

۵	فصل (۱) تعاریف متداول
۵	(۱.۱) نکات
۶	(۱.۲) نام‌گذاری
۶	(۱.۳) متغیرها
۷	(۱.۴) مدیریت JDK
۸	(۱.۴.۱) نصب JDK
۸	(۱.۴.۲) حذف JDK
۹	(۱.۴.۳) Java Web Start
۹	(۱.۵) نمونه کد ساده (Hello World)
۱۱	فصل (۲) Java Virtual Machine
۱۱	(۲.۱) حافظه Heap
۱۲	(۲.۱.۱) Garbage Collection
۱۲	(۲.۲) حافظه Stack
۱۲	(۲.۳) Class Loader
۱۲	(۲.۴) Class Path
۱۳	(۲.۵) Meta Space و Perm Gen (Permanent Generation)
۱۴	فصل (۳) مفاهیم پایه
۱۴	(۳.۱) دستورات شرطی
۱۵	(۳.۲) حلقه‌ها
۱۵	(۳.۳) فایل Jar
۱۶	(۳.۴) bytecode
۱۷	فصل (۴) شیء‌گرایی و مفاهیم شیء
۱۷	(۴.۱) مقداردهی اولیه
۱۸	(۴.۱.۱) متد Constructor
۲۰	(۴.۱.۲) بلوک Initialization
۲۰	(۴.۲) رهاسازی شیء
۲۰	(۴.۲.۱) متد Destructor
۲۰	(۴.۲.۲) متد finalize
۲۱	(۴.۳) عملگر Instanceof
۲۲	(۴.۴) کلاس‌های مهم در جاوا
۲۲	(۴.۴.۱) کلاس‌های «لغاف‌انواع اولیه»
۲۳	(۴.۵) متدهای مهم در جاوا
۲۳	(۴.۵.۱) متد ToString
۲۴	(۴.۵.۲) متد Equals
۲۶	(۴.۵.۳) متد getClass
۲۷	(۴.۵.۴) تکنیک Hash
۲۷	(۴.۵.۴.۱) متد hashCode
۲۸	فصل (۵) دسته‌بندی و سطوح دسترسی
۲۸	(۵.۱) Package [دسته‌بندی دیتا برنامه]

۲۸	پوشه‌بندی پروژه‌ها	(۵.۱.۱)
۲۹	Encapsulation یا محصورسازی	(۵.۲)
۲۹	Access Level یا سطوح دسترسی	(۵.۳)
۲۹	Public [مخصوص متدها و ویژگی‌ها و کلاس‌ها]	(۵.۳.۱)
۲۹	private [مخصوص متدها و ویژگی‌ها]	(۵.۳.۲)
۳۰	Package Access [مخصوص متدها و ویژگی‌ها و کلاس‌ها]	(۵.۳.۳)
۳۰	Protected [کاربرد در وراثت، برای متدها و ویژگی‌ها]	(۵.۳.۴)
۳۰	مرور سطح دسترسی	(۵.۳.۵)
۳۱	فصل ۶ شیء‌گرایی و مفاهیم اصلی	
۳۱	Method Overloading یا سربارکردن متد	(۶.۱)
۳۲	مفاهیم Static	(۶.۲)
۳۲	ویژگی استاتیک	(۶.۲.۱)
۳۳	متد استاتیک	(۶.۲.۲)
۳۳	مفاهیم Final	(۶.۳)
۳۳	متغیر final	(۶.۳.۱)
۳۴	متد final	(۶.۳.۲)
۳۴	کلاس final	(۶.۳.۳)
۳۵	اشیاء «تغییرپذیر» یا «تغییرناپذیر»	(۶.۳.۴)
۳۵	مفاهیم وراثت (Inheritance)	(۶.۴)
۳۶	Override	(۶.۴.۱)
۳۶	Super	(۶.۴.۲)
۳۸	کلاس Object	(۶.۴.۳)
۳۸	مقداردهی اولیه در وراثت	(۶.۴.۴)
۳۹	توضیحات تکمیلی	(۶.۴.۵)
۴۰	چندریختی یا Polymorphism	(۶.۵)
۴۲	شرایط کدنویسی بدون چندریختی	(۶.۵.۱)
۴۲	مفهوم انتزاعی (Abstract)	(۶.۶)
۴۲	متد انتزاعی	(۶.۶.۱)
۴۴	کلاس انتزاعی	(۶.۶.۲)
۴۵	مفهوم Interface	(۶.۶.۳)
۴۶	جزئیات ارث‌بری Interface	(۶.۶.۳.۱)
۴۷	متغیر	(۶.۶.۳.۲)
۴۸	annotation	(۶.۷)
۴۹	Data Structure and Containers [ساختمان داده‌وظرف‌ها]	فصل ۷ (۷)
۴۹	Collection	(۷.۱)
۵۰	تفاوت اصلی set و List	(۷.۱.۱)
۵۰	واسط List	(۷.۲)
۵۱	Array	(۷.۲.۱)
۵۲	کلاس String	(۷.۲.۱.۱)
۵۳	تبدیل Collection به آرایه	(۷.۲.۱.۲)
۵۴	کلاس ArrayList	(۷.۲.۲)

۵۵	متدهای مهم (۷.۲.۲.۱)
۵۶	تبدیل آرایه به ArrayList (۷.۲.۲.۲)
۵۶	تبدیل ArrayList به آرایه: (۷.۲.۲.۳)
۵۶	لیست پیوندی (Linked List) (۷.۲.۳)
۵۸	واسط Set [مجموعه] (۷.۳)
۵۸	کلاس HashSet (۷.۳.۱)
۵۹	واسط Map (۷.۴)
۶۰	کلاس HashMap (۷.۵)
۶۰	put (۷.۵.۱)
۶۱	واسط Iterator (۷.۶)
۶۲	متد Iterator (۷.۶.۱)
۶۲	تغییر همزمان یا Concurrent Modification (۷.۶.۲)
۶۴	واسط Comparable (۷.۷)
۶۴	متد compareTo (۷.۷.۱)
۶۵	واسط Comparator (۷.۷.۲)
۶۶	داده‌های Enumeration (۷.۸)
۶۷	Variable Argument List متغیر «متفاوت پارامتر» (۷.۹)
۶۸	کلاس Arrays (۷.۱۰)
۶۹	کلاس Collections (۷.۱۱)
۷۰	انواع داده عام Generic (۷.۱۲)
۷۰	نحوه تعریف کلاس عام (۷.۱۲.۱)
۷۱	ایجاد کلاس‌های عام دلخواه (۷.۱۲.۲)
۷۲	برخی کلاس‌ها نوع عام جاوا (۷.۱۲.۳)
۷۳	محدود کردن نوع عام (۷.۱۲.۴)
۷۴	نوع خام (Raw Type) (۷.۱۲.۵)
۷۴	استنتاج نوع یا Type Inference (۷.۱۲.۶)
۷۴	وراثت و انواع داده عام (۷.۱۲.۷)
۷۵	متدهای عام یا Generic Method (۷.۱۲.۸)
۷۶	فرآیند محو (Erasure) (۷.۱۲.۹)
۷۷	محدودیت‌ها در تعریف «نوع عام» (۷.۱۲.۱۰)
۷۹	مرور برخی واسط‌ها و کلاس‌های مهم دیگر (۷.۱۳)
۸۰	برخی مفاهیم (۷.۱۴)
۸۱	فصل ۸ Stream
۸۱	Scanner class (۸.۱)
۸۲	فصل ۹ مدیریت خطا (Exception)
۸۲	Exception Object (۹.۱)
۸۳	ایجاد Class Exception جدید (۹.۱.۱)
۸۴	Exception Handling Framework (۹.۲)
۸۶	Try (۹.۲.۱)
۸۶	امکان try-with-resources (۹.۲.۱.۱)
۸۶	Catch یا Exception Handler (۹.۲.۲)

۸۷throw (۹.۲.۳)
۸۸throws (۹.۲.۴)
۸۹finally (۹.۲.۵)
۸۹Throwable کلاس (۹.۳)
۸۹Unchecked Exception استثناء چک نشده (۹.۴)
۹۰checked Exception استثناء چک شده (۹.۵)
۹۰illegal Exception استثناء (۹.۶)
۹۱مفاهیم تکمیلی (۱۰ فصل)
۹۱Java Documents (۱۰.۱)
۹۲Refactoring (۱۰.۲)
۹۴تست نرم افزار (JUnit) (۱۱ فصل)
۹۴مفاهیم (۱۱.۱)
۹۵نکات Junit (۱۱.۲)
۹۵JUnit Framework (۱۱.۳)
۹۶قطعه کدها و مثال ها (۱۱.۴)

فصل ۱) تعاریف متداول

- طراحی اولیه جاوا به سال ۱۹۹۱ برمی‌گردد و اولین نسخه جاوا توسط شرکت «سان‌میکروسستم» در سال ۱۹۹۵ منتشر شد و شرکت سان توسط شرکت اوراکل در سال ۲۰۱۰ خریداری شد
- مروری بر نسخه‌های جاوا
 - J۲SE: یعنی نسخه دوم از جاوای Standard Edition
 - J۲ME: یعنی نسخه دوم از جاوای Micro Edition
 - J۲EE: یعنی نسخه دوم از جاوای Enterprise Edition

JDK Alpha and Beta (1995)
JDK 1.0 (January 21, 1996)
JDK 1.1 (February 19, 1997)
J2SE 1.2 (December 8, 1998)=Java2StandardEdition
J2SE 1.3 (May 8, 2000)
J2SE 1.4 (February 6, 2002)
Java SE 5.0 (September 30, 2004)
Java SE 6 (December 11, 2006)
Java SE 7 (July 28, 2011)
Java SE 8 (March 18, 2014)

JavaSE: مخفف عبارت Java Standard Edition است.

JavaEE: مخفف عبارت Java Enterprise Edition است.

JME یا **JavaME**: مخفف عبارت Java Micro/Mobile Edition و عنوان خیلی کوچک‌شده از جاوا حتی کوچکتر از Java Core که برای استفاده جاوا در دستگاه‌های الکترونیکی بکار می‌رود.

JDK: مخفف Java Development Kit است و نرم‌افزاری است که به کاربر اجازه اجرا و کامپایل برنامه‌های جاوا را می‌دهد. (بدرد برنامه نویسان جاوا می‌خوره)

JRE: مخفف عبارت Java Runtime Environment است.

بخش کوچکی از JDK که فقط برای اجرای یک برنامه جاوا احتیاج است و شامل موارد موجود در jdk نیست

Javac: مخفف عبارت Java Compiler است

IDE: مخفف عبارت Integrated Development Environment است. محیط‌هایی که مجموعه‌ای از امکانات مورد نیاز برنامه‌نویس را بصورت یکپارچه همانند یک جعبه‌ابزار فراهم می‌کنند. معروف‌ترین‌های آن نظیر: Eclipse یا NetBeans یا IntelliJ IDEA

Indentation: دندانه‌گذاری یا همان تورفتگی در دستورات کد جهت ظاهر زیبای قطعه کد نوشته شده

پیمایش: اگر عملی را روی تک تک عناصر یک آرایه انجام دهیم به اینکار پیمایش آرایه می‌گویند

Jshell: قابلیتی که از جاوای ۹ به بعد آمده و اجازه می‌دهد کدهای کوچک که در حد اسکریپتی است را بدون ساختن کلاس تست کنیم. که در IDE IntelliJ در مسیر Tools>Jshell Console قرار دارد

۱.۱) نکات

- ترتیب اجرای برنامه‌ها به صورت زیر است
[Fortran, C, Pascal] > High Level Language > Assembly Language > Machine Language > hardware
- زبان جاوا برای اجرا بصورت زیر عمل می‌نماید:
JVM > Byte Code(.class) < Javac compiler < Java Code(.Java) < User Interface

- زبان‌های برنامه‌نویسی نظیر Scala و groovy برای ران شدن نیاز به وجود JVM دارند
- بازه عددی متغیر int عددی بین منفی ۲ به توان ۳۱ تا یک عدد کمتر از مثبت ۲ به توان ۳۱ است.
- برای استفاده از اعداد کوچکتر از int می‌توان از short یا byte نیز استفاده نمود
- نکته: در جاوا وقتی از `/**` استفاده می‌نماییم یعنی داکیومنت کنونی با جاواداک در ارتباط است
- نکته: برای اینکه بدانیم یک متغیر در کجاها در workspace استفاده شده است می‌توانیم روی محل تعریف یک متغیر راست کلیک کرده و از گزینه‌های موجود references را باز کرده و از زیرمنوی workspace استفاده نماییم.
- نکته: برای رفتن به مکان تعریف یک متغیر یا کلاس یا متد یا غیره از کلید F۳ استفاده می‌شود
- برای ایجاد کامنت از شیوه زیر استفاده می‌شود

```
//This is a temporary variable
or
/* Data */
```

۱.۲ نام‌گذاری

- بصورت کلی پیشنهاد می‌شود که در نامگذاری‌ها از فعل استفاده نشود.
- جاوا به حروف بزرگ و کوچک حساس می‌باشد (Case Sensitive Names)
 - بسته‌ها: همه حروف کوچک مثل `com.sun.eng`
 - کلاس‌ها: Camel Case یعنی با حروف بزرگ شروع شود و همه حروف اول کلمه بزرگ و بقیه کوچک
 - مثل: `ImageSprite` یا `Raster` یا `StreamReader` و غیره
 - از اسم برای نام‌گذاری استفاده نمایید
 - متدها: شروع با حروف کوچک و سپس ادامه در وضعیت Camel Case
 - مثل: `getBackground` یا `runFast` و غیره
 - از فعل برای نام‌گذاری استفاده نمایید
 - متغیرها: شروع با حرف کوچک و ادامه با الگوی «کوهان‌شتر»
 - مثل: `myWidth` یا `maxNumber`
 - ثابت‌ها: همه حروف بزرگ، کلمات مختلف در نام با `Undeline` جدا شوند
 - مثل: `MIN_WIDTH`
- برای کلاس‌ها و متغیرها و متدها از اسامی با معنی و گویا استفاده شود و نام‌هایی از قبیل زیر مناسب نیستند:
 - `a, b, c, x, y, z, a۱, var۱, var۲, method۱, Class, MyClass, myMethod`
- برای متغیرهای شمارنده مثل استفاده در حلقه `for` نام‌هایی مثل `i` و `z` اشکالی ندارد
- اگر از نام‌های مناسب استفاده شود ممکن است نیاز به جاواداک را کمتر نماید

۱.۳ متغیرها

- انواع متغیرها:
 - انواع داده‌اولیه یا **Primitive Data Types**: همانند داده‌هایی از نوع `Byte` یا `Short` یا `Int` یا `Long` یا `Float` یا `Double` یا

Boolean یا Char که هر متغیر از این انواع، حاوی یک مقدار است و شیء نیست و حتی به شیء نیز اشاره نمی‌کند. بلکه مستقیماً یک مقدار را نگهداری می‌کنند. همچنین امکان استفاده از new ندارند.

○ **انواع داده ارتجاعی یا Reference Data Types:** همانند داده‌هایی از نوع String یا Scanner یا مشابه که از نوع کلاس هستند یا کلاس‌هایی که ما خودمان ایجاد می‌کنیم نظیر Book یا Person یا غیره که هر متغیر از این انواع ارجاع به شیء است. همچنین از new باید استفاده شود

• اگر عدد از نوع double باشد برای اینکه تصریح کنیم که عدد از نوع double است می‌توانیم در انتهای عدد یک علامت d بگذاریم

```
d=4763.129837D; // تصریح اینکه متغیر از نوع دابل است
f=4763.129837F; // تصریح اینکه متغیر از نوع فلوت است
l=4763.129837L; // تصریح اینکه متغیر از نوع لانگ است
```

```
int a=12;
b--a; // اول مقدار «آ» که ۱۲ هست را به ۱۱ تغییر بده و بعد آن را به «بی» وارد کن
```

```
int a=12;
b=a--; // «بی» به «بی» وارد کن و سپس یک واحد از عدد ۱۲ کم کن «ا» اول مقدار
```

• نمونه «تبدیل داده» Type Cast در جاوا

```
int varINT=2;
long varLONG=20;
varINT = (int) varLONG;
```

• برخی اعداد

```
double x = Double.NaN
double x = Double.NEGATIVE_INFINITY
double x = Double.POSITIVE_INFINITY
double x = Double.MAX_VALUE
int x = Integer.MAX_VALUE
```

۱.۴ مدیریت JDK

لینک مفید: [URL](#)

ابتدا توسط دستورات زیر از ورژن کنونی جاوای خود آگاه شوید:

```
java -version
output:
openjdk version "17.0.6" 2023-01-17
OpenJDK Runtime Environment (build 17.0.6+10-Debian-1deb11u1)
OpenJDK 64-Bit Server VM (build 17.0.6+10-Debian-1deb11u1, mixed mode, sharing)
```

1.4.1 نصب JDK

ابتدا باید نرم افزار oracle jdk را دانلود نمایید. [URL](#) یا [URL](#)

اگر در سیستم شما هیچ نسخه‌ای وجود ندارد آنگاه توسط دستور زیر اقدام نمایید

```
sudo update-alternatives --install "/usr/bin/java" "java" "/opt/<DIRECTORY>/bin/java" 1 #JRE
sudo update-alternatives --install "/usr/bin/javac" "javac" "/opt/<DIRECTORY>/bin/javac" 1 #JDK
sudo update-alternatives --install "/usr/bin/javaws" "javaws" "/opt/<DIRECTORY>/bin/javaws" 1 #Web
```

نکته: عدد صفر میزان اولویت را تعیین می‌کند که این ورژن که نصب می‌شود تحت چه اولویت در سیستم استفاده قرار گیرد
توسط دستور زیر نسخه‌های موجود در سیستم به نمایش درآمده و نیز می‌توان نسخه پیش فرض سیستم را انتخاب نمود:

```
sudo update-alternatives --config java
sudo update-alternatives --config javac
sudo update-alternatives --config javaws
```

توسط دستور زیر از وجود ورژن‌های قبلی آگاه شوید

```
java -version
sudo update-alternatives --list java
sudo update-alternatives --list javac
sudo update-alternatives --list javaws
```

توسط دستور زیر می‌توانیم یک ورژن جدید به سیستم بیافزاییم

```
sudo update-alternatives --set java <PATH>
sudo update-alternatives --set javac <PATH>
sudo update-alternatives --set javaws <PATH>
```

1.4.2 حذف JDK

برای حذف یک نسخه از جاوا بهتر است از دستور زیر اقدام نمایید
گام اول: مشاهده نسخه‌ها به همراه مسیرهای آنها

```
java -version
sudo update-alternatives --list java
sudo update-alternatives --list javac
sudo update-alternatives --list javaws
```

گام دوم: حذف نسخه مورد نظر توسط ارائه مسیر آن ورژن


```
sudo update-alternatives --remove java <Path>
sudo update-alternatives --remove javac <Path>
sudo update-alternatives --remove javaws <Path>
```

example:

```
sudo update-alternatives --remove java /opt/java8/bin/java
```

گام سوم: تعیین نسخه مطلوب باقیمانده از نسخه‌های جاوا بعنوان پیش‌فرض

```
sudo update-alternatives --config java
sudo update-alternatives --config javac
sudo update-alternatives --config javaws
```

Java Web Start (۱.۴.۳)

در ورژن‌های شماره ۷ یا ۸ مورد javaws یا java web start وجود داشت ولی در ورژن‌های بالاتر این مورد وجود ندارد که برای این کار با نصب بسته icedtea می‌توانیم javaws را به سیستم معرفی کنیم

```
sudo apt install icedtea-netx
sudo yum install icedtea-web
```

در این صورت برنامه‌های با فرمت jnlp می‌توانیم با دستور زیر اجرا نماییم

```
/usr/bin/javaws /tmp/launch.jnlp
```

۱.۵) نمونه کد ساده (Hello World)

برای نوشتن یک برنامه ساده به روش زیر عمل می‌نماییم:

```
vim First.java
Public class First {
    public static void main(String[] args) {
        System.out.println("!!! اولین برنامه")
    }
}
```

برای کامپایل کردن یک برنامه جاوا نوشته شده از دستور زیر استفاده می‌نماییم:

```
javac First.java
```

در این صورت اگر برنامه به درستی نوشته شده باشد فایل کامپایل شده آن بصورت First.Class ساخته می‌شود. در این صورت باید با دستور زیر برنامه را اجرا کنیم:

java First

نکته: اسم کلاس باید دقیقاً با اسم فایل منطبق باشد

فصل ۲) Java Virtual Machine

- یک برنامه نوشته شده به زبان C است که بعنوان یک میان افزار تلقی می شود
- هر برنامه در جاوا بر روی JVM اجرا می شود

نکته: در قطعه کد زیر موارد اشاره شده در حافظه های متناظر قرار می گیرند:

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

حافظه Stack متغیرهایی نظیر number و tmp و s1 و s2 را در خود جای می دهد که متغیر number مقدار نگهداری می کند ولی موارد دیگر آدرس (شیء) را نگهداری می کنند
حافظه Heap مو اردی نظیر : آدرس جایی که number و tmp و s1 و s2 دارند به آن اشاره میکنند

۲.۱) حافظه Heap

همه اشیاء در بخشی از حافظه بنام Heap در رم ذخیره می شود
عملگر new حافظه را در بخشی بنام Heap ایجاد می کند.
هر شیء که new می شود روی Heap قرار می گیرد

برای تعیین اندازه اولیه Heap از عبارت **Xms** استفاده می شود.
برای تعیین حداکثر اندازه Heap از عبارت **Xmx** استفاده می شود.
بطور مثال:

```
java person # فراخوانی کلاس پرسن  
java -Xms512m -Xmx375m Person  
Java -Xmx3750m -Xss4m Main
```

نکته: اگر برنامه ای اشیاء فراوانی را new نماید ممکن است با ارور Java Heap Space روبرو شویم یعنی یک ارور از نوع:

```
OutOfMemoryError: Java heap space
```

Garbage Collection (۲.۱.۱)

- موجودیت Garbage Collector که بخشی از jvm است بدون نیاز به دخالت برنامه‌نویس قابلیت شناسایی و آزادسازی حافظه Heap از اشیاء ساخته شده که دیگر در برنامه دیگر مورد استفاده قرار نمی‌گیرد را دارد.
- زباله روب یا همان Garbage Collection فقط درباره حافظه هست. و نمی‌تواند منابعی مثل فایل را که توسط یک شیء باز شده است را آزاد سازی نماید.

