی `hashCode` هم یکی است.

نمونه پیاده سازی (`hashCode()` : سرج کن راه های بهتر؟؟؟

## مثال

```
class Student {  
    private String name;  
    public Student(String name) {  
        this.name = name;  
    }  
}  
Set<Student> set = new HashSet<Student>();  
set.add(new Student("Ali"));  
System.out.println(set.contains(new Student("ALi")));  
false  
راه حل:
```

- باید برای کلاس Student هم متدهای hashCode و equals مناسبی پیاده کنیم
- پیاده سازی equals کافی نیست، زیرا HashSet مبتنی بر hashCode کار می کند
- مثلاً:

```
public int hashCode() {  
    return 31 + ((name == null) ? 0 : name.hashCode());  
}
```

or

```
17 @Override  
18 public int hashCode() {  
19     final int prime = 31;  
20     int result = 1;  
21     result = prime * result + length;  
22     result = prime * result + width;  
23     return result;  
24 }  
25  
26 @Override  
27 public boolean equals(Object obj) {  
28     if (this == obj)  
29         return true;  
30     if (obj != null)  
31         return false;  
32     if (getClass() != obj.getClass())
```

```
Console  
<terminated> Practice4 [Java Application] D:\java\jre8\bin\javaw.exe (Feb 5, 2016, 9:01:42 AM)  
true  
true
```

## Casting

ایجاد یک **ref** جدید، یک **obj** جدید در

با تغییر دادن **type** کلاس یک **obj** قدیمی **heap**

برای تبدیل انواع به نوع **String**:

تبدیل مقدار عددی به رشته ای:

```
Int a = 7;
```

```
String b = a + "";
```

برای تبدیل همه ی انواع به نوع رشته ای این روش کاربرد دارد

```
Double d = 1.1;
```

```
String b = d;"" +
```

برای تبدیل نوع **String** به انواع دیگر:

تبدیل مقدار رشته ای به عددی:

```
String a="12";
```

```
int b = Integer.parseInt(a);
```

```
int c = b+2;
```

همه ی کلاسهای داده ای دارای متدهستند برای تبدیل نوع **String** به نوع خودشان.

```
String a = "true"
```

```
Boolean b= Boolean.parseBoolean(a);
```

## Encapsulation

Data and method **Hiding**, access control **modifiers**

as Java Bean (private members and public **getter/setter**)

Class های دیگر چه چیزهایی از یک کلاس را در دسترس داشته باشند.

Class، پنهان سازی field های یک نوع obj state (و در نتیجه Class) از دید یک نوع

دیگر

: بوسیله

تعیین سطح دسترسی غیر **public** برای **field** ها (PA و Private)

تعیین متدهای **getter/ setter** برای آن **field** ها (mutate/ access، اصلاً نویسیم) برای آن **field** ها (public, private)

Advantages of encapsulation (getter setter): \*\*\*

مدیریت میزان دسترسی به **field** : read/ write only

مدیریت مقدار دهی به **field** ها

مدیریت اعتبار سنجی مقدار دهی به **field**

مدیریت مقدار دهی به سایر **field** ها

تغییرات در نوع **field** کلاس از دید انواع Class های دیگر پنهان میماند.

چرا **getter** و **setter** ، بصورت **public** تعریف میشوند:

چون اگر استفاده کننده داخل pack باشد، فرقی بین سه نوع سطح دسترسی نیست.(مراجعه به بخش مربوطه)

اگر هم خارج pack باشد ، فقط در صورتی دسترسی بهش هست که **public** باشد.

پس **public** تنها راه است.

```
&& a < 150)
```

- امکان اعتبارسنجی در setter ها
- اجازه هر مقداری را ندهیم.

- امکان شبیه‌سازی ویژگی‌هایی که در واقع وجود ندارند  
براساس setAge و getAge پنهان «تاریخ تولد»

- محدود کردن نحوه دسترسی
- مثلاً برای یک ویژگی خاص public getter را تعریف کنیم  
و private setter را تعریف کنیم (یا اصلاً تعریف نکنیم)

## درباره setter و getter

- هنگام تعریف کلاس‌ها در فرایند محصورسازی (Encapsulation)

- معمولاً ویژگی‌ها (Property) به صورت private تعریف می‌شوند

- برای تغییر ویژگی‌ها، متدهای setter تعریف می‌شوند

- برای دریافت مقدار ویژگی‌ها متدهای getter تعریف می‌شوند

- متدهای getter و setter به صورت public تعریف می‌شوند

- به متدهای accessor ، getter هم گفته می‌شود

- به متدهای mutator ، setter هم گفته می‌شود

## Polymorphism

The ability of a class to provide **different implementations of a method**

**Overloading:** compile time binding, depending on the **ref** of object

**Overriding:** runtime binding, depending on the **type** of object that the method is called on it. (inheritance/ abstraction)

**Compile time polymorphism:** overloading

**Run time polymorphism:** overriding in: inheritance/ abstraction

Advantages of polymorphism:

- Programmers code can be **reused** via Polymorphism.
- Supports a single variable name for multiple data types.
- **Reduces coupling** between different functionalities.

- معماری ها همه با poly قابل پیاده سازی اند.  
- هر مزیتی که inheritance/ abstract/ interface/overloading میدهد، بصورت کلی مزیت polymorphism است!

دو کلاس فرزند، یک کلاس پدر را ارث میبرند و متدهای آن را override میکنند. با اجرای متدهای روی شیی پدر، در زمان اجرا مشخص میشود کدام ریخت از متدهای خواهد شد. (Inheritance نوعی poly را ممکن میکند)

دو کلاس فرزند، یک کلاس پدر abstract را ارث میبرند و متدهای آن را override میکنند. با اجرای متدهای روی شیی پدر، در زمان اجرا مشخص میشود کدام ریخت از متدهای خواهد شد.

دو کلاس یک Interface را پیاده سازی کنند و هر دو یک متادانه برای خودشان پیاده سازی کنند. با اجرای متدهای روی interface، در زمان اجرا مشخص میشود کدام ریخت از متدهای خواهد شد.

## جایگاه وراثت در طراحی نرم افزار

- وراثت: راهی برای ایجاد کلاس‌های جدید با کمک کلاس‌های موجود
- استفاده مجدد از ویژگی‌ها و رفتارهای کلاس اصلی در کلاس جدید
- ایجاد امکانات جدید در کلاس جدید: زیرکلاس، آبرکلاس را توسعه می‌دهد (extends)

Composition

- راههای دیگری هم وجود دارد

- مثلاً استفاده از یک کلاس به عنوان نوع یک ویژگی
- زیرکلاس: گروه محدودتری از اشیاء (نمونه‌ها) را در بر می‌گیرد
- همه این اشیاء رفتار و ویژگی‌های آبرکلاس را دارند
- اما برخی از رفتارها در زیرکلاس تغییر می‌کند
- ممکن است ویژگی‌ها و رفتارهای جدیدی هم در زیرکلاس تعریف شوند

## با وجود امکان چندریختی

```
Drawable[] drawables = ...  
for (Drawable drawable : drawables) {  
    drawable.draw();  
}
```



- یک آرایه از جنس آبرکلاس می‌سازیم
- همه اشیاء را در این آرایه قرار می‌دهیم
- متدهای draw را برای اعضای این آرایه فراخوانی می‌کنیم

## Polymorphism

```
public class Vehicle extends Object  
public class Car extends Vehicle  
public class Benz extends Car
```



```
Benz benz=new Benz();  
Car car=benz;  
Vehicle vehicle=benz;  
Object object=benz;
```

چرا poly در جاوا میتواند وجود داشته باشد؟

چون:

- امکان چندریختی در زبان‌های شیء‌گرا:
- متدهی روی یک شیء فراخوانی می‌شود
- نوع دقیق شیء در زمان اجرا مشخص می‌شود
- در زمان اجرا رفتار دقیق این شیء (با توجه به نوع آن) معلوم می‌شود

```
Animal a ;
if(X) a = new Cat();
else a = new Fish();
a.move("right", 3.0);
```

مثال:

- ایجاد امکان چندریختی، از عهده کامپایلر بر نمی‌آید
- دقت کنید: کامپایلر نمی‌داند یک ارجاع، به شیئی از چه کلاسی اشاره خواهد کرد

```
Animal a ;
if(X) a = new Cat();
else a = new Fish();
a.move("right", 3.0);
```

در زمان اجرا مشخص می‌شود:

شیئی که یک متغیر به آن ارجاع می‌دهد و نوع (کلاس) این شیء

بسیار مهم: برخی کارها در زمان اجرا و برخی در زمان کامپایل انجام می‌شوند

### • Compile time & Runtime

## رفتار چندریخت (Polymorphic Behavior)

- دیدیم که ممکن است ارجاعی از نوع آبرکلاس، به شیئی از نوع زیرکلاس اشاره کند
- مثلاً: Person p = new Student(); و یا Animal a=new Dog();
- اگر یک متدهی از چنین ارجاعی فراخوانی شود، متدهی آبرکلاس اجرا می‌شود یا متدهی زیرکلاس؟
- مثالاً متدهی a.move از Animal move را اجرا می‌کند یا همین متدهی Dog ؟
- چندریختی: نوع دقیق شیء تعیین کننده رفتار شیء است، نه نوع ارجاع آن

```
Shape s = new Rectangle();
s.draw();
double d = s.getArea();

Circle c = new Circle();
s = c;
s.draw();
d = s.getArea();
```

توجه کنید: بخش‌هایی از برنامه مقابل، ظاهری یکسان ولی رفتاری متفاوت دارند

واسطی یکسان که شکل‌های مختلف رفتار را ایجاد می‌کند

به این وضعیت چندریختی می‌گویند

## **binding**

هر **ref** که یک method را call کرده، مربوطه به چه **Class Type** است؟ تا مشخص شود کدام فرم از **method** باید اجرا شود.

### **Binding in compile time:**

در compile Time polymorphism استفاده میشوند بعضی از متدها در زمان compile مشخص میشود روی چه نوع obj‌ای اجرا میشوند. آنها همیشه با obj کلاسی است که در آن تعریف شده اند و احتمال دیگری وجود ندارد.) bind Overloading متدها در compile time،

متدهایی که قابل override کردن نیستند bind، compile time (private, static, final) در میشوند. متدهای final یا static یا private باشند، آنها همیشه با obj binding از نوع کلاسی است که در آن تعریف شده اند و احتمال دیگری وجود ندارد. این متدها همیشه binding compile time هستند.

the compiler determines the method to be called based on **the declared type at compile-time**. the specific method to be executed is determined before the program runs.

یعنی وقتی برنامه compile میشود نوع obj‌ای که دارد مت را کال میکند کاملا declared میشود

### **Binding in run time:**

در runtime polymorphism استفاده میشوند. برای متدهایی که قابلیت override دارند ( یعنی private, static, final )

پس جاوا در compile time نمتواند مطمین باشد این مت روی چه نوع obj‌ای در نهایت اجرا میشود، چون ممکن است مثلا override شود و نوع ref فرزند را بخواهد اجرایش کند.

حتی ممکن است از poly هم استفاده نشود ولی باز هم جاوا در runtime ، binding را مشخص خواهد کرد چون خبر ندارد که poly‌ای رخ داده یا نه. پس اکثرها binding run time هستند.

Runtime binding occurs when the method to be called is determined based on the **actual type of the object at runtime**.

## انقیاد (Binding)

```
Animal a ;  
if(X) a = new Cat();  
else a = new Fish();  
a.move("right", 3.0);
```

- مثلًا در کد روبرو باید مشخص شود دقیقاً کدام متد move فراخوانی می‌شود

- متدهای که در Animal تعریف شده؟ یا متدهای که در Cat یا در Dog تعریف شده؟

- انقیاد متدهای (method binding): تعیین متدهای فراخوانی شده است

- بدینهی است که برای اجرای یک متد باید این کار (binding) انجام شود

- گاهی این کار به سادگی در زمان کامپایل ممکن است

- مثلًا وقتی یک متد خصوصی (private) را فراخوانی می‌کنیم

- گاهی این کار در زمان اجرا قابل انجام است

- مثلًا وقتی از چندربیختی استفاده می‌کنیم (مثل کد فوق)

## انقیاد در زمان کامپایل

- Compile-time binding یا Static binding یا Early binding

- زمانی که ابهامی در تشخیص متدهای فراخوانی شده، وجود ندارد

- کامپایلر به سادگی می‌فهمد دقیقاً چه متدهای فراخوانی شده

- مثال: فراخوانی f() در زمان کامپایل مقید می‌شود

```
class StaticBinding{  
    private void f(){...}  
    public void g(){  
        f();  
        ...  
    }  
}
```

- متدهای final یا private static bind می‌شوند
- چرا؟

## انقیاد در زمان اجرا

- Runtime binding یا Dynamic binding یا Late binding

• گاهی کامپایلر نمی‌تواند متدهای فراخوانی شده را تشخیص دهد

• مثلاً وقتی متدهای را روی ارجاعی از نوع آبرکلاس فراخوانی می‌کنیم

• مثال:

```
Animal a ;  
if(X) a = new Cat();  
else a = new Fish();  
a.move("right", 3.0);
```

• متدهای واقعی در زمان اجرا bind می‌شود

• زیرا ممکن است override شده باشد

## خروجی این قطعه کد چیست؟

```

Child child = new Child();
Parent parent = new Parent();
Parent parentRefToChild = new Child();

parent.f();
child.f();
parentRefToChild.f();

```

Output:  
f() in Parent  
f() in Child  
f() in Child

## خروجی این قطعه کد چیست؟

```

SomeClass square = new SomeClass();
square.method(parent);
square.method(child);
square.method(parentRefToChild);

```

Output:  
method(Parent)  
method(Child)  
**method(Parent)**

- نکته مهم:
- رفتار چندریختی برای ارجاع
- ارجاع قبل از نقطه
- نه برای پارامترها

برای پارامتر، ارجاع مهم است نه نوع شبی.

\*\*\*

## Overloading

compile time polymorphism , is resolved by the compiler

با تغییر پارامتر های یک متد، چند متد هم نام داشته باشیم.

در overloading پارامترها باید تغییر داده شوند.

Overloading در زمان compile time میشود، یعنی ref یک object رفتار یا متدی که باید اجرا شود است. یعنی در زمان compile معلوم باشد دقیقاً obj های چه کلاسی (فقط یک کلاس که ref نشان میدهد) میتوانند این متد را call کنند.

Answer: **Method Overloading Rules:** Two methods can be called overloaded if they follow below rules:

- Both have **same** method **name**
- Both have **different** **arguments**

If both methods follow above two rules, then they **may or may not**:

- Have **different** **access modifiers**
- Have **different** **return types**
- Throw **different** checked or unchecked exceptions

بعد از تغییر پارامترها میتوان نوع برگشتی را هم تغییر داد.

نمیتوان یک متده را با فقط نوع برگشتی overloading کرد بدون اینکه اول در پارامترها تغییر داد ( فقط نوع برگشتی overloading متفاوت باشند )

وقتی در یک کلاس چندین متده هم نام (نوع برگشتی مهم نیست) داریم، حتما در نوع یا تعداد پارامترها متفاوتند.

وقتی آن متده call میشود، متده اجرای آرگمانهایش با پارامترهای ارسالی نطابق داشته باشد.



## Overriding

نقض معنای متده پدر در فرزند، آنطور که فرزند میپسندد رفتار تغییر میکند.

در overriding پارامترها باید تغییر نکنند

از یک متده Superclass، فقط یک متده override شده میتوان در subclass ساخت.

تغییر سطح دسترسی در overriding فقط میتواند افزایشی باشد. (is a)

هنگام overriding میتوان کلاس Exception ای که throws میشود را خاص تر کرد.

متده override شده نمیتواند return type را عوض کند مگر اینکه نوع خاص تر (فرزنده) را نسبت به پدر برگرداند.

در زمان **bind** میشود، یعنی **object type** در زمان اجرا تعیین کننده رفتار یا متدهای که باید اجرا شود است. یعنی در زمان **compile** نیازی نیست معلوم باشد دقیقاً **obj** های چه کلاس‌هایی میتوانند این متدها را کنند. بلکه این وابسته است به اینکه در **runtime** این متدها روی **obj** چه **Class Type** ای **call** شده است.

امکان اجرای فرم‌های مختلفی از یک **method** وجود داشته باشد.

- It must have **same method name** as that of parent class method
- It must have **same arguments** as that of parent class method
- It must have either the **same return type or covariant return type** (child classes are covariant types to their parents)
- It must **not throw broader checked exceptions**
- It must not have a **more restrictive access modifier** (if parent method is public, then child method cannot be private/protected)

نوع **reference** تعیین کننده رفتار در **runtime** نیست، بلکه **Class Type** ای که به آن اشاره میکند تعیین کننده رفتار (چه متدهای اجرا شود) است. ولی در **overloading** بر عکس است.

چگونه از **run time binding** آگاه شویم؟ یکی از راه‌ها **instanceOf** است / راه دوم **getClass** است

این متدها قابل **override** کردن نیستند:

override کردن:

پیاده‌سازی و بدنۀ دادن به متدهای **abstract** و **interface** در **Child** و **Parent** کردن متدهای **Override**

## Overriding VS Overloading

\*\*\*\* در **overloading** پارامترها باید تغییر داده شوند. در **overriding** پارامترها باید تغییر نکنند تغییر نوع برگشتی و **Exception** در **overloading** ممکن است، ولی در **overriding** فقط میتواند خودش و یا خاصتر را برگرداند.

تغییر سطح دسترسی در **overloading** ممکن است، ولی در **overriding** فقط میتواند افزایشی باشد

**Overloading** در زمان compile time ، bind میشود، یعنی ref یک object تعیین کننده ی رفتار یا متodi که باید اجرا شود است. ولی **Overriding** در زمان run time ، bind میشود، یعنی object type در زمان اجرا تعیین کننده رفتار یا متodi که باید اجرا شود است.

compile-time type of the reference

VS runtime type of the object

**Subclass** میتواند متدهای **overload** یا **override** را **overload** کند. میتوان از هر متدهایی که در **Superclass** تعریف شده باشند، یک متدهایی داشت که در **Subclass** تعریف شده باشند. این متد را **override** میگویند.

متدهای static overload قابل کردن هستند:

چون هر دو binding **compile time** هستند.

مشخص compile time overloaded static methods class Object شده در قرار میگیرند و در مشخص است چه Class type ای میتواند اجرایشان کند.

متدها static override قابل کردن نیست:

چون static method در compile time bind میشود و در run time مشخص باشد چه obj ای میتواند اجرا کند overriding کند. ولی نیاز به این دارد که در run time مشخص شود چه obj type ای متواند اجرا کند باشد که اینها با هم در تناقض هستند پس شدنی نیست.

because method overriding is based on dynamic binding at runtime and the **static methods** are bonded using static binding at **compile time**. So, we cannot override static methods.

**Method hiding** قابل Redefine static subclass کردن در متدهایی است: \*\*\*\*

در زمان compile مشخص شده است که obj های sup و sub هر کدام میتوانند آن static method خود را فقط ببینند و اجرا کنند، یعنی اگر ref اشاره میکند به نوع sub ، متادستاپ فرزند اجرا میشود و اگر ref اشاره میکند به نوع sup ، متادستاپ فرزند پدر اجرا میشود.

When a static method is called, the method implementation is determined by the compile-time type of the reference, not the runtime type of the object.

the subclass method **hides** the superclass method

ظاهرش شبیه compile time Polymorphism هست و عملکردش شبیه runtime Polymorphism است، ولی کلا در دسته بندی Polymorphism نماید.

در مورد exception

## زیر کلاس ممکن است:

۱- متدها یا ویژگی‌های جدیدی تعریف کند

• ویژگی تعداد گلزده و رفتار شوت زدن در ورزشکار نیست و در فوتبالیست اضافه می‌شود

۲- از متدها و ویژگی‌های آبرکلاس استفاده کند

• استفاده از ویژگی‌هایی که قبل از انسان تعریف شده در تعریف متدهای کارمند

• فراخوانی متدهایی که در ورزشکار تعریف شده برای شبیه از نوع فوتبالیست

۳- برخی رفتارها را تغییر دهد: پیاده سازی برخی متدها در زیر کلاس تغییر پیدا کند



• به تغییر معنای یک متدهای زیر کلاس، **Override** کردن متدهای گویند

• مثلاً خارپشت بی دندان نوعی پستاندار است که با تخمگذاری تولید مثل می‌کند

• متدهای تولید مثل که در کلاس آبرکلاس (پستاندار) به شکل «جهه زایی» پیاده شده

در زیر کلاس (خارپشت بی دندان) به صورت «تخم‌گذاری» تغییر می‌کند (override)

• نکته: زیر کلاس نمی‌تواند ویژگی یا متدهای آبرکلاس را حذف کند

• هرگاه یک زیر کلاس می‌سازیم و متدهای **Override** می‌کنیم:

حق نداریم سطح دسترسی به این متدهای **Override** را کاهش دهیم و گرنده: خطای کامپایل

• مثلاً نمی‌توانیم متدهای در آبرکلاس **public** بوده را

در زیر کلاس، **private** تعریف کنیم (override کنیم)

• چرا؟ زیرا این کار قانون **IS-A** را نقض می‌کند

• هر شبیه از زیر کلاس، شبیه از جنس آبرکلاس هم هست

• هر رفتاری که در آبرکلاس هست، باید برای اشیاء زیر کلاس هم قابل فراخوانی باشد

• مثلاً اگر «غذاخوردن» یک متدهای **public** در کلاس «حیوان» است:

○ یعنی هر حیوانی این رفتار را دارد

○ مثلاً کلاس سگ نمی‌تواند این متدهای **public** را مخفی (غیرقابل فراخوانی) کند

• پس سطح دسترسی به متدهای در زیر کلاس‌ها قابل کاهش نیست

## حاشیه‌نگاری @Override

مفهوم حاشیه‌نگاری (Annotation)

- توضیحاتی که با `@` شروع می‌شوند (Metadata)
- شكلی از فراداده (Override)
- توضیحی درباره یک متدهای کلاس یا ... می‌دهد
- بر نحوه کامپایل یا اجرای آن تأثیر می‌گذارد

مثال: `@Override`

- قبل از تعریف یک متدهای آید
- تصریح می‌کند که این متدهای همین متدهای آبرکلاس را `Override` می‌کند
- اگر به درستی متدهای نظر را `Override` نکنیم: خطای کامپایل رخ می‌دهد
- این تصریح، می‌تواند اشتباهاتی ناخواسته برنامه‌نویس را کمتر کند

## مثال برای @Override

```
class Animal {  
    public void talk(){}}  
}  
class Dog extends Animal{  
    private String name;  
    @Override  
    public String toString() {  
        return name;  
    }  
    @Override  
    public void talk() {  
        System.out.println("Hop!");  
    }  
}
```

- چه می‌شد اگر به اشتباه، هنگام تعریف کلاس `Dog` و یا `toString()` را `toSrtting` تایپ می‌کردیم؟
- یا پارامترهایی برای `talk` در نظر می‌گرفتیم؟
- حاشیه‌نگاری `@Override` کمک می‌کرد تا این اشتباه را کشف کنیم:
- یک خطای کامپایل ایجاد می‌کرد

دقت کنید: مفهوم **Override** و مفهوم **Overload** متفاوت هستند

**Overload** (سریار کردن):

چند متدهای همان با امضاهای مختلف (مجموعه پارامترهای متفاوت)

```
int f(){return 2;}  
int f(int a){return a;}
```

**Override** (لغو کردن):

امضای متدهای متفاوت (مجموعه پارامترها) در زیرکلاس دقیقاً مثل امضا آن در آبرکلاس است

معنای متدهای که در آبرکلاس وجود داشته، تغییر می‌کند: معنی قبلی لغو می‌شود

نمونه‌های (اشیاء) زیرکلاس، رفتار تغییریافته را استفاده می‌کند

در یک کلاس می‌توانیم متدهای همان کلاس را **Overload** کنیم

در یک زیرکلاس می‌توانیم متدهای آبرکلاس را هم **Overload** کنیم

ولی **Override** مربوط به وراثت است (در زیرکلاس رخ می‌دهد)

در کلاس فرزند هر دو قابل انجام اند:

## Override یا Overload

```
class Parent{
    public void method(Parent p){
        System.out.println("method(Parent)");
    }
}
class Child extends Parent{
    public void method(Child p){
        System.out.println("method(Child)");
    }
}
```

Overload در کلاس Child method() شده است •

Override نشده است •

دو متده مستقل: متده دوم معنی اولی را لغو نکرده است



- وقتی متده equals() را برای یک کلاس تعریف می‌کنیم
- در واقع در حال override کردن این متده هستیم
- زیرا این متده در کلاس Object تعریف شده است:

```
public boolean equals(Object obj) { return (this == obj); }
```

چرا در هنگام تعریف متده equals، Object را به عنوان پارامتر پاس می‌کنیم؟

مثالاً کلاس Person متده equals ای به این شکل دارد:

```
public boolean equals(Object obj) {...}
```

اما نه به این شکل: { ... } ای به این شکل دارد!

چرا؟!

زیرا در شکل دوم، این متده overload شود. در حالی که باید override می‌شود.



## **instanceOf**

چطوری از `run time binding` در `object type` یا آگاه شویم؟

- یکی از راه ها `instanceOf` است

- راه دوم `getClass` است

آیا `obj` ای که داخل این ارجاع هست از نوع کلاس `Type` (یا بچه هایش) است؟ `boolean` بر میگرداند

```
Ref a = ...  
a instanceof Type
```

## عملگر instanceof

- یک شیء (a) و یک کلاس (Type) می‌گیرد

- اگر a نمونه‌ای از کلاس Type (با زیرکلاس آن) باشد، true برمی‌گرداند

```
Animal x = ...  
if(x instanceof Cat){...}  
else if(x instanceof Dog){...}
```

- توجه: رابطه is a

- قاعدها باید زیرکلاس Ref باشد

- اگر همان کلاس Ref یا آبرکلاس Ref باشد: همیشه true، مگر...

- اگر Type آبرکلاس، زیرکلاس یا خود Ref نباشد: خطای کامپایل (همیشه غلط)

- این بررسی در زمان اجرا انجام می‌شود

- قبل از هر تغییر نوع به پایین (Downcast)، بررسی نوع انجام دهد

```
Animal x = ...  
if(x instanceof Cat){Cat c = (Cat)x; c.mew();}
```

- نکته: اگر ارجاع موردنظر null باشد، این عملگر false برمی‌گرداند

استفاده مهم: instanceof downCasting

## خروجی این برنامه چیست؟

پاسخ صحیح:

```
Animal a = new Cat("Maloos");  
System.out.println(a instanceof Object); true  
System.out.println(a instanceof Animal); true  
System.out.println(a instanceof Cat); true  
System.out.println(a instanceof Dog); false  
System.out.println(a.getClass().getName());  
ir.javacup.polymorphism.Cat
```

\*\*\*\*

## تفاوت ماهیت عملگر instanceof و شیء کلاس

```
if(c instanceof Person)...
```

- این دو دستور چه تفاوتی دارند؟

```
if(c.getClass().equals(Person.class))...
```

- دستور اول (عملگر instanceof :

- اگر c از نوع Person یا یکی از زیرکلاس‌های Person باشد، true برمی‌گرداند

- رابطه is-a را بررسی می‌کند

- دستور دوم (استفاده از شیء کلاس) :

- اگر c دقیقاً از نوع Person باشد، true برمی‌گرداند

- نکته: عملگر instanceof همانند متدهای است

```
if(Person.class.isInstance(c))...
```



## Casting with inheritance

تغییر نوع یک obj بین sub و Super کلاس‌ها

### Upcasting:

اشاره ref نوع پدر به obj type ref یا نوع فرزند

```
p = c;  
Parent p = new Child();
```

همیشه معتبر است.

Object type تعیین کننده رفتار است نه ref (البته اگه ref بداره method را ببیند).

در تصویر بالا sup Reference، متدها و پروپریتی‌هایی که superclass دارد دیده میشوند، اگر آنها را subclass override کرده باشد پیاده سازی subclass را میبینند. توسعه‌های دیگر subclass دیده نمیشود.

- Shape s = new Rectangle();
- Circle c = new Circle();  
Shape s = c;
- Animal a = new Dog();
- Person p = new Student("Ali", 9430623);

## Downcasting:

اشاره ref نوع فرزند به ref نوع پدر و قابل اجرا است که قبلش حتما باید یک Upcasting \*\*\*\* انجام شده باشد.  
پس قبل از هر downCasting بررسی نوع لازم است.  
ای از subclass میتواند به ref ای از superclass اشاره کند که، آن ref پدر به type obj ای از همان subclass اشاره کرده باشد.

```
Shape s = new Circle();
Circle c = (Circle) s; 
```

حالا s و c ، هر دو ref به یک obj از Circle هستند  
قبل از DownCasting برای جلوگیری از خطا زمان اجرا بررسی نوع لازم است.  
همه ی توسعه های subclass بعد از downCasting دیده میشوند.

\*\*\*\*

## Compile error:

```
Student st = new Person();
```

## Runtime error: exception

```
Student st = (Student) new Person();
```

یا

```
Person p = new Person();
```

```
Student st = (Student) p;
```

با نوشتن compile خطای runtime رفع میشود ولی خطای downCasting (Student) ایجاد میشود.

بدون error

```
Person p = new Student();  
Student st = (Student) p;
```

```
Shape s = new Circle();  
Circle c = (Circle) s; 
```

```
Shape s = new Rectangle();  
Circle c = (Circle) s;  خطای در زمان اجرا
```

برای جلوگیری از خطا زمان اجرا

- قبل از هر تغییر نوع به پایین (Downcast)، بررسی نوع انجام دهید

```
Animal x = ...  
if(x instanceof Cat){Cat c = (Cat)x; c.mew();}
```

```
Child c = new Child();  
Parent p = new Parent();
```

## نکته

- فرض کنید Child زیرکلاسی از کلاس Parent باشد

- به یاد داشته باشید که همواره شیء کلاس Child شیء Parent نیز هست

• رابطه is-a

• بنابراین این خطوط معتبر هستند:

```
p = c;  
Parent p = new Child();
```

• اما این خطوط نامعتبر هستند:

```
c = p;  
Child c = new Parent();
```

- مثال: Animal a=new Dog(); صریح ولی Cat c=a; غلط است

• هر سگی یک حیوان است (ارجاع a هم قرار است به یک حیوان اشاره کند)

• هر حیوانی لزوماً یک گربه نیست (ارجاع c قرار است به یک گربه اشاره کند)

• تأکید: درباره عملگر = صحبت می‌کنیم که «نوع» سمت چپ و راست آن متفاوت است

## تغییر نوع به بالا (UpCasting)

- گاهی از یک شیء، به عنوان شیئی از نوع آن کلاس استفاده می‌کنیم
- مثال:

```
• Shape s = new Rectangle();
• Circle c = new Circle();
Shape s = c;
• Animal a = new Dog();
• Person p = new Student("Ali", 9430623);
```

• به این کار، «تغییر نوع به بالا» یا Upcasting می‌گویند

- تغییر نوع به بالا همواره معتبر است
- کامپایلر جلوی آن را نمی‌گیرد (خطای کامپایل ایجاد نمی‌شود)



## تغییر نوع به پایین (DownCasting)

- اگر از یک شیء، به عنوان شیئی از نوع زیرکلاس استفاده کنیم:

• به این کار، «تغییر نوع به پایین» یا Downcasting می‌گویند

```
Shape s = ...
Circle c = s; X
```

- تغییر نوع به پایین همواره معتبر نیست (گاهی معتبر و گاهی نامعتبر است)

• بنابراین کامپایلر جلوی آن را نمی‌گیرد (خطای کامپایل ایجاد می‌شود)

• مگر این که صراحتاً از عملگر «تغییر نوع» (Cast) استفاده شود

```
Shape s = new Circle();
Circle c = (Circle) s; V
```

- در این صورت در زمان کامپایل خطای گرفته نمی‌شود

• اما ممکن است منجر به خطا در زمان اجرا شود

```
Shape s = new Rectangle();
Circle c = (Circle) s; X
    خطای در زمان اجرا
```



## رفتار چندریخت (Polymorphic Behavior)

- دیدیم که ممکن است ارجاعی از نوع آبرکلاس، به شیئی از نوع زیرکلاس اشاره کند
- مثل `Person p = new Student();` و یا `Animal a=new Dog();`
- اگر یک متاد از چنین ارجاعی فراخوانی شود، متاد آبرکلاس اجرا می‌شود یا متاد زیرکلاس؟
- منلاً متاد `a.move` را اجرا می‌کند یا همین متاد `Dog`؟
- چندریختی: نوع دقیق شی تعیین کننده رفتار شی است، نه نوع ارجاع آن

```
Shape s = new Rectangle();
s.draw();
double d = s.getArea();

Circle c = new Circle();
s = c;
s.draw();
d = s.getArea();
```

### فرض کنید:

```
class Animal{}
class Cat extends Animal{}
class Dog extends Animal{}

Object o = new Object();
Animal a = new Animal();
Animal x = new Cat();
Cat c = new Cat();
Dog d = new Dog();
```

- کدام دستورات خطای کامپایل ایجاد می‌کنند؟
- کدام دستورات خطای در زمان اجرا ایجاد می‌کنند؟ (هر دستور را مستقل فرض کنید)

خطای کامپایل

`a = o;`

`a = c;`

`c = o;`

`c = a;`

`c = d;`

خطای کامپایل نمیخورد چون (cast) را انجام داده، ولی خطای run میخورد:

```
: o;
a;
d;
c = (Cat) x;
d = (Dog) x;
```

خطای ClassCastException در زمان اجرا

```
class A {  
    int a;  
    A(int a) {  
        this.a = a;  
    }  
}
```

```
String s;  
A a = new A(1);  
B b = new B(2);
```

```
enum Weather{GOOD, BAD}
```

```
class B {  
    int b;  
    B(int b) {  
        this.b = b;  
    }  
    public String toString() {  
        return String.valueOf(b);  
    }  
}
```

- در هر مورد مقدار `s` را حدس بزنید:

<code>s=1+2;</code>	<code>s="1"+b;</code>	<code>s = ""+(1+2)+b;</code>
<code>s = "1"+"2";</code>	<code>s = 1+2+b;</code>	<code>s = ""+Weather.BAD;</code>
<code>s="1"+a;</code>	<code>s = ""+1+2+b;</code>	<code>s = a+b;</code>

\*\*\*

<b>Syntax Error</b>		
<code>s=1+2;</code>	<code>s="1"+b; 12</code>	<code>s = ""+(1+2)+b; 32</code>
<code>s = "1"+"2"; 12</code>	<code>s = 1+2+b;</code>	<code>s = ""+Weather.BAD;</code>
<code>s="1"+a;</code>	<code>s = ""+1+2+b;122</code>	<code>s = a+b; BAD</code>

```

public class Pedar {
    int p = 2;
    public void sayPedar() { System.out.println("i'm pedar"); }
    public void sayHello() { System.out.println("Pedar : salam"); }
}

```

```

public class Farzand extends Pedar {
    int f = 2;
    public void sayFarzand() { System.out.println("i'm farzand"); }
    @Override
    public void sayHello() {
        System.out.println("Farzand : salam");
    }
}

```

```

public class Main {
    public static void main(String[] args) throws Exception {
        Pedar pedar = new Pedar();
        Farzand farzand = new Farzand();

        Pedar pedar2 = new Farzand();

        pedar2.sayHello();
        pedar2.sayPedar();
        System.out.println("pedar2.p:" + pedar2.p);

        Pedar pedar3 = new Farzand();
        Farzand farzand2 = (Farzand) pedar3;
    }
}

```

```

Farzand : salam
i'm pedar
pedar2.p:2
Farzand : salam
i'm farzand
i'm pedar
farzand2.f:2
farzand2.p:2

```

We cant: pedar3.sayFarzand(); // syntax compile error

## Inheritance

Inheritance is a mechanism in Java where a subclass inherits the properties and methods of its superclass. This allows for **code reuse** and the creation of **more specialized classes**.

یک کلاس فقط میتواند یک پدر داشته باشد، ولی میتواند پدر بزرگ و پدر پد ربزرنگ و پدر ... داشته باشد.

در ارث بری، پدر در پکیج خودش باشد یا نباشد مهم نیست.

هر obj از نوع subclass instance است (**is a**). یک نوع از superclass هایی همی با خاطر field ها و method هایی بیشتری که از superclass دارد، دارای obj هایی **more specific** تر است.

نوع subclass بخارط field ها و method هایی بیشتری که از superclass دارد، دارای obj هایی **more specific** تر است.

Advantages od inheritance:

- Reusability
- more specialized classes
- Impl Polymorphism
- Design patterns

## Access Levels modifiers مراجعه به بخش

:ram در

ابتدا Class Object از Class Object subclass ساخته میشوند.

## inheritance در Static field

اگر پدر دارای Static متغیر باشد، آن متغیر در فرزندها چگونه خواهد بود؟ obj ای از فرزند آنرا تغییر دهد در پدر هم تغییر خواهد کرد؟ بله . آن متغیر برای هر دو کلاس پدر و پسر دارای یک ref هستند.

یک Class میتواند فقط از یک Class دیگر (abstract باشد یا نباشد مهم نیست) ، Extend کند.

یک کلاس همان میتواند، implement و extends کند.

یک Class میتواند یک یا چند Interface را Implement کند.

یک interface میتواند از یک یا چند Class extend ، interface کند.

## Constructor in inheritance

وقتی new استفاده میشود، default constructor صدای زده میشود و یک ref برای obj جدید ساخته میشود و داده میشود به Class Obj اش.

\*\*\*\*

اگر پدر و فرزند overload default constructor را نکرده باشند، بصورت **ضمی** اول default constructor را فرزند نمیتواند باشد، فرزند میتواند constructor های خود را داشته باشد. پدر و بعد فرزند call خواهد شد.

اگر پدر overload default constructor را نکرده باشد، فرزند میتواند constructor های خود را داشته باشد. در خط اول بصورت **implicitly** call خواهد شد.

اگر پدر حتی یک constructor تعریف کند، دیگر default constructor نخواهد داشت (override شده است). چون دیگر پدر default constructor ندارد که **ضمی** call شود:

- فرزند باید constructor تعریف کند و در خط اول آن مشخص کند بصورت **explicitly** کدام

constructor از پدر را میخواهد اول اجرا کند. پس ناچارا فرزند هم مجبور میشود default constructor خود را override نماید.

- پدر constructor بدون پارامتر هم بنویسد. پس بطور **implicitly** همچنان call میشود.

اگر یکی از اینها انجام نشود خطای compile میخورد.

## Super

دو کاربرد دارد:

- فراخوانی constructor کلاس superclass

- در متدهای subclass میتوانیم، از field ها و method های superclass که در فرزند override شده اند استفاده کنیم از کلمه کلیدی super استفاده میکنیم.

## مقداردهی اولیه شیئی از نوع زیرکلاس

- وقتی یک زیرکلاس تعریف می‌کنیم:

باید سازنده (constructor) مشخصی از آبرکلاس در سازنده زیرکلاس فراخوانی شود

۵ این کار با کلیدواژه `super` انجام می‌شود

فراخوانی سازنده آبرکلاس، باید اولین دستور از سازنده زیرکلاس باشد

و گرنه سازندهای بدون پارامتر از آبرکلاس به صورت ضمنی فراخوانی می‌شود

۵ اگر چنین سازندهای در آبرکلاس نباشد، خطای کامپایل ایجاد می‌شود

- نکته: سازنده‌ها به ارث نمی‌رسند

۶ مثلاً اگر سازندهای در آبرکلاس باشد که یک پارامتر `int` می‌گیرد

۷ این سازنده به زیرکلاس به ارث نمی‌رسد

۸ اگر زیرکلاس به چنین سازندهای نیاز دارد، باید آن را صراحتاً تعریف کند

```
class Person{
    private String name;
    private String nationalID;
    public Person(String name, String nationalID) {
        this.name = name;
        this.nationalID = nationalID;
    }
} class Student extends Person{
    private long studentID;
    public Student(String name, String id, long studentID) {
        super(name, id);
        this.studentID = studentID;
    }
}

Person p = new Person("Ali Alavi", "1290562352");
Student s = new Student("Ali Alavi", "1290562352", 94072456);
```

```
package ir.javacup.inheritance.practice;
> public class Parent {
+     private int a;
>
5@     public Parent(int a) {
?         this.a = a;
3     }
> }
>
L class Child extends Parent{
2 }
```

compile error

```

1 package ir.javacup.inheritance.practice;
2
3 public class Parent {
4     private int a;
5
6     public Parent(int a) {
7         this.a = a;
8     }
9     public Parent() {
10
11 }
12 }
13
14 class Child extends Parent{
15
16 }

```

ok

```

class Person{
    private String name;
    private String nationalID;
    public Person(String name, String nationalID) {
        this.name = name;
        this.nationalID = nationalID;
    }
}
class Student extends Person{
    private long studentID;
}

Person p = new Person("Ali Alavi", "1290562352");
Student s = new Student();

```

خطای کامپایل کجاست?