Stack حافظه (۲.۲)

- متغیرهای محلی هر متد در این حافظه قرار خواهند گرفت.
- وقتی یک متد فراخوانی می‌شود به تعداد متغیرهای محلی آن در Stack فضا اشغال می‌شود و وقتی این متد تمام شد حافظه های اختصاص داده شده آزاد خواهد شد
- آزادسازی حافظه استک بصورت خودکار در تمامی زبان‌های برنامه نویسی وجود دارد و کاری سبک و پیچیده نیست
- برای تعیین حداکثر اندازه Stack از عبارت **Xss** استفاده می‌شود.

```
java person # فراخوانی کلاس پرسن
java -Xss4m person
Java -Xmx3750m -Xss4m Main
```

- نکته: اگر یک متد را بصورت بازگشتی صدا بزنینم طوریکه هیچ شرط پایانی نداشته باشد، آنگاه با ارور Stack Overflow روبرو خواهیم شد

```
int Func(int i){
    return Func(i+1);
}
```

Class Loader (۲.۳)

- وقتی کلاسی را برای اولین بار مورد استفاده قرار می‌دهیم، بخشی از JVM بنام Class Loader این کلاس را در قسمتی از حافظه بارگذاری می‌کند که بخشی از این حافظه مختص متغیرهای استاتیک آن کلاس خواهد بود.

Class Path (۲.۴)

- یک پارامتر برای کامپایلر جاوا یا JVM هست.
- مشخص می‌کند در چه محل‌هایی به دنبال کلاس‌ها و بسته‌ها بگردند
- این پارامتر به دستور java یا javac پاس می‌شود

- یا اینکه بصورت یک متغیر محیطی Environment Variable تعریف می‌شود
- همچنین محل کلاس‌های موجود در زبان جاوا نیازی به معرفی در Class Path ندارند مثل String
- با مفهوم path اشتباه نگیرید
- path مربوط به سیستم عامل است و محل فایل‌های اجرایی را مشخص می‌کند
- با مفهوم Working Directory اشتباه نگیرید (دایرکتوری که برنامه در آن مسیر دارد اجرا می‌شود)

عبارت زیر یعنی وقتی جاوا دارد اجرا می‌شود باید فولدر `myprogram` را بگردد تا بتواند کلاس `HelloWorld` را پیدا نماید. عبارتی یعنی کلاس‌های ما در مسیر `home/user/myprogram/` قرار دارند و اسم کامل کلاس هم `org.mypackage.HelloWorld` نام دارد.

```
Windows: java -cp D:\myprogram org.mypackage.HelloWorld
LINUX: java -cp /home/user/myprogram org.mypackage.HelloWorld
```

- میتوان از متغیر محیطی استفاده کرد:

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

- تعیین چندین فولدر یا JAR در مسیر `classpath`:

```
java -cp D:\myprogram;d:\lib\support.jar org.mypackage.HelloWorld
```

- برای جدا کردن بخش‌های مختلف `cp` در لینوکس از کاراکتر `:` و در ویندوز از کاراکتر `;` استفاده می‌شود.
- عبارت `خط -cp` معادل `-classpath` است.
- استفاده از چندین فایل JAR و شاخه جاری به عنوان `cp`:

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

۲.۵ Perm Gen(Permanent Generation) و Meta Space

- بخشی از حافظه مسئول نگهداری اشیاء از قبیل «شیء کلاس» یا «ClassObject» است.
- قبل از نسخه ۸ جاوا اطلاعات کلاس‌ها (شیء کلاس) در بخشی بنام PermGen ذخیره می‌شد
- اگر پروژه بسیار بزرگ داشته باشیم ممکن است این فضا پر شود و با خطای Out Of Memory Error مواجه شویم (برنامه‌ای که کلاس‌های زیادی شامل کتابخانه‌ها و فایل‌های Jar متنوع باشد)
- حجم حافظه PermGen قابل تنظیم است

```
java -XX:MaxPermSize=512m MyClass
```

- بعد از جاوا ۸ اطلاعات مربوط به کلاس‌ها در Meta Space نگهداری می‌شود و PermGen به کلی حذف شده است و دیگر نیازی به تنظیم اندازه PermGen نداریم.

فصل ۳) مفاهیم پایه

دستورات زیر برای چاپ در کنسول هست:

```
System.out.println(a);
System.out.print(X);
System.out.println(a*6+1);
```

۳.۱) دستورات شرطی

اگر شرط درست باشد مقدار value_true اجرا شود و اگر نبود مقدار value_false به اجرا در بیاید

```
TYPE value = CONDITION ? value_true : value_false;
```

عبارت زیر یعنی: یعنی اگر مقدار area از ۱۰ بزرگتر بود مقدار value برابر بشود با ۱ وگرنه مقدار آن بشود منفی یک

```
area=5;
int value = area > 10 ? 1 :-1;
```

عبارت زیر معادل عبارت استفاده شده در بالا است:

```
if (condition) {
    statements;
} else if (condition) {
    statements;
} else {
    statements;
}
```

Switch:

```
switch (I) {
    case 1:
        Commands;
        break;
    case 2:
        Commands;
        break;
    case 3:
        Commands;
        break;
    ...
    default:
        Commands;
}
```

- ترجیحاً برای مقایسه اعداد اعشاری از عملگر == یا != استفاده نشود

۳.۲ حلقه‌ها

For

```
for (opration ; Condition ; opration){دستورات}
for (int i=1;i<=10;i++){
for(String s : array)
    System.out.println(s);
```

do while

```
long counter=0;
do{
    counter++;
    System.out.println(counter);
}while(counter<10);
```

while

```
long counter=1;
while(counter<10){
    System.out.println(counter);
    counter++;
}
```

۳.۳ فایل Jar

- عبارت JAR مخفف Java Archive است.
- بطور معمول یک فایل jar شامل کلاس‌هایی با پسوند نقطه class هستند(و احتمالاً فایل‌های تکمیلی و فایل‌های تنظیماتی)
- معمولاً متن برنامه‌ها (فایل‌های پسوند java) در فایل Jar گنجانده نمی‌شوند

برای ایجاد فایل jar میتوان از دستور زیر استفاده کرد:

```
jar cf <JarFile> <Input Files>
```

jarFile: نام فایل جری که میخواهیم ایجاد کنیم
Input Files: مجموعه برنامه‌های که قرار است در فایل جر ضمیمه شوند
همچنین ابزارهایی نظیر Eclipse و Ant و Maven نیز میتوانند این کار را انجام دهند

bytecode (۳.۴)

- فایل کامپایل شده را ByteCode می‌گویند.
 - فایل با پسوند class که معمولاً یک فایل jar شامل کلاس‌هایی با پسوند نقطه class است.
 - با کمک دستورات javap (در JDK هست) می‌توانیم bytecode یک کلاس را ببینیم. (صفحه مشکی در مثال زیر)
- مثال: کلاس زیر که توسط کامپایلر کامپایل شده است، در ایجاد دو متغیر g1 و g2 به یک شکل ترجمه نموده است.

```
1) public class Generic<T> {
2)     void f() {
3)         Generic<String> g1;
4)         g1 = new Generic<String>();
5)         Generic<Integer> g2;
6)         g2 = new Generic<Integer>();
7)     }
8) }
```

شکل زیر نمونه بایت‌کد کلاس بالا 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
}
```

در شکل سیاه‌رنگ خطوط شماره ۱۰ و ۱۱ و ۱۲ و ۱۳ از خط شماره ۴ موجود در کلاس حاصل شده است (از عبارت new تا عبارت astore_1) و همچنین خطوط شماره ۱۴ و ۱۵ و ۱۶ و ۱۷ از خط شماره ۶ موجود در کلاس حاصل شده است (از عبارت new تا عبارت astore_2) که نشان می‌دهد در حالت generic به ازای هر نوع داده با طرز رفتار یکسان مواجه خواهیم بود.

فصل ۴) شیء‌گرایی و مفاهیم شیء

TODO: اینجا باید مقداری توضیح درباره شیء‌گرایی بدهم و بگویم یک شیء چیست و مزایای شیء‌گرایی چیست

۴.۱) مقداردهی اولیه

- ترتیب «مقداردهی اولیه» به «یک‌شیء» ساخته شده از کلاس بصورت زیر است:
 - همه مقداردهی‌هایی که در خطوط کلاس قرار دارد (Inline Initialization) مثلاً خط اول یا دوم یک متغیر ایجاد و مقداردهی کنیم
 - همه بلوک‌های مقداردهی اولیه یا Initialization Block ها (معمول نیست که در یک کلاس چند بلوک مقداردهی اولیه داشته باشیم)
 - یکی از توابع سازنده (همانی که صدا زده شده است)
 - ترتیب «مقداردهی اولیه» به یک شیء از «کلاس حاوی ویژگی استاتیک» بصورت زیر خواهد بود.
 - اول: مقداردهی به متغیرهای استاتیک
- نمونه مثال مقداردهی 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";}
```

- دوم: مقداردهی به متغیرهای استاتیک توسط «بلوک استاتیک» یا Static Block
- نمونه مثال استفاده از بلوک استاتیک یا همان Static Block

```
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();
}
}
```

- سوم: اقداماتی که مربوط به هر شیء است (فرض شود ویژگی استاتیک نداریم) انجام می‌شود که شامل زیر است:
 - مقداردهی به هر ویژگی به ازای ایجاد هر شیء
 - مقداردهی Inline یا همان در خط به ویژگی‌ها
 - «بلوک مقداردهی اولیه» یا Instance Initialization Block
 - تابع سازنده یا Constructor

- فرض شود بخواهیم تعداد اشیاء زنده از یک کلاس را بدانیم، می توان این موضوع را توسط یک ویژگی استاتیک مدیریت نمود. باتوجه به قطعه کد زیر، توجه نمایید:

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

- توضیحات قطعه کد به شرح زیر است:
 - **ویژگی liveInstances:** استاتیک است زیرا ویژگی کل کلاس است و مختص یک شیء نیست و کل کلاس در هر زمان باید بداند که چند شیء زنده وجود دارد. از نوع Private است زیرا باید دسترسی محدود باشد تا کسی نتواند از بیرون عدد را تغییر دهد.
 - «بلوک مقداردهی اولیه» بابت اینکه هر بار شیء ساخته شود پس دستورات بلاک انجام شود، پس یک واحد به این متغیر افزوده شود
 - متد **getLiveInstances** یک Getter برای ویژگی liveInstances است. از نوع استاتیک است زیرا متدی است که برای خود کلاس قرار است استفاده شود و نه مختص یک شیء خاص و public است زیرا هرکسی بخواهد بتواند این متد را فراخوانی کند
 - متد **finalize:** هرگاه Garbage Collector بخواهد شیء را حذف کند و دور بریزد، قبل از پاک کردن، این متد را فراخوانی می کند و هر گار شیء می میرد باید یک واحد از متغیر liveInstances کاسته شود

۴.۱.۱) متد Constructor

هر بار که از یک کلاس استفاده می شود، در صورتی که آن کلاس دارای تابع Constructor باشد، این تابع فراخوانی خواهد شد.

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

همچنین برای صدا زدن از فرمت زیر استفاده می شود:

```
public static void main(String[] args) {
    Circle c = new Circle(2);
    System.out.println(c.getArea());
}
```

```
}
```

- باید دقیقاً هم نام کلاس باشد. و هیچ نوع برگشتی نباید داشته باشد
- هنگام new نمودن و ایجاد شیء جدید از کلاس بصورت خودکار این تابع فراخوانی خواهد شد
- یک کلاس می تواند سازنده های مختلفی داشته باشد. (به شرط اینکه پارامترهای مختلفی داشته باشد)

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

- در صورت عدم تعریف تابع سازنده برای یک کلاس، کامپایلر جاوا بصورت خودکار، یک تابع سازنده بدون پارامتر ورودی با بدنه خالی در نظر خواهد گرفت
- وقتی برنامه نویس اولین constructor را تعریف نماید جاوا سازنده پیش فرض برای این کلاس اضافه نمی کند.
- هرگاه یک شیء ساخته می شود سازنده اش فراخوانی می شود. یعنی سازنده باید ویژگی های شیء را مقدار دهی نماید. در صورت لزوم ، ویژگی هایی که خودشان شیء هستند را هم ایجاد نماید.

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

- اگر در تابع سازنده مقدار یک ویژگی یا همان Property مشخص نشود، آنگاه هر ویژگی مقدار پیش فرض نوع داده خودش را میگیرد.
- یعنی integer , char , long مقدار صفر و Boolean مقدار False و متغیرهای ارجاعی (اشیاء) مقدار Null را به خود می گیرند
- گاهی یک تابع سازنده در بدنه خود یک سازنده دیگر را فراخوانی می نماید. در این صورت باید این فراخوانی حتماً اولین دستور سازنده باشد، این فراخوانی به کمک پارامتر this صورت می گیرد.

```
public class Country {  
    private String name;  
    Private int population;  
  
    public Country(){  
        name = "Iran" ;  
    }  
    public Country(int number){
```

```

        this();
        population = number;
    }
    public Country(String n, int number) {
        this(number);
        name = n;
    }
}

```

Initialization بلوک (۴.۱.۲)

همانند تابع سازنده سبب مقداردهی اولیه به کلاس است و هرگاه یک شیء جدید از کلاس مذکور ساخته شود این بلاک اجرا خواهد شد به قطعه کد زیر توجه نمایید:

```

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

```

۴.۲ رهاسازی شیء

۴.۲.۱ متد Destructor

- مفهوم Destructor در مقابل مفهوم سازنده (Constructor) است.
- جاوا نیازی به Destructor ندارد و مفهوم Garbage Collector کارهای مفهوم Destructor را انجام می دهد.
- تابع Destructor سبب آزادسازی شیءهای ساخته شده است
- برخی زبانهای برنامه نویسی مانند ++C از عنوان نابودگر پشتیبانی می کنند
-

۴.۲.۲ متد finalize

- هرچند جاوا Destructor ندارد اما برای هر کلاس متد ویژه ای با نام finalize قابل تعریف است و برنامه نویس می تواند این متد را ایجاد نماید.

- هروقت Garbage Collection بخواید یک شیء را آزاد نماید قبل از آزاد شدن شیء متد Finalize از آن شیء را فراخوانی می کند. (وصیت نامه یک شیء)
- می توان بصورت دستی فرایند زباله رویی را در این متد به اجرا در آورد:

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

- معمولاً برنامه نویس باید اجازه بدهد که Garbage Collector بصورت مقتضی خودش اتوماتیک فراخوانی شود و دستی آن را فراخوانی نکند.
- عملیات زباله روب یا همان Garbage Collection فقط پیرامون حافظه اقدام های خود را انجام می دهد
- ممکن است یک شیء منابعی غیر از حافظه را در اختیار گرفته باشد و این منابع را هنوز آزاد نکرده باشد مثل باز بودن یک فایل یا دسترسی به یک دیتابیس
- پیاده سازی Finalize کار رایجی نیست و کاربردهای خاصی دارد

۴.۳ عملگر instanceof

- این عملگر مشخص می کند که آیا شیء از نوع یک کلاس می باشد یا خیر
- اگر شیء از جنس آن کلاس باشد (نمونه ای از آن کلاس) مقدار True و در غیر این صورت همواره مقدار False را در خروجی برمی گرداند.
- نکته: اگر ارجاع مورد نظر null باشد، این عملگر false برمی گرداند
- این بررسی در زمان اجرا انجام می شود
- قاعدتاً Type باید زیر کلاس Ref باشد
 - اگر Type همان کلاس Ref یا اَبَر کلاس Ref باشد: همیشه true، مگر...
 - اگر Type اَبَر کلاس، زیر کلاس یا خود Ref نباشد: خطای کامپایل (همیشه غلط)
- اگر a نمونه ای از کلاس Type (یا زیر کلاس آن) باشد، true برمی گرداند

```
Ref a = ...           //a: شیء
a instanceof Type    //Type: کلاس
```

- قبل از هر Downcast بهتر است، بررسی صورت گیرد.

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

۴.۴ کلاس‌های مهم در جاوا

۴.۴.۱ کلاس‌های «لفاف انواع اولیه»

متناظر با هر داده نوع اولیه یک کلاس وجود دارد که به این کلاس‌ها Primitive Wrapper Class گفته می‌شود. یعنی هرگاه یک شیء مورد نیاز باشد، آنگاه باید بجای انواع اولیه از Wrapper Class استفاده شود. اشیاء این کلاس‌ها، برخلاف انواع اولیه، می‌توانند NULL باشند. که این موضوع زمانی اهمیت دارد که بخواهیم متغیری تعریف کنیم که می‌تواند بصورت اختیاری توسط کاربر تکمیل شود. یعنی مثلاً سن کاربر بصورت اختیاری در یک فرم قرار داده شده است. که در آن صورت بجای تعریف متغیر int از کلاس Integer برای متغیر سن استفاده می‌شود

- برای نوع اولیه byte یک Wrapper Class متناظر، تحت عنوان Byte وجود دارد.
- برای نوع اولیه short یک Wrapper Class متناظر، تحت عنوان Short وجود دارد.
- برای نوع اولیه int یک Wrapper Class متناظر، تحت عنوان Integer وجود دارد.
- برای نوع اولیه long یک Wrapper Class متناظر، تحت عنوان Long وجود دارد.
- برای نوع اولیه float یک Wrapper Class متناظر، تحت عنوان Float وجود دارد.
- برای نوع اولیه double یک Wrapper Class متناظر، تحت عنوان Double وجود دارد.
- برای نوع اولیه char یک Wrapper Class متناظر، تحت عنوان Character وجود دارد.
- برای نوع اولیه boolean یک Wrapper Class متناظر، تحت عنوان Boolean وجود دارد.

• یک مثال از کلاس‌های wrapper را در زیر مشاهده می‌نمایید:

```
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;
```

- **AutoBoxing**: از نسخه جاوا ۵ به بعد اگر یک مقدار primitive به عنوان شیء استفاده شود آنگاه بصورت خودکار به شیء از نوع متناظر Wrapper تبدیل می‌شود یعنی نمونه زیر در حالت عادی سبب خطا می‌شود ولی با این Auto Boxing مشکلی نخواهد بود

```
Integer i = 2;
```

- **Unboxing**: اگر یک شیء از نوع Wrapper به عنوان یک Primitive استفاده شود آنگاه بصورت خودکار به یک مقدار از نوع متناظر

Primitive تبدیل می‌شود. یعنی نمونه زیر درحالت عادی سبب بروز خطا می‌شود ولی با این unboxing مشکلی پیش نمی‌آید

```
int a = new Integer(12);
```

۴.۵ متدهای مهم در جاوا

۴.۵.۱ متد ToString

هرگاه بخواهیم یک شیء را به رشته تبدیل نماییم از این متد استفاده می‌شود. بطور مثال: برای چاپ اطلاعات یک شیء یا ذخیره در فایل همه کلاس‌ها این متد را دارند، حتی اگر برای آن کلاس تعریف نشده باشد. اما پیاده‌سازی صحیح آن را باید برای کلاس جدید تعریف کنیم

```
Sample1)
Person person = new Person("Ali", 25);
String s = person.toString();
```

مثال دیگر:

```
Sample2)
Integer number = new Integer(12);
String s = number.toString();
```

مثال دیگر:

```
package ir.javacup.oop;
public class Circle {
    private double radius;
    public Circle(double radius){
        this.radius = radius;
    }
    public String toString(){
        return "Circle [radius=" + radius + " ]";
    }
    public static void main(String[] args){
        Circle c = new Circle(2.0);
        System.out.println(c.toString());
    }
}
```

در مثال بالا اگر متد ToString را تعریف نمی‌کردیم خروجی متد پیش فرض ToString جاوا عبارت زیر می‌شد:

```
ir.javacup.oopadeeperlook.Circle@15db9742
```

اما وقتی این متد را تعریف کردیم خروجی مطلوب مشابه زیر شده است:

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

۴.۵.۲ متد equals

- عملگر «==»
 - **متغیرهای اولیه (Primitive):** برای بررسی تساوی بین دو متغیر از نوع Primitive (بطور مثال int یا boolean یا char)
 - عملگر «==» مناسب است
 - **متغیرهای ارجاع (Reference):** اگر متغیر از نوع Reference یا همان ارجاع باشد
 - عملگر «==» تساوی محتوای دو شیء را بررسی نمی‌کند
 - عملگر «==» تساوی ارجاع (اشاره‌گرها) را بررسی می‌کند
 - عملگر «==» به ما می‌گوید که آیا دو متغیر به یک جا از حافظه اشاره می‌کنند یا خیر

```
String s = new String("Ali");
String t = new String("Ali");
if(s == t)
    System.out.println("s==t");
else
    System.out.println("s!=t");
OUTPUT: s!=t
```

- **متد equal**
 - برای مقایسه محتوای دو شیء از متد equals استفاده می‌کنیم
 - بسیاری از کلاس‌های مهم جاوا، متد equals مناسبی دارند که تساوی محتوای دو شیء را بررسی می‌کند. مثل کلاس String یا کلاس‌های لفاف اولیه نظیر Integer و Character و ...

```
String s = new String("Ali");
String t = new String("Ali");
if(s.equals(t))
    System.out.println("s equals t");
else
    System.out.println("s is not equals t");
OUTPUT: s equals t
```

- اگر بخواهیم در کلاسی که خودمان ایجاد کردیم این متد را تعریف نماییم باید یک شیء بصورت پارامتر بگیرد و نهایتاً مقدار boolean برگرداند. این متد ویژگی‌های شیء موجود در پارامتر را با ویژگی‌های خودش توسط کلمه کلیدی this بررسی و مقایسه می‌کند و در نهایت وضعیت بصورت درست یا غلط برگردانده می‌شود. در تعریف متد equals محتوای پارامتر با this مقایسه می‌شود.
به مثال زیر توجه نمایید:


```

public class Person {
    private String nationalID;
    private String name;
    private String email;
    private int age;

    public boolean equals(object other) {
        return nationalID.equals(other.nationalID);
    }
    public static void main(String[] args) {
        Person p1 = new Person("1290786547", "Ali Alavi");
        Person p2 = new Person("1290786547", "Taghi Taghavi");
        Person p3 = new Person("0578905672", "Taghi Taghavi");
        System.out.println(p1.equals(p2)); ✓ return True
        System.out.println(p2.equals(p3)); ✗ return False
    }
}