چون کلاس فرزند خودش constructor نساخته و super نزد، کانسٹراکتور پدون پارامتر پدر بطور ضمنی فراخوانی میشه و چون پدر اونو نداره خطای کامپایل میخوره

```

1 package ir.javacup.inheritance.practice;
2
3 public class Parent {
4     private int a;
5
6     public Parent(int a) {
7         this.a = a;
8     }
9 }
10
11 class Child extends Parent{
12
13     public Child(int a) {
14         super(a);
15         // TODO Auto-generated constructor stub
16     }
17 }

```

ok

This in super:

## سؤال

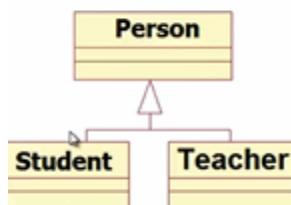
- در هنگام تعریف سازندهی زیرکلاس، در صورت فراخوانی سازندهی آبرکلاس (با کمک `super`). نمیتوانیم ویژگی‌های زیرکلاس را پاس کنیم
- مثال: `super(this.name)` (دچار خطای کامپایل می‌شود)
- چرا؟
- زیرا ویژگی‌های زیرکلاس، بعد از ویژگی‌های آبرکلاس آمده می‌شوند
- مقداردهی اولیه آن‌ها بعد از فراخوانی سازندهی آبرکلاس انجام می‌شود

آبرکلاس، نوع عام‌تری از زیرکلاس است (more general)

زیرکلاس، نوع خاص‌تری از آبرکلاس است (more specific)

تأکید: زیرکلاس و ابرکلاس هر دو «کلاس» هستند

هر شیء از زیرکلاس، شیئی از ابرکلاس هم هست



دانشجو زیرکلاس انسان است

دانشجو نوع خاص‌تری از کلاس انسان است

(دایره محدودتری از نمونه‌ها را شامل می‌شود)

همه ویژگی‌ها و رفتارهای انسان در دانشجو هم وجود دارد

مثل: نام، سن، غذاخوردن و ... البته دانشجو ویژگی‌ها و رفتارهای دیگر هم دارد

علی‌علوی یک دانشجو است (یک نمونه، شیء)، پس علی‌علوی، انسان هم هست

(اصطلاحات انگلیسی مهمتر هستند)

## واژه‌شناسی

کلاس اصلی:

(Base Class)

آبرکلاس (Superclass)

کلاس والد (Parent Class)

کلاس وارث:

(Derived Class)

زیرکلاس (Subclass)

کلاس فرزند (Child Class)

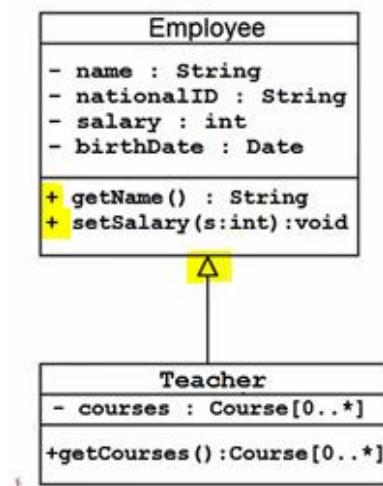
Rectangle extends Shape

• Rectangle is inherited/derived from Shape

• Rectangle is subclass/child of Shape

• Shape is the super-class/base-class/parent of Rectangle

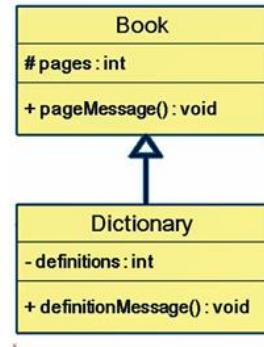
## نمودار UML برای کلاس‌ها



### UML Class Diagram •

- نموداری برای توصیف طراحی کلاس‌ها
- کاربردهای مختلفی دارد
- مثال: تعامل بین طراح و برنامه‌نویس
- نمودار UML قواعد خاصی دارد
- مخصوص زبان جاوا نیست
- نمودار UML شامل:
- متدها و ویژگی‌های کلاس‌ها
- سطوح دسترسی
- روابط بین کلاس‌ها
- (وراثت: یکی از انواع رابطه ممکن است)

## نمایش در UML



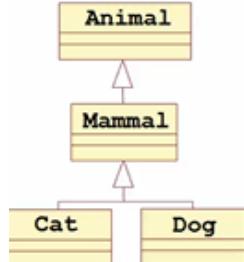
### UML Class Diagram در •

- اعضای `#` با: `protected`
- اعضای خصوصی: `بـ`
- اعضای عمومی: `بـ+`

اصطلاحاً: بین زیرکلاس و آنکلاس رابطه IS A برقرار است

- Rectangle is a Shape
- A rectangle instance is a Shape instance too

## سلسله مراتب کلاس‌ها



- زیرکلاس مستقیم:
- کلاس سگ زیرکلاس مستقیم کلاس پستانداران است
- زیرکلاس غیرمستقیم:
- کلاس گربه زیرکلاس غیرمستقیم کلاس حیوان است
- همه ویژگی‌ها و رفتارهای حیوان به گربه هم به ارث می‌رسد
- البته ممکن است کلاس پستانداران برخی از این متدها را تغییر داده باشد (Override)
- بدیهی است که یک کلاس می‌تواند چند زیرکلاس داشته باشد

### وراثت چندگانه (Multiple Inheritance): ارثبری از چند کلاس

البته جواه هم در شرایط  
خاصی امکان وراثت  
چندگانه را فراهم می‌کند

- در برخی زبان‌های برنامه‌نویسی ممکن است
- در زبان جاوا، یک زیرکلاس نمی‌تواند چند ابرکلاس داشته باشد
- یعنی یک کلاس نمی‌تواند از چند کلاس ارثبری کند

## جایگاه وراثت در طراحی نرم‌افزار

- وراثت: راهی برای ایجاد کلاس‌های جدید با کمک کلاس‌های موجود
- استفاده مجدد از ویژگی‌ها و رفتارهای کلاس اصلی در کلاس جدید
- ایجاد امکانات جدید در کلاس جدید: زیرکلاس، ابرکلاس را توسعه می‌دهد (extends)

ویرایشی: *Composition*

- راههای دیگری هم وجود دارد
- مثلًا استفاده از یک کلاس به عنوان نوع یک ویژگی
- زیرکلاس: گروه محدودتری از اشیاء (نمونه‌ها) را در بر می‌گیرد
- همه این اشیاء رفتار و ویژگی‌های ابرکلاس را دارند
- اما برخی از رفتارها در زیرکلاس تغییر می‌کند
- ممکن است ویژگی‌ها و رفتارهای جدیدی هم در زیرکلاس تعریف شوند

\*\*\*

## مثال

```

public class Sorting {
    public List sort(List list) {
        // sort implementation
        return list;
    }
} class DynamicDataSet extends Sorting {
    // DynamicDataSet implementation
}

```

- در برنامه فوق:

- طراح، کلاس DynamicDataSet را فرزند (زیرکلاس) Sorting قرار داده
- تا بتواند از امکانات مرتبسازی در کلاس DynamicDataSet استفاده کند

### اشتباه طراحی؟

- بین دو کلاس رابطه a is برقرار نیست. هر DynamicDataSet یک Sorting نیست

```

class DynamicDataSet{
    private Sorting sorting = new Sorting();
    //...
}

```

### روش بهتر: ترکیب

- شیوه از جنس DynamicDataSet در کلاس Sorting قرار گیرد

## کلیدواژه super

- گاهی در زیرکلاس می خواهیم از عضوی استفاده کنیم که در آبرکلاس تعریف شده
- مثلاً یک متدها یا ویژگی (متغیر) از آبرکلاس
- فرض کنید عضوی دقیقاً با همان نام در آبرکلاس هم وجود داشته باشد
- مثلاً متدهای موردنظر را در زیرکلاس override کرده باشیم
- در این شرایط با کمک نام این عضو، عضوی از همین کلاس فراخوانی می شود
- نه از آبرکلاس

### راه حل: کلیدواژه super

- با super می توانیم از اعضایی که در آبرکلاس تعریف شده اند استفاده کنیم
- برای فراخوانی سازنده ای آبرکلاس هم می توانیم از super استفاده کنیم

### مثال: super.f();

- با این کار متدهای f که در آبرکلاس تعریف شده فراخوانی می شود
- به ویژه اگر متدهای f در همین کلاس وجود داشته باشد

### مثال: s = super.name;

- با کمک super تصریح می کنیم که یک عضو از آبرکلاس موردنظر است
- با کمک this تصریح می کنیم که یک عضو از همین کلاس موردنظر است

- کاربرد مهم دیگر: فراخوانی سازنده ای آبرکلاس

### مثال: super(name, id);

- با این کار سازنده ای آبرکلاس فراخوانی می شود

```

class A {
    public int a;
}

public class B extends A{
    private int a;
    public void f() {
        int a ;
        this.a = 5;
        super.a = 6;
        a=4;

        System.out.println(a);
        System.out.println(this.a);
        System.out.println(super.a);
    }
    public static void main(String[] args) {
        new B().f();
    }
}

```

## خروجی این برنامه چیست؟

- نکته:
- مفهوم Override برای متدها معنی دارد
  - برای متغیرها معنی ندارد
  - تعریف ویژگی هایی در زیر کلاس که همانم ویژگی های آن کلاس هستند، کار رایجی نیست

### پاسخ صحیح:

4  
5  
6

اینکه فرزند پرопریتی هم نام با پدر داشته باشد override کردن محسوب نمیشود و کار رایجی نیست.

- مفهوم Override برای متدها معنی دارد
- برای متغیرها معنی ندارد
  - تعریف ویژگی هایی در زیر کلاس که همانم ویژگی های آن کلاس هستند، کار رایجی نیست

```

class Superclass{
    private void f(){ System.out.println("1"); }
    void f(int a){ System.out.println("2"); }
    protected void f(String a){ System.out.println("3"); }
    public void f(int a, int b){ System.out.println("4"); }
}

public class Subclass extends Superclass{
    public void f(){ System.out.println("5"); }
    protected void f(String a){ System.out.println("6"); }
    public void f(int a, int b){ System.out.println("7"); }
}

```

## خروجی هر یک از قطعه برنامه های زیر چیست؟

Subclass s = new Subclass(); s.f();      5	Superclass t = new Superclass(); t.f(1);      2
s.f(1);      2	t.f(1,2);    4
s.f("1");    6	t.f("1");    3
s.f(1,2);    7	

## Composition

Inheritance and Abstraction: is a

Composition: Has a

ساخت **composition** reference به واسطه

وقتی کلاس A دارای prop از کلاس دیگر باشد، وقتی کلاس A new شد خود compiler از کلاس های prop هم میسازد و در متغیر reference آنها null قرار میدهد.

```
Two two = new Two();  
  
System.out.println("two.one = " + two.one); //null  
  
System.out.println("two.long = " + two.aLong); //null  
  
  
  
two.one=new One();  
  
two.aLong=10L;  
  
System.out.println("two.one = " + two.one); // pack.One@4eec7777 (ref)  
  
System.out.println("two.long = " + two.aLong); // 10
```

## جایگاه و راثت در طراحی نرم افزار

- وراثت: راهی برای ایجاد کلاس‌های جدید با کمک کلاس‌های موجود
- استفاده مجدد از ویژگی‌ها و رفتارهای کلاس اصلی در کلاس جدید
- ایجاد امکانات جدید در کلاس جدید: زیرکلاس، ابرکلاس را توسعه می‌دهد (**extends**)

Composition

- راههای دیگری هم وجود دارد
- مثلاً استفاده از یک کلاس به عنوان نوع یک ویژگی
- زیرکلاس: گروه محدودتری از اشیاء (نمونه‌ها) را در بر می‌گیرد
- همه این اشیاء رفتار و ویژگی‌های ابرکلاس را دارند
- اما برخی از رفتارها در زیرکلاس تغییر می‌کند
- ممکن است ویژگی‌ها و رفتارهای جدیدی هم در زیرکلاس تعریف شوند

\*\*\*

```
public class Sorting {  
    public List sort(List list) {  
        // sort implementation  
        return list;  
    }  
}  
class DynamicDataSet extends Sorting {  
    // DynamicDataSet implementation  
}
```

### مثال

- در برنامه فوق:

- طراح، کلاس DynamicDataSet را فرزند (زیرکلاس) Sorting قرار داده
- تا بتواند از امکانات مرتبسازی در کلاس DynamicDataSet استفاده کند

### اشتباه طراحی؟

- بین دو کلاس رابطه is a برقرار نیست. هر DynamicDataSet یک Sorting نیست

```
class DynamicDataSet{  
    private Sorting sorting = new Sorting();  
    //...  
}
```

### روش بهتر: ترکیب

- شیئی از جنس Sorting در کلاس DynamicDataSet قرار گیرد

Advantages of composition:

- Reusability

Composition and inheritance make code reuse.

```
class Human{  
    private Heart heart;  
    private Hand leftHand;  
    private Hand rightHand;  
}
```

## ترکیب (Composition)

```
class Professor extends Human{  
    private University university;  
    private Course[] courses;  
}
```

ترکیب: روش دیگری برای استفاده مجدد (code reuse)

```
class Car extends Object{  
    private Engine engine;  
    private Tyre[] tyres;  
}
```

اگر بین دو شیء رابطه is a برقرار نیست، از وراثت استفاده نکنید

## درباره «ترکیب» (Composition)

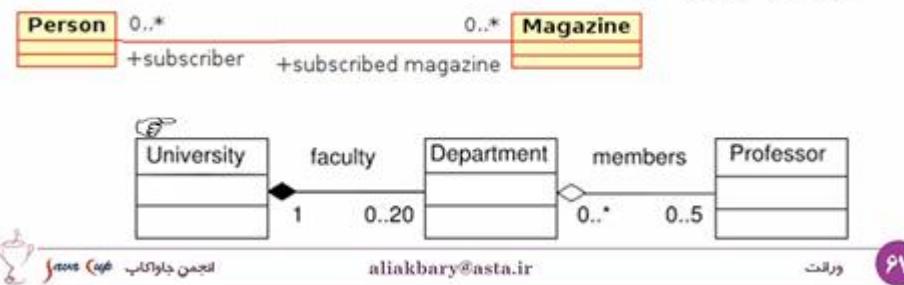
• در موقعي که شک دارد، Inheritance را بر Composition ترجیح دهد

• یک کلاس فقط از یک کلاس می‌تواند ارثبری کند

• ولی تعداد زیادی کلاس را می‌تواند در ترکیب به کار گیرد

Association, Composition, Aggregation •

نحوه نمایش در UML •



أنواع : اون سه تا بالا

## Abstract class

It can have abstract methods as well as concrete methods

If there is any abstract method in a class, then that class must be declared abstract  
cannot be instantiated

has constructor (first line of the subclasses constructor is always a call to the super  
class constructor)

subclasses have to implement all the abstract methods otherwise become abstract  
class

a convention between classes in our project, defining **common behavior for classes**.

Used for runtime poly

یکی از ابزارهای پیاده سازی polymorphism .Overriding

از کلاس abstract نمیتوان Class Object (برایش ساخته میشود) Instance Object ساخت.

فقط میتواند Parent Abstract class باشد.

کلاس abs میتواند متدهای abs داشته باشد (متدهای بدون بدن)

کلاسی که حتی فقط یک متدهای abstract داشته باشد و یا به ارث برید، یک کلاس abstract است.

میتوانیم کلاس abs بدون هیچ متدهای abs داشته باشیم: دلیل: تا نتوان از آن obj new کرد، و این کار را به عهده subclass ها گذاشت.

میتواند constructor داشته باشد. نه برای obj سازی، برای مقدار دهنده به فیلدها.

:Poly

از نوع کلاسی که abstract است، Ref ساخته میشود ولی یک obj از آن نمیتوان ایجاد کرد.

یک نوع abstract، میتواند به obj ای از subclass آن که abstract ref نیست اشاره کند.

کلاس فرزند که extend abs parent را کرده اختیار دارد همه ای abs method ها را override نکند، فقط باید پذیرد که خودش هم abstract شود و فقط بتواند parent باشد.

Advantages of abstract:

- Impl Polymorphism
- Reusability
- more specialize classes
- Design patterns

We can declare a constructor with no arguments in an abstract class .The purpose of the constructor in a class is used to initialize fields but not to build objects

```

abstract class Component{
    public abstract void show();
}

abstract class SelectableComponent extends Component{
    public abstract void select();
}

class Button extends SelectableComponent{
}

public class Application {
    public static void main(String[] args) {
        Component c = new Button();
        c.show();
        c.select();
    }
}

```

چون ref از **Component** است فقط متد **show()** را میتواند ببیند.

```

public class Application {
    public static void main(String[] args) {
        Component c = new Button();
        c.show();
        SelectableComponent s = (SelectableComponent) c;
        s.select();
        s.show();
    }
}

```

- کلاس انتزاعی: کلاسی که هیچ شیئی مستقیماً از آن ایجاد نمی‌شود
  - اگر شیئی از جنس این کلاس است، باید از یکی از زیرکلاس‌هایش تولید شود
  - بهویژه کلاس‌هایی که متد انتزاعی دارند، قطعاً کلاس انتزاعی هستند
  - چون کلاسی که متد انتزاعی دارد، تعریف برخی رفتارها را ندارد
  - این رفتارهای انتزاعی در زیرکلاس‌ها تعریف (واقعی) می‌شوند
  - مثال: Shape یک کلاس انتزاعی است، زیرا متدهای انتزاعی دارد
    - متدهای محاسبه مساحت و محیط انتزاعی هستند
    - هیچ شیئی مستقیماً از نوع Shape ساخته نمی‌شود
  - مثال: Animal یک کلاس انتزاعی است (متد حرکت کردن انتزاعی است)
- 

- انتزاعی بودن یک کلاس یا متد باید توسط برنامه‌نویس تصریح شود
- این کار با کلیدواژه abstract انجام می‌شود
- متد انتزاعی، دارای بدنه نیست
- ```
abstract class Animal { ... }
public abstract void talk();
```
- اگر در کلاسی یک متد انتزاعی تعریف کنید، باید آن کلاس را هم انتزاعی کنید
- در تعریف کلاس کلیدواژه abstract را اضافه کنید
- اگر کلاسی یک کلاس انتزاعی را به ارث ببرد، و همه متدهای انتزاعی آن را پیاده‌سازی نکند: کلاس جدید هم انتزاعی است و باید با پیشوند abstract تعریف شود
- از کلاس انتزاعی نمی‌توانیم نمونه‌ای بسازیم (چرا؟!)
- استفاده از new برای یک کلاس انتزاعی باعث خطای کامپایل می‌شود

- کلاسی که متد انتزاعی دارد: قطعاً باید به صورت انتزاعی تعریف شود
- کلاسی که متدهای انتزاعی به ارث برده است:
  - اگر همه متدهای انتزاعی که به ارث برده، پیاده‌سازی کند: واقعی می‌شود
  - اگر همه متدهای انتزاعی که به ارث برده، پیاده‌سازی نکند: انتزاعی می‌شود

● آیا می‌توانیم کلاسی که هیچ متدهای انتزاعی ندارد را انتزاعی تعریف کنیم؟

● حتی اگر هیچ متدهای انتزاعی به ارث هم نبرده باشد؟

● بله. طراح کلاس می‌تواند آن را انتزاعی تعریف کند

◦ مثلاً برای جلوگیری از ایجاد شیء از این کلاس

◦ و یا برای اجبار ایجاد زیرکلاس‌هایی از آن

```
abstract class Human{  
    private String name;  
    public String getName() {  
        return name;  
    }  
}
```

```
public abstract class Shape {  
    public abstract double getArea();  
    public abstract double getPerimeter();  
}  
  
public class Circle extends Shape{  
    private double radius;  
  
    public double getRadius() {  
        return radius;  
    }  
  
    public void setRadius(double radius) {  
        this.radius = radius;  
    }  
  
    public double getArea() {  
        return Math.pow(radius, 2) * Math.PI;  
    }  
    public double getPerimeter() {  
        return 2 * radius * Math.PI;  
    }  
}
```

```

abstract class Animal {
    private String name;
    public Animal(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public abstract void talk();
}

```

```

class Cat extends Animal{
    public Cat(String name) {
        super(name);
    }
    @Override
    public void talk() {
        System.out.println("Mew!");
    }
}

```

```

Animal[] animals = {
    new Cat("Maloos"),
    new Cat("loos"),
    new Dog("Fido")};

for (Animal a : animals) {
    System.out.println(a.getName());
    a.talk();
}

```

```

class Dog extends Animal{
    public Dog(String name) {
        super(name);
    }
    @Override
    public void talk() {
        System.out.println("Hop!");
    }
}

```

## مثال: حیوانات

- چرا متدهای انتزاعی (abstract) را تعریف می کردیم؟
- پاسخ:
- تا بتوانیم در رفتارهای چندریختی از آنها استفاده کنیم
- و گرنه خطای کامپایل می گرفتیم

```

Animal a ;
if(X) a = new Cat();
else a = new Fish();
a.move("right", 3.0);

```

• مثلاً در برنامه روبرو:

# اطلاعات نوع داده در زمان اجرا

## Runtime Type Identification

- اصلاً چرا متدهای انتزاعی را در تعریف کلاس بگنجانیم؟ چه فایده‌ای دارد؟
- وقتی نمی‌توانیم بدنه آن را تعریف کنیم، خوب اصلاً آن را اعلان نکنیم
- پاسخ: اگر آبرکلاس شامل یک متده باشد، نمی‌توانیم این متده را روی ارجاعی از نوع آبرکلاس فراخوانی کنیم

```
Animal a = new Cat();  
a.move();
```

• مثال:

- اگر کلاس Animal شامل متده move (انتزاعی یا واقعی) نباشد (انتزاعی یا واقعی):
- نمی‌توانیم move را روی شیء a فراخوانی کنیم
- حتی اگر در Cat تعریف شده باشد

\*

- چرا متدهای انتزاعی (abstract) را تعریف می‌کردیم؟

• پاسخ:

- تا بتوانیم در رفتارهای چندریختی از آنها استفاده کنیم
- و گرنم خطای کامپایل می‌گرفتیم

```
Animal a ;  
if(X) a = new Cat();  
else a = new Fish();  
a.move("right", 3.0);
```

• مثلاً در برنامه روبرو:

## Interface

A blueprint of a class

public static final properties

public abstract methods

default, static, private: concrete methods.

subclasses have to implement all the abstract methods otherwise become abstract class

We can instantiate

a convention between classes in our project, defining **common behavior for classes**.

But better: **contract between services**

Used for runtime poly

multiple inheritance.

\*\*\*\*

یکی از ابزارهای پیاده سازی Overriding polymorphism از جاوا 8 به بعد تفاوت چندانی با abstract نمیکند.

پک کلاس میتواند چندین interface را impl کند.

Interface میتواند static method داشته باشد، ولی static block نمیتواند داشته باشد.

جاوا interface های زیادی را impl کرده برای تولید کد.

• واسطه‌های مهم زبان جاوا

• Serializable, AutoCloseable, Runnable, ...

در interface، همه method‌ها بصورت **public** پیشفرض، **public** هستند (مناسب برای اونهایی که باید ارث برده شوند)

در interface، **abs** میتوانند: **abs** متدهای غیر default / private / static

## Advantages of Interface

- Impl Polymorphism

- Reusability
- Design patterns

- اگر subclass همه متد ها را impl نکند: کاربرد در الگوی composite .
- میتوان از interface، یک ref ساخت و اشاره داد به obj از subclass: در agile بر اساس قواعد open close princ.
- تغییر کد ورودی پارامتر ها بهتر است interface باشد.

یکبار در پروژه: `i = new A()`؛ اگر تصمیم گرفته شود از تاریخی به به بعد ، دیگر بجای A از کلاس B برای پیاده سازی های اینترفیس `i` در پروژه استفاده شود ؛ این کار فقط با تغییر دستور بالا انجام میشود و نیازی به تغییر همه جایی پروژه نیست. چون ما در سطح پروژه رفرنسی از `i` را میبینیم.

اگر برای عکس یک `i` داشته باشیم، دو کلاس A و B آنرا پیاده سازی کرده باشند، که A برای عکس های حجمی باشد و B برای عکس های سبک . هربار که عکسی آمد، با توجه به سایز آن ، آبجکتی از آن میسازیم و در رفرنس `i` از A قرار میدهیم. در متن پروژه خوب خود `i.method.call` شود با توجه به نوعی که از آن آبجکت ساخته شده رفتار خواهد شد.

## Interface Vs abstract

\*\*\*\*

شباهت ها

**both** provide mechanisms for **abstraction** and **defining common behavior** for classes.

**Both** provide mechanisms for **runtime poly.**

هر دو رابطه **is-a** دارند.

**both** provide abstract and none abstract methods

اگر child حتی فقط یک متد `abs` از interface `abstract` باشد یا فرقی نمیکند) Parent را هم نکند: یک **abstract class** میشود.

متد ها (با و بدون بدن) در هر دو قابل override است.

#### 1- Definition:

- interface is a blueprint or **collection of methods** but abstract is **a class**

#### 2- multiple inheritance of interfaces

A **subclass** can extend only **one abstract class** but it can implement **multiple interfaces**

An **abstract class** can **extend one other class** and can implement **multiple interfaces** but an **interface** can **extend multiple interfaces**

#### 3- Constructor:

- Interface: An **interface cannot have** a constructor because interfaces cannot be instantiated.
- Abstract class: An abstract class can have constructors that are called when its subclasses are instantiated.
- An abstract class in Java is **a class that cannot be instantiated**
  - ما میتوانیم new کنیم ولی abstract interface را خیر.

هر دو میتوانند ref داشته باشند، ولی constructor با اینکه abstract class دارد نمیتواند نمونه سازی شود، ولی interface با اینکه constructor ندارد میتواند instantiate شود.

#### 4- Fields:

- Interface: Fields in an interface are implicitly **public, static, and final**. They are **constants** and cannot be changed by implementing classes.
- Abstract class: An abstract class can have fields of **any access level**, including instance fields. **final and non-final, static and non-static variables**

#### 5- Static block:

نیتواند interface داشته باشد ولی abstract میتواند داشته باشد static blocks

## 6- Usage and Design:

- Interface: Interfaces are typically used to **define a contract** or behavior that **multiple unrelated classes can adhere to**. They provide a way to **achieve loose coupling**.
- Abstract class: Abstract classes are often used to provide a **common base implementation** among **related classes**. They can contain **reusable code** and serve as a template for subclasses.

## default method

فایده\*\*\*\*:

اضافه شدن متد جدید به interface : بدون نیاز به فورس پیاده سازی در childها

متد ها هم مثل static method ها اگر public باشند وقتی کلاسی آن Interface را impl میکند، این متد را هم با بدنه اش به ارث میرد. هر دو بصورت ضمنی public هستند.

متد default ، میتواند static باشد؟ خیر

Default method ها قابل override شدن هم هستند.

:default method call کردن روشن

روی **super** **InterfaceName.super** و یا **obj** child از قابل صدازده شدن هستند

اگر یک interface دیگر را extend کند، آیا میتواند بعضی از متدهای آن را بدنه بدهد؟

بله با default

```
public interface Vehicle {  
    String getBrand();  
    String speedUp();  
    String slowDown();  
  
    default String turnAlarmOn() {  
        return "Turning the vehicle alarm on.";  
    }  
  
    default String turnAlarmOff() {  
        return "Turning the vehicle alarm off.";  
    }  
}
```

```
public class Car implements Vehicle {  
    private String brand;  
  
    // constructors/getters  
  
    @Override  
    public String getBrand() {  
        return brand;  
    }  
  
    @Override  
    public String speedUp() {  
        return "The car is speeding up.";  
    }  
  
    @Override  
    public String slowDown() {  
        return "The car is slowing down."  
    }  
}
```

```
public static void main(String[] args) {  
    Vehicle car = new Car("BMW");  
    System.out.println(car.getBrand());  
    System.out.println(car.speedUp());  
    System.out.println(car.slowDown());  
    System.out.println(car.turnAlarmOn());  
    System.out.println(car.turnAlarmOff());  
}
```

Default method ها قابل override کردن هم هستند:

```
@Override  
public String turnAlarmOn() {  
    // custom implementation  
}
```

## متدهای پیشفرض برای واسطه‌ها

- یک واسط (interface):

- همانند کلاسی است که همه متدهای آن انتزاعی (abstract) هستند
- از جاوا 8 به بعد، یک واسط می‌تواند متدهای غیرانتزاعی داشته باشد
- به این متدها، متدهای پیشفرض (Default Method) گفته می‌شود.

```
interface Person {  
    Date getBirthDate();  
    default Integer age(){  
        long diff = new Date().getTime() - getBirthDate().getTime();  
        return (int) (diff / (1000L * 60 * 60 * 24 * 365));  
    }  
}
```

مثال:

## ارثبری از متدهای پیشفرض

- تعریف متدهای پیشفرض در کلاس‌هایی که واسط را پیاده‌سازی می‌کنند، اجباری نیست

```
class Student implements Person {  
    private Date birthDate;  
    public Date getBirthDate() {  
        return birthDate;  
    }  
    @Override  
    public Integer age() {  
        long yearMiliSeconds = 1000L * 60 * 60 * 24 * 365;  
        long currentYear = new Date().getTime() / yearMiliSeconds;  
        long birthYear = getBirthDate().getTime() / yearMiliSeconds;  
        return (int) (currentYear - birthYear);  
    }  
}
```

```
class Student implements Person {  
    private Date birthDate;  
    public Date getBirthDate() {return birthDate;}  
}
```

## Static method

\*\*\*\* اضافه شدن متدهای static به interface : بدون نیاز به فورس پیاده سازی در childها

فرقی با تعریف static method در یک کلاس ندارد. همان قوانین پابرجا هستند. کاربردش هم همون مثل static در کلاس هست که میتوانه utility باشه و روی نام (Class object) type صدا زده بشه.

Static method در همه جا shared است، ولی default فقط برای sub‌های خودش دیده میشود.

static method شدن نیستند چون static اند. ولی قابل override هستند.

This way of using Interface for defining utility classes is better as it helps in performance also, because using a class is more expensive operation than using an interface

Interface میتواند static method داشته باشد، ولی static block نمیتواند داشته باشد.

public اگر static method ها هم مثل default method باشند وقتی کلاسی آن Interface را impl میکند، این متدهای public را هم با بدنه اش به ارث میرد. هر دو بصورت ضمنی public هستند.

روش call static method کردن :

روی اسم **Interface** قابل صدازده شدن هستند

```
public interface Vehicle {  
    // regular / default interface methods  
  
    static int getHorsePower(int rpm, int torque) {  
        return (rpm * torque) / 5252;  
    }  
}
```

In another class:

```
Vehicle.getHorsePower(2500, 480);
```

## Private method

فایده \*\*\*\* encapsulation :

هایی که تعریف میشوند برای **confidential code** مناسبند

برای استفاده داخلی در یک **Default method**

برای استفاده داخلی در یک **static method**, باید **static** باشد.

و **public** **Default** متدها بصورت ضمنی **static** هستند.

برای استفاده داخلی در یک :Default method

```
public interface Foo {  
  
    default void bar() {  
        System.out.print("Hello");  
        baz();  
    }  
  
    private void baz() {  
        System.out.println(" world!");  
    }  
}
```

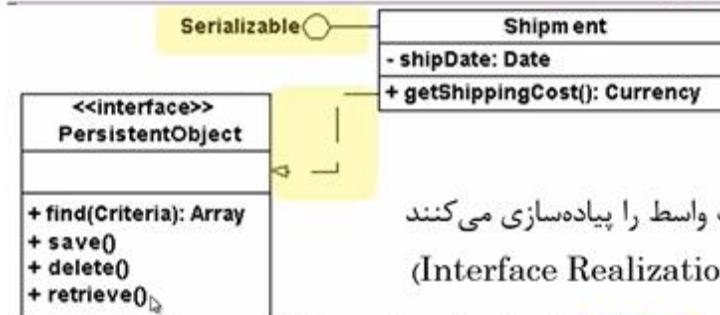
برای استفاده داخلی در یک : static method

```
public interface Foo {  
  
    static void buzz() {  
        System.out.print("Hello");  
        staticBaz();  
    }  
  
    private static void staticBaz() {  
        System.out.println(" static world!");  
    }  
}
```

.Default method قابل استفاده است هم در static method

فقط در Private static method قابل استفاده است .Default method

## نمایش واسط در UML



- کلاس‌هایی که یک واسط را پیاده‌سازی می‌کنند

(Interface Realization)

- تحقق واسط

• واسط در UML به دو شکل قابل نمایش است (هر دو شکل صحیح است) :

• واسطه‌ای ساده‌ای که متدهای مهمی ندارند: مثل واسط Serializable

• این گونه واسطها اصطلاحاً فقط یک پروتکل تعريف می‌کنند

• مثال: فقط اشیائی قابل ذخیره در فایل هستند که واسط Serializable را پیاده کنند

• واسطه‌ایی که متدهای خاصی دارند: مثل واسط PersistentObject

## تضاد اسامی

```
interface A{
    int f();
}
interface B{
    int f();
}
abstract class C implements A,B{ }
```



```
interface A{
    void f();
}
interface B{
    int f();
}
abstract class C implements A,B{ }
```



The return types are incompatible for the inherited methods A.f(), B.f()

چون در کلاس C نمیتوان دو متد را overload کرد که بدون تفاوت در یارامترها نوع برگشتیشون متفاوت باشند.

- گاهی همه متدهای یک کلاس انتزاعی هستند
- و هیچ ویژگی (Field) مشترکی در این آبرکلاس تعریف نمی‌شود
- چنین کلاسی عملاً یک واسط (interface) از عملکرد و رفتارهای زیرکلاس‌ها است

- کلاس **Instrument** یک کلاس کاملاً انتزاعی (pure abstract) است

• بهتر است به جای کلاس (class) یک واسط (interface) باشد

```
public interface Shape {
    double getArea();
    double getPerimeter();
}
```

**مثال:**

• معنا و کاربرد این واسط تقریباً مشابه این کلاس انتزاعی است:

```
public abstract class Shapes {
    public abstract double getArea();
    public abstract double getPerimeter();
}
```

• مثل کلاس انتزاعی: ایجاد نمونه (شیء) از واسط ممکن نیست

• عملگر new قابل اجرا روی یک واسط نیست

- برای ارثبری از یک کلاس، از کلیدواژه **extends** استفاده می‌شود
- برای ارثبری یک کلاس از یک واسطه، از کلیدواژه **implements** استفاده می‌شود

**class Rectangle implements Shape{...}**

- بین یک کلاس و واسطه که پیاده‌سازی کرده، رابطه **is-a** برقرار است
- Rectangle is a Shape

- اگر کلاسی یک واسط را پیاده‌سازی کند: باید همه متدهای آن را هم تعریف کند
- وگرنه این کلاس، متدهای انتزاعی را به ارت برده و خودش هم باید انتزاعی شود

**abstract class Rectangle implements Shape{**  
**public double getArea() { return ...; }**  
**}**

مثال: این کلاس باید به صورت انتزاعی تعریف شود  
 زیرا متدهای **getPerimiter** را پیاده‌سازی نکرده است

```
interface Human extends CanRun, CanTalk{
    void think();
}
```

```
interface CanFight {
    void fight();
}

interface CanSwim {
    void swim();
}

interface CanFly {
    void fly();
}
```

```
class ActionCharacter {
    private String name;
    public String getName() {
        return name;
    }
}
```

```
class Hero extends ActionCharacter
implements CanFight, CanSwim, CanFly {

    public void swim() {
    }

    public void fly() {
    }

    public void fight() {
    }
}
```

مثال



```

interface CanFight {
    void fight();
    void move();
}

interface CanSwim {
    void swim();
}

interface CanFly {
    void fly();
    void move();
}

class ActionCharacter {
    public void fight() {
    }
}

class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly {

```

## سؤال

- کلاس Hero باید چه متدهایی را پیاده‌سازی کند تا انتزاعی نباشد؟

پاسخ:  
move, fly, swim

## خلاصه: وراثت و واسط

- یک کلاس، فقط و فقط از یک کلاس می‌تواند ارثبری کند (extends)
- هر کلاس، آبرکلاس مشخص دارد که با کلیدواژه extends مشخص می‌شود
- وگرنه (اگر آبرکلاس تصریح نشود) زیرکلاس Object خواهد بود
  
- یک کلاس صفر یا چند واسط را پیاده‌سازی می‌کند (implements)
  
- یک واسط از صفر یا چند واسط ارثبری می‌کند (extends)

## جاوا ۸ و متدهای پیش فرض برای واسطه ها

- از جاوا ۸ به بعد: یک واسطه می تواند متدهای غیرانتزاعی داشته باشد

- به این متدها، متدهای پیش فرض (Default Method) گفته می شود.

**interface Person {  
    Date getBirthDate();  
    default Integer age(){**

مثال:

```
        long diff = new Date().getTime() - getBirthDate().getTime();  
        return (int) (diff / (1000L * 60 * 60 * 24 * 365));  
    }  
}
```

- همچنان یک کلاس می تواند چند واسطه را پیاده سازی کند

- بنابراین امکان وراثت چندگانه در جاوا ۸ (به شکلی محدود) وجود دارد

- بعداً به صورت مستقل در این زمینه (امکانات جاوا ۸) صحبت خواهیم کرد

**\_\_\_\_\_**

## واسطه: متغیرها و سازنده ها



- تعريف متغیر در یک واسطه رایج نیست

- در صورت تعريف متغیر:

- متغیرها به طور ضمنی ثابت، استاتیک و عمومی (public) خواهند بود

**interface Humans{  
    int MAX\_AGE=150; **~ public static final int MAX\_AGE=150;**  
}**

مثال:

- خلاصه: واسطه، وضعیت و حالت (state) اشیاء را توصیف نمی کند

- امکان تعريف سازنده (constructor) در واسطه وجود ندارد (چرا؟)

- هدف سازنده، مقداردهی اولیه ویژگی های شیء است (Property Field) یا Field

- سازنده: حالت (وضعیت) اولیه شیء را آماده می کند

- اما ویژگی خاصی در واسطه تعريف نمی شود (واسطه، حالت شیء را توصیف نمی کند)

**\_\_\_\_\_**

```
interface A {
    int a = 5;
    int f();
}

public class TestClass implements A{
    public static void main(String[] args) {
        System.out.println(a);
    }
}

...
```

```

abstract class PaintObject{
    public abstract int draw();
    private int x, y;
    public int getX() {
        return x;
    }
    public void setX(int x) {
        this.x = x;
    }
    public int getY() {
        return y;
    }
    public void setY(int y) {
        this.y = y;
    }
}
class R1 implements Shape{} ✖
abstract class R2 implements Shape{}
abstract class R3 extends PaintObject implements Shape {}
abstract class R4 extends PaintObject
    implements Shape,Drawable {} ✖

```

```

interface Shape{
    double getArea();
}

interface Drawable{
    void draw();
}

```

کدام تعاریف زیر ایجاد خطای کامپایل می‌کنند؟

## کاربرد واسط

- واسط: کلاس مجرد خالص (کاملاً انتزاعی): pure abstract
- واسط در گیر جزئیات پیاده‌سازی نمی‌شود و کلیات رفتار شیء را توصیف می‌کند
- از واسط می‌توانیم به عنوان «توصیف‌کننده طراحی کلاس» استفاده کنیم
- مثلاً طراح سیستم واسط‌ها را طراحی کند و برنامه‌نویس آن‌ها را پیاده‌سازی کند
- مثلاً برای کدی که نوشتم و در اختیار کاربران عمومی قرار می‌دهیم:  
     فقط واسط را توضیح دهیم (کاربران نحوه کار کلاس را خواهند فهمید)
- با کمک واسط به صورت قدرتمند از ارث‌بری و چندربختی بهره می‌گیریم
- بهتر است حتی‌الامکان طراحی کلاس‌ها و متدهای ما به واسط‌ها وابسته باشند
- مثلاً برای استریک متد، بهتر است واسط Shape باشد و نه زیرکلاس Circle
- زیرا وابستگی به یک زیرکلاس خاص، تغییر و تغییر داری برنامه را پرهزینه‌تر می‌کند

multiple inheritance of **Behavior**:

(multiple inheritance of **interfaces**: it existed before java 8, a class can implement multiple interfaces)

\*\*\*\* achieving a **form of** multiple inheritance of behavior.

در جاوا چه **class** ها ممکن نیست، با خاطر احتمال رخداد multiple inheritance Interface، ولی این با Diamond problem میشود

Answer: **Multiple inheritance** occurs when a class has **more than one parent** classes.

*Why Java does not allow this :* let us consider there are 2 parent classes having a method named *hello()* with same signature and one child class is extending these 2 classes, if you call this *hello()* method which is same in both parents, which parent class method will get executed – it results into an ambiguous situation, this is also called **Diamond Problem**.

You will get a compile time error if you try to extend more than one class.

```
class Parent1 {  
    public void hello() {  
        System.out.println("Hello from Parent1 class");  
    }  
}  
  
class Parent2 {  
    public void hello() {  
        System.out.println("Hello from Parent2 class");  
    }  
}  
  
public class Child extends Parent1, Parent2 {  
}
```

Errors (1 item)  
Syntax error on token ", , . expected

با **multiple inheritance** interface میشود داشت:

همینکه یک فرزند: **default method** که Interface چند دارد به نوعی multiple inheritance of behavior است.

```

class Duck implements Walkable, Swimmable {
    // Implements both Walkable and Swimmable
}

public class Main {
    public static void main(String[] args) {
        Human human = new Human();
        human.walk(); // Output: "Walking"

        Fish fish = new Fish();
        fish.swim(); // Output: "Swimming"

        Duck duck = new Duck();
        duck.walk(); // Output: "Walking"
        duck.swim(); // Output: "Swimming"
    }
}

```

به شرطی که اگر دو متد با یک signature داشت باید آن را **override** کنیم.

with the same default method, it needs to provide **its own implementation** to resolve the conflict

**Diamond Problem** (Refer to Question 9 , if you're not already familiar with this problem).

```

interface Interface1 {
    default void hello() {
        System.out.println("Hello from Interface1");
    }
}

interface Interface2 {
    default void hello() {
        System.out.println("Hello from Interface2");
    }
}

public class Child implements Interface1, Interface2 {
}

```

Errors (1 item)  
Duplicate default methods named hello with the parameters () and () are inherited from the types Interface2 and Interface1

برای حل این error باید **override** کنیم:

```

public class Child implements Interface1, Interface2 {
    @Override
    public void hello() {
        System.out.println("inside Child class hello method");
        Interface1.super.hello();
    }

    public static void main(String[] args) {
        Child obj = new Child();
        obj.hello();
    }
}

```

Output:

```

→inside Child class hello method
→Hello from Interface1

```

میتوانیم در override کردن، از پیاده سازی parent ها هم استفاده کنیم:

```

@Override
public String turnAlarmOn() {
    return Vehicle.super.turnAlarmOn();
}

```

or

```

@Override
public String turnAlarmOn() {
    return Vehicle.super.turnAlarmOn() + " " + Alarm.super.turnAlarmOn();
}

```

## multiple inheritance of Type

there is no direct mechanism for multiple inheritance of types in Java.

## multiple inheritance of State

: کلاسی بتواند متغیر های چند کلاس را به ارث ببرد که خطرناک است و در جاوا نداریم. (هر متغیری که در interface تعریف شده static خواهد بود)

## Architecture samples

برای قبل جاوا: 8

معمار/دولپر

### روش 1 : با کلاس معمولی معماری sequence method

کلاس print شامل متدهای: کار1(private) / دولپر نوشته است کار2(private) / دولپر کار3(private) / معمار نوشته است: درین متد فقط ترتیب صحیح انجام کارهای 1 و 2 و 3 و دستور اجرای متدهای انها مشخص شده است.

کلاس officer از کلاس print ارث میبرد یا obj میسازد. فقط متد اجرا که public است را میتواند داشته باشد و با همان کارش راه میوفتد.

بدین ترتیب کلاس officer به اطلاعات اضافی دسترسی نخواهد داشت و لازم نیست ترتیب صحیح را هم بداند.

میتوان بدون تغییر در کلاس officer روال انجام کارها را تغییر داد. (فقط با تغییر متد اجرا در کلاس print)

سعی میشود متدهایی که جنبه‌ی خصوصی دارد را private تعریف کنیم و فقط کلاس‌هایی که قرار است سرویس را ارائه بدهند (مثله متد run یا protected) public میکنیم.

### روش 2: با interface میتوان از Interface کمک گرفت:

معمار یک اینترفیس (Interface1) تعریف میکند و در آن متدهایی که باید پیاده سازی شوند تعریف میکند.