```

باتوجه به قطعه کد زیر تمام اشیاء توسط متد equals با هم برابر هستند ولی ...

```

String str1 = new String("Ali");
String str2 = new String("Ali");
String str3 = "Ali";
String str4 = "Ali";
→ str1 == str2 // ✗ مساوی نیستند زیرا هرکدام به یک شیء جدید اشاره دارد
→ str2 == str3 // ✗ مساوی نیستند
→ str3 == str4 // ✓ باهم مساوی هستند

Integer int1 = new Integer(2);
Integer int2 = new Integer(2);
Integer int3 = "Ali";
Integer int4 = "Ali";
→ int1 == int2 ✗ مساوی نیستند زیرا هرکدام به یک شیء جدید اشاره دارد
→ int2 == int3 ✗ مساوی نیستند #autoboxing
→ int3 == int4 ✓ باهم مساوی هستند #autoboxing

```

- همواره متدهای equals و hash Code باید باهم تولید شوند زیرا ایندو با هم وابستگی دارند
 - همواره باید پارامتر ورودی equals یک آبجکت باشد و نه چیز دیگر زیرا اگر چیزی جز object بگیرد وارد مفهوم overload می شود ولی با نوشتن متد equals قصد داریم که متد را override کنیم و نه اینکه overload کنیم
 - در ساختارهای داده Data Structure تابع Equals بسیار مهم است:
 - متدهای خود contains یا remove(Object o) یا indexOf(Object o) که به جستجوی شیء مساوی می پردازند از متد Equals استفاده می کنند
 - متد equals روی اشیاء فهرست فراخوانی می شود و شیء موردنظر به آن پاس می شود
 - در مجموعه ها (مثل HashSet) تکراری بودن عضو جدید با کمک equals بررسی می شود
- مثال: خروجی کد زیر False هست و این مشکل باید با پیاده سازی متد equals مناسب حل شود:

```

class Student {
    private String name;
    public Student(String name) {

```

```

        this.name = name;
    }
}

main(){
    List<Student> list = new ArrayList<Student>();
    list.add(new Student("Ali"));
    System.out.println(list.contains(new Student("Ali")));
}

```

در مثال بالا، شکل صحیح متد «Equals» که باید داخل کلاس نوشته شود بصورت زیر است (البته متد «Equals» زیر کامل و دقیق نیست و جزئیاتی مثل «NULL» بودن پارامتر را بررسی نمی‌کند)

```

public boolean equals(Object obj) {
    Student other = (Student) obj;
    if (!name.equals(other.name))
        return false;
    return true;
}

```

متد getClass (۴.۵.۳)

- اولین باری که یک کلاس استفاده می‌شود این کلاس توسط مکانیزم «Dynamic Loading» در حافظه بارگذاری می‌شود و اطلاعات مربوط به این کلاس، در یک شیئی مجزا تحت عنوان «شیء کلاس» یا «Class Object» در حافظه جای می‌گیرد
- بعنوان مثال در مورد کلاس String یک شیء در حافظه ایجاد می‌شود که اطلاعات این کلاس را نگهداری می‌کند. (اطلاعات ویژگی‌ها، متدها، متغیرها، مقادیر و غیره)
- هر شیء‌ای یک ارجاع به «ClassObject» خودش دارد. مثلاً اگر ۱۰ شیء از جنس Person ساخته شود، آنگاه یک شیء دیگر از جنس «ClassObject» برای Person هم خواهیم داشت.
- هر شیء ارجاعی به Object ای دارد که این آبجکت، اطلاعاتی درباره کلاس این شیء در خودش نگهداری می‌کند.
- این ارجاع به کمک متد getClass برمی‌گردد.
- متد getClass در کلاس Object و بصورت final پیاده‌سازی شده است.

```

Animal a = new Dog("Fido");
String s = a.getClass().getSimpleName();           //output: s → Dog
s = a.getClass().getName();                       //output: نام کامل به همراه بسته‌ها و مسیرهای آن کلاس

```

۴.۵.۴ تکنیک Hash

- برخی از ساختمان داده‌های جاوا مبتنی بر تکنیک Hash هستند (مانند HashSet و Hash Map)
- از هر شیء یک عدد صحیح (Hash) مبتنی بر ویژگی‌های داخل شیء استخراج می‌شود
- از hash برای محاسبه محل ذخیره شیء استفاده می‌شود
- ممکن است دو شیء مقدار hash مساوی داشته باشند (دو شیء با ویژگی‌های مساوی، باید مقدار hash مساوی برگردانند)
- تساوی مقدار Hash به معنی تساوی خود اشیاء نیست
- مقدار hash تصادفی نیست. یعنی اگر از یک شیء دو بار hash گرفته بشود باید دو مقدار یکسان حاصل گردد

۴.۵.۴.۱ متد hashCode

- بسیاری از ساختمان داده‌ها از تکنیک hash استفاده می‌کنند
- متد hashCode از کلاس Object به همه کلاس‌ها به ارث می‌رسد که قابلیت override آن وجود دارد
- با کمک تابع hashCode یک شیء به یک عدد صحیح (hash) تبدیل می‌شود
- از hash برای جایابی در حافظه و دسترسی سریع به اشیاء استفاده می‌شود
- متد hashCode مناسب، از فیلدهای شیء استفاده می‌کند و عددی حتی‌الامکان متفاوت برمی‌گرداند
- از امکانات IDE (مثلاً eclipse) برای تولید متدهای equals و hashCode استفاده کنید
- اگر برای مقایسه دو شیء متد equals مقدار true برمی‌گرداند، آنگاه باید خروجی متد hashCode این دو شیء هم با هم مساوی باشند، و نه لزوماً برعکس (یعنی اگر دو شیء هاش مساوی دارند لزوماً به این معنی نیست که با هم برابر هستند و متد Equals آن‌ها مقدار True بر نمی‌گرداند)
- مثال: مقدار False برمی‌گرداند.

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

راه حل: باید برای کلاس Student هم متد equals و هم hashCode مناسبی پیاده کنیم. پیاده‌سازی equals کافی نیست، زیرا HashSet مبتنی بر hashCode کار می‌کند
مثلاً:

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

فصل ۵) دسته‌بندی و سطوح دسترسی

۵.۱) Package [دسته‌بندی دیتا برنامه]

- عنوان Packaging در جاوا، امکانی برای طبقه‌بندی و گروه‌بندی کلاس‌های جاوا است (مانند مفهوم فولدر در مدیریت فایل) که هر بسته ممکن است شامل تعدادی کلاس باشد که کاربرد مشابهی دارند.
- یک فضای نام‌گذاری (name Space) یکتا برای کلاس‌ها فراهم می‌کند.
- نام دقیق کلاس = نام بسته + نام کلاس
- هر بسته می‌تواند شامل بسته‌های دیگر باشد (سلسله‌مراتبی از بسته‌ها)
- کلاس‌های یک بسته در یک شاخه (Folder) نگهداری می‌شوند
- برای قرار دادن یک بسته در یک کلاس، فایل کلاس را درون شاخه بسته قرار می‌دهیم
- می‌توانیم در اولین خط از تعریف کلاس، با استفاده از دستور package آن را معرفی می‌کنیم:

```
package ir.MyCompany;
```

- معمولاً نام‌گذاری بسته‌ها از کل به جزء صورت می‌گیرد: دامنه <شرکت> <دپارتمان> <پروژه> <زیرمجموعه>

```
Files.Securent.It.Behrooz.Com
```

- باکمک ستاره (*) همه کلاس‌های یک بسته قابل استفاده می‌شوند

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

- کلاس‌های بسته java.lang که شامل بسته‌های اصلی زبان جاوا است نیاز به import شدن ندارد. مثلاً کلاس String و Math از این بسته است.
- دستور import فقط برای کامپایلر است. (جایگزین شدن نام کلاس با نام کامل کلاس‌ها و آدرس‌ها و در نهایت کامپایل شدن برنامه بدون وجود داشتن دستورات import)
- دستور import در کلاس کامپایل شده دیده نمی‌شود (فایل class یا bytecode)

۵.۱.۱) پوشه‌بندی پروژه‌ها

- پوشه **src**: شامل مواردی نظیر: Servlet، سرویس‌ها، بخش لاجیک کد نوشته شده، انضمام لایه کنترلر
 - این پوشه محض فایل‌های باینتری ساخته می‌شود. پس معمولاً فایل‌های Text و XML در این پوشه نباید قرار بگیرد
- پوشه **web**: تمامی صفحات استاتیک [شامل اچ‌تی‌ام‌ال‌ها] و استاتیک ریسورس‌ها [سی‌اس‌اس و جاوا اسکریپت]
- پوشه **lib**: اگر Library مجزا داشتیم باید رد پوشه‌ای بنام lib در این فولدر کپی بشود
- پوشه **Resources**: بصورت پیش‌فرض معمولاً وجود خارجی نداشته و در زمان نیاز توسط کدنویس تولید می‌شود.
 - حین ساختن پوشه، توسط گزینه mark as Root Resources به تغییر وضعیت می‌دهیم و قرارداد می‌کنیم که فایل‌های پروژه را در این پوشه قرار دهیم
 - داخل آن میتوانیم چندین دایرکتوری بسازیم و دسته بندی را منسجم‌تر نماییم

۵.۲ محصورسازی یا Encapsulation

- یک از تکنیک‌های برنامه‌نویسی Object Oriented است که در مفهوم کلاس و دسترسی‌ها معنی پیدا می‌کند
- محدودسازی دسترسی به کلاس و پنهان‌سازی اجزای پیاده‌سازی
- در کد نوشته شده، منطقه عمومی و منطقه خصوصی تعریف شود. بطوری که بخش خصوصی از دید کاربران پنهان است
- مثلاً استفاده از متدهای get برای دریافت مقادیر ویژگی‌های کلاس، که به آن‌ها Getter گفته می‌شود یا استفاده از متدهای Set برای مقداردهی به ویژگی‌های کلاس، که به آن‌ها Setter گفته می‌شود.

۵.۳ سطوح دسترسی یا Access Level

توسط یک کلیدواژه می‌توان برای هر متد یا کلاس سطح دسترسی تعیین نمود که این موارد می‌تواند Public یا Private یا هیچی باشد.

- سطوح دسترسی برای کلاس‌ها: Public و Package Access است
- سطوح دسترسی برای متدها و ویژگی‌ها: Public و Private و Package Access است
- یک فایل جاوا:
 - می‌تواند صفر یا چند کلاس غیر public با اسم‌های متفاوت تعریف شود.
 - حداکثر یک کلاس Public می‌تواند داشته باشد
 - اگر حاوی کلاس public باشد باید حتماً یکی باشد و نمی‌تواند دو کلاس پابلیک باشد و باید همانم فایل باشد
 - ممکن است کلاس عمومی نداشته باشد

۵.۳.۱ Public [مخصوص متدها و ویژگی‌ها و کلاس‌ها]

- اگر یک متد یا ویژگی بصورت public تعریف شود، یعنی این متد یا ویژگی از هر کجا و از هر کلاسی و از هر package قابل استفاده و در دسترس است (یعنی دسترسی به آن آزاد می‌باشد)
- متدهای Getter و Setter بصورت Public تعریف می‌پذیرند.

۵.۳.۲ private [مخصوص متدها و ویژگی‌ها]

- سطح دسترسی از نوع private یعنی در «خارج کلاس» از هیچ‌کجا قابلیت استفاده (از ویژگی‌ها و متدها) وجود ندارد و کلاس‌های دیگر به آن ویژگی دسترسی نخواهند داشت.
- فقط دسترسی از داخل همان کلاس برقرار است
- سطح دسترسی Private برای یک کلاس معنی ندارد (مگر برای کلاس‌های داخلی یا همان inner Class ها)
- معمولاً ویژگی‌ها (property) بصورت private تعریف می‌شوند و برای تغییر در ویژگی‌ها از متدهای Setter و برای دریافت مقدار ویژگی‌ها از متدهای getter استفاده می‌شود. همچنین دلیل استفاده از setter . getter این است که توسط اعتبارسنجی اجازه دریافت

هر مقدار به ویژگی یا همان متغیرها داده نشود

```
public void setAge(int a) {
    if (a>0 && a<150)
        age=a;
}
public int getAge() {
    return age;
}
```

۵.۳.۳) سطح دسترسی Package Access [مخصوص متدها و ویژگی‌ها و کلاس‌ها]

- حالت پیش فرض برای زمانی است که برای ویژگی و متد سطح دسترسی تعیین نکرده باشیم.
- یعنی عبارت کلیدی public یا private استفاده نشده باشد که در این صورت، دسترسی به آن متد یا ویژگی فقط در داخل کلاس‌های همان بسته ممکن است.
- بگونه‌ای خواهد بود که گویا برای کلاس‌های داخل بسته، این متد یا ویژگی از نوع public می‌باشد
- بطور خلاصه: قابل استفاده داخل بسته و غیرقابل استفاده در خارج بسته
- اگر برای یک کلاس عبارت public را تعیین نکنیم آنگاه سطح دسترسی برابر Package Access خواهد شد

۵.۳.۴) سطح دسترسی Protected [کاربرد در وراثت، برای متدها و ویژگی‌ها]

گاهی لازم است که یک ویژگی یا متد در زیرکلاس قابل استفاده باشد ولی از سایر کلاس‌ها مخفی باشد که برای این نیازمندی سطح دسترسی Protected ایجاد شده است

- اگر عضوی (متد یا متغیر) protected باشد،
 - برای زیرکلاس‌ها در دسترس است
 - برای کلاس‌های داخل همان بسته (package) هم قابل استفاده است
 - برای سایر کلاس‌ها مخفی (غیرقابل استفاده) خواهد بود
- یک سطح دسترسی میانی: کمتر از public و بیشتر از package access
- برای سایر کلاس‌ها که زیرکلاس کلاس اصلی مان نیستند و یا در بسته قرار ندارند ویژگی‌ها مخفی خواهد بود، یعنی گویا private برایشان از نوع private است

۵.۳.۵) مرور سطح دسترسی

Public (عمومی): از همه کلاس‌ها قابل استفاده است
Protected (محفوظ): از زیرکلاس‌ها و سایر کلاس‌های همان بسته قابل استفاده است

Package access (دسترسی در بسته): از سایر کلاس‌های همان بسته هم قابل استفاده است
Private (خصوصی): فقط در همان کلاس قابل استفاده است (از دید هر کلاس دیگری مخفی است)

بصورت خلاصه سطوح دسترسی برای کلیدواژه‌ها بصورت زیر است:

Modifier	Class	Package	Subclass	World
Public	✓	✓	✓	✓
protected	✓	✓	✓	✗
No Modifier	✓	✓	✗	✗
Private	✓	✗	✗	✗

فصل ۶) شیء‌گرایی و مفاهیم اصلی

۶.۱ Method Overloading یا سربار کردن متد

در یک کلاس می‌توانیم متدهای مختلف هم‌نام تعریف کنیم، به شرطی که پارامترهای متفاوتی داشته باشند و همگی این متدها قابل استفاده خواهند بود.

```
Void func1(){
    System.out.println("f is called");
}
Void func1(int number){
    System.out.println("f is called with number = " + number);
}
Void func1(String s){
    System.out.println("f is called with s = " + s);
}
Void func1(String s, int number){
    System.out.println("f is called with s = " + s + ", number = " + number);
}
public static void main(String[] args) {
    Circle cir = new Circle();
    cir.func1();
    cir.func1(5);
    cir.func1("salam");
    cir.func1("salam",7);
}
```

- سربار کردن متد یا همان تکنیک Overloading را فقط براساس پارامتر ورودی تفکیک می کنیم و اساساً نمی توان برحسب مقدار بازگشتی Overloading تنظیم نمود. اما حالت زیر اشکالی ندارد:

```
void func2(int a){}
int func2(){
    return 0;
}
```

۶.۲ مفاهیم Static

مفهوم استاتیک هم برای ویژگی ها و هم برای متدها قابل استفاده است

- یک ویژگی غیراستاتیک، به ازای هر شیء یک خانه در حافظه ایجاد میکند.

۶.۲.۱ ویژگی استاتیک

- ویژگی های مشترک بین اشیاء ساخته شده از یک کلاس را در حالت استاتیک قرار می دهیم.
 - مثلاً در کلاس پراید، بدلیل اینکه طول این پراید برای همه پرایدها ثابت است، پس متغیر length را از نوع استاتیک قرار می دهیم. اما ویژگی های رنگ و قیمت ممکن است برای اشیاء متفاوت (پرایدهای متنوع) متفاوت باشد
 - یک متغیر استاتیک، درواقع یک ویژگی برای یک کلاس است و نه برای اشیاء
 - یعنی وقتی از آن متغیر نوع استاتیک صحبت به میان می رود، این صحبت معطوف به اشیاء نیست بلکه مربوط به کلاس است (بعبارتی مربوط به همه اشیاء ساخته شده از کلاس است)
 - بدون ساختن شیء می توانیم از متغیرهای (ویژگی های) استاتیک استفاده نماییم (با کمک اسم کلاس و اسم متغیر به همراه نقطه) **Pride.length**
- ویژگی تعریف شده از نوع استاتیک در یک کلاس، در بین تمام اشیاء آن کلاس مشترک است
- هرشیء ساخته شده از کلاس حاوی متغیر استاتیک، فقط و فقط یک خانه در حافظه برای نگهداری دیتای آن متغیر استاتیک نیاز دارد. (چون این ویژگی در تمام اشیاء یکسان است)
- مثال: کلاس Math در جاوا

```
public class Math {
    public static double PI = ۳.۱۴۱۵۹۲۶;
    ...
    ...
}
```


۶.۲.۲ متد استاتیک

- در حالت عادی هر متد روی یک شیء فراخوانی می‌شود. ولی برخی متدها به هیچ شیء خاصی از کلاس مرتبط نیستند. چنین متدهایی از نوع استاتیک تعریف می‌شوند.
- متد استاتیک بدون نیاز به ساختن شیء قابل فراخوانی هستند.

```
Double value = Math.pow(2,3); # تابع توان در کلاس ریاضی از نوع استاتیک تعریف شده است
String str = String.valueOf(12); # تابع «ولیوآف» در کلاس «استرینگ» از نوع استاتیک تعریف شده است
```

- مثال از متد استاتیک در کلاس Math در جاوا:

```
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) {...}
}
```

- متدهای استاتیک در واقع «مجموعه دستورات» برای کلاس هستند نه «مجموعه دستورات» که اختصاص به یک شیء داشته باشد
- متدهای استاتیک به «شیء» ساخته شده از کلاس یا «اجزای شیء» دسترسی ندارند.
- متدهای استاتیک فقط می‌توانند از ویژگی‌های استاتیک کلاس استفاده نمایند. اما متدهای معمولی می‌توانند از همه ویژگی‌های استاتیک و غیر استاتیک موجود در کلاس استفاده نمایند
- متغیرهای استاتیک هم مثل ویژگی‌های معمولی باید مقداردهی اولیه معتبر داشته باشند و اینکار یکبار برای همیشه در لحظه تعریف انجام می‌شود.

۶.۳ مفاهیم Final

اجزای final بمنزله نهایی شده تلقی می‌شود یعنی اگر یک شیء یا متغیر یا متد یا کلاس از نوع final تعریف شود، قابلیت تغییر نخواهد داشت. و این کار با افزودن کلیدواژه final در ابتدای متغیر انجام می‌شود

نکته مهم: مفاهیم final [ثابت بودن یا نبودن] و سطوح دسترسی [public, Private, Package Access] و استاتیک بودن یا نبودن، هر کدام یک مفهوم مستقل از هم هستند.

۶.۳.۱ متغیر final

دو نوع متغیر داریم ۱- Primitive و ۲- Reference که در هر کدام نوع رفتار متفاوت دارد.