دو عدد دولپر داریم. Mehrad و Reza. این دو Interface1 را implement میکنند.

حال معمار از Interface1 ای که خودش ایجاد کرده یک نوع میسازد: i1

حال بسته به تصمیمش میتواند از کلاس هر یک از دولپر ها obj بسازد و آن رادر i1 قرار دهد. با توجه به شرایط میتواند نوع دولپر را تعیین کند و متدهای مورد نظرش از Interface1 را با شیئی i1 call کند.

### روش 3 : با abstract میتوان از Abstract کمک گرفت.

با abstract ، هم متد با بدنه میتوانیم داشته باشیم هم بدون بدنه. (بخاطر همین میشه معماری sequence abstract را با abstract method run رو معمار مینویسد و بقیه‌ی متدها رو میکنیم تا دولپر ها پس کنند). حال اگه نحوه اجرای هر یک از متدها متفاوت است هم میتوان چند ارث برنده یا دولپر از این abstract داشت و از هر کدام در شرایط مناسبش استفاده کرد.

## Final property

مقدار primitive غیر قابل تغییر است

Ref برای obj غیر قابل تغییر است / ربطی به state برای obj ندارد.

باید مقدار دهی اولیه شود

• برخی از متغیرها یک بار مقدار می‌گیرند و هرگز تغییر نمی‌کنند

• به این متغیرها ثابت (constant) گفته می‌شود

• مثال: Math.PI و Integer.MAX\_VALUE

• در جاوا متغیرهای ثابت با کلیدوازه final مشخص می‌شوند

• مقدار یک متغیر ثابت (final) قابل تغییر نیست

• اگر متغیر ثابت از انواع داده اولیه باشد: مقدارش قابل تغییر نیست

• اگر متغیر ثابت، یک شیء باشد: دیگر به شیء دیگری نمی‌تواند ارجاع دهد

final int i = 2;

i = 3;

مقدار متغیرهایی از انواع اولیه

غیرقابل تغییر است (primitive)

final Person p1 = new Person();

Person p2 = new Person();

p1 = p2;

هویت یک شیء ثابت قابل تغییر نیست

p1 = new Person();

وضعیت (ویژگی‌ها، محتوا) یک

p1.setName("Ali");

شیء ثابت قابل تغییر است



## آشکال متغیرهای ثابت

- متغیرهای ثابت به شکل‌های مختلفی دیده می‌شوند:

```
class SomeClass{  
    private final String name;  
    public final int val = 12;  
    void f(final int a){  
        final int b = a+1;;  
    }  
    void g(){  
        final String s = "123";  
    }  
    public SomeClass(String name) {  
        this.name = name;  
    }  
}
```

- پارامتر ثابت
- متغیر محلی ثابت
- ویژگی ثابت
- متغیر استاتیک ثابت

- هر متغیر ثابت، باید بلافاصله مقداردهی شود

- مثلاً یک ویژگی ثابت، باید در فرایند مقداردهی اولیه شی، مقداردهی شود

• مثلاً در سازنده

- این مفاهیم مستقل از هم هستند:

- سطح دسترسی (public, private, package access)

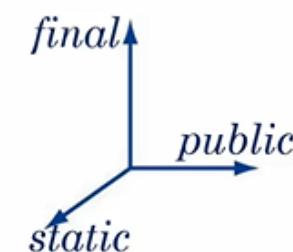
- استاتیک بودن یا نبودن

- ثابت (final) بودن یا نبودن

- مثلاً هر متغیر استاتیک:

- ممکن است final باشد یا نباشد

- ممکن است public باشد یا نباشد



وقتی final ای field تعریف می‌شود، باید همون اول مثلاً در constructor مقدار دهی بشه و دیگه نمی‌شه اونرو آپدیت کرد. / وقتی final field ای تعریف می‌شود، نمی‌توان برای آن setter نوشت.

## Immutable

ای Object immutable است که نتوان state آن را تغییر داد.

Primitive wrapper class ها ، immutable هستند چون متغیرهای primitive داخشون final تعریف شدند، پس نمیتوانند تغییر داده شوند. پس state آنها نمیتوانند تغییر بکند.

وقتی برای فیلدی از کلاس getter مینویسیم یک کپی از ارجاع به ما میدهد و با تغییرش کپی ارجاع از دست میرود ولی باعث تغییر state برای obj نمیشود.

وقتی Setter را برای obj میکنیم یک ارجاع دریافت میکند که اگر بره بذاره جای ارجاع قبلیه آن فیلد، obj تغییر میکند.

\*\*\*\*

چطور یک class را immutable کنیم؟

primitive ها field -

private final (wrapper class) آنها تغییر نمیکند مثل state immutable ها field -

تعریف شوند و setter نوشته نشود. (برای فیلدهای final نمیتوان setter نوشت)

در این حالت:

مقدار field های primitive رو نمیشه تغییر داد

فیلدهای ref را نمیتوان تغییر داد (چون final و هم immutable هستند)

فیلدهای primitive wrapper state را نمیتوان تغییر داد چون immutable هستند

```
public class MenuComponent {
    private final String url;
    private final String name;

    public MenuComponent(String url, String name) {
        this.url = url;
        this.name = name;
    }

    public String getUrl() {
        return url;
    }

    public String getName() {
        return name;
    }
}
```

- اگر نوع Class B از class A composition field بعنوان استفاده کند در حالی که immutable نباشد؛ هر دو روش بالا را انجام دهیم باز هم باز هم باز هم state obj قابل تغییر است.

```
B b= a.getB();  
b.setF1("x");
```

○ راه حل

<https://codepumpkin.com/immutable-class-with-mutable-member-fields-in-java/>

۹۹۹

برای اینکه استفاده درستی از بعضی collection ها برای کلاس هایی که خودمان میتوانیم کنیم باید کلاسمون باشند. کلاسهای پدری هستند که اگر ازشون ارث ببریم immutable میشون.

- اشیاء به دو دسته تقسیم می‌شوند: تغییرپذیر و تغییرناپذیر
- **Mutable & Immutable**
  - ویژگی‌های اشیاء تغییرناپذیر بعد از ساخت این اشیاء قابل تغییر نیست
  - امکان تغییر وضعیت اشیاء تغییرپذیر وجود دارد
  - مثلاً متدهای **(mutator) setter** دارند
  - موضوع «تغییرناپذیری» با «ثابت بودن» متفاوت است
  - ثابت بودن درباره ثبات **هویت** است و با کلیدوازه **final** مشخص می‌شود
  - تغییرناپذیری درباره ثبات **وضعیت (state)** است
  - تغییرناپذیری یک مفهوم است و کلیدوازه خاصی ندارد
  - طراح یک کلاس تصمیم می‌گیرد نمونه‌های این کلاس تغییرپذیر باشند یا خیر



- اشیاء تغییرناپذیر مزایایی دارند

- ساده‌تر هستند

- فهمشان آسان‌تر است

- مزایایی در کارایی برنامه دارند

- مزایایی در برنامه‌های همرونده و موازی دارند (Thread-safe)

- اشیاء برخی از کلاس‌هایی که می‌شناسیم، تغییرناپذیر هستند. مثال:

- (مثلاً متدهای `setValue` `String` ندارد)

- همه کلاس‌های لفاف انواع اولیه (`Double`, `Boolean`, `Integer` و غیره)

```
package ir.javacup.javashortstories;
```

```
public class Person {  
    private final String name;  
    private Integer age = null;  
    private Double length = null;  
    public Person(String name){  
        this.name = name;  
    }  
    public Person(String name, int age){  
        this.name = name;  
        this.age = age;  
    }  
    public String toString(){  
        return name + "(+age+)";  
    }  
}
```

هر تغییری در مقدار آنها، در حقیقت `ref` آنها تغییر کرده نه `primitive value` داخل آنها.

## بعد از فرآخوانی badSwap مقدار a و b چه خواهد بود؟

```
static void badSwap(String var1, String var2){  
    String temp = var1;  
    var1 = var2;  
    var2 = temp;  
}  
  
public static void main(String[] args) {  
    String a = "5";  
    String b = "4";  
    badSwap(a, b);  
}
```

مقدار a و b در مبدا تغییر نخواهد کرد، `ref` که پاس داده میشود در داخل متده مقصد ارجاعات برای `var1` و `var2` جابجا میشود.

نمیشه goodSwap برای wrapper class String ها و String نوشت! چون حالتشان قابل تغییر نیست.  
چون نهایتا هر جا خواستم str1= strX کنم این state را کاری ندارد، ارجاع موردنظر بحث است و در متده دیگر، ارجاع کپی خواهد شد، در مقصده setter ای برای Immutable String که نداریم، ارجاع کپی شده هم اگر تغییر دهیم فقط ارجاع را از دست داده ایم و تاثیری روی مبدا نمیگذارد.

حتی concat هم خروجی آبجکت String جدید میده و String قبلی رو update نمیکنه.  
هر جا هم خواستم str1="AA" کنم، خود جاوا یه strX String temp (strX) اون پشت ساخته و تغییری در مبدا ایجاد نمیکنه.

تلash های من:

```
public static void goodSwap(String s1, String s2){  
  
    s1.concat("****");  
  
    System.out.println("s1 = " + s1); // A
```

```

String s3= "C";
s2=s3;
System.out.println("s2 = " + s2); // C

s2="D";
System.out.println("s2 = " + s2); // D

}

public static void main(String[] args) {

String a="A";
String b="B";
goodSwap(a, b);

System.out.println("a = " + a); // A
System.out.println("b = " + b); // B

```

هر کاری کردم مقدار a و b تغییری نکردند.

```

String name = new String("ali");
String name = "ali";

```

این دو خط کد دقیقاً یکی هستند. خود جاوا در جریان فرایند **autoBoxing** ، یه OBJ String اون پشت میسازه.

پس وقتی مقدار یک String را تغییر میدهیم:

```
String s;  
s= "A";  
s= "B";
```

در حقیقت دو تا Object String ساخته شده، s اول به اونی که مقدارش A بوده اشاره میکرده، و بعد به اونی که مقدارش B هست.

چون state immutable است، آن قابل تغییر نیست. فقط میتوان هر بار ارجاع یک obj را عوض کند به obj ای دیگر که state آن چیز دیگریست.

اگر بجای که Student mutable String داشتیم، میشد با dot یا ... ref یکی از فیلدهای مثل name را تغییر داد:

```
public static void swapNames(Student s1, Student s2){  
    String tmp = s1.name;  
    s1.name = s2.name;  
    s2.name = tmp;  
}  
  
Student a = new Student();  
Student b = new Student();  
a.setName("Ali");  
b.setName("Taghi");  
swapNames(a, b);  
System.out.println(a.name);  
System.out.println(b.name);
```

دو obj سمت مبدا هم تغییر میکند.

ولی در مورد کلاس هایی که برنامه نویس میسازند مثل student به راحتی میشه حالت obj را تغییر داد چون دیفالت immutable نیستند.



**final Class/ final method**

کلاسی که به صورت final تعریف شود به سایر کلاس‌ها اجازه ارث بری و توسعه نمی‌دهد.

کلاس abstract final دقیقاً بر عکس کلاس abstract عمل می‌کند. فحش: abstract final

از کلاس final میتوان obj ساخت، نمیشه extend و توسعه داد

از کلاس abstract نمیتوان obj ساخت، میشه extend و توسعه داد

متدهای static یا final یا private ، binding آنها همیشه با obj‌ای از نوع کلاسی است که در آن تعریف شده اند و احتمال دیگری وجود ندارد. این متدها همیشه binding compile time هستند. و قابل override کردن نیستند.

کلاس‌های String یا wrapper class هم هستند .final هستند هم

Final هستند برای اینکه توسعه نیابند.

برای اینکه obj state های آنها تغییری نمیکند.

Ref‌های آنها میتوانند تغییر کند مگر اینکه وقتی جایی new هم field final میشنوند، آن هم obj field final تعریف شود یا .setter بدون private

خود کلاس final میتواند دارای متدهای main باشد و اجرا شود؟ بله

در بحث exception handling finally block استفاده میشود.

### • کلاس final :

- هیچ کلاسی نمی‌تواند یک کلاس final را extend کند
- معنای کلاس final ، نهایی است و قبل تغییر و گسترش نیست
- کلاس final هیچ زیرکلاسی نخواهد داشت
- ایجاد زیرکلاس برای یک کلاس final : ایجاد خطای کامپایل

### • متدهای final :

- معنای متدهای final ، نهایی است: قابل تغییر نیست
- هیچ زیرکلاسی نمی‌تواند تعریف یک متدهای final را لغو (override) کند
- لغو (override) کردن یک متدهای final : ایجاد خطای کامپایل

- تعریف متدها در کلاس به صورت **final** : یک تصمیم طراحی است
- با این کار طراح کلاس اجازه نمی‌دهد دیگران معنی کلاس یا متدها را تغییر دهند
- اگر طراح چنین هدفی داشته باشد، متدها را **final** می‌کند
- مثال: کلاس **String** ، یک کلاس **final** است
- نمی‌توانیم آن را **extend** کنیم
- در مورد متدها و کلاس‌های **final** : چندريختی وجود ندارد
- چون معنی نهايی مشخص است

- تعریف متدها در کلاس به صورت **final** : یک تصمیم طراحی است
- با این کار طراح کلاس اجازه نمی‌دهد دیگران معنی کلاس یا متدها را تغییر دهند
- اگر طراح چنین هدفی داشته باشد، متدها را **final** می‌کند
- مثال: کلاس **String** ، یک کلاس **final** است
- نمی‌توانیم آن را **extend** کنیم
- در مورد متدها و کلاس‌های **final** : چندريختی وجود ندارد
- چون معنی نهايی مشخص است

```
class Dog extends Animal{
    public final void bark() { .. }
}
```

```
Dog d = ...;
d.bark();
```

## Generics

نگاه Class Type به یک کلاس ، مثلاً یک کلاس Data Structure که ساختاری از تعدادی Class Type را در خود جای میدهد.

میخواهیم در زمان ایجاد شیی ، و نه در زمان تعریف کلاس ، نوع را تعیین کنیم.

فایده:

برای هر نوع Data Structure جدایگانه ننویسیم.

میخواهیم برای هر Data Type ای قابل استفاده باشد. Object همه‌ی نوعی را پوشش میدهد و خوب نیست.

استفاده‌ی نادرست از GDT: برای استفاده همزمان چند نوع Obj بریزیم توش. GT object را داده باشیم.

GDT میتوانند abstract و Interface و Class باشند

### Generic Data Type (GDT) / Parameterized type

که Class ای generic های Data Type میپذیرد.

```
public class Pair<T1, T2> {
    private T1 first;
    private T2 second;
    public T1 getFirst() {return first;}
    public void setFirst(T1 first) {this.first = first;}
    public T2 getSecond() {return second;}
    public void setSecond(T2 second) {this.second = second;}
    public Pair(T1 first, T2 second) {
        this.first = first;
        this.second = second;
    }
}
```

### شال: کلاس Pair

- فرض کنید می‌خواهیم کلاس Pair تعریف کنیم
- هر شیء از این کلاس یک جفت شیء (زوج مرتب) در درون خود نگه می‌دارد

می‌خواهیم در زمان ایجاد شیء (و نه در زمان تعریف کلاس) نوع این دو شیء را تعیین کنیم

```
Pair<String, Double> p1;
p1 = new Pair<String, Double>("Ali", 19.0);
String name = p1.getFirst();
Double avg = p1.getSecond();
```

```
Pair<String, String> p2;
p2 = new Pair<String, String>("Ali", "Alavi");
String fname = p2.getFirst();
String lname = p2.getSecond();
```

## محدود کردن نوع عام

- هنگام استفاده از یک نوع عام، از هر کلاسی به عنوان پارامتر نوع می‌توانیم استفاده کنیم

- ولی گاهی نیازمندیم که پارامتر نوع را محدود به انواع خاصی کنیم

```
class NumbersQueue<T extends Number>{...}
```

مثال:

:diamond operator

نوع generic در زمان new کردن یک instance با عنوان GDT از آن مشخص می‌شود. parameter type

می‌توان در هنگام GT، new را دوباره اعلام نکرد، همانی که سمت ساخت ref اعلام کردیم کافیست می‌کند.

:Raw Type

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

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

class NumbersQueue<T extends Number>
{
    public void enqueue(T o){}
    public T dequeue(){...}
}

NumbersQueue queue;
queue = new NumbersQueue();
queue.enqueue(new Integer(1));
queue.enqueue(new Double(3.14));
queue.enqueue("Ali");
```

### :Generic method

متوازد GT مربوط به GDT را استفاده نکند.

حتی GM متوازد در یک Class Type غیر Generic باشد.

برای GDT در زمان new کردن instance از GDT مشخص میشود

هنگام ارسال پارامتر خودبخود مشخص میشود

```

class NotGeneric{
    public <T> T chooseRandom(T p1, T p2){
        if(new Random().nextFloat()>0.5)
            return p1;
        return p2;
    }
    public static <E extends Comparable> E max(E p1, E p2){
        return p1.compareTo(p2) > 0 ? p1 : p2;
    }
}

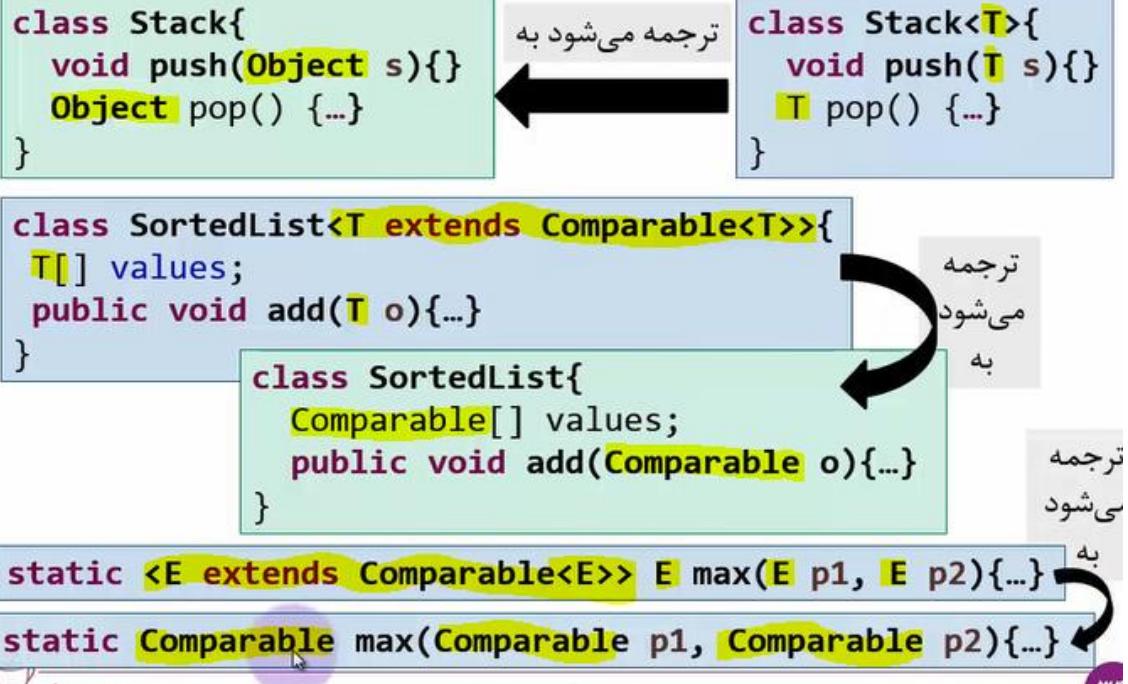
String s = new NotGeneric().chooseRandom("A", "B");
Integer num = new NotGeneric().chooseRandom(1, 2);
num = NotGeneric.max(1, 2);
s = NotGeneric.max("A", "B");

```

### Erasure mechanism

در GT ، compile time چک میشود ولی در نهایت به خاص ترین حالت ممکن (raw type) (اگر هیچ محدودیتی نباشد: compile code میشود و در run time قرار میگیرد. بنابرین در Jre خبر ندارد GT ای در compile time وجود داشته.

## مثال



## کویز

- چرا نمی‌توانیم متدهای `f` را به شکل زیر سربار (overload) کنیم؟ (خطای کامپایل می‌گیریم)

```
public static <E extends Number> void f(E i){...}  
public static void f(Number i){...}
```

پاسخ:

- زیرا براساس فرایند محو، هر دو متدهای `f` یک شکل ترجمه می‌شوند

پیغام کامپایلر:

- Erasures of methods `f(E)` and `f(Number)` are the same as another method

## محدودیت‌ها در تعریف انواع عام

فرض کنید:  
class Stack<T>{...}

- نمونه‌سازی از «پارامتر نوع» (T) در کلاس فوق) ممکن نیست

T ref = new T(); ← Stack

- ایجاد خطای کامپایل در داخل کلاس

T[] elements = new T[size]; ←

- ایجاد (new) آرایه از پارامتر نوع ممکن نیست

خطای کامپایل

- عملگر instanceof بر روی پارامتر نوع قابل فراخوانی نیست

if(o instanceof T);

- تعریف متغیر استاتیک از جنس «پارامتر نوع» ممکن نیست

private static T obj;

- خطای کامپایل

\*\*\* ۹۹۹

چون runtime از نوع دقیق تعریف شده برای GT آگاهی ندارد:

نمیتوان از instance ، GT سازی کرد.

هنگام ساخت array نباید نام GT را بیاریم

عملگر () instanceOf کار نمیکند

نمیتوان از GT Static field ساخت

\*\*\*\*

instanceOf cannot be directly used with generic types **due to type erasure**.

**Since the type information is not available at runtime**, the instanceof operator cannot determine the actual type of a generic object.

بنابرین فکر میکنم خطای compile برای استفاده از instanceof برای جلوگیری از اشتباه در runtime در آینده است.

خطا در زمان :runtime

```
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) {  
12            System.out.println(s);  
13        }  
14    }  
15 }
```

خروجی؟

- الف) بدون خطا
- ب) خطای کامپایل در خط ۹ و ۱۰
- ج) خطای کامپایل در خط ۱۱ یا ۱۲
- د) خطا در زمان اجرا در خط ۹
- ه) خطا در زمان اجرا در خط ۱۱
- و) خطا در خط ۳

\*\*\*\* این list در compile time میشه بهش اضافه بشه، ولی در متدهای main و String باشد. runtime است، نوع های Object و Integer هم میتوانند به آن اضافه شوند.

ولی وقتی میخواهد چاپ کنه باید باعث run time Error کنه که String cast به Integer میشود.

راه حل در فایل **java2** آمده است.

## نوع عام نمی‌تواند Exception باشد

- یک کلاس عام نمی‌تواند به عنوان استثنا (exception) استفاده شود

- یک کلاس عام نمی‌تواند از Throwable ارثبری کند:

```
class GenExc<T> extends Exception {} → syntax error
```

- بنابراین شیئی از نوع عام را نمی‌توان پرتاب (throw) یا دریافت (catch) کرد

- البته از «پارامتر نوع» (و نه از خود نوع عام) می‌توان به صورت استثنا استفاده کرد:

```
class Generic<T extends Exception> {
    void f() throws T {...}
    <E extends Throwable> void g() throws E {...}
}
```

یک GDT نمی‌تواند از throwable ارث ببرد. چرا؟؟؟

در هنگام overloading Erasure Compile error ممکن است به بخوریم.؟؟؟

به polymorphism ربطی ندارد، چون از قبل در زمان compile مشخص شده است Generic Data Type یا raw Type هستند و هر نوع type خاص تری که در آن باشد فرقی نمی‌کند تنها یک form متد احتمال obj گردید. اجرا شدن دارد.

## انواع داده عام (Generic)

- گاهی منطق پیاده‌سازی یک کلاس، برای انواع داده مختلف یکسان است
  - مثلاً منطق متدهای add در کلاس ArrayList به ازای لیستی از رشته یا عدد متفاوت نیست
  - راه اول: به ازای هر نوع داده، یک کلاس ArrayList بسازیم
  - مثال: StudentArrayList و IntegerArrayList و StringArrayList
  - این کلاس‌ها مشابه (کپی) یکدیگرند
  - راه دوم: یک کلاس ArrayList تعریف کنیم و در هنگام استفاده نوع آن را محدود کنیم
- ```
ArrayList<String> list1 = new ArrayList<String>();  
list1.add("Ali");  
ArrayList<Integer> list2 = new ArrayList<Integer>();  
list2.add(new Integer(2));
```
- کلاس ArrayList یک نوع داده عام (Generic) است

## مسئله چیست؟ (ادامه)

- فرض کنید می‌خواهیم کلاس MyList یا ArrayList را تعریف کنیم
- متدهای add را چگونه تعریف کنیم؟
- اگر این متدهای این‌گونه باشد:

```
void add(String obj) {...}
```

- این کلاس فقط برای رشته‌ها کار خواهد کرد (لیستی از رشته‌ها)
- اگر این متدهای این‌گونه باشد:

```
void add(Object obj) {...}
```

- نوع اشیاء این کلاس محدودیتی ندارد (لیستی از هر نوع شیء)
- ممکن است در یک لیست، همزمان اشیائی از نوع رشته، عدد یا دانشجو داشته باشیم
- معمولًاً علاقمندیم یک ظرف (مثلاً لیست)، اشیائی از یک نوع داشته باشد
- مثل ظرف‌های جawa (ArrayList و ...): این ظرف‌ها چگونه تعریف شده‌اند؟

به شیوه پیاده سازی الگوریتم stack و `LinkedList` هم توجه کن

نحوه تعریف کلاس عام

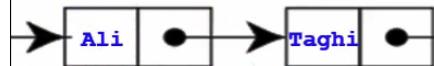
```
public class Stack<E> {
    private E[] elements;
    private int top;
    public void push(E pushValue) {
        if (top == elements.length - 1) throw new FullStackException();
        elements[++top] = pushValue;
    }
    public E pop() {
        if (top == -1) throw new EmptyStackException();
        return elements[top--];
    }
    public Stack(int maxsize) {
        top = -1;
        elements = (E[]) new Object[maxsize];
    }
}
Stack<String> st1;
st1 = new Stack<String>(10);
st1.push("A");
st1.push("B");
String p1 = st1.pop();
```

پارامتر نوع  
(Type Parameter)

```
Stack<Integer> st2;
st2 = new Stack<Integer>(10);
st2.push(new Integer(1));
st2.push(new Integer(2));
Integer p2 = st2.pop();
st2.push("A"); X
```

## مسال

- برای هر گره از یک لیست پیوندی، می خواهیم یک کلاس با نام `Node` تعریف شود



- هر گره دو فیلد مهم دارد:

۱- مقدار (از هر نوعی می تواند باشد)

۲- ارجاع به گره بعدی (ارجاعی به یک `Node`)

```
class Node<E> {
    E item;
    Node<E> next;
    Node(E element, Node<E> next) {
        this.item = element;
        this.next = next;
    }
}
```

```
Node<String> last = new Node("Taghi", null);
Node<String> first = new Node("Ali", last);
```

## مثال

```
class ArrayList<E> implements List<E>
//extends... implements...
{
    public int size() {...}
    public E get(int index) {...}
    public E set(int index, E element) {...}
    public boolean add(E e) {...}
    ...
}

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

مانند یک کلاس، یک واسطه یا  
یک کلاس مجرد هم می‌تواند  
عام (generic) باشد

اما به عنوان «پارامتر نوع»، نمی‌توانیم از انواع داده اولیه (مثل double) استفاده کنیم

• جایگزین E در مثال فوق فقط یک «کلاس» می‌تواند باشد



List<int> error1;  
List<double> error2;

خطای کامپایل:

## مثال

```
interface Map<K,V> {
    int size();
    V get(Object key);
    V put(K key, V value);
    V remove(Object key);
    Set<K> keySet();
    Collection<V> values();
}
```

امکان استفاده از چند «پارامتر نوع»  
در یک کلاس عام (generic type)

```
public class HashMap<K,V> implements Map<K,V>
//extends... implements...
{
    ...
}
```

- در زمان کامپایل، از اشتباه برنامه‌نویس جلوگیری می‌کند (پیش از اجرای برنامه)
- اگر برنامه‌نویس متغیر `List<String> strs` را به این شکل تعریف کند:  
یعنی قرار است `strs` لیستی از رشته‌ها باشد
- با ذکر پارامتر نوع (رشته) و نظارت کامپایلر، برنامه نویس نمی‌تواند سهواً اشتباه کند
- مثلاً برنامه‌نویس نمی‌تواند `strs.add(new Integer(5))` را فراخوانی کند
- زیرا کامپایلر با یک syntax error جلوی آن را می‌گیرد
- به نوع عام (Parameterized Types) نمی‌تواند (Generic Type) هم می‌گویند
- در مثال فوق، `List` یک نوع عام یا پارامتردار است: `String` به عنوان پارامتر نوع

## محدود کردن نوع عام

- هنگام استفاده از یک نوع عام، از هر کلاسی به عنوان پارامتر نوع می‌توانیم استفاده کنیم
- ولی گاهی نیازمندیم که پارامتر نوع را محدود به انواع خاصی کنیم
- `class NumbersQueue<T extends Number>{...}` مثال:
- `interface SortedList<E extends Comparable>{...}`
- در اینجا، extends یعنی «پارامتر نوع» باید زیرکلاس یا زیرواسط نوع مشخص شده باشد
- مثال: فرض کنید:

`NumbersQueue<Integer> n;` ✓  
`NumbersQueue<Double> d;` ✓  
`NumbersQueue<String> s;` ✗  
`NumbersQueue<Person> p;` ✗

`SortedList<Long> l;` ✓  
`SortedList<Float> f;` ✓  
`SortedList<String> s;` ✓  
`SortedList<Person> p;` ✗

## نوع خام (Raw Type)

- انواع داده عام را بدون تصریح «پارامتر نوع» هم می‌توان استفاده کرد

- در این صورت، کامپایلر حداقل محدودیت ممکن را برای این انواع اعمال می‌کند

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

در این مثال، هر شیئی قابل افزودن به List است (محدودیتی نیست)

```
class NumbersQueue<T extends Number>{  
    public void enqueue(T o){}  
    public T dequeue(){...}  
}
```

در این مثال، فقط اشیائی از نوع

```
NumbersQueue queue;  
queue = new NumbersQueue();  
queue.enqueue(new Integer(1));  
queue.enqueue(new Double(3.14));  
queue.enqueue("Ali");  
queue.dequeue();
```

قابل استفاده در queue هستند

الواع داده عام

19

- از نسخه ۷ (java 1.7) به بعد، «استنتاج نوع» برای انواع عام ممکن شده است

- Type Inference

- بهویژه، ذکر نوع عام در هنگام نمونه‌سازی از انواع عام لازم نیست (نوع آن استنتاج می‌شود)

- به این امکان، عملگر لوزی (diamond operator) می‌گویند

## • مثال:

قدیمی      جدید

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

```
Map<String, List<Student>> map = new HashMap<String, List<Student>>();
```

```
Map<String, List<Student>> map = new HashMap<>();
```

یک نوع عام یا غیرعام می‌تواند از  
یک نوع عام یا غیرعام ارثبری کند

## وراثت و انواع داده عام

```
class A{}  
class B extends A{}
```

۱- یک نوع غیرعام، فرزند نوع غیرعام دیگری باشد

```
class Box<T> extends B{}
```

۲- یک نوع عام، فرزند یک نوع غیرعام باشد

```
class IntList implements List<Integer>{  
    public boolean add(Integer e){...}  
}
```

۳- یک نوع غیرعام،  
فرزند یک نوع عام باشد

• در این صورت، زیرکلاس عام بودن را کنار می‌گذارد

• زیرکلاس تعیین می‌کند از چه نوع خاصی به جای پارامتر نوع آبرکلاس استفاده می‌کند

۴- یک نوع عام، فرزند یک نوع عام باشد

• در این صورت می‌تواند پارامتر نوع را محدودتر کند

```
interface NumberList<T extends Number> extends List<T>{}
```

## متدهای عام (Generic Method)

• دیدیم متدهای یک کلاس عام می‌توانند از نوع داده عام آن کلاس استفاده کنند

```
class ArrayList<E> {  
    public E get(int index) {...}  
    public boolean add(E e) {...}  
}
```

• به عنوان پارامتر یا نوع داده برگشتی:

• اما یک متدهای خود نیز می‌تواند

نوعی عام را به عنوان «پارامتر نوع» معرفی کند

```
interface NotGeneric{  
    public <E> E f(E p1);  
}
```

• نوعی غیر از آن چه در کلاس تعیین شده

(مثلًاً نوعی غیر از E در کلاس فوق)

• حتی برای متدهایی که در کلاس‌های غیرعام قرار دارند

• به این متدها، متدهای عام (Generic Method) گفته می‌شود

```
class GenericClass<T>{  
    public <E> void f(E p1, T p2){}  
}
```

- می‌دانیم واسط Map به شکل زیر تعریف شده است:

```
interface Map<K,V> { ... }
```

- امضای متدهای values و keySet در این واسط چگونه است؟
- یادآوری: keySet کلیدها و values مقادیر موجود در map را برمی‌گردانند

پاسخ:

```
Set<K> keySet();
Collection<V> values();
```

## فرایند محو (Erasure)



- کنترل انواع داده عام (Generics) فقط مربوط به زمان کامپایل است
- در زمان اجرا، اطلاع و اثری از «پارامتر نوع» (Type Parameter) نیست
- اگر از یک «پارامتر نوع» برای ایجاد یک شیء استفاده کنیم، این «پارامتر نوع» فقط توسط کامپایلر چک می‌شود
- در زمان اجرا اثری از این «پارامتر نوع» نیست
- مثلاً در زمان اجرا معلوم نیست که یک شیء از نوع Stack به شکل Stack<Integer> ایجاد (new) شده یا به شکل Stack<String>
- در bytecode (فایل کامپایل شده یا .class) اطلاعاتی درباره «پارامتر نوع» یک شیء نیست
- در واقع همه انواع عام، به صورت «نوع خام» (raw type) خود ترجمه می‌شوند
- به این رفتار جاوا در قبال انواع عام، فرایند محو (Erasure) گفته می‌شود

- وقتی کامپایلر، کدی شامل داده‌عام را ترجمه می‌کند، بخش «پارامتر نوع» ترجمه نمی‌شود
- در ترجمه، به جای «پارامتر نوع» خاص‌ترین نوع ممکن را جایگزین می‌کند
- به عنوان مثال، هر چهار خط زیر به یک شکل ترجمه می‌شوند
- کد ترجمه‌شده‌ی هر چهار دستور زیر (در byte code) یکسان است

```
ArrayList<String> list = new ArrayList<>();
ArrayList<Integer> list = new ArrayList<>();
ArrayList<Object> list = new ArrayList<>();
ArrayList list = new ArrayList(); // raw type
```

- البته قبل از اجرا (در زمان کامپایل) این متغیرها متفاوتند
- کامپایلر مراقب است شیء اول فقط با رشته‌ها فراخوانی شود، و گرنۀ خطای کامپایل
- مثلاً هنگام فراخوانی متده add روی این شیء، باید پارامتر از نوع رشته باشد

## ۲. ساختار داده

- `class SortedList<T extends Comparable>{}`
- مثال دیگر: کلاس SortedList فوق را در نظر بگیرید
  - در ترجمه این کلاس، هر جا پارامتر نوع (T) استفاده شده، در هنگام ترجمه با Comparable جایگزین می‌شود
  - همچنین کد ترجمه‌شده‌ی هر سه دستور زیر (در byte code) یکسان است

```
SortedList<Integer> list = new SortedList<>();
SortedList<String> list = new SortedList<>();
SortedList<Comparable> list = new SortedList<>();
```

- خلاصه: اعمال محدودیت‌ها در زمینه داده‌های عام بر عهده کامپایلر است
- بعد از کامپایلر (در زمان اجرا) محدودیتی در زمینه داده‌های عام اعمال نمی‌شود

**مثال از فرایند محو**

```

public class Generic<T> {
    void f() {
        Generic<String> g1;
        g1 = new Generic<String>();
        Generic<Integer> g2;
        g2 = new Generic<Integer>();
    }
}

```

با کمک دستور javap (در JDK هست) می‌توانیم bytecode را بینیم:

- ایجاد دو متغیر g1 و g2 به یک شکل ترجمه شده است

```

D:\java>javap -c Generic.class
Compiled from "Generic.java"
public class Generic<T> {
    public Generic();
    Code:
        0: aload_0
        1: invokespecial #1
        4: return

    void f();
    Code:
        0: new           #2
        3: dup
        4: invokespecial #3
        7: astore_1
        8: new           #2
        11: dup
        12: invokespecial #3
        15: astore_2
        16: return
}

```

انجمن جاوایاب aliakbary@asta.ir اندیع داده عالم ۳۵

## حدودیت‌ها در استفاده از انواع عام

فرض کنید:

```
class Stack<T>{...}
```

- اگر پارامتر نوع را قید کنیم، ایجاد آرایه از نوع عام ممکن نیست
- خطای کامپایل: Stack<String>[] s = new Stack<String>[8];
- بدون خطای: Stack[] t = new Stack[8];
- استفاده از انواع داده اولیه (primitives) به عنوان پارامتر نوع ممکن نیست
- خطای کامپایل: Stack<int> s; ←
- در صورت تعیین پارامتر نوع، امکان instanceof برای نوع عام وجود ندارد
- خطای کامپایل: if(o instanceof Stack<String>); ←
- بدون خطای: if(o instanceof Stack); ←

## کویز

گفتیم امکان استفاده از انواع اولیه به عنوان پارامتر نوع نیست

پس چرا این کد خطا ندارد؟ مگر ۵ یا num از انواع اولیه (int) نیستند؟

```
List<Integer> list = new ArrayList<>();
int num = 4;
list.add(num);
list.add(5);
```

پاسخ:

- به دلیل **autoboxing**
- از جوا ۵ به بعد، به صورت ضمنی مقدار ۵ و num به Integer تبدیل می‌شوند
- به همان دلیل که این کد خطا ندارد: Integer i = 3;
- در واقع list.add(new Integer(5)); تقریباً معادل list.add(5); است

## Enum

تعداد مشخص `obj` با تعریف پیشفرض `public static final` بعنوان `fields` instance پیشفرض Constructor

• فرض کنید یک کلاس، تعداد محدود و مشخصی شیء خواهد داشت

• نمونه‌های این کلاس محدود هستند

• نمونه جدیدی در آینده اضافه نخواهد شد.

• مثلاً:

- Student Type : <BS, MS, PhD>
- SMS Status : <Sent, Delivered, Pending, Error>
- Color : <Blue, Green, Black, Red>

```
class Color{  
    public static final Color BLACK = new Color();  
    public static final Color BLUE = new Color();  
    public static final Color GREEN = new Color();  
    public static final Color RED = new Color();  
  
    private Color() {  
    }  
}  
مثال از کاربرد این کلاس:  
Color c = Color.RED;  
  
راه ساده‌تری که جاوا پیشنهاد می‌کند:  
enum Color {  
    BLACK, BLUE, GREEN, RED  
}
```

# انواع داده شماری (enum)

- Enumerated type یا enumeration یا enum

- اگر یک کلاس، تعداد محدود و مشخصی شیء دارد

- بهتر است به جای کلاس، با کلیدواژه enum تعریف شود

- و همانجا همه اشیاء (نمونه‌ها) آن مشخص شود

```
enum Color {  
    BLACK, BLUE, GREEN, RED  
}  
  
enum StudentType{  
    BS, MS, PHD  
}  
  
enum Shape {  
    RECTANGLE, CIRCLE, SQUARE  
}
```

- همه این نمونه‌ها، به صورت ضمنی static، public و final هستند

آیا immu هم هستند؟

- هیچ نمونه (شیء) جدیدی نمی‌تواند ایجاد شود

- نمونه‌سازی با عملگر new منجر به خطای کامپایل می‌شود

- ارثبری از انواع enum ممکن نیست

- مفهوم وراثت را بعداً خواهیم دید

- معمول‌اً یک enum تعریفی بسیار ساده شامل اسم نمونه‌ها دارد

مثال: `enum Color{ BLACK, BLUE, GREEN, RED };`

- اما یک enum می‌تواند کلاس پیچیده‌تری باشد

- با سازنده‌های مختلف و ویژگی‌ها و متدهای متنوع

های دارای متدهای پیشفرض هستند و میتوان آنها را پیچیده تر نوشت:

## تعريف انواع پیچیده تر enum

```
enum Shape {  
    RECTANGLE(1),  
    CIRCLE(2),  
    SQUARE(3);  
  
    private int number;  
    Shape(int i) {  
        number = i;  
    }  
    public int getNumber() {  
        return number;  
    }  
}
```

```
Shape sh = Shape.CIRCLE;  
print(sh.getNumber());  
  
sh = Shape.valueOf("CIRCLE");  
print(sh.getNumber());  
  
Shape[] array = Shape.values();  
for (Shape s : array) {  
    print(s.name());  
}  
  
// Runtime Error:  
sh = Shape.valueOf("PYRAMID");
```

وقتی یک object با یک + String میشود : مثل call toString کردن آن است:

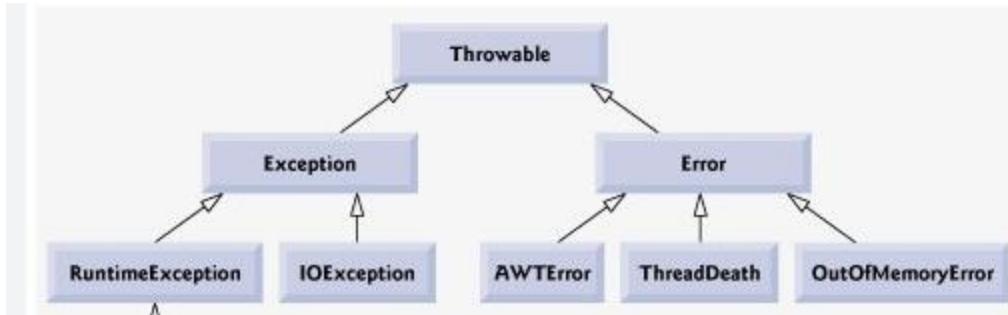
```
class A {  
    int a;  
    A(int a) {  
        this.a = a;  
    }  
}  
  
String s;  
A a = new A(1);  
B b = new B(2);  
  
enum Weather{GOOD, BAD}
```

```
class B {  
    int b;  
    B(int b) {  
        this.b = b;  
    }  
    public String toString() {  
        return String.valueOf(b);  
    }  
}
```

• در هر مورد مقدار s را حدس بزنید:

Syntax Error s=1+2;	s="1"+b; 12	s = ""+(1+2)+b; 32
s = "1"+"2"; 12	s = 1+2+b;	s = ""+Weather.BAD;
s="1"+a;	s = "+1+2+b;122	s = a+b; BAD

## Exceptions



An **Error** is **irrecoverable**, when something **severe** has happened in runtime, the application **gets terminated**.

running out of memory (heap): [OutOfMemoryError](#)

making too many recursive calls (stack): [StackOverflowError](#)

\*\*\*\*

An **exception** is an event (is an **object** which is thrown at runtime) that **disrupts** the normal flow of the program.