هر متغیر final (در هر دو حالت) باید بلافاصله مقداردهی شود که این مقداردهی می‌تواند بصورت inline یا Initiator Block یا تابع سازنده باشد

- **Primitive Variable:** اگر از نوع final تعریف شود آنگاه

- باید در هنگام تعریف مقداردهی شود
- دیگر هرگز مقدار آن قابل تغییر نیست
- به این متغیرها Constant گفته می‌شود.
- بطور مثال مقداری Math.PI یا Integer.MAX_VALUE که در جاوا متغیرهای ثابت می‌باشد، با کلید واژه final مشخص شده است
- **Reference Variable**: اگر از نوع final تعریف شود آنگاه
 - به شیء دیگری نمی‌تواند ارجاع دهد
 - ارجاع و اشاره‌گر ثابت خواهد بود و به شیء یا جایی نمی‌تواند اشاره نماید

به مثال‌های زیر توجه شود

```
final int I = 2;
I = 3; ❌ # مقدار متغیرهایی از انواع اولیه غیرقابل تغییر است

final Person p1 = new Person();
p1 = p2; ❌ # هویت یک شیء ثابت قابل تغییر نیست
p1 = new Person(); ❌ # هویت یک شیء ثابت قابل تغییر نیست
p1.setName("Ali"); ✅ # وضعیت(ویژگی، محتوی) یک شیء ثابت قابل تغییر است
```

۶.۳.۲) متد final

- متدهای final، قابلیت override کردن در زیر کلاس‌ها را ندارند
- اگر یک متد final در یک زیر کلاس override شود آنگاه سبب بروز خطای کامپایلر می‌شود
- یعنی تعریف یک متد نهایی شده و هیچ‌کس قابلیت تغییر این متد را ندارد تا سبب تغییر کارایی متد شود
- تعریف یک متد بصورت final یک تصمیم دلخواه از سوی برنامه‌نویس است. تا به دیگران اجازه ندهد کارایی متد را تغییر دهند.
- در بیشتر موارد متدهای نوشته شده را چهارچوب final قرار نمی‌دهیم.
- نگرشی در بین برنامه‌نویسان وجود دارد که متدهایشان را final می‌کنند تا مفهوم polymorphism در آن متد bind نشود و سبب افزایش سرعت و کارایی شود. این اشتباه است که با هدف افزایش performance متدها را final تعریف نماییم.
- مثال:

```
class Dog extends Animal{
    @Override
    public void talk() { ... }
    public final void bark() { ... }
}
```

۶.۳.۳) کلاس final

- کلاس‌های final قابل extend کردن نیستند.
- ارث‌بری از آن غیر مجاز است
- اگر کسی زیر کلاس تولید نماید سبب خطای کامپایلر خواهد شد
- یعنی مفهوم و کارایی کلاس دقیق پیاده‌سازی شده و این کلاس قابلیت تغییر ندارد تا کارایی کلاس تغییر نماید
- در بیشتر موارد کلاس‌های نوشته شده را چهارچوب final قرار نمی‌دهیم

```
final class Dog extends Animal{
    ...
}
```

- مثال: کلاس String که در جاوا قرار دارد یک کلاس final می‌باشد. که قابلیت extend ندارد زیرا این کلاس نهایی شده است

۶.۳.۴) اشیاء «تغییر پذیر» یا «تغییر ناپذیر»

اشیاء به دو دسته تقسیم می‌شوند: برخی از نوع تغییر پذیر و برخی از نوع تغییر ناپذیر هستند. طراح کلاس تصمیم می‌گیرد نمونه‌های یک کلاس تغییر پذیر باشند یا خیر ثابت بودن درباره ثبات هویت یا identity شیء است و با final مشخص می‌شود ولی تغییر ناپذیر بودن درباره ثبات وضعیت (state) است

• Mutable Object یا اشیاء تغییر پذیر

- امکان تغییر وضعیت اشیاء در اینگونه اشیاء وجود دارد
- اشیاء ای که تغییر پذیرند معمولاً Setter دارند
- به متدهای Setter یک متغیر اصطلاحاً mutator می‌گویند

• Immutable Object یا اشیاء تغییر ناپذیر

- محتوای آن‌ها قابل تغییر نیست. متدی برای تغییر وضعیت یا ویژگی‌ها یا Property ها و فیلدها داخل اون شیء وجود ندارد
- تغییر ناپذیری یک مفهوم است و کلیدواژه خاصی ندارد
- ویژگی‌های اینگونه اشیاء بعد از ساخت این اشیاء قابل تغییر نیست
- مزایای آن: سادگی و فهم ساده‌تر و کارایی بهتر برنامه و سهولت و بهینگی در موازی کاری یا همان Thread-safe
- خیلی از کلاس‌هایی که می‌شناسیم از این نوع است: مثل String که متد setValue ندارد
- همه کلاس‌های لفاف اولیه (Integer یا Character یا غیره) از نوع Immutable هستند و متد Setter ندارند که بتوانیم محتوای آن را تغییر دهیم
- توجه شود که وقتی گفته می‌شود یک شیء «تغییر ناپذیر» است، مفهوم جداگانه‌ای از مفهوم «ثابت بودن» final دارد.

۶.۴) مفاهیم وراثت (Inheritance)

- به کلاس اصلی، کلاس پایه (Base Class) یا اَبَر کلاس (Superclass) یا کلاس والد (Parent Class) می‌گویند
- به کلاس وارث، کلاس مشتق (Derived Class) یا زیر کلاس (Subclass) یا کلاس فرزند (Child Class) می‌گویند
- زیر کلاس از نوع کلاس می‌باشد و ویژگی‌های و رفتار اَبَر کلاس خود را به ارث می‌برد
- هر شیء از زیر کلاس، شیء ای یا نمونه‌ای از اَبَر کلاس هم هست.
- یک کلاس می‌تواند زیر کلاس‌های مختلفی داشته باشد.
- زیر کلاس نمی‌تواند ویژگی یا متد ابر کلاس را حذف نماید
- در جاوا وراثت با کلید واژه Extends معرفی می‌شود

```
class Shape{
    int color;
    int positionX, positionY;
}
```

```

class Circle extends Shape{
    private int radius;
    public double getArea(){
        return 3.14*radius*radius;
    }
}
class Rectangle extends Shape{
    private int width, length;
    public double getArea(){
        return width*length;
    }
}

```

- یک زیر کلاس نمی تواند بصورت همزمان از چند ابر کلاس ارث ببرد. (multiple Inheritance یا ارث بری چندگانه در جاوا پشتیبانی نمی شود)
- تعریف متغیر در زیر کلاس که هم نام متغیر ابر کلاس هستند، کار رایجی نیست.

Override (۶.۴.۱)

- هنگامی که یک زیر کلاس از یک ابر کلاس ارث بری می کند، در چنین شرایطی قابلیت تغییر و اصلاحات در **متدها** وجود دارد. یعنی متدی که در والد تعریف شده را بطور دلخواه **تغییر** دهیم و ساختار و بدنه آن تغییر نمایم (و اینکه باید تعداد آرگومان ورودی یکسان باشند)
- هرگاه یک زیر کلاس می سازیم و متدی را **Override** می کنیم در این صورت **حق نداریم سطح دسترسی** به این متد را کاهش دهیم.
 - مثلاً حق نداریم متدی را که در ابر کلاس **public** بوده است را در زیر کلاس **private** تعریف کنیم چرا؟ این کار قانون IS-A را نقض می کند و هر شیء از زیر کلاس، شیئی از جنس ابر کلاس هم هست و هر رفتاری که در ابر کلاس هست، باید برای اشیاء زیر کلاس هم قابل فراخوانی باشد. پس سطح دسترسی به متدها در زیر کلاس ها قابل کاهش نیست.
- در صورت یکسان بودن نام ویژگی یا متد در هر دو کلاس والد و فرزند، در چنین شرایطی در صورت فراخوانی آن متد یا ویژگی، آنگاه آن متد یا ویژگی موجود در زیر کلاس فراخوانی خواهد شد.
- مفهوم **Override** برای متدها معنی داشته و برای متغیرهای معنی ندارد

Super (۶.۴.۲)

- در مفهوم وراثت گاهی نیاز می شود که متد یا ویژگی خاصی را که در ابر کلاس تعریف شده است مورد استفاده قرار گیرد و نه متد یا ویژگی متناظر در زیر کلاس، پس در چنین شرایطی توسط کلیدواژه **Super** میتوان متد یا ویژگی موجود در ابر کلاس را فراخوانی نمود
- مثال زیر اگر تابع **Func** از کلاس ابر کلاس فراخوانی خواهد شد، حتی اگر متد **Func** مجدداً در زیر کلاس تعریف شده باشد

```
super.Func();
```

- عبارت زیر بیانگر متدسازنده ای از ابر کلاس است که دارای دو پارامتر ورودی است

```
super(name, id);
```

- عبارت `this` و `Super` در مقابل هم هستند یعنی عبارت `this` به اجزای خود کلاس دسترسی ایجاد می‌کند ولی در `super` دسترسی به اجزای اَبَر کلاس ایجاد خواهد شد.
مثال یک:

```
class Person{
    private String name;
    private String nationalID;
    public Person(String name, String nationalID) {
        this.name = name;
        this.nationalID = nationalID;
    }
}
```

ایجاد زیر کلاس برای مثال یک:

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

- **متدهای سازنده به ارث نمی‌رسند.** یعنی مثلاً اگر متد سازنده‌ای در اَبَر کلاس باشد که یک پارامتر `int` می‌گیرد این متد سازنده به زیر کلاس به ارث نمی‌رسد.
- وقتی یک زیر کلاس تعریف می‌کنیم باید متد سازنده مشخصی از اَبَر کلاس در متد سازنده‌ی زیر کلاس فراخوانی شود وگرنه سازنده‌ای بدون پارامتر از اَبَر کلاس بصورت ضمنی فراخوانی خواهد شد و این کار با کلیدواژه `super` که باید در اولین دستور در زیر کلاس باشد صورت می‌گیرد. همچنین اگر سازنده‌ای در اَبَر کلاس نباشد، خطای کامپایل ایجاد می‌شود. مثال:

```
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;
    }
}
```

ادامه مثال؛ برای ایجاد اشیاء از کلاس‌های بالا، باید به صورت زیر اقدام کرد:

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

۶.۴.۳ کلاس Object

- در جاوا هر کلاس، یک زیرکلاسی از ابرکلاس Object است. یعنی همه کلاس‌ها از Object ارث‌بری می‌کنند
 - کلاس‌هایی که هنگام تعریف از کلیدواژه extends استفاده نمی‌کنند به صورت ضمنی از Object ارث‌بری می‌کنند
 - کلاس‌هایی که هنگام تعریف از کلیدواژه extends استفاده می‌کنند به صورت غیرمستقیم از Object ارث‌بری می‌کنند
- مثلاً: در تعریف کلاس Parent گویی نوشته‌ایم

```
class Parent extends Object {
    ...
}
```

- کلاس Object متدهای آشنایی دارد نظیر: hashCode, equals, finalize, toString و غیره

۶.۴.۴ مقداردهی اولیه در وراثت

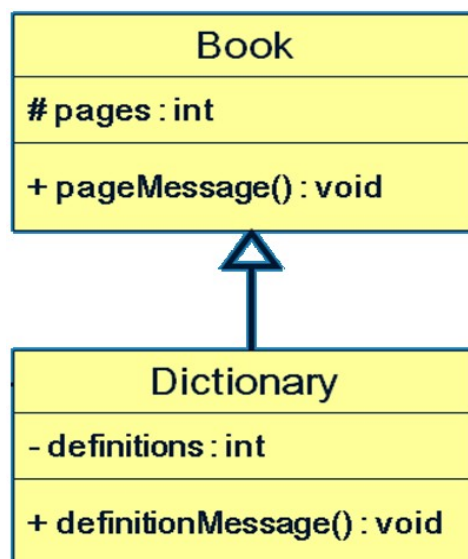
- وقتی یک زیرکلاس تعریف می‌کنیم:
 - ابتدا باید در نظر داشته باشیم که متدهای سازنده به ارث نمی‌رسند و با این نکته باید متد سازنده مشخصی از ابرکلاس در متد سازنده زیرکلاس فراخوانی شود که این کار با کلیدواژه super انجام می‌شود و این فراخوانی در اولین خط از متد سازنده زیرکلاس باشد
 - در غیر این صورت متد سازنده‌ای بدون پارامتر از ابرکلاس به صورت ضمنی توسط کامپایلر فراخوانی می‌شود

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

- بارگذاری کلاس‌ها در وراثت به صورت زیر است:
 - **اول:** اگر اَبَرکلاس از کلاس دیگری ارث‌بری کرده باشد، آن کلاس بارگذاری می‌شود(به همین صورت تا برسیم به بالاترین کلاس والد)
 - **دوم:** اَبَرکلاس بارگذاری می‌شود
 - **سوم:** یکبار برای همیشه زیرکلاس بارگذاری می‌شود
 - **یادآوری:** هنگام بارگذاری کلاس، ویژگی‌های استاتیک مقدردهی اولیه می‌شوند
 - **چهارم:** هنگام ایجاد شیء از زیرکلاس (هر بار که یک شیء از نوع زیرکلاس ایجاد می‌شود)
 - **پنجم:** ابتدا بخشی از شیء که در اَبَرکلاس تعریف شده، مقدار دهی اولیه می‌شود
 - **ششم:** سپس سایر ویژگی‌های شیء که در زیرکلاس تعریف شده، آماده می‌شود.

۶.۴.۵) توضیحات تکمیلی

- در نمودارهای UML در Class Diagram ها برای نمایش دسترسی
 - اعضای protected از علامت استفاده می‌نماییم
 - اعضای Private از علامت - استفاده می‌نماییم
 - اعضای Public از علامت + استفاده می‌نماییم



۶.۵ چندریختی یا Polymorphism

- پس از محصورسازی (Encapsulation) و وراثت (Inheritance) موضوع چندریختی (Polymorphism) مهمترین و حیاتی ترین امکان در زبان های شیء گرای به حساب می آید
- متدی روی یک شیء فراخوانی می شود. نوع شیء، در زمان اجرا مشخص می شود. یعنی در زمان اجرا رفتار دقیق این شیء (بر حسب نوع آن) معلوم می شود.
- ایجاد امکان چندریختی، از عهده کامپایلر بر نمی آید و کامپایلر نمی داند یک ارجاع، به شیئی از چه کلاسی اشاره خواهد کرد و در زمان اجرا مشخص می شود

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

- به نوع Cast توجه شود که شیء از نوع والد در شیء از نوع فرزند ریخته نمی شود و شیء فرزند در شیء والد ریخته می شود (توجه: درباره عملگر تساوی صحبت می کنیم که «نوع» سمت چپ و راست آن متفاوت است)

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

- مثال در حالت Up casting یا «تغییر نوع به بالا» که همواره معتبر است

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

- مثال در حالت Down casting یا «تغییر نوع به پایین» که همواره معتبر نیست و گاهی معتبر و گاهی نامعتبر است و باید کلیدواژه Cast ذکر شود

```
shape s = ...  
Circle c = s; ✗ خطا  
  
shape s = new Circle();  
Circle c = (Circle) s; ✓  
  
shape s = new Rectangle();  
Circle c = (Circle) s; ✗ خطای زمان اجرا #
```

- در نمونه چندریختی زیر: تابع draw ظاهری یکسان ولی رفتاری متفاوت دارند.

```
Shape s = new Rectangle();  
s.draw();
```



```
double d = s.getArea();
```

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

```
s = c;
```

```
s.draw();
```

```
d = s.getArea();
```

• مثال دیگر

```
class Parent {
    Public void f() {
        System.out.println("f() in Parent");
    }
}
public class Child extends Parent {
    Public void f() {
        System.out.println("f() in Child");
    }

    public static void main(String[] args) {
        Parent p new Parent();
        p.f();
        Child c = new Child();
        c.f();
        p = c;
        p.f();
    }
}
```

• مثال

```
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();
```

```
o = a;
```

```
o = c;
```

```
a = o; ❌ خطای کامپایل
```

```
a = c;
```

```
c = o; ❌ خطای کامپایل
```

```
c = a; ❌ خطای کامپایل
```

```
c = d; ❌ خطای کامپایل
```

```
c = (Cat) x;
```

```
d = (Dog) x; ❌ در زمان اجرا ClassCastException خطا
```

۶.۵.۱ شرایط کدنویسی بدون چندریختی

• در یک بازی فوتبال صدها نوع شیء وجود دارد که باید به ازای هر نوع یک حلقه ایجاد کنیم

```
Player[] players = ...
Referee[] refs = ...
Ball ball = ...

for (Player ply : players) {
    ply.draw();
}

for (Referee ref : refs) {
    ref.draw();
}

ball.draw();
```

• با وجود چند ریختی شکل ساده‌سازی شده قطعه کد بالا به فرم زیر درخواهد آمد:

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

• بلی

۶.۶ مفهوم انتزاعی (Abstract)

انتزاعی بودن یک کلاس یا متد باید توسط برنامه‌نویس مشخص شود و اینکار با کلیدواژه `abstract` انجام می‌شود

۶.۶.۱ متد انتزاعی

- انتزاعی بودن یک متد باید توسط برنامه‌نویس با کلیدواژه `abstract` تصریح شود
- **متد بدون بدنه** که در داخل کلاس تعریف می‌شود. و هر شیء که از این کلاس ساخته شود به آن متد دسترسی دارد. با این تفاوت که بدنه این متد باید در زیرکلاس تعریف شود.
- چنین متدی در آبرکلاس، متد انتزاعی خوانده می‌شود.
- متد `Concrete` متدی است که پیاده‌سازی کاملاً مشخصی داشته و انتزاعی نیست.
- `Concrete method ≠ Abstract Method`
- گاهی همه متدهای یک کلاس، انتزاعی هستند.
- متد انتزاعی، دارای بدنه و پیاده‌سازی نیست. قرار نیست که پیاده‌سازی شود و باید توسط یک زیرکلاس پیاده‌سازی شود.

```

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

```

مثال دوم:

```

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;
    }
}
public class Rectangle extends Shape{
    private double width, length;

```

```

public double getWidth() {
    return width;
}
public void setWidth(double width) {
    this.width=width;
}
public double getLength() {
    return length;
}
public void setLength(double length) {
    this.length=length;
}
public double getArea() {
    return length * width;
}
public double getPremiter() {
    return 2 *(length + width);
}
}

public class Square extends Shape{
    private double length;
    public double getLength(){
        return length;
    }
    public void setLength(double length) {
        this.length=length;
    }
    public double getArea() {
        return length * length;
    }
    public double getPrimeter() {
        return 4 * length;
    }
}

```

۶.۶.۲ کلاس انتزاعی

- انتزاعی بودن یک کلاس باید توسط برنامه نویس با کلیدواژه abstract تصریح شود
- کلاسی که هیچ شیئی مستقیماً از آن ایجاد نمی‌شود و اگر شیئی از جنس این کلاس است، باید از یکی از زیرکلاس‌هایش تولید شود.
- کلاس‌هایی که متد انتزاعی دارند(حتی یک متد انتزاعی):
 - قطعاً این کلاس یک کلاس انتزاعی می‌باشد و حتماً باید قبل از تعریف کلاس، کلیدواژه abstract اضافه شود.
 - این کلاس قطعاً تعریف برخی رفتارها را ندارد. پس باید در زمان پیاده‌سازی در «زیرکلاس» بدنه این متدها را(حداقل یک متد) تعریف کنیم تا به یک Concrete method تبدیل شود
 - نمی‌توانیم نمونه‌ای بسازیم زیرا استفاده از new برای یک کلاس انتزاعی باعث خطای کامپایل می‌شود.
- در «زیرکلاس» مشتق شده، اگر حتی یک متد انتزاعی پیاده‌سازی یا بدنه‌دار نشود، آنگاه بازم انتزاعی خواهد بود و باید با پیشوند abstract تعریف شود
- کلاسی که هیچ متد انتزاعی در آن نیست(ارث‌بری هم نداشته) هم می‌تواند بصورت انتزاعی تعریف شود

- باید در ابتدای نام آن کلاس، عبارت `abstract` آورده شود.
- این کار معمولاً به منظور جلوگیری از ایجاد شیء از این کلاس کاربرد دارد.
- گاهی برای اجباری کردن ایجاد زیرکلاس‌هایی از آن نیز کاربرد دارد.
-

مثال: این کلاس توسط برنامه‌نویس بگونه‌ای طراحی شده است که اجازه ایجاد شیء را نمی‌دهد:

```
abstract class Human {
    private String name ;
    public String getName(){
        return name;
    }
}
```

۶.۶.۳ مفهوم Interface

- اگر در یک «آبرکلاس»، تمام متدها انتزاعی باشد و هیچ متغیری هم تعریف نشود، آنگاه به آن ابرکلاس Interface یا «واسط» می‌گویند
 - بجای کلیدواژه Class از کلیدواژه interface استفاده می‌شود
 - هیچ متد غیر Abstract نباید داشته‌باشد
 - همه متدها بدون نوشتن کلیدواژه، بصورت ضمنی `Abstract` و `Public` تلقی می‌شوند
 - Interface تنها هدف ریل‌گذاری دارند. و مانند یک کلاس کاملاً انتزاعی است (pure abstract class)
 - همانند کلاس انتزاعی امکان ایجاد نمونه(شیء) وجود ندارد و عملگر `new` قابل اجرا روی یک واسط نیست
- دو مثال زیر از لحاظ معنا و کاربرد تقریباً مشابه هستند: **حالت اول:**

```
public interface Shape {
    double getArea();
    double getPerimeter();
}
```

حالت دوم:

```
public abstract class Shape {
    public abstract double getArea();
    public abstract double getPrimeter();
}
```

مثال:

```
public interface Shape {
    double getArea();
    double getPerimeter();
}

class Rectangle implements Shape{
```

```

private double width, length;
private int color;
public Rectangle(double width, double length, int color){
    this.width=width;
    this.length=length;
    this.color=color;
}
public int getColor(){
    return color;
}
public double getArea(){
    return width * length;
}
public double getPerimeter(){
    return 2 * (width * length);
}
}

```

۶.۶.۳.۱) جزئیات ارث‌بری Interface

- ارث‌بری یک کلاس از یک واسط، با کلیدواژه implements است
- اگر کلاسی یک واسط را پیاده‌سازی نماید، آنگاه باید همه متدهای آن را هم تعریف کند وگرنه این کلاس متدهای انتزاعی را به ارث برده است و خودش هم باید انتزاعی شود

```

abstract class Rectangle implements Shape{
    public double getArea() {
        return ...;
    }
}

```

- ارث‌بری یک واسط از واسط دیگر، با کلیدواژه extends انجام می‌شود.
- یک واسط نمی‌تواند از یک کلاس ارث‌بری نماید.

```

interface CanRun{
    void run();
}
interface CanThink{
    void think();
}
interface CanTalk extends CanThink{
    void talk();
}
interface Human extends CanRun, CanTalk{
    void think();
}

```

- یک کلاس می‌تواند صفر یا چند واسط را پیاده‌سازی کند (implements) همچنین یک کلاس می‌تواند از صفر یا چند واسط ارث‌بری

نماید (extends) که در مثال زیر آورده شده است

```
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(){}
}
```

• ارث‌بری زیر بدلیل «تضاد اسامی» غلط است

```
interface A{ void f(); }
interface B{ int f(); }
abstract class C implements A,B{} // غلط است بدلیل تضاد اسامی
```

- امکان تعریف سازنده (Constructor) در واسط وجود ندارد
 - زیرا هدف سازنده مقداردهی اولیه ویژگی‌های شیء است (Field یا Property) و نیز سازنده وضعیت اولیه شیء را آماده می‌کند.
 - اما واسط حالت شیء را توصیف نمی‌کند.
- بهتر است حتی‌الامکان طراحی کلاس‌ها و متدهای ما به واسطه‌ها وابسته باشند.
- از جاوا ۸ به بعد یک واسط میتواند متدهای غیر انتزاعی داشته باشد. که به این متدها، متد پیش‌فرض (Default Method) گفته می‌شود.

```
interface Person {
    Date getBirthDate(); // این متد انتزاعی است
    default Integer age(){
        long diff = new Date().getTime()-getBirthDate().getTime();
        return (int) (diff / (1000L*60*60*24*365));
    }
}
```

۶.۶.۳.۲ متغیر

- تعریف یک متغیر در یک واسط رایج نیست
 - اگر در واسط متغیر تعریف شود بدون نوشتن کلیدواژه بصورت ضمنی: Public و static و final خواهد بود.

```
Interface Humans{
    int MAX_AGE=150; //public static final int MAX_AGE=150;
}
```

annotation (۶.۷)

- به مفهوم حاشیه‌نگاری، که با کاراکتر @ آغاز می‌شود گفته می‌شود
- بخشی از Metadata به حساب می‌آید
- توضیحی پیرامون یک کلاس یا متد یا غیره می‌دهد.
- بر نحوه کامپایل مؤثر است.
- مثال:
 - @Override
 - قبل از تعریف یک متد می‌آید (بالای متد)
 - تصریح می‌کند که این متد برآمده از ابرکلاس Override می‌شود
 - در این صورت اگر به درستی متد را Override نکنیم سبب بروز خطا در کامپایل می‌شود
 - این تصریح می‌تواند سبب کاهش در بروز اشتباهات شود
- بلی
-

فصل ۷) Data Structure and Containers [ساختمان داده و ظرفها]

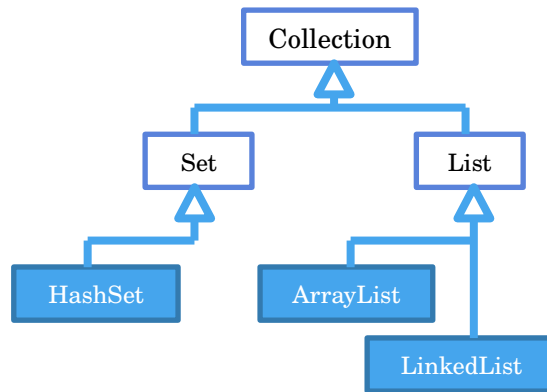
- جاوا امکانات متنوعی برای نگهداری اشیاء دارد نظیر: کلاس‌هایی مثل انواع لیست‌ها، مجموعه‌ها، جدول‌ها و ...
- هر یک از این کلاس‌ها، یک ساختمان داده (Data Structure) است
- هر نمونه ساختمان داده، یک ظرف (container) برای نگهداری اشیاء است
- امکانات و الگوریتم‌هایی بر روی اشیاء داخل ظرف هم پشتیبانی می‌شود. مانند جستجو، تبدیل به انواع دیگر، مرتب‌سازی و ...
- امکاناتی که جاوا به این منظور ساخته : Java collections framework است که کتابخانه‌ای از کلاس‌ها و واسط‌هایی که ساختمان‌های داده مختلف را ایجاد می‌کنند
- در جاوا برای اهداف متفاوت ساختار متفاوت بوجود آمده است و راه‌حل مورد نیاز گاهی آرایه و گاهی ArrayList و گاهی دیگر ساختارها است
- در زیر خلاصه‌ای از شکل ساختمان داده‌ها آورده شده است:
 - Collection: همانند فهرستی از اشیاء عمل می‌کنند
 - List: نظیر array و ArrayList و LinkedList
 - Set: نظیر HashSet و غیره
 - Map: همانند جدول یا نگاشت از اشیاء عمل می‌کنند. جدول دو ستون (زوج مرتب‌دوتایی) است که ستون اول مقدار Key و ستون دوم مقدار Value است.
- نکته خیلی مهم: انواع داده اولیه مثل int و double در هیچ‌یک از ظرف‌های جاوا همانند map یا LinkedList یا ArrayList یا غیره قابل استفاده نیستند و برای استفاده باید بجای int یا double از مقادیری همچون Integer یا Double استفاده نمایند
- مقایسه کلی بین Array و ArrayList
 - Array
 - امکان استفاده از انواع داده اولیه نظیر int و double در صورتی که در ArrayList نمی‌توان از Primitive Data لیست ساخت
 - کارایی یا performance بیشتر نسبت به ArrayList
 - ArrayList
 - ارائه متدها و امکاناتی که در آرایه نیست مانند اضافه و کم کردن اعضا به صورت پویا، جستجو در لیست

Collection (۷.۱)

- واسط List و Set زیرواسط java.util.Collection در جاوا هستند.
- برخی از متدهای مهم Collection در زیر آورده شده است:

```
int size();
boolean isEmpty();
boolean contains(Object o);
boolean add(E e);
boolean remove(Object o);
void clear();
```

- نمودار Collection به شکل زیر است:



۷.۱.۱ تفاوت اصلی set و List

- اشیاء داخل یک Set متمایز هستند، شیء تکراری در Set وجود ندارد
- اگر تلاش کنید شیئی تکراری به مجموعه اضافه شود، مجموعه تغییری نمی‌کند
- شیء تکراری: شیئی که با یکی از اعضای موجود مجموعه برابر است (براساس متد equals)
- اعضای List ترتیب دارند. بین اعضای Set لزوماً ترتیبی وجود ندارد
- واسط Set هیچ متدی که با اندیس کار کند، ندارد
- مثلاً در واسط Set، متد get(i) نداریم، ولی در List داریم
- متدهای دیگری مثل موارد زیر هم در Set وجود ندارد:

```

set(int index, E element)
int indexOf(Object o)
int lastIndexOf(Object o)
remove(int index)
  
```

۷.۲ واسط List

ساختمان داده لیست از نوع Collection می‌باشد.
 یک واسط یا interface تحت عنوان List وجود دارد که ArrayList آمده و متدهای آن را پیاده‌سازی کرده است

```

interface List<E> {
int size();
boolean isEmpty();
boolean contains(Object o);
boolean add(E e);
boolean remove(Object o);
void clear();
  
```

```

E get(int index);
E set(int index, E element);
void add(int index, E element);
E remove(int index);
int indexOf(Object o);
int lastIndexOf(Object o);
List<E> subList(int fromIndex, int toIndex);
}

```

Array (۷.۲.۱)

تعریف: مجموعه‌ای از داده‌های مرتبط به هم که همه این داده‌ها «از یک نوع» هستند

- طول آرایه ثابت است و اندازه آن تنها یکبار و آن هم در زمان ایجاد معین می‌شود. و امکانی برای افزایش طول آرایه وجود ندارد و باید سراغ ساختارهای دیگر برویم
- شماره ترتیبی هر عنصر در آرایه، اندیس یا ایندکس نامیده می‌شود و اندیس از صفر شروع می‌شود
- عناصر آرایه به همان ترتیب در حافظه جای داده می‌شود
- هر آرایه در واقع یک شیء است و مثل همه اشیاء در بخشی از حافظه بنام Heap ذخیره می‌شوند
- در آرایه محدودیت‌هایی هست هندل کردن آن کمی سخت است مانند: افزایش طول سایز آرایه (Size). حذف برخی عناصر آرایه
- برای تعریف آرایه از روش زیر استفاده می‌نماییم:

```

String[] strs; # یعنی «اس‌تی‌آرز» یک ارجاع به شیء‌ای از جنس آرایه است
Person[] people; # یعنی «پی‌پل» یک ارجاع به شیء‌ای از جنس آرایه است
people = new Person[5]; # ایجاد یک شیء از جنس آرایه (یعنی اختصاص مکان ۵ تایی در حافظه) که حاوی پنج پرسن است
float[] RealNumber = new float[10]; # ایجاد یک شیء از جنس آرایه که حاوی ۱۰ عدد فلوت است
int[] array = new int[10]; # ایجاد یک آرایه با ۱۰ عنصر از نوع عدد صحیح
int array[] = new int[10]; # ایجاد یک آرایه با ۱۰ عنصر از نوع عدد صحیح
int[][] matrix = {{1,2},{8,4}} # ایجاد آرایه دو بعدی
char[] array = {'a', 's', 't'};

```

به قطعه کد زیر توجه کنید

```

Student[] students; # ایجاد یک ارجاع به آرایه‌ای از نوع استیودنت یعنی تولید متغیری که قرار است ارجاعی به یک آرایه در آن ذخیره شود
students=new student[4]; # ایجاد یک جا در حافظه (حاوی ۴ ارجاع) برای استیودنت‌ها یک شیء جدید از جنس آرایه در حافظه [ ]
# و اینکه «استیودنتس» خط بالاتر به آن اشاره دارد یعنی آدرس محل ۴ تایی در آن ذخیره شده است
students[0]=new Student(); # اولین خونه آرایه ۴ تایی که هم‌اکنون نال است، به جایی در حافظه که یک شیء جدید است اشاره خواهد کرد
students[0].setName("Ali"); # تعیین یک مقدار برای شیء

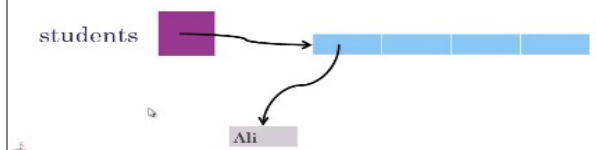
```

باتوجه به قطعه کد زیر می‌توان تصویر را تفسیر نمود

```

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

```



String کلاس (۷.۲.۱.۱)

رشته یا **String**: مجموعه‌ای از کاراکترهایی است که کنار هم قرار گرفته‌است

- هر رشته String یک متغیر از نوع Reference Datatype است
- در یک آرایه از جنس String[]، هر مؤلفه «یک ارجاع به رشته» است
- در آرایه‌ای از جنس Int[]، هر مؤلفه «یک مقدار int» است
- هر رشته String در واقع یک شیء است

```

String str = "salam bar Hossein";
char ch = str.charAt(0);
int i = str.indexOf("Salam");
int j = str.lastIndexOf("Salam");
String str2 = str + "mazloom" + i;
String str3 = str.replace("salam", "Hi");

```

نکته مهم: با تساوی دو عبارت String، اشاره‌گر کپی می‌شود و محتوی کپی نمی‌شود یعنی متغیر s و t هر دو به یک شیء اشاره می‌کنند

- 1) String s; # اختصاص بخشی از حافظه به متغیر اس (نگهداری آدرس حافظه برای ارجاع به شیء دیگر)
- 2) s = new String("Ali"); # ایجاد شیء جدید و برگرداندن ارجاع آن به متغیر اس
- 3) String t = new String("Taghi"); # اختصاص بخشی از حافظه به متغیر و در ادامه نگهداری آدرس حافظه شیء در متغیر
- 4) s = t; # تغییر «آدرس ارجاع» متغیر اس به همان آدرسی که متغیر تی ارجاع دارد

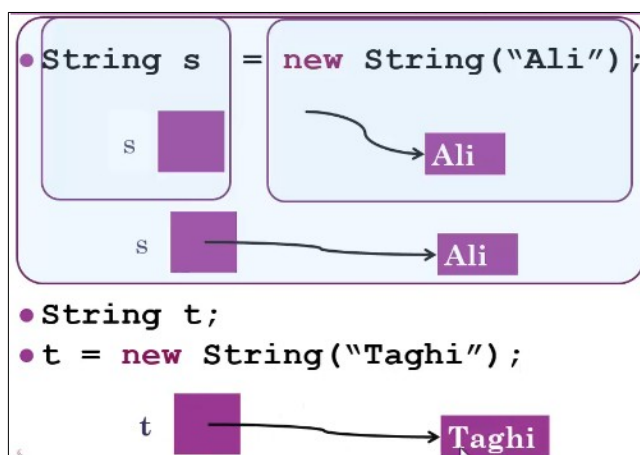
مفهوم بالا در شکل زیر بیان شده است

- برخی ساختار تولید متن در جاوا وجود دارد که در زیر آورده شده است

```

System.out.println(MessageFormat.format("Name:{0} , FamilyName: {1}",name,familyName);
System.out.println(String.format("Name:%s , FamilyName: %s",name,familyName);

```



۷.۲.۱.۲ تبدیل Collection به آرایه

واسط Collection دو متد با نام `toArray` برای تبدیل به آرایه معرفی می‌کند:
روش اول: با استفاده از متد زیر:

```
Object[ ] toArray()
```

این متد پارامتری نمی‌گیرد و فهرست را به یک آرایه از `Object` ها تبدیل می‌کند. بدین ترتیب نوع واقعی اشیاء در آرایه معلوم نیست

روش دوم: با استفاده از متد زیر:

```
T[] toArray(T[] a)
```

- در این روش، آرایه‌ای از اشیاء به عنوان پارامتر ارسال می‌شود
 - مقدار برگشتی از نوع داده مشخص شده است
 - اگر پارامتر موردنظر به اندازه کافی فضا داشته باشد، همان را پر می‌کند وگرنه، یک آرایه جدید از همان جنس می‌سازد
- مثال:**

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

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

Integer[] array2 = list.toArray(new Integer[list.size()]);
for (Integer i : array2)
```

```

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

```

ArrayList کلاسی (۷.۲.۲)

- در مسیر java.util.ArrayList قرار دارد
- لیستی که امکان تغییر سایز طول آرایه در آن وجود دارد. مثلاً زمانی که نمیدانیم تعداد یک آرایه چقدر است، استفاده از این مورد مطلوب می‌باشد
- در ابتدا خالی است و به مرور می‌توانیم عناصر به این فهرست اضافه یا کم کنیم.
- در مثال زیر شیء students ظرفی از اشیاء هست که هر نوع از اشیاء را میتوان به آن اضافه کرد. (معمولاً همه اجزای یک لیست دارای جنس مشابه می‌باشند)

```

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

```

- ولی معمولاً نمی‌خواهیم اجازه دهیم که یک ظرف اشیائی از انواع مختلف را نگه دارد

```

ArrayList s = new ArrayList();
s.add(new Student("Ali Alavi"));
s.add("Taghi Taghavi"); // توصیه نمی‌شود که از این نحو استفاده شود
s.add(new Object()); // توصیه نمی‌شود که از این نحو استفاده شود

```

- لیست‌ها میتوانند نوع اشیاء درون خود را مشخص کنند که در مثال زیر لیستی از اشیاء کلاس Student را ذخیره میکند

به این لیست فقط باید اشیاء «استیودنت» افزوده گردد //

```

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

```

Example:

```

students.add(new Student("Ali Alavi"));
students.add(new Student("Taghi Taghavi"));
students.remove(0); // حذف شیء صفر یعنی اولین شیء که به این لیست افزوده شده است
students.remove(new Student("Behrooz MohammadiNasab")); // حذف چنین شیء از لیست
Student stud = students.get(0); // دستیابی به یک شیء خاص
System.out.println(stud);

```

Limitation:

```

students.add("Taghi Taghavi"); // نمی‌توانیم رشته اضافه نماییم و فقط باید کلاس استیودنت اضافه گردد
students.add(new Object()); // نمی‌توانیم اوجکت اضافه نماییم و فقط باید کلاس استیودنت اضافه گردد

```

- لیست‌ها می‌توانند نوع اشیاء درون خود را مشخص کنند که در مثال زیر لیستی از اشیاء نوع رشته String را ذخیره میکند

```
ArrayList<String> names = new ArrayList<String>();
names.add("Ali Alavi");
names.add("Taghi Taghavi");
names.remove(0);
names.remove("Ali Alavi");
String nm = names.get(0);
System.out.println(nm);
```

- درواقع ArrayList لیستی هست که به کمک آرایه پیاده‌سازی شده است.
- برای حذف یک عضو ArrayList، خانه‌های بعدی در خانه قبلی کپی می‌شوند. و این کپی‌های ناخواسته سبب کاهش performance یا کارایی می‌شود.
- لیست ArrayList ظرفی از اشیاء است: هر یک از مقادیر داخل آن، یک شیء است
 - انواع داده اولیه (primitive types) نمی‌توانند در ArrayList قرار گیرند
 - این محدودیت برای سایر انواع ظرف‌ها (مثل Set و Linked List و ...) هم وجود دارد.
 - در واقع این محدودیت برای همه انواع عام (generics)، از جمله ظرف‌ها، وجود دارد.
 - این محدودیت برای آرایه وجود ندارد.

```
ArrayList<int>           // غلط است ❌
int[]                   // صحیح است ✅
```

- در مجموع، کلاس ArrayList پرکاربردتر است

۷.۲.۲.۱ متدهای مهم

```
int size()                // طول فهرست
boolean isEmpty()        // فهرست خالی است یا خیر
boolean contains(Object o) // وجود شیء موردنظر در فهرست
add(E e)                 // یک عضو به فهرست اضافه می‌کند
remove(Object o)         // یک عضو از فهرست حذف می‌کند
remove(int index)        // عضوی با شماره اندیس موردنظر را حذف می‌کند
clear()                  // همه اعضای فهرست را حذف می‌کند
get(int index)           // عضوی که در اندیس موردنظر است را برمی‌گرداند
int indexOf(Object o)    // شماره اندیس عضو موردنظر را برمی‌گرداند
```

- درواقع ArrayList یک زیر کلاس از واسط java.util.List می‌باشد و متدهای بالا همگی از واسط List ارث‌بری شده است و توسط کلاس ArrayList پیاده‌سازی شده است

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

مثال:

```

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

```

۷.۲.۲.۲ تبدیل آرایه به ArrayList

```

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

```

۷.۲.۲.۳ تبدیل ArrayList به آرایه:

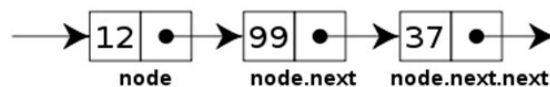
```

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

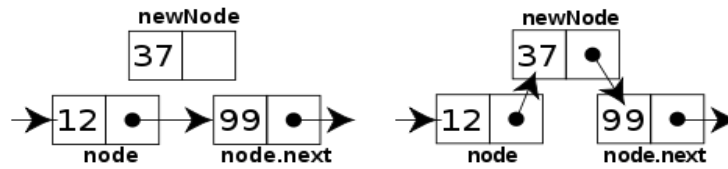
```

۷.۲.۳ لیست پیوندی (Linked List)

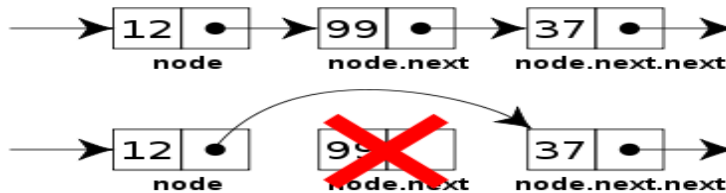
- برخلاف آرایه، همه اعضا پشت سرهم در حافظه قرار نمی‌گیرند. بلکه هر عضو فهرست، محل (آدرس یا ارجاع) عضو بعدی را نگه می‌دارد



- برای افزودن عضو به فهرست ابتدا یک شیء جدید ایجاد می‌کنیم و آخرین ارجاع (اشاره‌گر) به این شیء جدید اشاره خواهد کرد.



• برای حذف عضو از فهرست کافیت اشاره گر عضو کنونی، به شیء بعدی اشاره کند



- تحت کلاس java.util.LinkedList در جاوا پیاده سازی شده است
- کلاس Linked List هم مانند ArrayList واسط List را پیاده سازی کرده است. همه متدهای مهم List را دارد، مانند ، get ، add و remove ... بنابراین نحوه کاربرد Linked List مشابه ArrayList است

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

مثال از لیست پیوندی:

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

- کلاس های Linked List و ArrayList واسط List را پیاده کرده اند اما پیاده سازی متفاوتی دارند.
- درون هر Linked List یک آرایه نیست بلکه یک لیست پیوندی است.
- در زمانی که تعداد زیادی add و remove در لیست داریم بهتر است از لیست پیوندی استفاده شود.
- در هنگامی که دسترسی فراوان به عناصر (با استفاده از اندیس) وجود دارد بهتر است از نوع ArrayList پیاده سازی شود.
- هزینه اجرای get(i) در ArrayList کم است ولی در لیست پیوندی آن عنصر باید پیمایش شوند تا به عنصری با اندیس آن برسیم.
- در کد زیر بهتر است بخاطر ارجاع های مکرر از لیست پیوندی استفاده شود تا سریعتر اجرا شود

```
for(int i=0;i<1000000;i++){
for(int j=0;j<100;j++)
```

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

۷.۳ [مجموعه] Set واسط

- ساختمان داده Set از نوع Collection می‌باشد.
- شبیه معنای «مجموعه» در ریاضیات است. تعدادی شیء متمایز که لزوماً بین اعضا ترتیبی وجود ندارد
- Set اجازه افزودن عضو تکراری به مجموعه را نمی‌دهد. تکراری بودن عضو جدید را چک می‌کند (سربار محاسباتی) پس به همین لحاظ می‌تواند از هدر رفتن حافظه جلوگیری کند (کاهش حافظه مصرفی)
- مثلاً دو مجموعه زیر با هم برابر هستند

```
{1,2,3,1,4,2} = {4,3,2,1}
```

۷.۳.۱ کلاس HashSet

- کلاس HashSet واسط Set در مسیر java.util.Set را پیاده‌سازی کرده است. مثال:

```
HashSet<String> set= new HashSet<String>();
set.add("Ali");
set.add("Taghi");
set.add("Naghi");
```

مثال:

```
Set<String> set = new HashSet<String>();
set.add("Ali");
set.add("Taghi");
set.add("Taghi");
set.add("Ali");
set.add("Taghi");
System.out.println(set.size());           // OUTPUT: 2
for (String str : set)
    System.out.println(str);             // OUTPUT: Taghi "\n" Ali
set.remove("Ali");
System.out.println(set.contains("Ali"));  // OUTPUT: False
System.out.println(set.contains("Taghi")); // OUTPUT: True
set.clear();
System.out.println(set.size());           // OUTPUT: 0
```

۷.۴ واسط Map

واسط MAP در مسیر java.util.Map قرار دارد. باید توجه شود که یک MAP از نوع Collection نیست.

- **Map**: همانند جدول یا نگاشت از اشیاء عمل می‌کند. جدول دو ستون (زوج مرتب دوتایی) است که ستون اول مقدار Key و ستون دوم مقدار Value است.
 - اعضای ستون اول (کلیدها) **یکتا** هستند: کلید تکراری نداریم
 - اعضای ستون دوم (مقادیر) ممکن است **تکراری** باشند
 - نوع ستون اول و ستون دوم قابل تعیین است.
 - **نکته خیلی مهم**: انواع داده اولیه مثل int و double در هیچ‌یک از ظرف‌های جاوا قابل استفاده نیستند و برای استفاده باید از مقادیری همچون Integer یا Double استفاده نمایید و نه int یا double
- مثال: mp یک شیء از نوع map است که Key آن (ستون اول) از نوع رشته و Value یا مقادیر آن (ستون دوم) از نوع اعداد حقیقی است. به عبارتی می‌شود گفت که «نگاشتی از رشته به عدد حقیقی»

```
Map<String, Double> mp;
```

مثال: «نگاشتی از اعداد صحیح به دانشجو»

```
Map<Integer, Student> mp;
```

نگاهی به واسط Map:

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

۷.۵ کلاس HashMap

- کلاس java.util.HashMap واسط map را implement کرده است و به این واسط ساختار داده است
- در جاوا ساده‌ترین نوع cache را با کلاس HashMap می‌توان پیاده‌سازی کرد. کوچکترین واحدی که در آن می‌توان کش تولید کرد از آبجکت static HashMap است. که نام و مقدار دارد که در کش‌های داخل مموری مورد استفاده قرار می‌گیرد.

به قطعه کد زیر توجه نمایید:

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

- در کد بالا به جای Integer از int استفاده شده است
- تبدیل int به Integer از جاوا ۵ به بعد بصورت خودکار انجام می‌شود (auto Boxing) یعنی قسمت‌هایی توسط تابع put مقدار مستقیم int (مثلاً ۸۷۳۰۰۸۷۶) داده شده اشتباه بوده و بخاطر auto Boxing این موضوع ارور گرفته نشده است
- نوع مورد استفاده در همه کلاس‌های java collections framework باید شیء باشند

۷.۵.۱ put

- تغییر مقدار با کمک put صورت می‌گیرد
 - اگر سطری با کلید تکراری به یک map اضافه شود: مقدار قبلی آن کلید حذف می‌شود.
- مثال:

```
Map<Integer, String> map = new HashMap<Integer, String>();
map.put(76, "Ali")
map.put(31, "Taghi")
map.put(76, "Naghi") // 76: Naghi
```

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

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

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

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

```

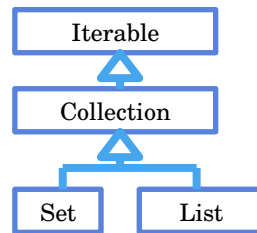
Double totalSum = 0.0;
for (Double avg : map.values())
    totalSum += avg;

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

```

7.6 واسط Iterator

- واسط Collection، از واسط Iterable ارث‌بری کرده است
- بنابراین List ها و Set ها همگی Iterable هستند



- از جاوا 5 به بعد، for each برای آرایه‌ها و collection ها ممکن شد

Example1:

```

int[] array = {1,2,3,7};
for (int i : array)
    System.out.println(i);

```

Example2:

```

List<Integer> list ;
...
for (Integer i : list)
    System.out.println(i);

```

- قبل از جاوا 5 با کمک iterator پیمایش روی collection ها انجام می‌شد. این امکان همچنان وجود دارد و کاربردهایی نیز دارد
مثال:

```

List<Integer> list = ... ;
Iterator<Integer> iterator = list.iterator();
while(iterator.hasNext()){
    Integer i = iterator.next();
    System.out.println(i);
}

```

- واسط Iterator که سبب شد تا امکان for each در نسخه‌های جدید جاوا با کمک آن پیاده شود.

```

public interface Iterator<E> {
    boolean hasNext();
    E next(); // رفتن به خونه بعدی
}

```

```
void remove();
Iterator<T> iterator();
}
```

- واسط‌های زیرمجموعه Collection هم Iterator را ارث‌بری می‌کنند:

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

۷.۶.۱ متد Iterator

- متد iterator در واسط java.lang.Iterable تعریف شده است. هر کلاسی که Iterable را Implement نماید باید متد Iterator را هم پیاده‌سازی نماید
- هر for each توسط Iterator انجام می‌شود
- در واقع همه کلاس‌هایی که Iterable هستند، امکان for each دارند

مثال

```
List<Integer> list = new ArrayList<Integer>();
for(int i=0;i<10;i++)
    list.add(i);

Iterator<Integer> iterator = list.iterator();
while(iterator.hasNext()){
    Integer value = iterator.next();
    if(value%2==0)
        iterator.remove();
}
System.out.println(list.size()); // OUTPUT: 5
```

۷.۶.۲ تغییر همزمان یا Concurrent Modification

- فرض کنید: چند بخش برنامه به صورت همزمان در حال استفاده از یک ظرف باشند(مثلاً یک لیست یا مجموعه)و در همین حال، یک بخش از برنامه، تغییری در ظرف ایجاد کند(مثلاً شیئی به آن اضافه یا کم کند) این تغییر همزمان نباید ممکن باشد زیرا یک ظرف توسط یک بخش در حال پیمایش است و در بخش دیگری تغییر می‌کند
 - مثلاً شاید در بخشی که پیمایش انجام می‌شود، روی طول ظرف حساب شده باشد که سبب تغییر روی این محور محاسبه بشود و تمامی معادلات به هم بریزد
 - مثلاً شیئی که پیمایش و پردازش شده، توسط بخش دیگری از برنامه حذف شود
- جاوا از تغییرات همزمان از Containerها (مواردی ذیل واسط Collection یا موارد ذیل واسط Map) جلوگیری می‌کند.
- با تغییر حتی روی یک متد در Container ها سبب «غیرمعتبر» نمودن تمامی iterator های آن Container می‌شود و در ادامه هر عملیات روی iterator های غیرمعتبر شده سبب پرتاب خطای **Concurrent Modification Exception** می‌شود. به این تکنیک، «شکست سریع» یا **Fail Fast** گفته می‌شود
 - با تغییر یک ظرف توسط یک iterator ، سایر iterator ها غیرقابل استفاده می‌شوند. این تکنیک، روش جاوا برای جلوگیری از تغییر همزمان است.

- مثال: در این مثال itr غیر معتبر می‌شود و منجر به ارسال خطای ConcurrentModificationException می‌شود.

```
Collection<String> c = new ArrayList<String>();
Iterator<String> itr = c.iterator();
c.add("An object"); // این خط سبب می‌شود که «آی‌تی‌آر» نامعتبر شود
// و هر گونه پیمایش روی «آی‌تی‌آر» سبب ارور می‌شود
String s = itr.next(); // سبب بروز خطا می‌شود زیرا «آی‌تی‌آر» نامعتبر است
```

- مثال: بلافاصله بعد از اجرای متد remove روی این خط خطا دریافت می‌کنیم و iterator ای که با کمک آن حلقه for را به اجرا درآورده‌ایم نامعتبر می‌شود:

```
List<String> list = new ArrayList<String>();
list.add("A");
list.add("B");
list.add("C");
for (String s : list)
    if(s.equals("A"))
        list.remove(s); // با تغییر یافتن کالکشن «لیست» ایتريتورهاى ديگرى كه بر روى ليست ساخته شده‌اند نامعتبر مى‌شوند
// كه يكى از اين ايتريتورها ، حلقه فور است
// و ميدانيم كه هر فورايج توسط يك ايتريتور انجام مى‌شود
// پس هنگامى كه دوباره به حلقه رجوع مى‌کند و ميخواهد مقدار بگيرد ارور نامعتبر بودن ليست را ميدهد
// نکته مهم: ارور روى خط حلقه فور داده مى‌شود يعنى خط 5 ونه خط حذف يك پارامتر
```

- مثال غلط: حذف یک رکورد از یک لیست: این راه حل منجر به Concurrent Modification Exception می‌شود. (از این مورد استفاده نکنید):

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

- مثال صحیح: راه حل (اول) صحیح مثال بالا (از این روش استفاده نمایید):

```
public static void removeAli(List<String> list){
    Iterator<String> iterator = list.iterator();
    while(iterator.hasNext()) {
        String string = iterator.next();
        if(string.startsWith("Ali"))
            iterator.remove();
    }
}
```

- مثال صحیح: راه حل (دوم) صحیح مثال بالا (از این روش استفاده نمایید):

```
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);
}
```

۷.۷ واسط Comparable

گاهی لازم است دو شیء با هم مقایسه شوند و ترتیب آن‌ها مشخص شود (کدام یک کوچک‌تر است و کدام یک بزرگ‌تر!) کاربرد مقایسه اشیاء: مرتب‌سازی و جستجوی سریع‌تر (مثلاً در ظرفی از اشیاء)

- متد equals، فقط تساوی اشیاء را بررسی میکند
- برای بسیاری از انواع داده (کلاس) معنای مشخصی برای «ترتیب» اشیاء وجود دارد
 - در موارد عددی به کمک عملگرها می‌توان بزرگ‌تر و کوچک‌تر را جستجو نمود
 - در موارد غیر عددی عملگرهای مقایسه‌ای (> و < و <= و >=) کار نمی‌کنند
- اگر کلاسی واسط Comparable را پیاده کند، یعنی برای اشیاء این کلاس، عنوان ترتیب معنا داشته و مهم است.

نگاهی به واسط Comparable:

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

۷.۷.۱ متد compareTo

- تابع compareTo شیء جاری را با پارامتر متد مقایسه می‌کند و یک عدد برمی‌گرداند
 - عدد خروجی اگر مثبت باشد یعنی شیء جاری بزرگ‌تر از پارامتر متد است
 - عدد خروجی اگر صفر باشد یعنی شیء جاری مساوی از پارامتر متد است
 - عدد خروجی اگر منفی باشد یعنی شیء جاری کوچک‌تر از پارامتر متد است
- مثال: در خط شماره ۳ پارامتر this همان date است که تعداد میلی‌ثانیه تا کنون را برمی‌گرداند. خط ۴ تعداد میلی‌ثانیه زمانی خاص را برمی‌گرداند و خط بعدی مقایسه می‌کند

```
class java.util.Date implements Comparable<Date>{  
    public int compareTo(Date anotherDate) {  
        long thisTime = getMillisOf(this);  
        long anotherTime = getMillisOf(anotherDate);  
        return (thisTime<anotherTime ? -1 : (thisTime==anotherTime ? 0 : 1));  
    }  
}
```

نمونه کاربردی کلاس بالا در قطعه کد زیر استفاده شده است

```
//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)); // OUTPUT:1  
System.out.println(d2.compareTo(d1)); // OUTPUT:-1  
System.out.println(d2.compareTo(d3)); // OUTPUT:0
```


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

۷.۷.۲) واسط Comparator

واسط دیگری بنام Comparator داریم که در مسیر java.util.Comparator قرار دارد.

- گاهی می‌خواهیم اشیاء را با ترتیبی غیر از آنچه خودشان تعریف کرده‌اند مقایسه کنیم. مثلاً کلاس دانشجو واسط Comparable را پیاده‌سازی کرده و متد compareTo را بر اساس معدل دانشجو تعریف کرده ولی ما می‌خواهیم فهرست دانشجویان را بر اساس «سن» مرتب کنیم (ترتیب بر اساس سن) گاهی نیز می‌خواهیم اشیائی را مقایسه کنیم که کلاسشان Comparable نیست. در این موارد واسط Comparator را برای مقایسه این اشیاء پیاده‌سازی می‌کنیم

ساختار واسط Comparator :

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

مثال: Comparator برای کلاس Student: در زیر دو دانشجو توسط متد compareTo بر حسب نمره با هم مقایسه می‌شوند

```
class Student implements Comparable<Student> {
    int age;
    double grade;
    // مقایسه زیر بر حسب نمره صورت می‌گیرد
    public int compareTo(Student s) {
        return (this.grade<s.grade ? -1 : (this.grade == s.grade ? 0 : +1));
    }
    //Constructor for this class
    public Student(int age, double grade) {
        this.age = age;
        this.grade = grade;
    }
}
```

ادامه: اگر بخواهیم مقایسه دو دانشجو نه بر حسب نمره بلکه بر حسب سن مقایسه نماییم. به همین خاطر مطابق زیر عمل می‌کنیم

```
class StudentComparator implements Comparator<Student>{
    // مقایسه بر حسب سن صورت می‌گیرد
    public int compare(Student s1, Student s2) {
        return s1.age<s2.age? -1:(s1.age==s2.age?0 : +1);
    }
}
```

ادامه: استفاده از کلاس فوق مطابق زیر است

```
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));
```

output:

```
-1
1
```

۷.۸ داده‌های Enumeration

- فرض شود که یک کلاس، تعداد مشخص و محدود شیء خواهد داشت و در آینده اضافه نخواهد شد. بهتر است بجای کلاس با کلیدواژه enum تعریف شود و همان جا همه اشیاء (نمونه‌ها) آن مشخص شود.
- همه نمونه‌های enum بصورت ضمنی **public, static, final** هستند.
- هیچ نمونه (شیء) جدیدی نمی‌تواند ایجاد شود (نمونه سازی با کمک new با ارور مواجه خواهد شد)
- ارث‌بری از enum امکان‌پذیر نیست
- مثال اول: وضعیت یک اس ام اس که می‌تواند در وضعیت ۱- ارسال شده ۲- در دست ارسال ۳-مواجه با خطا
- مثال دوم: رنگ که آبی و سبز و مشکی و قرمز (که برای پیاده‌سازی چنین مواردی از enum استفاده می‌شود).
- مثال سوم: کلاس نوع دانشجو که حتماً چیزی بین لیسانس و فوق لیسانس و دکترا و این قبیل است
- مثال رنگ را با دو حالت زیر که یکی با enum و دومی بدون enum است پیاده‌سازی می‌کنیم:

حالت بدون Enum:

```
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() { }
}
```

حالت Enum:

```
enum Color { BLACK, BLUE, GREEN, RED }
```

برای استفاده از این مورد باید به شکل زیر اقدام کرد:

```
Color color = Color.BLACK;
```

مثال متفاوت:

```
enum Shape { RECTANGLE, CIRCLE, SQUARE}
enum StudentType {BS, MS, PHD }
```

بعنوان نمونه به قطعه کد زیر توجه نمایید:

```
enum Shape {
    Rectangle(1),    // نحوه خاص استفاده از تابع سازنده در این نوع داده‌ها
    Circle(2),      // نحوه خاص استفاده از تابع سازنده در این نوع داده‌ها
    Square(3);      // نحوه خاص استفاده از تابع سازنده در این نوع داده‌ها

    private int number;
    Shape(int i){    // تعریف تابع سازنده
        number= i;
    }
    public int getNumber(){
        return number;
    }
}
```

برای استفاده از این مورد باید به شکل زیر اقدام کرد:

```
Shape sh = Shape.CIRCLE;    // ایجاد یک ارجاع بنام «اس‌اچ» که یکی از اشیاء تحت عنوان «سیرکل» اشاره می‌کند
print (sh.getNumber());    // برگرداندن عدد ۲
sh = Shape.valueOf("CIRCLE"); // یک متد استاتیک «ولیو آف» درون این «اینام‌ها» وجود دارد که یک رشته می‌گیرد و
                             // شیء که با این نام است را برمی‌گرداند یعنی شیء را برمی‌گرداند
print(sh.getNumber());    // متد استاتیک «ولیز» بصورت پیش‌فرض وجود دارد که تمام مقادیر را درون آرایه می‌ریزد
Shape[] array = Shape.values(); // متد نام یک متد استاتیک است که بصورت پیش‌فرض وجود دارد
for (Shape s : array) {    // متد «نام» و متد «تو استرینگ» مشابه هم کار می‌کنند
    print(s.name());
}

// Runtime Error:
sh = Shape.valueOf("PYRAMID"); // در زمان استفاده از داده غیر از موارد تعریف شده با ارور مواجه خواهیم شد
```

۷.۹ Variable Argument List متغیر «متفاوت پارامتر»

- امکان تعریف متد به نحوی که بعضی از پارامترهای این متدها تعداد متغیری از پارامترها رو دریافت کنند.
- درواقع آرگومان‌های این متد بگونه‌ای است که می‌شود تعداد صفر یا یک یا چند پارامتر ورودی بعنوان آرگومان به این متد ارسال کرد
- پیاده‌سازی آن به شکل زیر است.

```
void Func(Type... args){...}
```

```

void func1(String... args){...} # هنگام فراخوانی تابع می‌توانیم به این تابع یک یا چند رشته پاس بدهیم
func1(); # طول آرایه زیر صفر است
func1("Ali"); # طول آرایه زیر یک است
func1("A", "B", "C", "D"); # طول آرایه زیر چهار است

```

- پارامتر ارسالی به این گونه متدها، به شکل یک آرایه قابل استفاده است.

```

void Func2(String... params){
    String[] array = params;
    System.out.println(array.length);
    for(String p : params) {
        System.out.println(p);
    }
}

```

- تفاوت استفاده از varargs با آرایه را در دو قطه زیر می‌توان دید.

```

1) void func1(String[] args){...}
2) void func1(String... args){...}

```

روش‌های فراخوانی دو تابع بالا به اشکال زیر است

```

1)
String[] array = {"A", "B"};
func1(array); # فراخوانی متد اول فقط به این شکل است

```

```

2) فراخوانی متد دوم به همه اشکال زیر صحیح است و دست برنامه‌نویس باز است
func2();
func2("A");
func2("A", "B");
func2("A", "B", "C");
func2(array);

```

7.10 کلاس Arrays

جاوا یک کلاس برای کار با آرایه‌های تحت عنوان java.util.Arrays فراهم آورده است که این کلاس دارای متدهای فراوانی است که بصورت استاتیک در آن تعبیه شده است و برای استفاده از آن نیاز به ساختن شیء نیست و مستقیماً می‌توان از آن استفاده نمود

- این کلاس‌ها دارای متدهای استاتیک متنوعی هستند، نظیر
 - کپی اشیاء درون آرایه
 - پر کردن همه اعضا با یک مقدار مشخص (fill)
 - جستجو (search)
 - مرتب‌سازی (sort) و ...

```

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

```

کلاس Collections (۷.۱۱)

جاوا یک کلاس برای کار با آرایه‌های تحت عنوان `java.util.Collections` فراهم آورده است که این کلاس دارای متدهای فراوانی است که بصورت استاتیک در آن تعبیه شده است و برای استفاده از آن نیاز به ساختن شیء نیست و مستقیماً می‌توان از آن استفاده نمود.

- این کلاس‌ها دارای متدهای استاتیک متنوعی هستند، نظیر
 - کپی اشیاء درون ظرف‌ها
 - پر کردن همه اعضا با یک مقدار مشخص (`fill`)
 - جستجو (`search`)
 - مرتب‌سازی (`sort`) و ...

```

List<String> list = new ArrayList<String>();
Collections.addAll(list, "A", "Book", "Car", "A");
int freq = Collections.frequency(list, "A");
Collections.sort(list);
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);
Collections.reverse(list);

String min = Collections.min(list);
String max = Collections.max(list);

```

2

A, A, Book, Car

A, A, Car, Book

A

car

Book

```
String max2 = Collections.max(list, comp);

Collections.shuffle(list);
Collections.fill(list, "B");
```

۷.۱۲ انواع داده عام Generic

- داده عام از نسخه ۵ به جاوا اضافه شد. قبل از JDK ۱.۵ اصلاً Generic نداشتیم
- کلاس ArrayList بگونه‌ای پیاده‌سازی شده که انواع متغیر String و Integer و Double یا غیره را بتواند به لیست تبدیل نماید.
- گاهی کلاس باید بگونه‌ای نوشته‌شود که انواع داده‌ها را بتواند (همانند کلاس ArrayList) پوشش دهد.
- کلاس ArrayList یک نوع داده عام (Generic) است
 - این قابلیت هنگامی برای کلاس ArrayList فراهم می‌شود که نوع داده آن، از نوع عام استفاده شده باشد
 - اگر کلاس ArrayList از نوع داده عام تعریف نمی‌شد آنگاه باید برای هر کدام از انواع داده کلاس مستقل مثلاً StringArrayList و IntegerArrayList و StudentArrayList تعریف می‌نمودیم.

```
ArrayList<String> list1 = new ArrayList<String>(); # نمونه لیست از نوع متن
list1.add("Ali");
OR
ArrayList<Integer> list2 = new ArrayList<Integer>(); # نمونه لیست از نوع عدد
list2.add(new Integer(2));
```

- یک واسط یا کلاس مجرد نیز می‌تواند همانند کلاس عام (generic) از نوع عام تعریف شود
- قابلیت استفاده از چند «پارامتر نوع» در یک کلاس عام (generic type) وجود دارد
 - بطور مثال، کلاس Map که دارای دو نوع عام K و V است. در ادامه مثال آن آورده شده است.
- در زمان استفاده از کلاس‌های نوع عام، پارامترهای داده شده باید حتماً از نوع کلاس باشند و Primitive Data نمی‌توانند بعنوان ورودی تعریف شوند

```
interface List<E>
List<String> str1; // ✓
List<Integer> ints; // ✓
List<Student> stus; // ✓
List<int> error1; // ✗ خطای کامپایل
List<double> error2; // ✗ خطای کامپایل
```

۷.۱۲.۱ نحوه تعریف کلاس عام

- در مثال زیر فرض شود بخواهیم کلاس Stack (که در جاوا وجود دارد) را پیاده‌سازی نماییم:
 - این کلاس قرار است پشت‌پشته‌ای از اشیاء را نگهداری نماید
 - طرفی از اشیاء که به کمک متد push دیتا را ذخیره کند
 - طرفی از اشیاء که به کمک متد pop دیتا را بازیابی می‌کند
 - هر جا کاراکتر E را مشاهده کردیم یعنی این مقدار می‌تواند هر نوع باشد.

○ کاراکتر E استفاده شده در این مثال یک اسم دلخواه است و می‌توانست T باشد

```
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];
    }
}
// Sample1
Stack<String> st1;
st1 = new Stack<String>(10);
st1.push("A");
st1.push("B");
String p1 = st1.pop();
// Sample2
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"); // در لیست اعداد حروف جای نمی‌گیرد و تولید ارور می‌نماید
```

۷.۱۲.۲ ایجاد کلاس‌های عام دلخواه

مثال ۱: 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

```
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;
    }
}
// Sample1
Pair<String, Double> p1;
p1 = new Pair<String, Double>("Ali", 19.0);
String name = p1.getFirst();
Double avg = p1.getSecond();
// Sample2
Pair<String, String> p2;
p2 = new Pair<String, String>("Ali", "Alavi");
String fname = p2.getFirst();
String lname = p2.getSecond();

```

مثال ۲ (لیست پیوندی):

○ هر گره دو فیلد مهم دارد: ۱- مقدار (از هر نوعی می‌تواند 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;
    }
}
// Sample1
Node<String> last = new Node("Taghi", null);
Node<String> first = new Node("Ali", last);

```

۷.۱۲.۳) برخی کلاس‌ها نوع عام جاوا

نگاهی به کلاس ArrayList

```

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);
}

```

نگاهی به کلاس ArrayList

```

public class HashMap<K,V> implements Map<K,V>
//extends... implements...
{
...
}
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();
}

```

۷.۱۲.۴) 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.

مثال: در این مثال، extends یعنی «پارامتر نوع» باید زیر کلاس Number باشد (کلاس‌هایی که عدد را ارث بری می‌کنند یعنی باید عدد باشد و رشته نمی‌تواند باشد)

```

class NumbersQueue<T extends Number>{...}
interface SortedList<E extends Comparable>{...}
class Person{}
// در این صورت موارد زیر را بررسی نمایید
NumbersQueue<Integer> n; // ✓ باید عدد باشد پس صحیح است
NumbersQueue<Double> d; // ✓ باید عدد باشد پس صحیح است
NumbersQueue<String> s; // ✗ باشد عدد باشد ولی رشته گرفته پس غلط است
NumbersQueue<Person> p; // ✗ باید عدد باشد ولی کلاس گرفته پس غلط است
SortedList<Long> l; // ✓ باید مواردی باشد که «کامپریبل» باشد و نوع لانگ هست
SortedList<Float> f; // ✓ باید مواردی باشد که «کامپریبل» باشد و نوع اعشاری هست
SortedList<String> s; // ✓ باید مواردی باشد که «کامپریبل» باشد و نوع رشته هست
SortedList<Person> p; // ✗ باید مواردی باشد که «کامپریبل» باشد و نوع کلاس نیست

```

۷.۱۲.۵) نوع خام (Raw Type)

انواع داده عام را بدون تصریح «پارامتر نوع» هم می توان استفاده کرد. که در این صورت، کامپایلر حداقل محدودیت ممکن را برای این انواع اعمال می کند.

مثال ۱: کلاس ArrayList را اگر بدون پارامتر استفاده نمایید آنگاه هیچ محدودیتی نداشته و هر نوع را به خود می تواند بگیرد. در این مثال، هر شیئی قابل افزودن به List است (محدودیتی نیست)

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

مثال ۲: در این مثال، فقط اشیائی از نوع Number (یا فرزندان Number) قابل استفاده در queue هستند

```
class NumbersQueue<T extends Number>{ // درج «اکستند» به منزله محدود کردن است که تنها از آن نوع باشد
    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"); // رشته نمی تواند بگیرد و فقط باید از نوع اعداد باشد
```

۷.۱۲.۶) استنتاج نوع یا Type Inference

- از نسخه ۷ (Java ۱.۷) به بعد، «استنتاج نوع» برای انواع عام ممکن شده است.
- به ویژه، ذکر نوع عام در هنگام نمونه سازی از انواع عام لازم نیست (نوع آن استنتاج می شود)

```
ArrayList<String> list = new ArrayList<String>(); // در قدیم باید حتماً مقدار استرینگ را مشخص می کردیم
ArrayList<String> list = new ArrayList<>(); // در جاوای جدید نیاز به معرفی نیست و کامپایلر استنتاج می کند

Map<String, List<Student>> map = new HashMap<String, List<Student>>(); // نسخه قدیمی باید حتماً می نوشتیم
Map<String, List<Student>> map = new HashMap<>(); // نسخه جدید نیاز به نوشتن نیست
```

۷.۱۲.۷) وراثت و انواع داده عام

- یک نوع عام یا غیرعام می تواند از یک نوع عام یا غیرعام ارث بری کند که در این صورت چهار حالت پیش می آید
- حالت اول: یک نوع غیرعام، فرزند نوع غیرعام دیگری باشد
 - حالت دوم: یک نوع عام، فرزند یک نوع غیرعام باشد
 - حالت سوم: یک نوع غیرعام، فرزند یک نوع عام باشد
 - در این صورت، زیرکلاس، عام بودن را کنار می گذارد

- زیر کلاس تعیین می‌کند از چه نوع خاصی به جای پارامتر نوع اَبَر کلاس استفاده می‌کند
- **حالت چهارم:** یک نوع عام، فرزند یک نوع عام باشد
- در این صورت می‌تواند پارامتر نوع را محدودتر کند

مثال حالت اول:

```
class A{}
class B extends A{}
```

مثال حالت دوم:

```
class Box<T> extends B{}
```

مثال حالت سوم:

```
class IntList implements List<Integer>{
    public boolean add(Integer e){...}
}
```

مثال حالت چهارم:

```
interface NumberList<T extends Number> extends List<T>{}
```

7.12.8 متدهای عام یا Generic Method

متدهای یک کلاس عام می‌توانند از نوع داده عام آن کلاس استفاده کنند. به عنوان پارامتر یا نوع داده برگشتی:

```
class ArrayList<E> {
    public E get(int index) {...}
    public boolean add(E e) {...}
}
```

اما یک متد، خود نیز می‌تواند، نوعی عام را به عنوان «پارامتر نوع» معرفی کند.

- و این نوع عام می‌تواند نوعی غیر از چیزی باشد که در خود کلاس تعیین شده است باشد مثلاً نوعی غیر از E در کلاس بالا

```
class GenericClass<T>{
    public <E> void function(E p1, T p2){}
}
```

- حتی برای متدهایی که در کلاس‌های غیر عام قرار دارند هم ممکن است
- به این متدها، متد عام (Generic Method) گفته می‌شود

```
interface NotGeneric{
```

```
public <E> E function(E p1);
}
```

مثال:

```
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;
    }
}
// استفاده از کلاس بالا به صورت زیر است
String s = new NotGeneric().chooseRandom("A", "B");
Integer num = new NotGeneric().chooseRandom(1, 2);
num = NotGeneric.max(1, 2);
s = NotGeneric.max("A", "B");
```

۷.۱۲.۹ فرآیند مَحُو (Erasure)

- کنترلی که بر روی انواع داده عام (Generic) اعمال می شود مربوط به زمان کامپایل توسط کامپایلر است. و تنها کامپایلر هست که از عدم بروز اشتباه در نحوه استفاده از Generic مراقبت می کند.
- هنگامی که کامپایل تمام می شود و برنامه به اجرا درمی آید، اثری از «پارامترنوع» در زمان اجرا وجود نخواهد داشت و در زمان اجرا پارامتر نوع کلاً حذف خواهد شد. عبارتی در ByteCode (منظور فایل کامپایل شده یا فایل با پسوند class هست) اطلاعاتی درباره «پارامترنوع» یک شیء نیست. که به این نحوه رفتار Erasure گفته می شود.
- به عنوان مثال، هر چهار خط زیر به یک شکل ترجمه می شوند. کد ترجمه شده ی 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
```

- مثال دیگر: در ترجمه کلاس SortedList زیر، هر جا پارامتر نوع (T) استفاده شده، در هنگام ترجمه با Comparable جایگزین می شود و همچنین در ادامه کد ترجمه شده ی هر سه دستور زیر (در byte code) یکسان است

```
class SortedList<T extends Comparable>{}
SortedList<Integer> list = new SortedList<>();
SortedList<String> list = new SortedList<>();
SortedList<Comparable> list = new SortedList<>();
```

- خلاصه: اعمال محدودیت‌ها در زمینه داده‌های عام برعهده کامپایلر است. بعد از کامپایلر (در زمان اجرا) محدودیتی در زمینه داده‌های عام اعمال نمی‌شود.

مثال: در مثال زیر عبارتهای اول به عبارتهای دوم ترجمه می‌شود

```
// عبارت اول
class Stack<T>{
    void push(T s){}
    T pop() {...}
}
// عبارت دوم
class Stack{
    void push(Object s){}
    Object pop() {...}
}
```

مثال: در مثال زیر عبارتهای اول به عبارتهای دوم ترجمه می‌شود

```
// عبارت اول
class SortedList<T extends Comparable<T>>{
    T[] values;
    public void add(T o){...}
}
// عبارت دوم
class SortedList{
    Comparable[] values;
    public void add(Comparable o){...}
}
```

مثال: در مثال زیر عبارتهای اول به عبارتهای دوم ترجمه می‌شود

```
// عبارت اول
static <E extends Comparable<E>> E max(E p1, E p2){...}
// عبارت دوم
static Comparable max(Comparable p1, Comparable p2){...}
```

۷.۱۲.۱۰ محدودیت‌ها در تعریف «نوع عام»

- در زمان اجرا اگر از یک «شیء نوع عام» استفاده شده باشد، «پارامتر نوع» اصلاً معلوم نخواهد بود. مثلاً در زمان اجرا هنگام ایجاد (new) نمودن یک شیء از نوع لیست دو حالت زیر یکسان خواهند بود

```
List<String>
List<Integer>
```

- در زمان اجرا، عملیات بر روی «پارامتر نوع» غیرممکن است (عملیات زمان اجرا نظیر new و instanceof)
- نمونه‌سازی (با کمک new) از «پارامتر نوع» غیر ممکن است

```
class Name<T>{
    T ref = new T(); // ❌ خطای کامپایل
}
```

- ایجاد (new) آرایه از پارامتر نوع ممکن نیست (البته قسمت قبل از مساوی اشکال ندارد و قسمت بعد از مساوی سبب بروز ارور می شود)

```
T[] elements = new T[size]; // ❌ 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;
```

- اگر پارامتر نوع را قید کنیم، ایجاد آرایه از نوع عام ممکن نیست

```
Stack<String>[] s = new Stack<String>[8]; // ❌ خطای کامپایل
Stack[] t = new Stack[8]; // ✅ بدون خطای کامپایل
```

- استفاده از انواع داده اولیه (primitives) به عنوان پارامتر نوع ممکن نیست

```
Stack<int> s; // ❌ خطای کامپایل
```

- در صورت تعیین پارامتر نوع، امکان instanceof برای نوع عام وجود ندارد

```
if(o instanceof Stack<String>); // ❌ خطای کامپایل
if(o instanceof Stack); // ✅ بدون خطای کامپایل
```

- یک کلاس عام نمی تواند به عنوان استثنا (exception) استفاده شود. یک کلاس عام نمی تواند از Throwable ارث بری کند. بنابراین شیئی از نوع عام را نمی توان پرتاب (throw) یا دریافت (catch) کرد

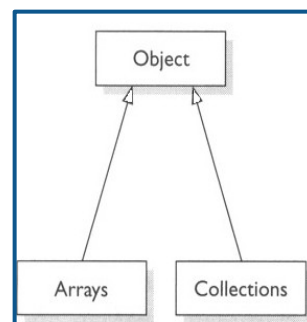
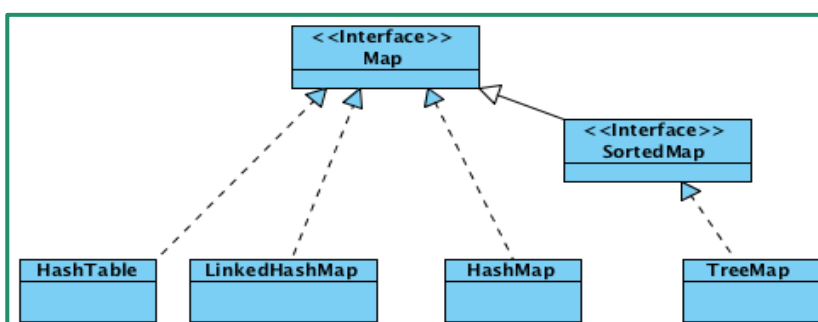
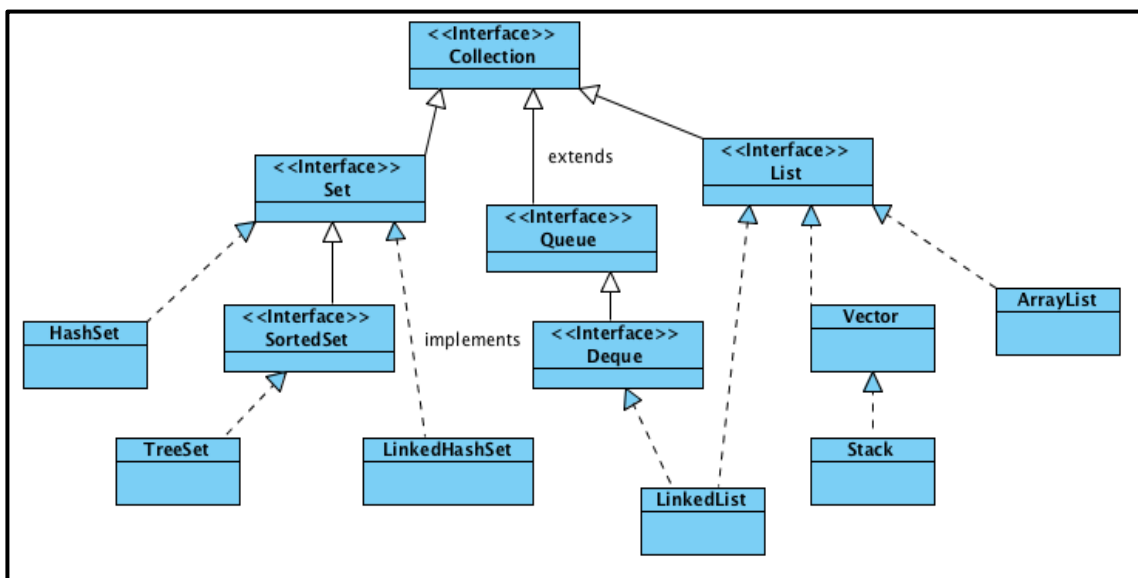
```
class GenExc<T> extends Exception {} // ❌ syntax error
```

- البته از «پارامتر نوع» (و نه از خود نوع عام) می توان به صورت استثنا استفاده کرد:

```
class Generic<T extends Exception> {
    void f() throws T {...}
    <E extends Throwable> void g() throws E {...}
}
```

۷.۱۳) مرور برخی واسط‌ها و کلاس‌های مهم دیگر

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



۷.۱۴ برخی مفاهیم

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

مثال:

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

نکته (از جاوا ۷ به بعد): امکان عدم ذکر نوع عام (**generic**) هنگام ایجاد شیء

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

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

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


فصل ۸ Stream

- سه نوع ورودی رایج در برنامه‌ها به شکل زیر است:
 - ورودی استاندارد (Standard Input)
 - خروجی استاندارد (Standard Output)
 - خروجی ارور استاندارد (Standard Error)
- ورودی‌های System.in از نوع

نمایش در خروجی استاندارد

```
System.out.println("")
System.out.println(variable)
System.out.format("min double = %5.2f", floatVariable); # فرمت مناسب برای نمایش اعداد اعشاری
```

کلاس‌های زیادی در جاوا پیرامون مبحث ورودی و خروجی مطرح است که برخی از آن‌ها در ذیل آورده شده است

Scanner class (۸.۱)

این کلاس در مسیر java.util.Scanner قرار دارد و برای دریافت و پردازش رشته‌های متنی کاربرد دارد. مثلاً اگر بخواهیم درخصوص برنامه‌ای از کاربر دیتا دریافت کنیم

```
Scanner scanner = new Scanner(System.in);
String line = scanner.nextLine(); # دریافت رشته از کاربر
int i = scanner.nextInt(); # دریافت عدد اینتیجر از کاربر
double d = scanner.nextDouble(); # دریافت عدد دابل از کاربر
```

کلاس Scanner می‌تواند با انواع Input Stream ها و Reader ها کار کند

```
s = new Scanner("1.txt");
s = new Scanner(new File("1.txt"));
s = new Scanner(new FileInputStream("1.txt"));
s = new Scanner(new FileReader("1.txt"));
```

فصل ۹) مدیریت خطا (Exception)

Exception: خطا یا اتفاقی غیرعادی که در جریان اجرای برنامه رخ می‌دهد و سبب اختلال در روند اجرای طبیعی برنامه می‌شود. بصورت پیش‌فرض اگر در زمان اجرا خطا یا استثنایی رخ دهد، این استثنا توسط JVM کشف می‌شود و توضیحاتی درباره این استثنا به خروجی standard output هدایت می‌شود و اجرای برنامه قطع می‌شود و خاتمه می‌یابد (اگر آن Exception مدیریت نشده باشد). این روند پیش‌فرض می‌باشد ولی معمولاً برنامه‌نویس باید عکس‌العمل بهتری برای زمان بروز استثنا پیاده‌سازی کند.

- برخی از عناوین مهم که سبب بروز خطا در زمان اجرا خواهند شد:
 - ورودی نامعتبر
 - تقسیم عدد بر صفر
 - دسترسی به مقداری از آرایه که خارج از محدوده آرایه است
 - خرابی دیسک
 - باز کردن استریم (سوکت یا فایل یا دیوایس یا غیره) که وجود ندارد
- استثنا یک مفهوم در زمان اجرا است و یک کد در زمان کامپایل، هیچ خطایی ندارد

9.1 Exception Object

- هر استثنا یک شیء است و هنگام رخداد Exception، «شیء استثنا» یا «Exception Object» به JVM تحویل می‌شود که به آن «پرتاب استثنا» یا «Throwing Exception» گفته می‌شود.
- «شیء استثنا» شامل اطلاعاتی نظیر موارد زیر است:
 - پیغام خطا
 - اطلاعاتی پیرامون نوع خطا
 - شماره خطی از برنامه که استثنا در آن رخ داده است
 - محل اصلی پرتاب شدن استثنا
 - **Stack Trace:** مجموعه متدهایی که استثنا از آن‌ها عبور کرده و متدهای **getStackTrace** و **printStackTrace** قابلیت دسترسی به اطلاعات را از طریق «شیء استثنا» فراهم می‌آورد

```
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();
    }
}
```

OUTPUT:

```
java.lang.NullPointerException
at Third.g(Third.java:18)
at Third.f(Third.java:13)
at Third.main(Third.java:5)
```

۹.۱.۱ ایجاد Class Exception جدید

- کلاس جدید معمولاً کلاس‌های ساده‌ای از «ابرکلاس» `java.lang.Exception` هستند
 - می‌توانند در وضعیت پرتاب (`throw`) یا در وضعیت دریافت (`catch`) باشند
 - متدها و ویژگی‌های کم و مختصری دارند(البته مثل همه کلاس‌ها می‌توانند سازنده، ویژگی و متدهای متنوعی داشته باشند)
 - معمولاً یک سازنده بدون پارامتر و یک سازنده با پارامتر رشته(دریافت پیغام خطا) دارند.

```
public class IOException extends Exception {
    public IOException() {
        super();
    }
    public IOException(String message) {
        super(message);
    }
    ...
}
```

مثال از نحوه استفاده از کلاس `IOException` :

```
if(...)
    throw new IOException();
if(...)
    throw new IOException("Internal state failure");
```

مثال:

```
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!");
}
```

مثال:

```
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) {...}

```

- در هنگام ایجاد یک کلاس Exception جدید در صورت تمایل می‌توانید نوع آن را Checked Exception یا Unchecked Exception قرار دهید.
 - Checked Exception ← کلاس جدید را فرزند Exception قرار دهید
 - Unchecked Exception ← کلاس جدید را فرزند RuntimeException قرار دهید
- اگر نمی‌خواهید کامپایلر آن را چک کند، آن را چک‌نشده تعریف کنید
- برای این کار، کلاس جدید را فرزند RuntimeException قرار دهید

Exception Handling Framework (۹.۲)

- مکانیزمی برای برنامه‌نویسی بهتر در تشخیص و گزارش و مدیریت استثناءها
- سبب می‌شود تا بدنه اصلی برنامه و بلوک‌های مدیریت خطاها تفکیک شود
- مدیریت خطا در بخشی از کد ممکن می‌شود که این کار امکان‌پذیر است و نه قسمتی که خطا رخ می‌دهد.
 - ممکن است خطا در یک بخشی از برنامه رخ دهد ولی عکس‌العمل مناسب در جایی دیگر از برنامه صورت گیرد. مثلاً در یکی از متدهایی که آن را فراخوانی نموده‌اند
- امکان گروه‌بندی و دسته‌بندی نمودن خطاها و پیاده‌سازی خاص پیرامون نوع خطا خاص صورت گیرد.
- هر «شیء‌استثنا»، نوعی دارد، مثلاً نوع «خطا هنگام خواندن فایل» و «خطای تقسیم بر صفر» متفاوت است. بنابراین می‌توان خطاها را برحسب نوع دسته‌بندی نمود.

```

try {
    Commands;
} catch (ArithmeticException e1) {
    Handler;
} catch (ArrayIndexOutOfBoundsException e2) {
    Handler;
} catch (IOException e3){
    Handler;
} catch (SQLException e4){
    Handler;
} catch (ClassCastException | NullPointerException e5){
    Handler;
} catch (Exception e){
    Handler;
}

```

- اگر چندین catch تنظیم شده باشد:
 - اگر در catch اول یک نوع استثنا تنظیم شود، آنگاه نمی‌توانیم در catch دوم یک «زیرکلاس» از آن نوع استثنا دریافت کنیم.
 - یعنی بلوک اول باید جزئی‌تر از بلوک‌های بعدی باشند وگرنه استثنا توسط آن بلوک دریافت و هیچ‌گاه به بلوک دوم نمی‌رسد
 - در چنین شرایطی ارور Unreachable catch block به شما اجازه نخواهد داد.
 - در مثال زیر این موضوع بیان شده است:

```
try {
    int a = Integer.parseInt(args[0]);
    int b = Integer.parseInt(args[1]);
    System.out.println(a/b);
} catch (Exception ex) {
    //..
} catch (ArrayIndexOutOfBoundsException e) { //X
    //..
}
// در مثال بالا اگر ترتیب جایگزین شود مشکلی بوجود نخواهد آمد
```

- اگر متدی در زیرکلاس خود override می‌شود، اجازه ندارد دایره Exception ها(که به کمک throws پرتاب می‌کند) را گسترده‌تر نماید. یعنی زیر کلاس حق ندارد استثناهای بیشتری از ابرکلاس برای پرتاب تعریف نماید
- مثال ۱:

```
class Parent{
    void f(){} // بدون مدیریت استثنا
}
class Child extends Parent{
    void f()throws Exception{} //X
}
```

مثال ۲:

```
class Parent{
    void f() throws ArithmeticException{} // مدیریت تنهای یک نوع استثنا
}
class Child extends Parent{
    void f() throws ArithmeticException, IOException{} //X
}
```

مثال ۳:

```
class Parent{
    void f()throws ArithmeticException{} // استثنا از نوع خاص تر
}
class Child extends Parent{
    void f() throws Exception{} //X
}
```

مثال ۴:

```

class Parent{
    void f() throws Exception{
    }
class Child extends Parent{
    void f() throws ArithmeticException{
    }
}
// بدون ارور ✓

```

Try(۹.۲.۱)

- قطعه کدی که ممکن است سبب بروز خطا شود داخل این بلوک قرار داده می‌شود.
- اگر خطایی در قطعه Try رخ دهد، برنامه همان قسمت متوقف شده و دنبال یک «Exception Handler» می‌گردد و در صورت یافتن آن ، خطا مستقیماً به قسمت Catch مربوط به Handler خود هدایت می‌شود.
- هنگام رخداد یک Exception ، یک «شیء-استثنا» یا «Exception Object» ایجاد می‌شود

امکان try-with-resources (۹.۲.۱.۱)

دو قطعه کد زیر یکسان هستند و این امکان برای ریسورس‌ها نظیر فایل‌ها بوجود آمده است که خودکار یک ریسورس close گردد قبل از جاوا ۷

```

BufferedReader br = new BufferedReader(new FileReader(path));
try {
    return br.readLine();
} finally {
    if (br != null) br.close();
}

```

بعد از جاوا ۷

```

try (BufferedReader br =
    new BufferedReader(new FileReader(path))) {

    return br.readLine();
}

```

Exception Handler یا Catch (۹.۲.۲)

- مجموعه اقدام‌های جانبی در صورت بروز خطا نیز در این بلوک قرار داده می‌شود.

- «شیء‌استثنا» یا «Exception Object» که پرتاب «Throw» شده است، توسط Catch دریافت می‌شود.
- این قسمت از اطلاعات موجود در «شیء‌استثنا» یا «Exception Object» برای مدیریت بهتر استثناء استفاده می‌کند.
- اگر این بخش نباشد: «فشار پیش‌فرض جاوا» در مقابله با استثنا اجرا می‌شود. یعنی پیغام خطا در خروجی استاندارد چاپ و اجرای برنامه خاتمه خواهد یافت.
- در بسیاری از موارد باید بروز خطا را لاگ بزنییم و این کار در بلاک catch قابل انجام است-البته بهتر است از فناوری مخصوص لاگ مثل SLF4J استفاده نمایید. استفاده از System.out.println یا printStackTrace برای اینکار مناسب نیست
- پیام مناسب و گویا به عنوان message استفاده کنید:

```
throw new IOException(message);
```

- استثناء را نادیده نگیرید-مثلا کد فوق SQLException را خفه میکند(کار خوبی نیست)

```
try {
    db.save(entity);
} catch (SQLException ex) {}
```

throw(۹.۲.۳)

- عبارت زیر سبب پرتاب یک استثناء در یک تابع خواهد شد

```
throw new Exception("متن دلخواه");
```

- گاهی استثنا باید re-throw شود. یعنی catch شود، کارهایی انجام شود، و سپس دوباره throw شود

```
try {
    ...
} catch (IOException ex) {
    ...
    throw ex;
}
```

- گاهی هم خطای جدیدی در بلاک catch پرتاب می‌شود. یعنی هر کاری که ممکن است در catch انجام می‌دهیم و سپس خطای جدیدی ایجاد و پرتاب می‌کنیم

```
try {
    ...
} catch (IOException e) {
    ...
    throw new ReportDataException(e);
}
```

throws(۹.۲.۴)

- هرگاه در انتهای یک متد عبارت **throw Exception** مشاهده شد یعنی:
 - احتمال بروز خطا و پرتاب استثناء در این متد وجود دارد
 - این تابع هیچ عملی درخصوص Exception Handler نخواهد کرد و فقط استثناء را به تابع بالایی پرتاب می‌کند
- مثال:

```
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;
}
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); // فرض شود محتوی متغیر، معتبر نباشد
        System.out.println(year);
    }
    catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```

مثال:

```
public class Test1{
public static void main(String[] args){
    try{
        Scanner scanner = new Scanner(System.in);
        int first = Scanner.nextInt();
        int second = Scanner.nextInt();
        int div = division(first,second);
        System.out.println(div);
    }catch(Exception e) {
        System.out.println(e.getMessage());
    }
}
private static int division(int first, int second) throws Exception {
    if(second == 0)
        throw new Exception("مخرج صفر هست");
    return first/second;
}
}
```


finally (۹.۲.۵)

```
Try{
    Commands;
} catch(ExceptionType e) {
    Handler;
} finally {
    Command;
}
```

- بخشی که در finally می‌آید، تحت هر شرایطی در انتهای اجرای try-catch حتماً اجرا می‌شود. شرایطی از قبیل:
 - تمام طبیعی اجرای بلاک try بدون پرتاب خطا
 - خروج اجباری از بلاک try (مثلاً با break, return یا continue)
 - خطایی در try پرتاب شود و در catch دریافت شود
 - خطایی در try پرتاب شود و در هیچ یک از بلاک‌های catch، دریافت نشود
 - موارد دیگر، تحت هر شرایطی...
- اگر خطا پرتاب شود یا نشود، در انتهای کار اجرای بخش finally تضمین می‌شود.
- معمولاً برای آزادسازی منابع گرفته شده در Try استفاده می‌شود.
 - همانند بستن فایل یا اتمام اتصال به دیتابیس البته هر منبعی بجز حافظه که این حافظه توسط ذباله روب بصورت خودکار آزاد می‌شود.

۹.۳ کلاس Throwable

- در واقع هر آنچه درباره Exception گفته شده، درباره Throwable نیز صادق است مثلاً هر شیء از جنس Throwable قابل پرتاب (throw) یا دریافت (catch) است
- دو نوع Throwable اصلی وجود دارد
 - **Exception** : اکثر کلاس‌های استثنا که با آنها سروکار داریم
 - **Error (خطا)** : معمولاً تلاش نمی‌کنیم که آنها را در برنامه catch کنیم
 - حتی اگر آن را catch کنیم، کار مهمی در قبال این خطاها نمی‌توانیم انجام دهیم
 - مانند: OutOfMemoryError

۹.۴ Unchecked Exception استثناء چک نشده

- برای برخی استثناها ذکر کلیدواژه throws واجب نیست.
- کامپایلر در زمان کامپایل احتمال دریافت (Catch) یا اعلان پرتاب (throw) توسط متد را اجبار نمی‌کند
- کلاس‌های این نوع، عبارتند از :
 - کلاس Error
 - کلاس Runtime Exception
 - زیر کلاس‌های Error و Runtime Exception

به قطعه کد زیر توجه نمایید:

```
private static void function(String arg) {
    System.out.println(1 / arg.length()); // ممکن است صفر باشد
}
public static void main(String[] args) {
    function("");
}
```

بطور مثال در قطعه کد بالا متد function ممکن است Arithmetic Exception پرتاب نماید ولی کلیدواژه throws تصریح نشده است. با التزام این کار برنامه‌ها پر از throws های نامهم می‌شدند.

- کلاس‌های ArrayIndexOutOfBoundsException یا ArithmeticException از این قبیل هستند.

۹.۵ checked Exception استثناء چک شده

- کامپایلر جاوا بررسی می‌کند که اگر در یک متد این نوع Exception ها پرتاب می‌شود پس متد باید آن را Catch کند یا احتمال پرتاب شدن را به کمک throws اعلام نماید.
به قطعه کد زیر توجه نمایید:

```
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(); // اینجا سبب ارور می‌شود
                                // Syntax Error:Unhandled Exception Type IOException
}
```

فرم درست این قطعه کد بصورت زیر است

```
void example(int x) throws IOException {
    ...
    if(x==3)
        throw new IOException();
}
```

۹.۶ استثناء Illegal Exception

- از نوع Runtime است
- هنگامی throw می‌شود که هم‌اکنون وضعیت مطلوب است ولی ممکن است در آینده بنا بر دلایل زیادی که هنوز آن را هندل نکردیم خطا رخ دهد.
- این استثناء را ارسال می‌کنیم تا برنامه‌نویس را نسبت به ملاحظه‌خیز بودن قطعه کد آگاه کنیم
- اصطلاح خودمانی: ممکن هست چیزهایی درست نباشد و در حال ریسک است. پس این استثناء را پرتاب می‌کنیم.

• انواع زیادی از این نوع وجود دارد که برخی در زیر آورده شده است:

- illegalStateException
- illegalSelectorException
- illegalAccessException
- illegalArgumentException
- illegalCallerException
- illegalFormatException
- illegalThreadException
- illegalAnnotationException
- illegalReceiveException
- illegalFormatWidthException
- illegalClassFormatException

```
...
@Override
public void play(String mediaPath) throws MediaPlayerException {
    try {
        ...
    } catch(Exception e) {
        throw new IllegalStateException(e);
    }
}
...
```

فصل ۱۰ مفاهیم تکمیلی

Java Documents (۱۰.۱)

- کامنتی که با `/**` شروع شود تحت عنوان جاواداک در نظر گرفته می‌شود
- این جاواداک قبل از هر چیزی قرار گرفته باشد به توصیف همان جزء می‌پردازد(متد یا کلاس یا سازنده یا ویژگی یا هر چیز دیگر)
- دستور javadocs این توصیفات را به یک مستند HTML تبدیل می‌کند

```
/**
 * This class represents a human. Objects of this class are immutable...
 * @author Sadegh Aliakbary
```

```

* @see java.lang.String
*/
public class Person {
    /** national ID (SSN) */
    private String ID;
    private String name; //No javadoc

    /**
     * The only constructor of the class
     * @param id The social security number (national ID)
     * @param name The full-name, including first-name and last-name
     */
    public Person(String id, String name) {
        ID = id;
        this.name = name;
    }
    /**
     * 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){...}
}

```

Refactoring (۱۰.۲)

برخی تکنیک‌های کدنویسی برای منظم‌نمودن و سادگی و تمیزسازی قطعات کد نوشته شده

- قواعد کلی برای کد
 - استفاده از نام‌های مناسب در کد
 - کاهش پیچیدگی در هر سطح و در هر قسمت که میسر است
- «منظم‌سازی» متد
 - استخراج متد برای هر هدف
 - برپایه کشف رابطه بین اجزای کد
 - هدف مستقل و منسجم
 - یک متد خوب کاری مستقل و منسجم انجام می‌دهد.
 - حداقل امکان هر متد هدفی واحد انجام دهد (استقلال)
 - پرهیز از پیچیدگی وابستگی بین متدها
 - پرهیز از تکراری نویسی و تبدیل به متد
 - دو متد نباید یک کار را انجام دهند
 - هیچ‌گاه نباید یک قطعه کد دوبار استفاده شود و حتماً باید به متد تبدیل شود
 - ساده‌سازی
 - عدم استفاده از متدهای طولانی و ساده‌سازی کارها
 - عدم استفاده از تعداد متغیر زیاد در متدها
 - کاهش تعداد آرگومان ورودی به متد و ساده‌سازی متغیرهای ورودی به متد بعنوان آرگومان

- دیتای مورد نیاز برای استفاده در هر متد توسط خود متد تولید شود و در متغیر محلی ریخته شود. (حداامکان) و نه اینکه بعنوان آرگومان ورودی پاس داده شود
- استفاده از تکنیک فراخوانی متد به جای پاس شدن مقدار پارامتر همانند استفاده از Getterها
- استفاده از متدهای inline
- «منظم‌سازی» کلاس
 - استخراج کلاس بر پایه کشف روابط بین اجزای کد
 - عدم تعریف کلاس بزرگ و حداامکان استخراج «زیرکلاس» یا «ابركلاس»
 - اگر یک متد از یک کلاسی به کررات استفاده می‌شود پس شاید بهتر باشد به نحوی به کلاس خویش منتقل شود
 - اگر در تغییر یک قطعه کد، جاهای زیادی با پراکندگی زیادی از کلاس‌ها در حال تغییر است، این «بوی‌بد» تلقی می‌شود.
 - کلاس‌های کوچک و مختصر که معمولاً از آن‌ها استفاده نمی‌شود باید با تدبیر مناسب به کلاس‌های دیگر بپیوندند و ترکیب شوند
- «منظم‌سازی» متغیر
 - استخراج متغیر برای اهداف خاص
 - تکنیک: تبدیل مجموعه پارامترها به یک شیء (Introduce Parameter Object)

تحقیق برای بهروز: در اینترنت عنوان «بوی‌بد» یا «Bad Smell» را جستجو نمایید و تکنیک‌های کدنویسی بهتر را استخراج نمایید
<http://refactoring.com/catalog>

فصل ۱۱) تست نرم افزار (JUnit)

پس از اتمام یک برنامه توسط برنامه نویس، پیش از تحویل محصول به مشتری، باید ابعاد کنترل و تضمین کیفیت محصول بررسی و آزمایش شود. تا از کیفیت آن مطمئن شویم. نرم افزاری که آزمایش نشده، هنوز کامل نیست. انواع آزمایش‌ها، کیفیت نرم افزار را از دیدگاههای مختلف می‌آزمایند.

تست نرم افزار : فرایندی برای آزمایش ویژگی‌های موردنظر نرم افزار و بررسی عمل کرد برنامه نوشته شده

۱۱.۱ مفاهیم

- **Business Code** : متن اصلی کد برنامه‌ها
- **Test Code** : کدهایی که برای آزمایش برنامه‌ها نوشته شوند
- **سطح آزمون** : در چه حدی و ریز شدن نرم افزار تست می‌شود
 - آزمون واحد: تست یک مولفه یا بخش کوچک
 - آزمون یکپارچگی: تست چند بخش که با هم یکپارچه شده و اقدام واحدی را انجام می‌دهند
 - آزمون سیستم: تست تمام نرم افزار
- **نوع آزمون**: آزمون عملکرد، آزمون ویژگی‌های کیفی
- **روش آزمون**:
 - **white box**: در تست جزئیات پیاده‌سازی را مد نظر قرار دهیم
 - **black box**: به نحوه پیاده‌سازی کاری نداریم و فقط ورودی و خروجی را تست می‌کنیم و توقع داریم این سیستم همانند یک جعبه سیاه کار نماید
- **شکل آزمون**: آزمون خودکار، آزمون دستی
- **نقش آزمون‌گر**: برنامه‌نویس، تیم تست، کاربر یا ...
- آزمون واحد یا JUnit از وظایف برنامه‌نویس است.
- تنها آزمون واحد کافی نیست و انواع دیگر آزمون‌ها برای تضمین کیفیت نرم افزار لازم است.
- **چارچوب‌های xUnit**: چارچوب‌هایی برای آزمون واحد در زبان‌های مختلف ایجاد شده است. این ابزارها، اجرا و تشخیص نتیجه آزمون را خودکار می‌کنند. مجموعه این ابزارها xUnit نامیده می‌شوند. مثل CppUnit در زبان سی پلاس پلاس و JUnit در زبان جاوا

معايب تست سنتی:

- تست‌های نوشته شده دور ریخته می‌شود و استفاده مجدد ممکن نیست (software reuse) برای test code
- در هر لحظه یک تست انجام می‌شود
- برنامه نویس باید به صورت دستی تست‌ها را اجرا کند
- برنامه نویس باید شخصاً از صحت آنها مطمئن شود و تشخیص موفقیت آمیز بودن تستها خودکار نیست

ویژگی‌های آزمون واحد (Unit Test)

- اجرای خودکار
- تشخیص خودکار موفقیت تست

- قابل تکرار و استفاده مجدد
- یک آزمون white box است (نه black box)
- یک آزمون واحد است (نه آزمون سیستم، و نه حتی آزمون چند بخش یکپارچه شده)
- معمولاً هدف آزمون عملکردی (نه آزمون ویژگی‌های کیفی)
- آزمون خودکار انجام می‌شود (نه دستی)
- تولید و انجام آن توسط خود برنامه‌نویس است (نه توسط تیم تست یا کاربر)
- بهبود کیفیت برنامه‌ها
- کاهش اشکالات (bugs) و وقت کمتری به خاطر debugging هدر میدهد
- به تأخیر نیافتادن زمان کشف اشکالات برنامه
- بهبود ساختار و طراحی برنامه
- افزایش اطمینان به اجزای برنامه و سطوح پایین کد
- مستند گویا، زنده و قابل اجرا برای برنامه

۱۱.۲ نکات Junit

- هر متد تست یک نمونه‌آزمون (test-case) خوانده د
- نمونه‌آزمون‌ها متد main ندارند و با کمک Test Runner اجرا می‌شوند
 - غالباً از طریق محیط‌های برنامه‌نویسی (مثل Eclipse) یا Maven اجرا می‌شوند
 - یادآوری: موفقیت‌آمیز بودن آزمون باید به صورت خودکار بررسی شود
 - نه به صورت دستی: **سندروم println** ☹️
- نتیجه آزمون در JUnit با کمک assertion بررسی می‌شود. مثال: متد assertEquals
- هر نمونه‌آزمون شامل این بخش‌هاست:
 - **test data**: یک ورودی برای متد موردآزمون تعیین می‌شود
 - **expected result**: خروجی و رفتار موردنظر برای این ورودی مشخص می‌شود
 - **invocation**: متد موردآزمون با این ورودی فراخوانی می‌شود
 - **assertion**: خروجی و رفتار متد با خروجی موردنظر تطبیق داده می‌شود
- اگر اجرای تست موفقیت‌آمیز باشد: تست **pass** شده است
- اگر اجرای تست موفقیت‌آمیز نباشد: تست **fail** شده است
-

۱۱.۳ Junit Framework

- مثلاً در Eclipse باید Library فریم‌ورک JUnit را به سیستم اضافه نماییم
- قبل از متد از annotation تحت عنوان Test استفاده می‌کنیم

- هر متدی که قبل از آن انوتیشن test آمده باشد بعنوان یک تست در نرم افزار تلقی می شود. و نرم افزار خروجی تک تک این متدها را نشان می دهد که آیا صحیح انجام شده است یاخیر.
- معمولاً یک کلاس جدید ایجاد می کنند به نام تست و تمام تست های مربوطه به یک کلاس را در آن قرار می دهند.
- مجموعه متدهای **JUnit Assertions**

```

assertNull(x)           // بررسی نال بودن مقدار
assertNotNull(x)        // بررسی نال نبودن مقدار
assertTrue(boolean x)   // بررسی صحیح بودن مقدار
assertFalse(boolean x)  // بررسی ناصحیح بودن مقدار
assertEquals(x, y)      // بررسی مساوی بودن که توسط تابع صورت می گیرد x.equals(y)
assertSame(x, y)        // بررسی برابر بودن از جنس مساوی بودن و از اپراتور مساوی استفاده می شود x ==y
assertNotSame           // بررسی نابرابر بودن مقدار
fail()                  // اگر اجزای تست به آن برسد آنگاه آزمون شکست می خورد
                        // و تنها در خطی از این تابع استفاده می کنیم که نمی خواهیم اجرای برنامه به هیچ وجه به اون خط برسد
و غیره

```

- **متد Before**
 - قبل از هر متد تست اجرا می شود
 - مسؤؤل کارهایی که قبل از اجرای هر تست لازم است. مثل اتصال به پایگاه داده، مقداردهی به فیلدها و غیره
- **متد BeforeClass** : یک بار قبل از همه تست ها
- **متد After**
 - اجرا بعد از هر متد تست. مثل بستن اتصال به پایگاه داده، بستن فایل و غیره
- **متد AfterClass** : یکبار بعد از همه تست ها

۱۱.۴ قطعه کدها و مثال ها

مثال ۱: یک آرایه داریم و می خواهیم مرتب سازی شود و در انتها چک شود که آیا مرتب سازی بدرستی صورت گرفت است یا خیر

```

@Test // استفاده از انوتیشن
public void testSort(){
    int[] array = {3,2,1,5,6,4}; // آرایه ای که قصد مرتب سازی داریم
    sort(array); // اعمال عملیات مرتب سازی
    int[] sortArray={1,2,3,4,5,6}; // تعریف یک آرایه که انتظار داریم آرایه قبلی همانند این آرایه بشود
    assertEquals(array, sortArray); // بررسی تساوی آرایه اصلی و آرایه ای که انتظار داریم باشد
}

```

در مثال بالا اگر تساوی دو آرایه برقرار نبود پس متد assertEquals خروجی عدم تساوی گزارش می دهد

مثال ۲: بخش **Business Code**

```

public class MyMath {

```



```

private static final MyMath instance = new MyMath();

public static MyMath getInstance(){
    return instance;
}
public int division(int a, int b){
    if( b == 0 ) throw new ArithmeticException("Division by zero");
    return a//b;
}
public int multiply(int a, int b)
    return a*b;
}
}

```

مثال ۲: بخش Test Code

```

@Test
public void testDivision(){
    MyMath math = MyMath.getInstance();
    assertNotNull(math);
    assertEquals(math.division(4,2),2);
    assertEquals(math.division(5,3),1);
    try{
        math.division(5,0);
        // the test should never reach this line:
        fail();
    }catch(Exception e){
        //The Exception is OK
        assertTrue(e instanceof ArithmeticException);
    }
}

```

مثال ۲:

```

public class TestDB {
    private static Database db;
    private static List<String> storage;

    @BeforeClass // یکبار برای همه تست‌ها باید این متد انجام شود
    public static void startup(){
        db = new Database();
        storage = new ArrayList<String>();
        db.setStorage(storage);
    }

    @Afterclass // یکبار برای همه تست‌ها باید این متد انجام شود
    public static void shutdown(){
        db = null;
        storage = null;
    }

    @Before // قبل از هر تست
    public void setup(){
        db.open();
    }
}

```

```
@After // بعد از هر تست
public void teardown(){
    db.close();
}
}
```

مثال ۲: بخش تست

```
@Test
public void testNormal(){
    String name = "Ali Alavi"
    db.insert(name);
    assertTrue(storage.contain(name));
    db.delete(name);
    assertFalse(storage.contain(name));
}
@Test
public void TestNull(){
    db.insert(null);
    assertTrue(storage.contain(null));
    assertEquals(1,storage.size());
}
```