**exception handling** is a **mechanism** that **keeps** the normal flow of the program.

An **Exception** is an **event object** (made by runtime errors) which is **thrown** at runtime and **disrupts** the normal flow of the program.

Runtime Error → throw an instance of Exception Class → jvm: **print msg and disrupt app** (and the program may terminate **or** behave unexpectedly)

**With Handling:**

so an **exception handling** is a **mechanism** that we **define** for **what the program can do** when it happened.

Runtime Error → throw an instance of Exception Class → jvm: **run catch**

در حالت اول کنترل از دست برنامه خارج میشود، ولی در حالت دوم کنترل در دست برنامه میماند بلکه بجای اجرای خطوط برنامه که خطأ خورده است، خطوط handling اجرا میشود.

در EE، قبل از jvm در اختیار app server قرار میگیرد.

```
public class TestException {  
    public static void main(String[] args) {  
        System.out.println("Program Started");  
        int a = 15/0;  
        System.out.println("Program End");  
    }  
}
```

Output:

```
Program Started  
Exception in thread "main" java.lang.ArithmetiException: / by zero  
at com.exception.TestException.main(TestException.java:6)
```

we can recover them: by exception handling:

```
public class TestException {  
    public static void main(String[] args) {  
        System.out.println("Program Started");  
        try{  
            int a = 15/0;  
        } catch (ArithmetiException e) {  
            System.out.println("Exception handled");  
        }  
        System.out.println("Program End");  
    }  
}
```

Output:

```
Program Started  
Exception handled  
Program End
```

برای کدهایی که احتمال بروز خطای runtime دارند باید **handling** بنویسیم:

\*\*\*\*

If you think (unchecked ex.) or java forces you (checked ex.) that certain statements may throw an exception, surround them with try block.

- از نظر منطق جاوا یک خطای زمان اجراست.

مثل تقسیم بر صفر، کار با فایلی که نیست،...

وقت خطأ، خود جاوا در کدهای کتابخانه های داخلی اش throw new Exception() میکند و میرود داخل **catch** ای که ما تعییه کردیم.

- از نظر منطق برنامه نویس یک خطای زمان اجراست.

مثلاً ورودی متدهای مناسب نباشد و نتوان خروجی را محاسبه کرد.

خدمان با نوشتن دستور **throw new Exception("fff");** و برایش هم **catch** بنویسیم. اگر ننویسیم جلوتر خودش جایی **throw** میشود که ما هندلش نکردیم و روال اجرای برنامه **disrupted** میشود.

پس در هر دو حالت نوشتن **catch** با ماست، ولی فقط در حالت دوم نوشتن **throw** هم با ماست.

کتابخانه هایی برای ارتباط با DB، کار با file ، IO ، port exception را با تعریف **throws** اعلام کرده اند یعنی در آنها دستور **throw new Exception** موجود هست. پس developer مجبور میکند که تعیین کند اگر آن **ex** رخداد چطور **Handle** شود. یعنی **catch** بنویسد.

کلاس **Exception** یک **listener** دارد که مدام دارد گوش میدهد آیا خطایی روی jvm رخ میدهد یا نه سعی میشه همه ی بلوک ها (**try**،**catch**،...) رو کوچک در نظر گرفت.

کلاس **Exception** خود از **Throwable** است. به ارث برده شده است، یعنی یک نوع قابل **throw** است و **catch** است. حتی **Error** هم **subClass** از **Throwable** است که با **throw** و **catch** کردن آن کاری از دست برنامه نویس بر نمیآید. (مثلاً **out of memory**).

کلاس subclass به هایی توسعه یافته است. وقتی هر نوع آن جدگانه throws میشوند، در catch جدگانه خودش میتواند هندل شود.

علاوه بر Subclass های Exception که خود java توسعه داده، برنامه نویس هم میتواند های خود را از Exception توسعه دهد.

you can write a single catch block using a pipe symbol (|) to separate the exceptions.

### :Exception handling

#### try/ catch -

منطق برنامه را در try مینویسیم که در آن ممکن است خطای runtime رخ دهد. و در catch برنامه نویس تعیین میکند اگر runtime error رخداد اجرای برنامه چه شود تا jvm اجرای برنامه را مختل نکند.

- پاس دادن به لایه بالاتر:

اعلان throws exception در تعریف متدها اگر لایه ای بلوغه try/catch ننویسیم مرسه به jvm و برنامه مختل میشود. یا در برنامه های EE به app server این کار توسط لایبریری های داخلی جوا انجام میشود، در uncheckedExceptions نوشتن ما ممکن و بی فایده است.

بلک finally در هر صورت اجرا میشود. Return موجود در finally نسبت به try و catch اولویت دارد.

we can write a try block with finally, but we cannot write a try block alone.

When finally block will not get executed?

- when System.exit() is called
- when JVM crashes

#### Exception Propagation stack

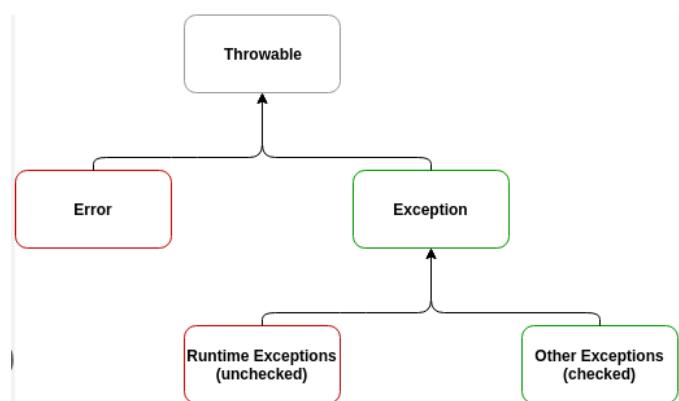
کلمه کلیدی throws در امضای متدها باعث میشود propagation stack شکل بگیرد.

کلمه کلیدی checked throws در امضای متدها باعث میشود propagation stack شکل بگیرد. این برای Exceptions واجب است و اگر ننویسیم، propagation رخ نمیدهد و نمیتوانیم در لایه های بالاتر handle کنیم.

ولی checked exception با کدهای داخلی جاوا، propagate میشوند. این یکی از تفاوت های مهم unchecked exceptions است. و checked exceptions هاست.

throws can be used with **unchecked exceptions** also, though it is of **no use** because unchecked exceptions are by default propagated.

## Checked/ unchecked Exception



### Checked Excep.

های checked exception یعنی `checked exception` هایی که **compiler** اجبار میکند (با syntax error برنامه نویس آنها را runtime error بنویسد). وقتی `catch` استفاده کند یا `throws Exception` رخ میدهد، `try` میشود و سیس `Handel` `throw ex` میشود. (برنامه مختلف نخواهد شد)

**Checked exceptions** are checked at **compile time** and must be handled by the programmer, Checked such as: **FileNotFoundException**, **SQLException**, **IOException**

### Un-Checked Excep.

در **unchecked** های **Exception** برای اینکه برنامه شلوغ نشود!!!، compiler بیخیال شده syntax Error

برنامه نویس اختیار دارد **Handel** کند یا نکند؛ اگر زمان اجرا آن **ex**، **throw** شود اگر با

throws Exception های داخلی در نهایت این **ex** ها به jvm میرسند و برنامه مختل میشود. حتی

نشده باشد با **throws Exception** نوشتن هم اضافه کاری است و کمکی نمیکند.

unchecked (runtime ex.) such as:

**NullPointerException**, **ArrayIndexOutOfBoundsException**, **Arithmatic**

**Checked exceptions** are checked at **compile time** and must be handled by the programmer while, unchecked exceptions are not checked at compile time and do not need to be handled. But as a programmer, it is **our responsibility** to handle runtime exceptions (unchecked).

```
public class DemoException {  
    public static void main(String[] args) {  
  
        try {  
            FileReader f = new FileReader("C:\\temp\\dummy.txt");  
        } finally {  
            System.out.println("Inside finally block");  
        }  
    }  
}  
Errors (1 item)  
Unhandled exception type FileNotFoundException
```

- کلاس‌های این نوع، عبارتند از :

- **Error**

- **RuntimeException**

- **RuntimeException** و **Error**

- زیرکلاس‌های **Error**

**ArrayIndexOutOfBoundsException** و **ArithmaticException** مثل

منظور این است که **exc** های چک شده را یا باید برایشان **catch** نوشت و یا اعلان **throws Exc** کرد تا خطای **compile** نخوریم. (رخدادن **exc** های **check** و **uncheck** میتواند با لایبرری های جاوا یا با **throw** باشد).

## مثال

```
void example(int x) {  
    if(x==1)  
        throw new Error();  
    if(x==2)  
        throw new RuntimeException();  
    if(x==3)  
        throw new NullPointerException();  
    if(x==3)  
        throw new IOException();  
}  
زیرا IOException یک استثنای چک شده (Checked Exception) است  
و RuntimeException یک استثنای چک نشده (Unchecked Exception) است  
برای این استثنای NullPointerException هستند
```

برای ساخت یک نوع exception ارث بری کنیم، باید از RuntimeException، یعنی unchecked exception، که اصلاً توصیه نمی‌شود

## مثال: ایجاد کلاس استثنای جدید

```
class BadIranianNationalID extends Exception {}  
  
try {  
    if (input.length()!=10) {  
        throw new BadIranianNationalID();  
    }  
    System.out.println("Accept NationalID.");  
} catch (BadIranianNationalID e) {  
    System.out.println("Bad ID!");  
}
```

هنگام overriding throws exception که ای ای می‌شود را بزرگ تر کرد.

## Exception chaining

```
} catch (Exception e) {  
    throw new MediaPlayerException(e);  
}
```

دلیل رخدان یک exception یک دیگری بوده است.

## چرا متدی در زیر کلاس نمی‌تواند استثناهای بیشتری پرتاب کند؟

- اگر این قانون وجود نداشت، تعریف کلاس Child بدون خطا می‌شد:

```
class Parent{  
    void f() {}  
}  
class Child extends Parent{  
    void f() throws IOException {}  
}
```

- در این تعریف، به نوعی رابطه is-a بین Child و Parent نقض شده است

```
void example() {  
    Parent p = new Child();  
    p.f();  
}
```

- مثلاً کامپایلر نمی‌تواند متد example را مجبور کند که IOException را throws کند یا catch

## دریافت (catch) مناسب

```
try {  
    db.save(entity);  
} catch (SQLException ex) {}
```

- استثنای SQLException را خفه می‌کند (کار خوبی نیست)

- مثلاً کد فوق SQLException را خفه می‌کند (کار خوبی نیست)
- به جای دریافت استثنای کلی (مثل Exception)، استثنای مشخصی (مثل IOException) را دریافت کنید
- در هنگام اعلان استثنای پرتالی با کمک throws هم این قاعده را رعایت کنید

- استثنای در محل مناسب دریافت (catch) کنید

- اگر در یک محل نمی‌دانید با خطای که کنید، آن را catch نکنید
- مثلاً اجازه دهید به متدهای بالادستی (که متد شما را فراخوانده‌اند) پرتاب شود

Throw early catch late

## چند نکته

- پیام مناسب و گویا به عنوان message استفاده کنید  
`throw new IOException(message);`
- لاغ (Log): در بسیاری از موارد باید بروز خطا را لاغ بزنیم (یعنی این اتفاق را ثبت کنیم)
  - این کار در بلاک catch قابل انجام است
  - البته بهتر است از فناوری‌های مخصوص لاغ (مثل SLF4J) استفاده کنید
  - استفاده از printStackTrace یا System.out.println برای این کار مناسب نیست
- مستندسازی مناسب رفتار استثناهای در برنامه شما با کمک جاوداک

● `String[] java.lang.String.split(String regex)`

Splits this string around matches of the given [regular expression](#).

**Parameters:**  
regex the delimiting regular expression

**Throws:**  
[PatternSyntaxException](#) - if the regular expression's syntax is invalid

```
/*
 * ...
 * @throws PatternSyntaxException
 *   if the regular expression's syntax is invalid
 */
public String[] split(String regex) {...}
```

تفاوت Exception و Error

## استثنا (Exception) چیست؟

- خطای اتفاقی غیرعادی که در جریان اجرای برنامه رخ می‌دهد
- روند اجرای طبیعی برنامه را مختل می‌کند
  - مثال:
    - ورودی نامعتبر
    - تقسیم به صفر
  - دسترسی به مقداری از آرایه که خارج محدوده است
  - خرابی هارد دیسک
  - باز کردن فایلی که وجود ندارد



## رفتار پیش‌فرض جاوا در زمان بروز استثنا

- به صورت پیش‌فرض، اگر در زمان اجرا خطأ یا استثنایی رخ دهد:
  - این استثنا توسط اجراء‌گر جاوا (JVM) کشف می‌شود
  - توضیحاتی درباره این استثنا در خروجی چاپ می‌شود
  - اجرای برنامه قطع می‌شود و خاتمه می‌یابد
  - اما معمولاً این رفتار پیش‌فرض مناسب نیست
- برنامه‌نویس باید عکس العمل بهتری برای زمان بروز استثنا پیاده‌سازی کند

```
17 public class DivByZero {  
18     public static void main(String a[]) {  
19         System.out.println(3/0);  
20     }  
21 }
```

Exception in thread "main"  
java.lang.ArithmetricException: / by zero  
at DivByZero.main(DivByZero.java:19)

- نکته: استثنا یک مفهوم در زمان اجراست
- این کد هیچ خطایی در زمان کامپایل ندارد

## مدیریت استثنا (Exception Handling)

- برای مدیریت و کنترل خطاهای استثناها، چارچوبی وجود دارد
- Exception Handling Framework
  - بسیاری از زبان‌های برنامه‌نویسی از این چارچوب کلی پشتیبانی می‌کنند
  - C++, Java, C#, ...
- این چارچوب، مدیریت استثناها را ساده می‌کند
- بخش اصلی برنامه را از بخش مدیریت استثناها تفکیک می‌کند
- به این ترتیب: برنامه‌نویسی و فهم برنامه‌ها ساده‌تر می‌شود

- در این موارد، هنگام برنامه‌نویسی، فقط بروز خطا را گزارش (برتاب) می‌کنیم
- بخش دیگری از برنامه گزارش خطای خطا را دریافت می‌کند و عکس العمل مناسبی اجرا می‌کند
- بهتر است این متدهای بروز استثنای آن را فراخوانی کرده گزارش کند
- و آن متدهای خاص عکس العمل مناسبی نشان دهد

```

public class ExceptionHandling {
    public static void main(String[] args) {
        try{
            f();
            g();
        }catch(Exception e){
            System.out.println(e.getMessage());
        }
        m();
    }
    private static void f() {...}
    private static void g() { h(); }
    private static void h() {...}
    private static void m() {...}
}

```

Exception Handler

اگر خطایی در بلوک try-catch (مثلًاً در متدهای f یا g) رخ دهد، روال عادی اجرای برنامه قطع و بخش catch اجرا می‌شود.

## وقتی یک استثنا رخ می‌دهد، چه می‌شود؟

۱- یک «شیء استثنا» ایجاد می‌شود (Exception Object)

۲- شیء استثنا به اجراگر جاوا (JVM) تحويل داده می‌شود

- به این عمل "پرتاپ استثنا" گفته می‌شود (Throwing an Exception)

• شیء استثنا شامل اطلاعاتی مانند این موارد است:

- پیغام خطا
- اطلاعاتی درباره نوع خطا
- شماره خطی از برنامه که استثنا در آن رخ داده است

۳- روند اجرای طبیعی برنامه متوقف می‌شود

۴- اجراگر جاوا به دنبال مسؤول بررسی استثنا (بخش catch) می‌گردد

- به این مسؤول exception handler می‌گویند

• پشته (stack) فراخوانی متدها را به ترتیب می‌گردد تا این بخش را پیدا کند

• اگر چنین بخشی (exception handler) را پیدا کند:

• شیء استثنا که پرتاپ (throw) شده، توسط این بخش گرفته (catch) می‌شود

• اجرای برنامه از این بخش ادامه می‌یابد (اجرای طبیعی متوقف شده)

• از اطلاعات موجود در شیء استثنا برای مدیریت بهتر این حالت خاص استفاده می‌شود

• اگر این بخش نباشد: «رفتار پیش‌فرض جاوا» در مقابله با استثنا اجرا می‌شود

• (پیغام خطا در خروجی استاندارد جاپ می‌شود و اجرای برنامه خاتمه می‌یابد)

## کلیدواژه‌های جاوا در چارچوب مدیریت استثنا

### throw •

```
throw new Exception("Bad Parameter");
```

- یک استثنا را پرتاب می‌کند

### throws •

```
int getYear(String d) throws Exception{}
```

- اگر متوجه احتمال دارد یک استثنا پرتاب کند، باید آن را اعلام کند

### try •

```
try {  
    ...  
} catch (Exception e) {  
    ...  
}
```

- یک بلوک برای مدیریت استثنا را شروع می‌کند

### catch •

- یک استثنا را دریافت و مدیریت می‌کند

```
public static void main(String[] args) {  
    Scanner scanner = new Scanner(System.in);  
    System.out.print("Enter a date: ");  
    String date = scanner.next();  
    try {  
        Integer year = getYear(date);  
        System.out.println(year);  
    } catch (Exception e) {  
        System.out.println(e.getMessage());  
    }  
}
```

```
public static int getYear(String day) throws Exception{  
    if (day == null || day.length() == 0)  
        throw new Exception("Bad Parameter");  
  
    String yearString = day.substring(0, 4);  
    int year = Integer.parseInt(yearString);  
    return year;  
}
```

متدهای میتوانند throws Exception را پیاده سازی کنند:

```
try {
    if (input.length()!=10) {
        throw new BadIranianNationalID();
    }
    System.out.println("Accept NationalID.");
} catch (BadIranianNationalID e) {
    System.out.println("Bad ID!");
}
```

: ویا

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

## مزایای چارچوب مدیریت استثناهای

- جداسازی بخش اصلی برنامه‌ها از کدهای مدیریت خطا و استثنای
- مدیریت خطا در بخشی که این کار امکان‌پذیر است
- و نه لزوماً در بخشی که خطا رخ داده است
- امکان گروه‌بندی خطاهای (استثنایها)
- و مدیریت آن‌ها با توجه به نوع آن‌ها
- عکس العمل مناسب به ازای هر نوع خطا
- نکته:

- همچنان باید برای تشخیص، گزارش و مدیریت استثنایها برنامه‌نویسی کنیم
- چارچوب مدیریت خطاهای مسئول رسیدگی به این امور نیست
- این چارچوب فقط ما را در سازماندهی مؤثر این کارها کمک می‌کند

## stack trace مفهوم

- وقتی استثنای را دریافت (catch) می‌کنیم، این اطلاعات در شیء استثنای موجود است:
- محل اصلی پرتاب شدن استثنای
- مجموعه (stack) متدهایی که استثنای از آن‌ها رد شده است
- به مجموعه این اطلاعات stack trace گفته می‌شود
- در موقع اشکال‌یابی برنامه، به این اطلاعات احتیاج داریم
- برخی متدهای دستیابی به stack trace از طریق شیء استثنای:

- printStackTrace();
- getStackTrace();

```

public class StackTrace {
    public static void main(String[] args) {
        try{
            f();
        }catch(Exception e){
            e.printStackTrace();
        }
    }
    private static void f() {
        g();
    }
    private static void g() {
        throw new NullPointerException();
    }
}

```

خروجی:

```

java.lang.NullPointerException
at Third.g(Third.java:18)
at Third.f(Third.java:13)
at Third.main(Third.java:5)

```

## دسته‌بندی انواع خطاها و استثناهای

- هر استثنا، نوعی دارد
- مثلاً نوع «خطا هنگام خواندن فایل» و «خطای تقسیم بر صفر» متفاوت است
- هر استثنا یک شیء است (شیء استثنا)
- هر شیء نوعی (type) یا کلاس) دارد
- بنابراین می‌توانیم استثناهای را با کمک نوع آن‌ها دسته‌بندی کنیم
- نوع استثناهای به مدیریت بهتر آن‌ها کمک می‌کند
- جاوا کلاس‌های مختلفی برای این منظور دارد
- مانند ClassCastException یا NullPointerException
- می‌توانیم کلاس‌های جدید استثنا هم ایجاد کیم
- مثلاً IranianBadNationalIdException

بالا بی هندل کن:

**مثال**

```
private void program() {  
    try{  
        int first = readInt();  
        int second = readInt();  
        int div = division(first, second);  
        System.out.println(div);  
    }catch (IOException e) {  
        System.out.println(e.getMessage());  
    }catch (ArithmaticException e) {  
        System.out.println(e.getMessage());  
    }  
}  
  
private int readInt() throws IOException {  
    String str = scanner.next();  
    if(str.matches("[\\d]+"))  
        return Integer.parseInt(str);  
    throw new IOException("Bad input");  
}  
  
private static int division(int first, int second)  
throws ArithmaticException{  
  
    if(second == 0)  
        throw new ArithmaticException("OOPS! Makhraj Sefre!");  
    return first/second;  
}  
  
class MultipleCatch {  
    public static void main(String args[]) {  
        try {  
            int den = Integer.parseInt(args[0]);  
            System.out.println(3/den);  
        } catch (ArithmaticException e1) {  
            System.out.println("Divisor is zero");  
        } catch (ArrayIndexOutOfBoundsException e2) {  
            System.out.println("Missing argument");  
        }  
        System.out.println("After exception");  
    }  
}
```

```

public static Integer getYear(String day)
    throws Exception {

    if (day == null)
        throw new NullPointerException();
    if (day.length() == 0)
        throw new EmptyValueException();
    if (!matchesDateFormat(day))
        throw new MalformedValueException();
    String yearString = day.substring(0, 4);
    int year = Integer.parseInt(yearString);
    return year;
}

private static boolean matchesDateFormat(String input) {...}

```

## نحوه ایجاد کلاس Exception جدید

- کلاس جدید باید زیرکلاس **Exception** باشد
- کلاسی با عنوان **java.lang.Exception** در جاوا وجود دارد
- زیرکلاسهای **Exception** می‌توانند پرتاب (**throw**) یا دریافت (**catch**) شوند
- کلاس‌های **Exception** معمولاً کلاس‌های ساده‌ای هستند
  - متدها و ویژگی‌های کم و مختصراً دارند
  - البته مثل همه کلاس‌ها می‌توانند سازنده، ویژگی و متدهای متنوعی داشته باشند
  - معمولاً یک سازنده بدون پارامتر دارند
  - و یک سازنده با پارامتر رشته دارند که پیغام خطأ را مشخص می‌کند

## finally مفهوم

```
try {
    //...
} catch (ExceptionType e) {
    //...
} finally {
    //...
}
```

- بخشی که در finally می‌آید، در انتهای اجرای try-catch حتماً اجرا می‌شود
- اگر خطا پرتاب شود یا نشود، در انتهای کار اجرای بخش finally تضمین می‌شود

```
try {
    //...
} catch (ExceptionType e) {
    //...
} finally {
    //...
}
```

## finally بلاک

- این بلاک حتماً اجرا می‌شود
- در هر شرایطی:
- اتمام طبیعی اجرای بلاک try بدون پرتاب خطا
- خروج اجباری از بلاک try (مثلاً با continue ، return یا break)
- خطای در try پرتاب شود و در catch دریافت شود
- خطای در try پرتاب شود و در هیچ یک از بلاک‌های catch ، دریافت نشود
- ...
- بلاک finally برای آزادسازی منابع گرفته شده در try مناسب است
- مثال: بستن فایل یا اتمام اتصال به دیتابیس
- البته هر منبعی به جز حافظه. حافظه را زباله‌روب به صورت خودکار آزاد می‌کند

```

int myMethod(int n) {
    try {
        switch (n) {
            case 1:
                System.out.println("One");
                return 1;
            case 2:
                System.out.println("Two");
                throwMyException();
            case 3:
                System.out.println("Three");
        }
        return 4;
    } catch (Exception e) {
        System.out.println("catch");
        return 5;
    } finally {
        System.out.println("finally");
    }
}                                void throwMyException() throws MyException {
}                                throw new MyException();

```

خروجی این قطعه برنامه چیست؟

System.out.println(myMethod(1));  
 System.out.println(myMethod(2));  
 System.out.println(myMethod(3));

: باسخ

One  
 finally  
 6  
 Two  
 catch  
 finally  
 6  
 Three  
 finally  
 6



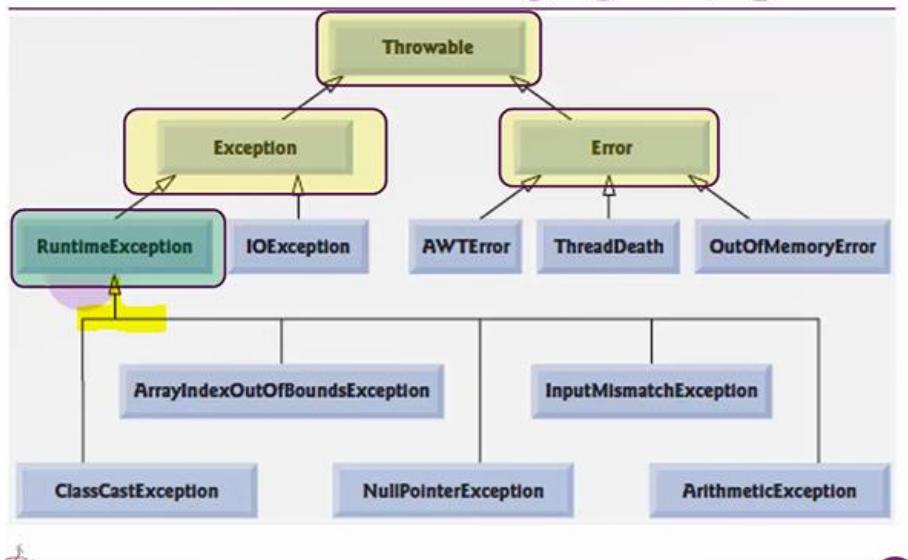
## کلاس‌های استثنا: نگاهی دقیق‌تر



### • کلاس Throwable

- در واقع هر آنچه درباره Exception گفتیم، درباره Throwable صادق است
  - مثلاً هر شیء از جنس Throwable قابل پرتاب (throw) یا دریافت (catch) است
- دو نوع Throwable اصلی وجود دارد
  - Exception - ۱ : قبلًا دیدیدم (اکثر کلاس‌های استثنا که با آن‌ها سروکار داریم)
  - Error - ۲ (خطا) : معمولاً تلاش نمی‌کنیم که آن‌ها را در برنامه catch کنیم
    - حتی اگر آن را catch کنیم، کار مهمی در قبال این خطاهای نمی‌توانیم انجام دهیم
    - مانند: OutOfMemoryError

## سلسله‌مراب کلاس‌های استثنا



- مهمنه: استثناهای چکنشده، فقط توسط کامپایلر چک نمی‌شوند
- رفتار استثناهای چکشده و چکنشده در زمان اجرا کاملاً مشابه است

## استثناهای چکشده و چکنشده

### • استثناهای چکشده (Checked Exception)

- کامپایلر جاوا بررسی می‌کند:
- برنامه باید استثنای پیش آمده را دریافت کند یا احتمال پرتابشدن آن را اعلام کند
- وگرن، خطای کامپایلر رخ می‌دهد

### • استثناهای چکنشده (Unchecked Exceptions)

- کامپایلر دریافت یا اعلان پرتاب را اجبار نمی‌کند (در زمان کامپایل چک نمی‌شود)
- کلاس‌های این نوع، عبارتند از :
  - کلاس Error
  - کلاس RuntimeException
  - زیرکلاس‌های RuntimeException و Error

## استثنای چکنشده

```
private static void function(String arg) {  
    System.out.println(1 / arg.length());  
}  
public static void main(String[] args) {  
    function("");  
}
```

- متده است `function()` ممکن است `ArithmaticException` پرتاب کند

- ولی کلیدوازه `throws` را تصریح نکرده است

- برای بعضی از استثنایها، مثل `ArithmaticException`، ذکر `throws` واجب نیست

- اگر این کار اجباری بود، هر متده که عملگر تقسیم ریاضی داشت باید `throws` را ذکر می‌کرد

- با این کار برنامه‌ها پر از `throws` های نامهم می‌شوند

- انواع استثنای چکنشده (`Unchecked Exceptions`)

- مثل `ArrayIndexOutOfBoundsException` و `ArithmaticException`

```
void example(int x) {  
    if(x==1)  
        throw new Error();  
    if(x==2)  
        throw new RuntimeException();  
    if(x==3)  
        throw new NullPointerException();  
    if(x==3)  
        throw new IOException();  
}
```

### مثال

بدون اشکال: زیرا `RuntimeException` و `NullPointerException` استثنای `Unchecked Exception` چکنشده هستند

Syntax Error:  
Unhandled Exception Type IOException

زیرا `IOException` یک استثنای چکشده (`Checked Exception`) است

- اگر می‌خواهید یک نوع `Exception` جدید ایجاد کنید

- اگر نمی‌خواهید کامپایلر آن را چک کند، آن را چکنشده تعریف کنید

- برای این کار، کلاس جدید را فرزند `RuntimeException` قرار دهید

## کلاس‌های استثنا و سلسله مراتب

- در یک عبارت try-catch :

• اگر در یک بلاک catch یک نوع استثنا را دریافت کنیم، نمی‌توانیم در یک catch بعدی زیرکلاس آن نوع استثنا را دریافت کنیم

- در این صورت، کامپایلر اعلام خطای می‌کند :

```
try {
    int a = Integer.parseInt(args[0]);
    int b = Integer.parseInt(args[1]);
    System.out.println(a/b);
} catch (Exception ex) {
    //..
} catch (ArrayIndexOutOfBoundsException e) {
    //..
}
```

مثال:

چرا؟

## چرا متدهای در زیرکلاس نمی‌تواند استثناهای بیشتری پرتاب کند؟

- اگر این قانون وجود نداشت، تعریف کلاس Child بدون خطای می‌شد:

```
class Parent{
    void f() {}
}
class Child extends Parent{
    void f() throws IOException {}
}
```

• در این تعریف، به نوعی رابطه is-a بین Child و Parent نقض شده است

```
void example() {
    Parent p = new Child();
    p.f();
}
```

• مثلاً کامپایلر نمی‌تواند متدهای example را مجبور کند که خطای IOException را throws یا catch کند

نتیجه؟

```
class Parent{  
    void f() throws ArithmeticException{}  
}  
  
class Child extends Parent{  
    void f()  
        throws ArithmeticException,  
        IOException{}  
}
```

خطای کامپایلر

نتیجه؟

```
class Parent{  
    void f() throws ArithmeticException{}  
}  
  
class Child extends Parent{  
    void f() throws Exception{}  
}
```

خطای کامپایلر

نتیجه؟

```
class Parent{  
    void f() throws Exception{}  
}  
  
class Child extends Parent{  
    void f() throws ArithmeticException{}  
}
```

بدون خطأ

```
try{
    f();
}catch (IOException ex) {
    Log(ex);
}catch (SQLException ex) {
    Log(ex);
}catch (ClassCastException ex){
    throw ex;
}
```

قبل از جاوا ۷:

از جاوا ۷ به بعد می‌توانیم:

```
try{
    f();
}catch (IOException | SQLException ex) {
    Log(ex);
}catch (ClassCastException ex){
    throw ex;
}
```

## امکان try-with-resources

قبل از جاوا ۷:

```
BufferedReader br = new BufferedReader(new FileReader(path));
try {
    return br.readLine();
} finally {
    if (br != null) br.close();
}
```

از جاوا ۷ به بعد می‌توانیم:

```
try (BufferedReader br =
      new BufferedReader(new FileReader(path))) {

    return br.readLine();
}
```

## استفاده نادرست از استثنایها

- ابزار کنترل جریان اجرای برنامه (Flow Control) دستورات شرطی (if ، حلقه‌ها (مثل for) و ...)

- نباید از چارچوب استثنایها برای کنترل فرایند اجرا استفاده کنیم
- از Exception فقط برای مدیریت خطأ و استثنایها استفاده کنید

```
void useExceptionsForFlowControl() {  
    try {  
        while (true) {  
            increaseCount();  
        }  
    } catch (MaxReachedException ex) {}  
    // Continue execution  
}  
  
void increaseCount() throws MaxReachedException {  
    if (count++ >= 5000)  
        throw new MaxReachedException();  
}
```

- مثال از کاربرد نامناسب استثنایها

## بازپرتاب استثنا و پرتاپ استثنای جدید

```
try {  
    ...  
}catch (IOException ex) {  
    ...  
    throw ex;  
}
```

- گاهی استثنا باید re-throw شود

- یعنی catch شود، کارهایی انجام شود، و سپس دوباره throw شود

- گاهی هم خطای جدیدی در بلاک catch پرتاپ می‌شود

- یعنی هر کاری که ممکن است در catch انجام می‌دهیم و سپس خطای جدیدی ایجاد و پرتاپ می‌کنیم

```
try {  
    ...  
}catch(IOException e) {  
    ...  
    throw new ReportDataException(e);  
}
```

- دستور assert

- تفاوت آن با JUnit assertions

- بسیاری از متخصصان معتقدند «استثنای چک شده» تجربه خوبی نبود

○ <http://www.mindview.net/Etc/Discussions/CheckedExceptions>

## Annotation

adding metadata information

alternative to the use of XML descriptors and marker interfaces

### compilation time:

Inform the compiler about warnings and errors

Manipulate source code at

### Runtime:

Modify or examine behavior at runtime

### Sample of Creating our annotation: serializing

mark the fields that we are going to include in the generated JSON:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface JsonElement {
    public String key() default "";
}
```

The annotation declares one String parameter with the name "key" and an empty string as the default value.

When creating custom annotations with methods, we should be aware that these methods must have no parameters, and cannot throw an exception. Also, the return types are restricted to primitives, String, Class, enums, annotations, and arrays of these types, and the default value cannot be null.

mark classes that can be serialized into JSON:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.Type)
public @interface JsonSerializable {
}
```

## runtime visibility / apply it to types (classes)

Let's imagine that before serializing an object to a JSON string, we want to execute some method to initialize an object. For that reason, we're going to create an annotation to mark this method:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Init{
}
```



We declared a public annotation with runtime visibility that we can apply to our classes' methods.

This type has a method that capitalizes the first letter of the first and last names. We'll want to call this method **before serializing the object**:

حالا استفاده از annotation‌هایی که ساختیم :

```
@JsonSerializable
public class Person {

    @JsonElement
    private String firstName;

    @JsonElement
    private String lastName;

    @JsonElement(key = "personAge")
    private String age;

    private String address;

    @Init
    private void initNames() {
        this.firstName = this.firstName.substring(0, 1).toUpperCase()
            + this.firstName.substring(1);
        this.lastName = this.lastName.substring(0, 1).toUpperCase()
            + this.lastName.substring(1);
    }

    // Standard getters and setters
}
```

By using our custom annotations, we're indicating that we can serialize a *Person* object to a JSON string. In addition, the output should contain only the *firstName*, *lastName*, and *age* fields of that object. Moreover, we want the *initNames()* method to be called before serialization.

By setting the *key* parameter of the *@JsonElement* annotation to "personAge," we are indicating that we'll use this name as the identifier for the field in the JSON output.

For the sake of demonstration, we made *initNames()* private, so we can't initialize our object by calling it manually, and our constructors aren't using it either.

## Annotation ۱۴.۲

گهگاه تیاز است تا توضیحاتی در سطح برنامه نوشته شود و از این توضیحات در روند اجرای برنامه استفاده نمود. برای اینکار می‌توان از Annotation‌ها استفاده نمود.

### Annotation

```
@Retention(RetentionPolicy.RUNTIME)

@interface Check {

    public String checkerName() default "Nadarad";

    public boolean status() default false;
}
```

در این حالت نیاز است تا ما توضیحات را قاعده مند کنیم، در مثال بالا یک توضیح به نام Check ایجاد کرده ایم که این چک دارای دو خصوصیت checkerName و status می باشد مقدار پیش فرض این خصوصیات به ترتیب false و Nadarad می باشد.

#### Annotation

```
@Check(checkerName = "RezaAmini", status = true)

class Amirsam {

    public static void main(String[] args) {

        Annotation[] annotations = Amirsam.class.getAnnotations();

        if (annotations[0].toString().equals("@test.controller.Check(
            status=true, checkerName=RezaAmini)")) {

            System.out.println("in class tavasote aghaye RezaAmini
                check shode ast");

        } else {

            System.out.println("in class check nashode ast!");

        }
    }
}
```

در این حالت خروجی برنامه RezaAmini check shode ast خواهد بود.

#### Annotation

```
@Check(checkerName = "RezaAmini", status = false)

class Amirsam {

    public static void main(String[] args) {

        Annotation[] annotations = Amirsam.class.getAnnotations();

        if (annotations[0].toString().equals("@test.controller.Check(
            status=true, checkerName=RezaAmini)")) {

            System.out.println("in class tavasote aghaye RezaAmini
                check shode ast");

        } else {

            System.out.println("in class check nashode ast!");

        }
    }
}
```

حال اگر مقدار status را همانند مثال بالا به false تغیر دهیم خروجی برنامه خواهد بود.

\*\*\*

## Serialize

کلاس های **Serializable** قایل تبدیل به byte هستند.

اگر (ram(heap) پاک بشه همه ی Class object ها و obj instance هاها از بین میروند. میتوان با serialize کردن، یک obj را تبدیل به String کرد (برای حفظ state آن) و اون رو در Hard ذخیره کرد و در صورت نیاز به heap برگرداند.

با impl کردن یک کلاس میتوان obj های آن را file binary کنیم و بر عکس. serializable ها wrapper class را impl کرده اند.

کلاس های فیلد ها هم اگر Serializable باشند بصورت زنجیروار serialize میشوند. برای کپی کردن یک شبی؛ میتواند معادل cleanable از نوع deep باشد. (prototype design pattern) Serializable میگیرد. ( مثل cleanable از نوع deep copy ) : deep copy انجام میدهد، یعنی obj رفته و برگشته تماما ref های جدید میگیرد.

Deep copy یعنی نه تنها برای object جدید (و فیلد های primitive wrapper هایش) ref جدید میدهد، بلکه برای کلاس های composition هم، copy کامل انجام میشود و ref جدید میدهد. البته در serialize کردن یا jackson؛ فایلی هست که در مرحله اول write میشه و سپس read میشه، ولی در clone یکسره این کار انجام نمیشود. روی فیلد های static clone انجام نمیشود.

یک راه دیگر استفاده از **jackson** است که در فایل خودش توضیح داده شده است.

## شیاء Serializable

- بسیاری از کلاس‌ها در جاوا **Serializable** هستند
- مثل `ArrayList`, `Integer`, `String` و ...
- اشیاء این کلاس‌ها قابل تبدیل به «جریانی از بایت‌ها» هستند
- و قابل بازسازی از «جریانی از بایت‌ها» هستند
- عملیات **Serialization** یعنی تبدیل شیء به یک جریان با پرتری
- در این عملیات، همه ویژگی‌های درون شیء (یعنی فیلدها) ذخیره می‌شوند
- البته به جز فیلدی‌ای که با کلیدواژه **transient** مشخص شده باشند
- معنای برچسب **transient** برای فیلدی‌ای کلاس:
- هنگام عملیات **Serialization** فیلدی‌ای **transient** ذخیره نمی‌شوند

```
class User implements Serializable {  
    private String username;  
    private transient String password;  
    ...  
}
```

```
class Student implements Serializable {  
    private String name;  
    private double[] grades;  
    private transient double average = 17.27;  
}
```

```

FileOutputStream f1 = new FileOutputStream("c:/1.txt");
ObjectOutputStream out = new ObjectOutputStream(f1);
Student st = new Student("Ali", new double[]{17.0, 18.0});
System.out.println(st.name);
System.out.println(st.average);
out.writeObject(st);
out.close();

```

Ali  
17.5

مثال

```

FileInputStream f2 = new FileInputStream("c:/1.txt");
ObjectInputStream in = new ObjectInputStream(f2);
Student s2 = (Student) in.readObject();
System.out.println(s2.name);
System.out.println(s2.average);
in.close();

```

Ali  
0.0

```

class Student implements Serializable {
    String name; double[] grades;
    transient double average;
    ...
}

```

```

public static void main(String[] args) throws Exception {
    DocumentBuilderFactory abstractFactory
        = DocumentBuilderFactory.newInstance();
    DocumentBuilder documentBuilder = abstractFactory.newDocumentBuilder();
    System.out.println(abstractFactory.getClass());
    System.out.println(documentBuilder.getClass());
    byte[] s = ("<person><firstName>Hosein"
        "</firstName><lastName>Zare</lastName></person>").getBytes( charsetName: "UTF-8" );
    Document parse
        = documentBuilder
            .parse( new ByteArrayInputStream(s))
    ;
    parse.normalizeDocument();
    System.out.println(parse);
}

```

با این روش هم میشه کرد: serialize

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.Type)
public @interface JsonSerializable {
}
```

## Logger

```
public static void main(String[] args) throws IOException {
    Logger logger = Logger.getLogger("Log");
    logger.setLevel(Level.FINEST);

    ConsoleHandler handler = new ConsoleHandler();
    handler.setLevel(Level.FINER);
    logger.addHandler(handler);

    FileHandler fileHandler = new FileHandler(pattern: "log.xml");
    fileHandler.setLevel(Level.ALL);
    logger.addHandler(fileHandler);

    logger.finest(msg: "Finest Message");
    logger.finer(msg: "Finer Message");
    logger.fine(msg: "Fine Message");
}
```

```
Feb 25, 2018 4:22:43 PM com.javaland.chain.Test main
FINER: Finer Message
Feb 25, 2018 4:22:43 PM com.javaland.chain.Test main
FINE: Fine Message
```

```
<message>Finest Message</message>
</record>
<record>
<date>2018-02-25T16:22:43</date>
<millis>1519563163890</millis>
<sequence>1</sequence>
<logger>Log</logger>
<level>FINER</level>
<class>com.javaland.chain.Test</class>
<method>main</method>
<thread>1</thread>
<message>Finer Message</message>
</record> I
<record>
<date>2018-02-25T16:22:43</date>
<millis>1519563163930</millis>
```

به هر از responsibility گفتیم در سطح console chain اجرایی شود. مثلاً به گفتیم در سطح .Finer یا میتوانیم برای یک chain تعریف کنیم برای هر کلاس در چه سطحی اجرای شود. مثلاً به file chain برای بگیم برای hibernate در سطح Error و spring Trace را در سطح در این کار کن.

میخواهیم بصورت داینامیک بر اساس انتخاب کلاینت از بین آپشن های destination، سیستم یک دنباله از chain درست کند: