



دانشگاه صنعتی امیر کبیر  
( پلی تکنیک تهران )

**دانشکده ریاضی و علوم کامپیوتر**

**برنامه سازی پیشرفته و کارگاه**

**مقدمه ای بر شی گرای**

استاد درس

**دکتر مهدی قطعی**

استاد دوم

**بهنام یوسفی مهر**

نگارش

**شهاب گریوانی، یونس کاظمی و آرمان حسینی**

**بهار ۱۴۰۳**

## فهرست

4	مقدمه
5	تایپ‌ها در جاوا
7	کلاس‌ها
10	تعریف کلاس
10	ایجاد Object
11	Constructorها
13	Default Constructor
13	کلاس‌ها با چند کانستراکتور
15	fieldها
16	فیلدهای static
18	methodها
19	متغیرهای محلی (local variables)
20	مقداردهی اولیه به متغیرهای محلی
22	Shadowing
23	this
24	متدهای استاتیک
27	Method overloading
30	Reference typeها
30	تفاوت primitive type و reference typeها
34	کلاس‌های wrapper برای primitive typeها
36	Garbage Collection
36	آشنایی با Memory Leak
37	Garbage Collection: یه راه‌حل خوب
38	Garbage Collector حواسش هست!

---

40	یک نکته در مورد کلاس‌ها
41	Packages
41	نیاز به منظم کردن فایل‌ها
41	پکیج چیست؟
42	چطور یک پکیج ایجاد کنیم؟
45	نام‌گذاری متداول پکیج‌ها (Naming Conventions)
46	کلیدواژه Import
46	یک مثال عملی
47	در اهمیت پکیج‌ها
47	import کردن کل پکیج
48	دو مثال دیگر: مرور خاطرات
48	java.util.Scanner
49	javax.swing.JFrame
50	چند نکته در مورد پکیج‌ها
50	هر کلاس دقیقا به یک پکیج تعلق دارد
51	استفاده از یک کلاس بدون import کردن اون

## مقدمه

توی این داکيومنت و چند داکيومنت بعدی، قراره با شی‌گرایی آشنا بشیم. شی‌گرایی یا object-oriented programming (به اختصار OOP)، یک طرز فکره برای تقسیم یک برنامه بزرگ، به واحدهای کوچیک‌تری به اسم class. خیلی از چیزهایی که قبلاً بهتون گفته بودیم «این‌ها رو در آینده بهتون توضیح می‌دیم» رو قراره این‌جا بهتون توضیح بدیم!

در طول خوندن این داکيومنت، اگر سوالی داشتین حتماً از تدریس‌یارهاتون یا ChatGPT بپرسین، چون که این داکيومنت یکی از مهم‌ترین داکيومنت‌های شماست که توی هفته‌های آینده بر پایه‌اون به بررسی مفاهیم دیگه‌ای مثل ارث‌بری، کپسوله‌سازی و مباحث مشابه می‌پردازیم.

## تایپ‌ها در جاوا

شما تا حالا با خیلی از type‌ها کار کردین و با اون‌ها آشنا شدین؛ مثل `int`، `float`، `String`، `JFrame`، `ArrayList` و امثال اون‌ها. هر برنامه‌نویسی توی جاوا، می‌تونه به راحتی تایپ‌های جدید برای خودش بسازه و به همین خاطر، احتمالا تا به حال میلیون‌ها تایپ توی جاوا نوشته شده. خود شما هم توی این داک قراره یاد بگیرین که چطور می‌تونین تایپ‌های جدید بسازین.

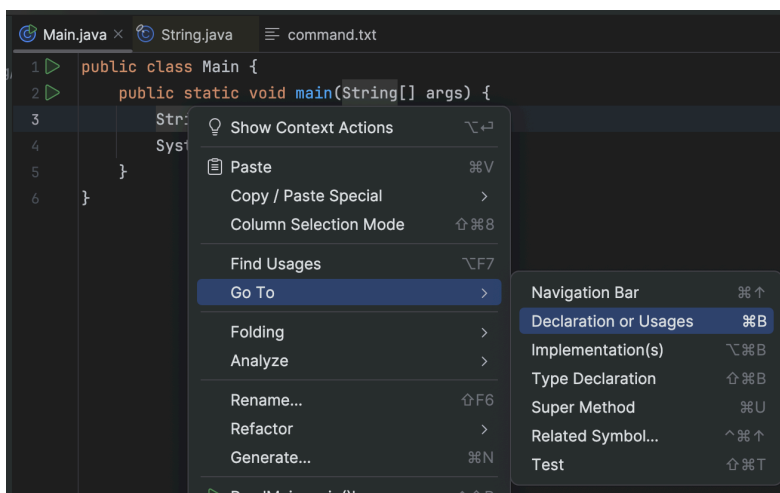
همون‌طور که توی جلسهٔ صفر دیدین، تایپ‌های `int`، `float`، `double`، `long` و `char`، به اسم «تایپ‌های اولیه» یا `primitive types` شناخته می‌شن. فهرست کامل اون‌ها به شکل زیره:

Type	Definition	Approximate range or precision
<code>boolean</code>	Logical value	true or false
<code>char</code>	16-bit, Unicode character	64K characters
<code>byte</code>	8-bit, signed integer	-128 to 127
<code>short</code>	16-bit, signed integer	-32,768 to 32,767
<code>int</code>	32-bit, signed integer	-2.1e9 to 2.1e9
<code>long</code>	64-bit, signed integer	-9.2e18 to 9.2e18
<code>float</code>	32-bit, IEEE 754, floating-point value	6-7 significant decimal places
<code>double</code>	64-bit, IEEE 754	15 significant decimal places

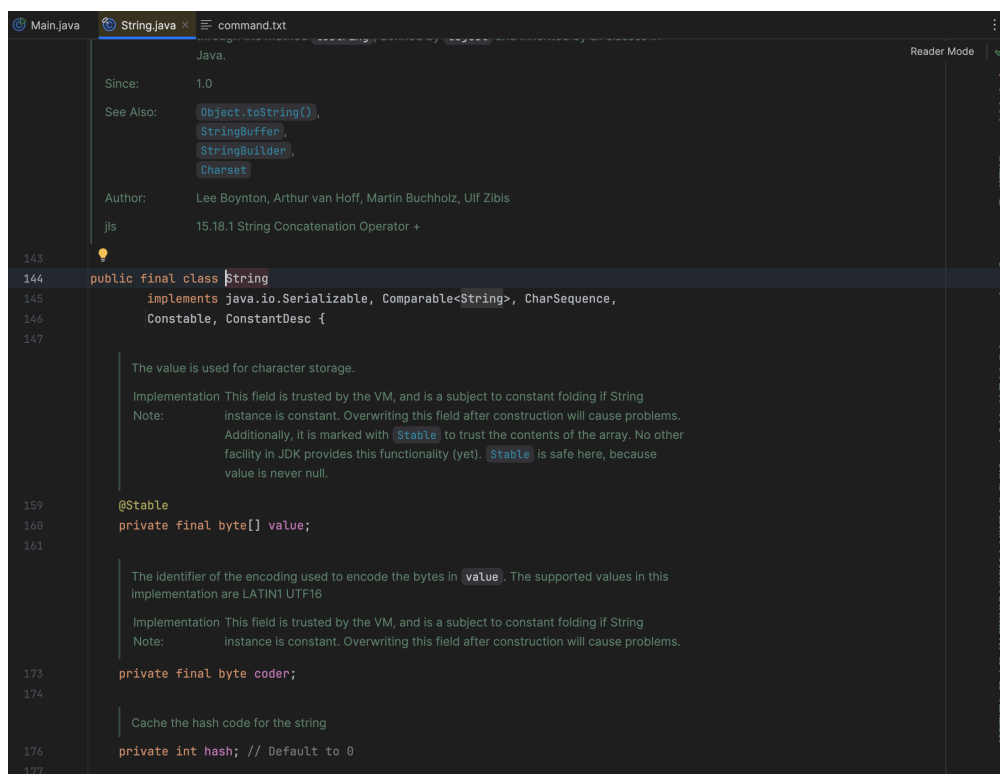
به این تایپ‌ها، مثل تیکه‌های کوچیک لگو نگاه کنید که تایپ‌های پیچیده‌تر مثل `String`، `JFrame`، `ArrayList` و هر تایپ دیگه‌ای رو تشکیل می‌دن. به این تایپ‌های پیچیده‌تر، `reference type` می‌گن. بیاید یکی از اون‌ها رو ببینیم. کد زیر رو توی IntelliJ کپی کنید:

```
public class Main {
    public static void main(String[] args) {
        String str = "Hello World";
        System.out.println(str);
    }
}
```

موستون رو روی `String` ببرین و کلیک راست کنید. از صفحه‌ای که باز می‌شه، منوی `Go To` و بعد از اون، `Declaration or Usages` رو انتخاب کنید:



صفحه جدیدی براتون باز می‌شه که شامل پیاده‌سازی تایپ Stringئه:



شما احتمالاً هنوز کامل این‌کد رو نمی‌فهمید، ولی توی این‌کد، نشون داده می‌شه که تایپ String از کنار هم گذاشتن چه تایپ‌های دیگه‌ای درست شده. مثلاً، توی خط ۱۶۰ همین تصویر می‌بینید که String، توی خودش یک آرایه از byte‌ها داره که اسمش valueئه. پایین‌تر می‌بینید که توی خودش، یه int به اسم hash داره. اگر دوست دارید، یه مقدار توی این‌کد بالا و پایین بشین، خیلی از تیکه‌های اون براتون غریبه‌ن و البته خیلی از بخش‌هاش هم براتون آشنا به نظر میاد.

## کلاس‌ها

کلاس‌ها، به شما اجازه می‌دن تا type‌های جدید خودتون رو درست کنید. بهترین راه یادگیری اونا، اینه که یه خورده باهاشون کار کنید، پس بیاید تا با هم یه کلاس جدید به اسم Student درست کنیم. کد زیر رو توی IntelliJ بنویسید:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}

class Student {
    public int age;
    public String name;
    public String studentID;
}
```

توی این کد، ما یه class جدید، به اسم Student تعریف کردیم. این کلاس، از چهار متغیر تشکیل شده:

- age: متغیری از جنس int که سن دانشجو رو نشون می‌ده.
- grade: متغیری از جنس double که معدل دانشجو رو نشون می‌ده.
- name: متغیر Stringی که اسم دانشجو رو نشون می‌ده.
- studentID: متغیر Stringی که شماره دانشجویی شخص رو نشون می‌ده.

به این متغیرها که یک کلاس رو تشکیل می‌دن، field‌های اون کلاس می‌گن. نگران کلیدواژه public نباشید، دو جلسه دیگه بهتون مفهوم اون رو توضیح می‌دیم ولی برای الآن، لازمه که اون رو قبل از همه field‌های کلاس‌هاتون بذارین. حالا، با کلیدواژه new، یه دانشجوی جدید به اسم قلی بسازید و فیلدهای اون رو مقداردهی کنید:

```
public static void main(String[] args) {
    Student gholi = new Student();

    gholi.age = 20;
    gholi.name = "Gholi";
    gholi.studentID = "40413099";
}
```

حالا، مشخصات این دانشجو رو چاپ کنید:

```
System.out.println("New student:");
System.out.println("\t+ Name: " + gholi.name);
System.out.println("\t+ StudentID: " + gholi.studentID);
System.out.println("\t+ Age: " + gholi.age);
```

کدتون رو اجرا کنید. خروجی زیر رو می‌بینید:

```
New student:
+ Name: Gholi
+ StudentID: 40413099
+ Age: 20
```

می‌بینید که مشخصات قلی، به درستی روی صفحه چاپ می‌شه. تبریک! شما اولین type خودتون رو ساختین و یه متغیر هم از جنس اون درست کردین. بیاین به این type جدید، چیزهای بیشتری اضافه کنیم. به Student، فیلدی به اسم grades از جنس ArrayList<Double> رو اضافه کنید. این فیلد، نمرات دانشجو توی درس‌های مختلف رو نشون می‌ده:

```
public ArrayList<Double> grades;
```

حالا، وقتی دارین فیلدهای مختلف قلی رو توی main مقداردهی می‌کنید، این آرایه هم با یه آرایه خالی مقداردهی کنید:

```
gholi.age = 20;
gholi.name = "Gholi";
gholi.studentID = "40413099";
gholi.grades = new ArrayList<Double>();
```

بعد از این کار، چندتا نمره رندوم به قلی بدین:

```
gholi.grades.add(20.0);
gholi.grades.add(17.0);
gholi.grades.add(18.0);
gholi.grades.add(0.0);
```

حالا، اون‌جا که دارین مشخصات قلی رو چاپ می‌کنید، این نمرات هم چاپ کنید:

```
System.out.println("New student:");
System.out.println("\t+ Name: " + gholi.name);
System.out.println("\t+ StudentID: " + gholi.studentID);
System.out.println("\t+ Age: " + gholi.age);
System.out.print("\t+ Grades: ");
for (var grade: gholi.grades) {
    System.out.print(grade + ", ");
}
```

کدتون رو اجرا کنید. خروجی کدتون باید به این شکل باشه:

```
New student:
+ Name: Gholi
```



```
+ StudentID: 40413099
+ Age: 20
+ Grades: 20.0, 17.0, 18.0, 0.0,
```

می‌خوایم به Student، قابلیت محاسبهٔ معدل هم بدیم. برای این کار، یه متد جدید به اسم `getAverageGrade` توی Student تعریف می‌کنیم:

```
class Student {
    public int age;
    public String name;
    public String studentID;
    public ArrayList<Double> grades;

    public double getAverageGrade() {
        if (grades.size() == 0) {
            return 0;
        }

        double gradeSum = 0;
        for (double grade: grades) {
            gradeSum += grade;
        }

        return gradeSum / grades.size();
    }
}
```

اون‌جای کدتون که دارین مشخصات `gholi` رو چاپ می‌کنید، خطوط زیر رو اضافه کنید:

```
System.out.println();
System.out.println("\t+ Average Grade: " + gholi.getAverageGrade());
```

کدتون رو دوباره اجرا کنید. خروجی‌ای مثل زیر می‌بینید:

```
New student:
+ Name: Gholi
+ StudentID: 40413099
+ Age: 20
+ Grades: 20.0, 17.0, 18.0, 0.0,
+ Average Grade: 13.75
```

یه لحظه به خود تابع `main` توجه کنید. می‌بینید که خود اون هم توی کلاسی به اسم `Main`! شما توی تمام این مدت، داشتین کلاس `Main` رو تعریف می‌کردین بدون این که حتی خبر داشته باشید:

```
public class Main {
    public static void main(String[] args) {
        // Your code here
    }
}
```

حتی شما می‌تونید از این کلاس هم یک متغیر درست کنید:

```
public class Main {
    public static void main(String[] args) {
        Main a = new Main();
    }
}
```

البته که متغیر ساختن از جنس Main خیلی کار خوبی نیست! ولی جالبه که تا همین الان هم شما از classها استفاده می‌کردین، بدون این که بدونین (=)))

بیاین در ادامه، یک خورده رسمی‌تر و قدم به قدم‌تر کلاس‌ها رو بررسی کنیم.

## تعریف کلاس

تعریف کلاس، کار راحتی، فقط از کلیدواژه class استفاده کنید و اسم کلاستون رو بنویسید:

```
class Student {
    // Everything an student can do
}
```

هر چیزی که بین دو براکت میاد، متعلق به اون کلاسه. کلاس‌ها، به شما اجازه می‌دن که کدهای خیلی خیلی بزرگ رو، به تیکه‌های کوچیک‌تر تقسیم کنید و با این کار، برنامه‌های منظم‌تر و بهتری داشته باشین.

یادآوری می‌کنیم که type‌هایی که به وسیله classها تعریف می‌شه، همگی از reference type محسوب می‌شن.

## ایجاد Object

به متغیرهایی که از جنس یک reference type درست می‌شن، object یا instance می‌گن. توی کد زیر، قلی و ممد و سلطان همگی objectهایی از جنس Studentان:

```
public static void main(String[] args) {
    Student gholi;
    Student mamad;
    Student soltan;
}
```

تلاش کنید این کد رو اجرا کنید. می‌بینید که طبیعتاً، بدون هیچ مشکلی اجرا می‌شه. شما توی این کد سه object جدید تعریف کردید و هیچ‌جایی اون‌ها رو مقداردهی نکردین، و جایی هم ازشون استفاده نشده.

برای این که اون‌ها رو مقداردهی کنید، از کلیدواژه `new` استفاده کنید:

```
Student gholi = new Student();
Student mamad = new Student();
Student soltan = new Student();
```

حالا فیلدهای اون‌ها رو مقداردهی کنید:

```
gholi.age = 21;
gholi.name = "Gholi";
gholi.studentID = "40513089";
gholi.grades = new ArrayList<>();

mamad.age = 25;
mamad.name = "Mamad";
mamad.studentID = "40513090";
mamad.grades = new ArrayList<>();

soltan.age = 103;
soltan.name = "Soltan";
soltan.studentID = "40513091";
soltan.grades = new ArrayList<>();
```

اوه اوه، چقدر توی این کد، مجبور شدیم کد تکراری بزنیم! کدی که مربوط به بخش مقداردهی `field`های `object` سه بار تکرار شده و همون‌طور که توی داک کلین کد گفتیم، این یعنی خوبه این کدها رو به یه متد تبدیل کنیم. `Constructor`ها، متدهایی هستن که دقیقا برای همین کار، به کمک ما میان.

## Constructorها

به داخل کلاس `Student`، متد زیر رو اضافه کنید:

```
public Student(int newStudentAge, String newStudentName, String newStudentID)
{
    age = newStudentAge;
    name = newStudentName;
    studentID = newStudentID;

    grades = new ArrayList<>();
}
```

می‌بینید که این متد، یه مقدار ظاهر عجیب و غریبی داره! نه نوع خروجی‌ش مشخص شده، نه جایی از کلیدواژه `return` استفاده شده. به این متد، `Constructor` می‌گن و وظیفه اون، مقداردهی یه `object` جدید از کلاس `Student`ه. توی `constructor` بالا، ما برای ساخت یه `Student` جدید، سن، اسم و شماره دانشجویی‌ش رو ورودی گرفتیم و با استفاده از اون‌ها، فیلدهای `Student` رو برای `object` جدیدمون مقداردهی کردیم.

برای این که از این constructor استفاده کنید، کد توی main رو با کد زیر جایگزین کنید:

```
public static void main(String[] args) {
    Student gholi = new Student(21, "Gholi", "40513089");
    Student mamad = new Student(25, "Mamad", "40513090");
    Student soltan = new Student(103, "Soltan", "40513091");
}
```

می‌بینید که با استفاده از یه constructor خوب، چقدر کد main کوتاه‌تر و تمیزتر شد! حالا متد زیر رو به Student اضافه کنید:

```
public void printInfo() {
    System.out.println("Student info:");
    System.out.println("\t+ Name: " + name);
    System.out.println("\t+ StudentID: " + studentID);
    System.out.println("\t+ Age: " + age);
    System.out.print("\t+ Grades: ");
    for (var grade: grades) {
        System.out.print(grade + ", ");
    }
    System.out.println();
    System.out.println("\t+ Average Grade: " + getAverageGrade());
}
```

و توی main، این متد رو صدا بزنید تا اطلاعات ممد و قلی و سلطان چاپ بشه:

```
gholi.printInfo();
mamad.printInfo();
soltan.printInfo();
```

خروجی‌ای مثل زیر می‌بینید:

```
Student info:
+ Name: Gholi
+ StudentID: 40513089
+ Age: 21
+ Grades:
+ Average Grade: 0.0
Student info:
+ Name: Mamad
+ StudentID: 40513090
+ Age: 25
+ Grades:
+ Average Grade: 0.0
Student info:
+ Name: Soltan
+ StudentID: 40513091
+ Age: 103
+ Grades:
+ Average Grade: 0.0
```

constructorها کار کردن و کد ما هم، کوتاه و تمیزه!

## Default Constructor

تا قبل از این که ما برای کلاس Student کانستراکتور تعریف کرده باشیم، می‌تونستیم هنوز با استفاده از کلیدواژه new از اون object بسازیم:

```
Student gholi = new Student();
```

اما چطوری؟ ما که constructor ای برای Student نداشتیم! با نوشتن این کد، دقیقا چه constructor ای صدا زده می‌شه؟

وقتی که کلاسی constructor نداشته باشه، جاوا خودش یه constructor خالی براش می‌نویسه. این constructor، به شکل زیره:

```
public Student() {}
```

توی این کانستراکتور هیچ اتفاقی نمی‌افته و هیچ field ای مقداردهی نمی‌شه. صرفا اضافه تا شما بتونید به راحتی با استفاده از new Student()، آبجکتهای جدید از این کلاس بسازید. به محض این که شما اولین constructor کلاس Student رو بنویسید، دیگه نمی‌تونید از default constructor جاوا استفاده کنید. توی کد فعلی‌تون این خط رو بنویسید و کد رو اجرا کنید:

```
Student pedram = new Student();
```

همچنین خطایی می‌گیرید:



از اونجایی که کلاس student، الآن یک constructor برای خودش داره، جاوا برای شما کانستراکتور دیفالت رو درست نمی‌کنه.

## کلاس‌ها با چند کانستراکتور

شما می‌تونید برای رفع مشکلی که بالاتر به اون برخورد کردین، خودتون یه کانستراکتور خالی برای Student بنویسید:

---

```
public Student() {}
```

دقت کنید که لازم نیست کانستراکتور قبلی‌تون رو پاک کنید! هر کلاس، می‌تونه چندین constructor داشته باشه. شما می‌تونید این constructor رو در کنار کانستراکتور قبلی‌تون بنویسید و کدتون همچنان کار می‌کنه.

## fieldها

fieldها، یا متغیرهای یک کلاس، ویژگی‌های اون کلاس رو نشون می‌دن. شما تا الآن تعداد خوبی field برای کلاس student تعریف کردین و با اون‌ها آشنا شدین:

```
class Student {
    public int age;
    public String name;
    public String studentID;
    public ArrayList<Double> grades;

    // Other things happening in the Student class
}
```

شما توی یک کلاس، حتی می‌تونید fieldهایی از جنس همون کلاس تعریف کنید! مثلاً، می‌تونید فیلد friend رو برای Student، از جنس خود Student تعریف کنید:

```
class Student {
    // Other fields

    public Student friend;

    // Other things happening in the Student class
}
```

حالا، یه student جدید به اسم شهرام توی کدتون تعریف کنید:

```
var shahram = new Student();
```

برای الآن، بدون این که فیلدهای شهرام رو مقداردهی کنید، سعی کنید تا اطلاعات اون رو چاپ کنید:

```
System.out.println("Shahram: ");
System.out.println("\t+ Name: " + shahram.name);
System.out.println("\t+ Age: " + shahram.age);
System.out.println("\t+ StudentID: " + shahram.studentID);
```

کدتون رو اجرا کنید، خروجی زیر رو برای شهرام می‌بینید:

```
Shahram:
+ Name: null
+ Age: 0
+ StudentID: null
```

می‌بینید که علی‌رغم این که شما به شهرام اسم و سن و شماره دانشجویی ندادین، خود جاوا یه سری مقدار به اون‌ها داده. مقدار دیفالت جاوا برای فیلدهایی که مقداردهی نشدن، به این شکله:

- **متغیرهای عددی (مثل int، float و امثال اون‌ها):** مقدار 0 رو به خودشون می‌گیرن.
- **متغیرهای char:** مقدار '\0' رو به خودشون می‌گیرن.
- **متغیرهای boolean:** مقدار false به خودشون می‌گیرن.
- **متغیرهای از جنس reference type:** مقدار null به خودشون می‌گیرن.

null، یکی از کلیدواژه‌های خاص جاواست که نشون‌دهندهٔ اینه که یک متغیر از جنس reference type، هنوز مقداری به خودش نگرفته. اگر توی کد قبلی مون، خط زیر رو بنویسیم:

```
if (shahram.studentID == null) {
    System.out.println("Shahram does not have a studentID");
}
```

می‌بینید که پیام زیر چاپ می‌شه:

```
Shahram does not have a studentID
```

جلوتر، با این کلیدواژه بهتر آشنا می‌شیم.

## فیلدهای static

بعضی ویژگی‌ها، متعلق به هیچ objectی نیستن، ولی بی‌ربط به خود class هم نیستن. مثلاً توی کلاس Student، ویژگی «تعداد کل دانشجوها» متعلق به هیچ کدوم از قلی، ممد یا سلطان نیست، ولی به کلاس Student ربط داره.

به این ویژگی‌ها، ویژگی‌های static می‌گیم. اون‌ها به خود class مرتبطن و بین تمام instance‌های اون class مشترکن. فیلد static زیر رو برای دانشجوها تعریف کنید:

```
class Student {
    public static int totalNumberOfStudents = 0;

    // other stuff
}
```

سپس توی همهٔ constructorهایی که برای Student نوشتین، به مقدار اون یکی اضافه کنین. با این کار، با ساخت هر دانشجو، تعداد کل دانشجوها یکی زیاد می‌شه:

```
public Student() {
    totalNumberOfStudents++;
}
```



```
public Student(int newStudentAge, String newStudentName, String newStudentID)
{
    age = newStudentAge;
    name = newStudentName;
    studentID = newStudentID;
    grades = new ArrayList<>();

    totalNumberOfStudents++;
}
```

حالا، کد زیر رو توی main بنویسین:

```
public static void main(String[] args) {
    var gholi = new Student();
    System.out.println("Current number of students: " +
Student.totalNumberOfStudents);

    var mamad = new Student();
    System.out.println("Current number of students: " +
Student.totalNumberOfStudents);

    var javad = new Student();
    System.out.println("Current number of students: " +
Student.totalNumberOfStudents);
}
```

همچین خروجی‌ای می‌بینید:

```
Current number of students: 1
Current number of students: 2
Current number of students: 3
```

می‌بینید که ما برای دسترسی به `totalNumberOfStudents`، از خود کلاس `Student` استفاده کردیم. می‌تونستید با کد زیر، از هر کدوم از instance های `student` هم به اون دسترسی پیدا کنین، ولی کار چندان خوبی نیست:

```
System.out.println("Current number of students: " +
gholi.totalNumberOfStudents);
```

## methodها

تا اینجا کار، با کلاس‌های نسبتاً ساده‌ای سر و کار داشتید. اما جاهای مختلف به "رفتار کلاس" یا این ایده که کلاس ما کاری انجام بده اشاره کردیم، اینجاست که متدها وارد عمل میشن: به طور کلی هر وقت بخواهید توی کدتون تصمیمی بگیرید یا عملیات منطقی انجام بدید یا کلا کاری انجام بدید، باید از متدها استفاده کنید. متدها انقدر مهمن که حتی توی اولین مواجهه‌تون با جاوا از متد main استفاده کردید و توی اون کدتون رو نوشتید! توی این بخش قراره دقیق‌تر و کامل‌تر با متدها آشنا بشید. کد زیر، یک مثال ساده از یک متده:

```
public class Refrigerator {
    int numberOfBananas;

    public void getBananas(int n) {
        boolean enoughBananas = numberOfBananas >= n;
        if (enoughBananas) {
            numberOfBananas -= n;
            System.out.println(
                "You took " + n + " bananas out of your fridge!"
            );
        } else {
            System.out.println(
                "You don't have that many bananas in your fridge!"
            );
        }
    }
}
```

توی این مثال، یه کلاس Refrigerator داریم که یک فیلد از نوع int داره به نام numberOfBananas و یک متد داره که نوع خروجیش voidئه (خروجی نداره) و یک ورودی (argument) از نوع int داره. با استفاده از این متد می‌تونید از توی یخچالتون موز بردارید! حالا خودتون یه متد اضافه کنید که باهاش بتونید توی یخچالتون موز بذارید. متدتون احتمالاً چیزی شبیه به این میشه:

```
public void putBananas(int n) {
    numberOfBananas += n;
    System.out.println("You put " + n + " bananas in your fridge!");
}
```

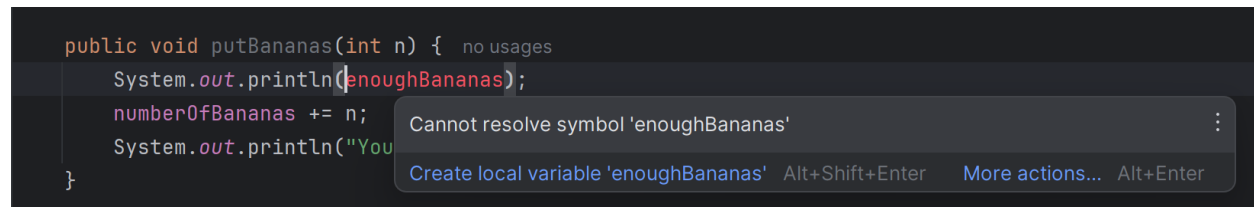
اینجا، متدمون تعداد مشخص و ثابتی ورودی داره (یکی)؛ اما می‌تونید متدهایی تعریف کنید که تعداد ورودی‌هاشون ثابت نباشه. برای این درس لازم نیست اینو یاد بگیرید، ولی اگه خودتون دوست دارید بیشتر راجع بهش بدونید، می‌تونید کلمه "varargs" رو جستجو کنید یا از [این لینک](#) راجع بهش بخونید.

## متغیرهای محلی (local variables)

متد `getBananas()` که توی مثال بخش قبل تعریف کردیم، قبل از هر چیزی چک می‌کنه که توی یخچال به اندازه کافی موز وجود داشته باشه و این رو توی یک متغیر محلی به اسم `enoughBananas` ذخیره می‌کنه. متغیرهای محلی موقتی هستن و فقط توی همون متدی که تعریف شدن قابل استفاده‌اند. این متغیرها وقتی متد صدا زده می‌شه، ساخته می‌شن و معمولاً بعد از تموم شدن متد از بین میرن. همچنین از بیرون متد هم نمی‌تونید بهشون دسترسی داشته باشید. برای این که خودتون ببینید، توی همین کلاس `Refrigerator` سعی کنید توی متد `putBananas` از متغیر `enoughBananas` استفاده کنید؛ همچنین چیزی مثلاً:

```
public void putBananas(int n) {
    System.out.println(enoughBananas);
    numberOfBananas += n;
    System.out.println("You put " + n + " bananas in your fridge!");
}
```

احتمالاً می‌بینید که `enoughBananas` قرمز شده. موس‌تون رو ببرید روش؛ با همچنین چیزی مواجه می‌شید:



```
public void putBananas(int n) { no usages
    System.out.println(enoughBananas);
    numberOfBananas += n;
    System.out.println("You
}
```

Cannot resolve symbol 'enoughBananas'

Create local variable 'enoughBananas' Alt+Shift+Enter More actions... Alt+Enter

اینجا IntelliJ داره بهتون می‌گه که نمی‌تونه `enoughBananas` رو پیدا کنه! دلیلش هم همونطور که گفتیم اینه که `enoughBananas` توی متد `getBananas` تعریف شده و مربوط به همون متده و توی `putBananas` همچنین متغیری وجود نداره! حالا یک متد `main` خالی توی کلاستون بنویسید و سعی کنید اجراش کنید:

```
public class Refrigerator {
    int numberOfBananas;

    public void getBananas(int n) {
        boolean enoughBananas = numberOfBananas >= n;
        if (enoughBananas) {
            numberOfBananas -= n;
            System.out.println(
                "You took " + n + " bananas out of your fridge!"
            );
        } else {
```

```

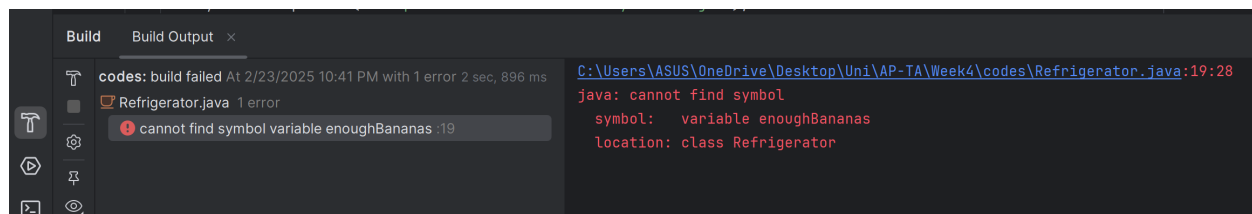
        System.out.println(
            "You don't have that many bananas in your fridge!"
        );
    }

    public void putBananas(int n) {
        System.out.println(enoughBananas);
        numberOfBananas += n;
        System.out.println("You put " + n + " bananas in your fridge!");
    }

    public static void main(String[] args) {
    }
}

```

کدتون کامپایل نمیشه، و با همچین چیزی مواجه می‌شید:



اینجا جاوا نتونسته کدتون رو کامپایل کنه و بهتون میگه که نمی‌تونه enoughBananas رو پیدا کنه. باید هشدارهای IntelliJ رو جدی می‌گرفتید!

ورودی‌های یه متد هم جزو متغیرهای محلی اون متد حساب میشن، با این تفاوت که مقدار اولیه‌شون موقعی که متد صدا زده می‌شه، از طرف کسی که متد رو فراخوانی کرده، مشخص میشه.

## مقداردهی اولیه به متغیرهای محلی

بر خلاف فیلدهای آبجکت که اگه مقداردهی‌شون نکنید، جاوا براشون مقدار پیش‌فرضی قرار میده، متغیرهای محلی رو باید قبل از استفاده کردن مقداردهی کنید وگرنه خطای کامپایل می‌گیرید:

```

public class SomeClass {
    // instance variables always get default values if
    // you don't initialize them
    int foo;

    void myMethod() {
        // local variables do not get default values
        int bar;
        foo += 1; // This is ok, foo has the value 0
        bar += 1; // compile-time error, bar is uninitialized
    }
}

```

```
public static void main(String[] args) {
    SomeClass something = new SomeClass();
    something.myMethod();
}
```

سعی کنید کد بالا رو اجرا کنید. می‌بینید که کدتون حتی کامپایل هم نمیشه! باید اول bar رو مقداردهی کنید:

```
bar = 99; // This is ok, we're setting bar's initial value
bar += 1; // Now this calculation is ok
```

البته دقت کنید که لازم نیست حتماً موقع تعریف کردن یک متغیر بهش مقداردهی کنید؛ صرفاً قبل از این که ازش استفاده کنید باید مقداردهیش کنید. موضوع وقتی پیچیده‌تر میشه که مقداردهی رو داخل یک شرط انجام بدید:

```
void myMethod() {
    int bar;
    if (someCondition) {
        bar = 42;
    }
    bar += 1; // Still a compile-time error, bar may not be initialized
}
```

توی این مثال، bar فقط در صورتی مقداردهی میشه که شرط someCondition برقرار باشه. یعنی همچنان ممکنه که قبل از خط `bar += 1`، متغیر bar مقداردهی نشده باشه. کامپایلر به شما اجازه نمیده همچین کاری بکنید و این کد هم خطای کامپایل میده.

برای حل این مشکل، چند راه حل وجود داره. می‌تونید متغیر رو قبل از شرطتون مقداردهی کنید، یا استفاده ای که از متغیر می‌کنید رو هم داخل شرط ببرید، یا می‌تونید با توجه به برنامه‌ای که دارید می‌نویسید، به نحوی مطمئن بشید که متغیر قبل از مقداردهی استفاده نمیشه. برای مثال، توی کد بالا می‌تونید bar رو هم در بلوک if و هم در بلوک else مقدار دهی کنید یا در صورتی که someCondition برقرار نبود، متد رو تموم کنید و return کنید:

```
void myMethod() {
    int bar;
    if (someCondition) {
        bar = 42;
    } else {
        return;
    }
    bar += 1; // This is ok!
}
```

توی این کد، یا bar مقداردهی میشه و بعد ارزش استفاده میشه، یا کلا متد قطع میشه و return می‌کنه. جاوا این رو ازتون می‌پذیره!

حالا چرا اصلاً جاوا انقدر روی این موضوع حساسه؟ یکی از متداول‌ترین مشکلاتی که توی زبان‌هایی مثل C و C++ به وجود میاد اینه که یادتون میره متغیری رو مقداردهی کنید. توی این زبان‌ها، متغیرهای مقداردهی نشده، مقادیر ظاهراً رندومی اختیار می‌کنند و این می‌تونه دردرساز باشه و باعث بشه دیباگ کردن برنامه‌ها سخت‌تر بشه. جاوا با مجبور کردن شما به مقداردهی به متغیرها، باعث جلوگیری از این مشکلات میشه.

## Shadowing

وقتی که یک متغیر محلی یا یک ورودی متد اسمش با اسم یکی از فیلدهای کلاسمون یکی باشه، اون متغیر محلی، اصطلاحاً روی اون فیلد "سایه می‌اندازه" و جلوی دسترسی ما به اون فیلد رو می‌گیره. شاید فکر کنید این مشکل به ندرت پیش میاد، ولی shadowing اتفاق نسبتاً متداولیه مخصوصاً وقتی که متغیرهامون اسم‌های متداولی داشته باشن. بیاید با یه مثال ببینیم:

```
public class Car {
    double x;
    double y;

    public void moveTo(double x, double y) {
        System.out.println("The car is moving to " + x + ", " + y);
    }
}
```

اینجا ما یک کلاس به نام Car داریم که فعلاً فقط دو تا فیلد برای مختصات داره (x و y). یک متد moveTo براش تعریف کردیم که قراره ماشین رو برامون حرکت بده. همونطور که می‌بینید، فعلاً متد moveTo فقط داره x و y رو چاپ می‌کنه. اما این x و y، کدوم x و y هستن؟ اگر مثلاً مختصات ماشینمون الان (3,4) باشه و ما متد moveTo رو روی ماشین صدا بزنیم و بهش مقادیر (6,7) رو بدیم، چه چیزی چاپ میشه؟ خودتون امتحان کنید! توی همین کلاس یک متد main بنویسید، توش یک آبجکت جدید از Car بسازید، بهش x و y بدید و متد moveTo رو روش صدا بزنید.

همونطور که می‌بینید، moveTo همون مقادیری رو چاپ می‌کنه که بهش ورودی دادیم؛ ولی ما اگر بخوایم ماشین رو حرکت بدیم، باید بتونیم مختصاتش رو تغییر بدیم، ولی چطور می‌تونیم به فیلدهای x و y که مربوط به آبجکتمون هستن دسترسی پیدا کنیم؟

**this**

هر وقت نیاز دارید که صریحاً به آبجکتی که توش هستیم یا یکی از اعضای اون اشاره کنید، می‌تونید از کلیدواژه `this` استفاده کنید. بیاید دوباره با مثال `moveTo` ببینیم:

```
public class Car {
    double x;
    double y;
    double gas;

    public void moveTo(double x, double y) {
        double distance = Math.sqrt(
            (this.x - x) * (this.x - x) + (this.y - y) * (this.y - y)
        );
        if (5 * distance > gas) {
            System.out.println("Not enough gas!");
        } else {
            this.x = x;
            this.y = y;
            gas -= 5 * distance;
            System.out.println("The car is moving to " + x + ", " + y);
        }
    }
}
```

اینجا، اول فاصله ای که قراره طی بشه رو حساب کردیم و توی متغیر محلی `distance` ریختیم. همونطور که می‌بینید، برای دسترسی به `x` و `y` مربوط به آبجکت (مختصات فعلی ماشین)، از `this.x` و `this.y` استفاده کردیم. `this` در واقع به همون آبجکتی که توش هستیم اشاره می‌کنه.

اینجا یک فیلد `gas` هم به `Car` اضافه کردیم که قراره مقدار بنزین ماشین رو نشون بده. در ادامه ی متد اول مطمئن می‌شیم که ماشین به اندازه کافی بنزین داره و بعد اگه بنزین داشت ماشین رو حرکت می‌دیم. می‌بینید که برای دسترسی به `gas` از `this` استفاده نکردیم؛ این به این دلیله که اشاره به آبجکتی که توش هستیم به طور ضمنی برقراره و `gas` و `this.gas` اینجا یک چیز هستند. مشکل جایی به وجود میاد که اسم یکی از متغیرهای محلیمون با اسم یکی از فیلدهای کلاسمون یکی باشه. اون وقت اگه بخوایم از فیلد کلاس استفاده کنیم، باید صریحاً این رو مشخص کنیم وگرنه پیش‌فرض جاوا استفاده از متغیر محلیه.

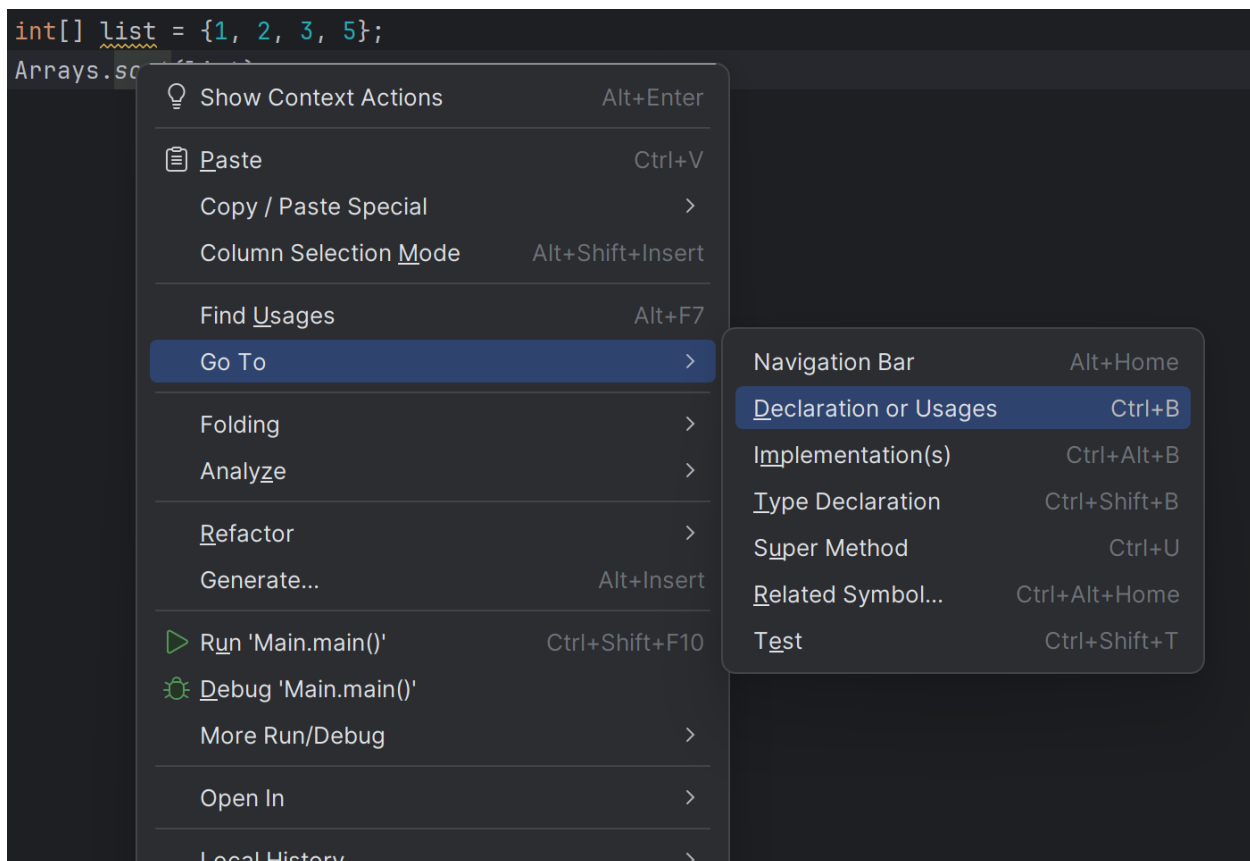
استفاده از `this` برای دسترسی به فیلدهایی که روشن سایه افتاده، روش مرسومیه و باعث می‌شه که بتونیم از اسم‌های متداولی که برای متغیرهای مختلف وجود داره (مثلاً `x` و `y` برای مختصات) استفاده کنیم و لازم نباشه هر بار دنبال یه اسم جدید برای متغیرهامون بگردیم. علاوه بر این، هر

جای دیگه‌ای که بخواید به آبجکتی که داخلش هستید اشاره کنید، می‌تونید از `this` استفاده کنید. مثلاً وقتی بخواید خود همین آبجکت رو به عنوان ورودی به یه متد بفرستید.

## متدهای استاتیک

متدهای استاتیک (static methods)، مثل فیلدهای استاتیک، به خود کلاس تعلق دارند، نه به آبجکت‌های مستقلی که ما از روی اون کلاس می‌سازیم. اما این یعنی چی؟ اول از همه، متدهای استاتیک خارج از آبجکت‌ها وجود دارند و برای صدا زدنشون لازم نیست آبجکتی وجود داشته باشه؛ شما می‌تونید اسم کلاس رو بنویسید و با عملگر نقطه متدهای استاتیک رو صدا بزنید. قبلاً از متدهای استاتیک زیاد استفاده کردید، مثلاً برای مرتب کردن آرایه‌ها از `Arrays.sort()` استفاده می‌کردید؛ ولی اینجا آبجکتی از کلاس `Arrays` نساختید و مستقیماً متد `sort` رو روی کلاس `Arrays` صدا زدید؛ این کار رو می‌تونید بکنید چون `sort` یک متد استاتیکه.

دوباره از `Arrays.sort()` استفاده کنید یا جایی که اون رو نوشتید رو بیارید، بعد روش راست کلیک کنید و گزینه `Go To` و بعد `Declaration or Usages` رو انتخاب کنید:





با همچین کدی مواجه می‌شید:

```
@Contract(mutates = "param1")
public static void sort( @NotNull int[] a) {
    DualPivotQuicksort.sort(a, parallelism: 0, low: 0, a.length);
}
```

همونطور که می‌بینید، پشت این متد از کلیدواژه static استفاده شده. این نشون میده که این متد، یک متد استاتیکه. حالا که می‌دونید متدهای استاتیک چجوری تعریف می‌شن، بیاید برای کلاس Car که تا الان داشتیم یک متد استاتیک تعریف کنیم:

```
public class Car {
    public static final int SUV = 0;
    public static final int SEDAN = 1;
    public static final int HATCHBACK = 2;

    double x;
    double y;
    double gas;
    int model;

    public static String[] getCarModels() {
        return new String[]{"SUV", "SEDAN", "HATCHBACK"};
    }
    // ...
}
```

اینجا، اول از همه یک فیلد جدید به ماشین هامون اضافه کردیم به اسم model که مدل ماشینمون رو نشون میده: ماشینمون می‌تونه شاسی‌بلند (model = 0)، سواری (model = 1) یا هاچ‌بک (model = 2) باشه. برای راحتی، مدل‌های مختلف ماشین رو به صورت فیلدهای static final تعریف کردیم. حالا فرض کنید به اسم این مدل‌ها به صورت String نیاز داشته باشیم، می‌تونیم مثل بالا یک متد استاتیک تعریف کنیم که این اطلاعات رو بهمون بده. دقت کنید که مدل‌های مختلف ماشین‌ها هیچ ارتباطی به یه ماشین خاص یا یه آبجکت مشخص از نوع Car ندارن و به‌طور کلی برای همه ماشین‌ها یکسان هستن. به همین خاطر، استفاده از فیلدها و متدهای استاتیک بهترین انتخابه.

اصلی‌ترین کاربرد متدهای استاتیک، برای تعریف متدهای کمکیه؛ متدهایی که یا مستقل از آبجکت‌ها کار می‌کنن، یا روی آبجکت‌هایی که از اون کلاس (یا حتی کلاس‌های دیگه) می‌سازیم، عملی انجام میدن و منطقشون به یک instance خاص تعلق نداره و به‌طور کلی عمل می‌کنند.

حالا بیاید یه متد استاتیک دیگه برای Car بنویسیم:

```
public void printModelsCount(ArrayList<Car> list) {
    int suvCount = 0;
    int sedanCount = 0;
    int hatchbackCount = 0;
    for (Car car : list) {
        switch (car.model) {
            case 0:
                suvCount++;
                break;
            case 1:
                sedanCount++;
                break;
            case 2:
                hatchbackCount++;
                break;
        }
    }
    System.out.println("SUV: " + suvCount);
    System.out.println("SEDAN: " + sedanCount);
    System.out.println("HATCHBACK: " + hatchbackCount);
}
```

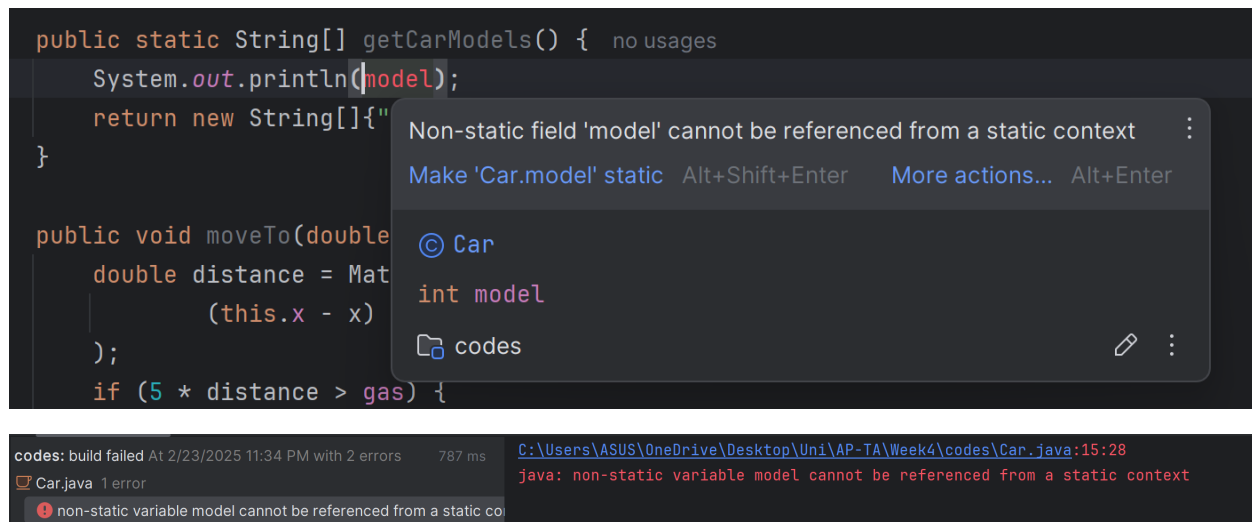
این متد، یک ArrayList از ماشین‌ها می‌گیرد، تعداد مدل‌های مختلف ماشین‌ها رو می‌شماره و چاپ می‌کنه. همونطور که می‌بینید کارکرد این متد هیچ ربطی به یک instance خاص از Car نداره و به همین دلیل استاتیک تعریفش می‌کنیم.

مثال خوب دیگه‌ای برای کاربرد متدهای استاتیک، کلاس Math هست. این کلاس قراره مجموعه‌ای از عملیات‌های ریاضی باشه؛ به همین دلیل تمام متدهای کلاس Math استاتیک هستند. البته Math یک مرحله فراتر میره، شما نمی‌تونید اصلاً آبجکتی از Math بسازید! اصلاً این که یک آبجکت از روی Math بسازید، معنی نداره و نیازی بهش نیست. این کلاس صرفاً قراره که مجموعه‌ای از متدها و متغیرها برای انجام عملیات ریاضی باشه. شما چند "ریاضی" مختلف ندارید که بخواید instance های مختلفی از Math بسازید!

حالا سعی کنید توی یکی از متدهای استاتیک Car از یکی از فیلدها یا متدهای غیر استاتیک Car استفاده کنید. همچین چیزی مثلاً:

```
public static String[] getCarModels() {
    System.out.println(model);
    return new String[]{"SUV", "SEDAN", "HATCHBACK"};
}
```

اگر موستون رو ببرید روی model یا سعی کنید کلاس Car رو کامپایل کنید و جایی ازش استفاده کنید، با همچین خطاهایی مواجه می‌شید:



همونطور که می‌بینید، جاوا داره بهتون میگه که نمی‌تونید یک فیلد غیر استاتیک مثل `model` رو تو یه یک متد استاتیک استفاده کنید. شما می‌تونید متد `getCarModels` رو بدون داشتن هیچ آبجکتی صدا بزنید؛ پس این `model`ی که سعی دارید ازش استفاده کنید، مربوط به کدوم آبجکت؟ از اون جایی که متدهای استاتیک مربوط به کلاس و از آبجکت‌ها جدا هستن، طبیعیه که نمی‌تونند به فیلدها و متدهای عادی که مربوط به هر آبجکت هستند دسترسی داشته باشن و فقط به متد ها و متغیر های استاتیک دسترسی دارند.

## Method overloading

Method overloading، این قابلیتیه که شما چند متد رو با یک اسم، ولی با جنس و تعداد ورودی متفاوت توی یک کلاس تعریف کنید؛ وقتی که متد رو صدا می‌زنید، کامپایلر با توجه به نوع ورودی، متد درست رو انتخاب می‌کنه و اجرا می‌کنه.

Method overloading، قابلیت بسیار قدرتمند و پرکاربردی. ایده اصلی اینه که متدهایی درست کنیم که روی ورودی های مختلف، کارهای یکسانی انجام میدن. با این کار می‌تونید این توهم رو ایجاد کنید که یک متد می‌تونه روی انواع مختلفی از ورودی ها کار کنه. متد `println()` که از اولین جلسه باهاش کار کردید، مثال خیلی خوبی از method overloading هست؛ شما به `println()` می‌تونید هر ورودی دلخواهی بدید و اون به نحوی یک نمایش متنی از اون ورودی رو براتون چاپ می‌کنه. توی زبان هایی که method overloading ندارند، کار سخت‌تر میشه. مثلاً برای چاپ چیزهای مختلف باید متدهای

مختلف با اسم‌های مختلف تعریف کنیم و در اون صورت، این مسئولیت روی دوش شما می‌افته که متد درست رو انتخاب کنید. بیاید یه مثال دیگه از method overloading ببینیم:

```
public class Sum {
    // Overloaded sum(). This sum takes two int parameters
    public int sum(int x, int y) {
        return (x + y);
    }

    // Overloaded sum(). This sum takes three int parameters
    public int sum(int x, int y, int z) {
        return (x + y + z);
    }

    // Overloaded sum(). This sum takes two double
    // parameters
    public double sum(double x, double y) {
        return (x + y);
    }

    public static void main(String[] args) {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
    }
}
```

همونطور که می‌بینید، اینجا سه تا متد با نام یکسان sum داریم، ولی ورودی‌هاشون فرق می‌کنه. هر سه تای این متدها دارن عمل جمع کردن رو انجام میدن، ولی یکی دو تا double رو جمع می‌کنه، یکی دو تا int رو جمع می‌کنه و یکی 3 تا int رو جمع می‌کنه!

به غیر از نوع ورودی‌ها و تعدادشون، با تغییر دادن ترتیب ورودی‌ها هم میشه متدها رو overload کرد:

```
class Student {
    // Method 1
    public void getStudentInfo() {
        System.out.println("Name :" + name + " "
            + "ID :" + roll_ studentID);
    }

    // Method 2
    public void getStudentInfo (String name) {
        // Again printing name and id of person
        System.out.println("ID :" + studentID + " "
            + "Name :" + name);
    }
}
```

---

بعد از این که با مباحث مربوط به polymorphism و متدهای override شده آشنا شدید، به method overloading دوباره برمیگردیم.

## Reference type ها

همون‌طور که تا الآن به خوبی می‌دونید، توی جاوا، type ها به دو دسته primitive type و reference type تقسیم‌بندی می‌شن. primitive type ها، تایپ‌های بسیار ساده‌ای مثل int، char، boolean و امثال اون‌ها هستن. فهرست کامل اون‌ها توی لیست زیر اومده:

Type	Definition	Approximate range or precision
boolean	Logical value	true or false
char	16-bit, Unicode character	64K characters
byte	8-bit, signed integer	-128 to 127
short	16-bit, signed integer	-32,768 to 32,767
int	32-bit, signed integer	-2.1e9 to 2.1e9
long	64-bit, signed integer	-9.2e18 to 9.2e18
float	32-bit, IEEE 754, floating-point value	6-7 significant decimal places
double	64-bit, IEEE 754	15 significant decimal places

هر تایپ دیگه‌ای توی جاوا، reference type. String، JFrame، ArrayList و حتی تایپ‌هایی مثل Car و Student که تا این‌جای کار تعریف کردیم، همگی reference type ان. هر reference type با یک کلاس تعریف شده.

## تفاوت primitive type ها و reference type ها

همون‌طور که می‌دونید، تمام متغیرهای برنامه‌های شما، توی حافظه خاصی به اسم RAM ذخیره می‌شن. سیستم عامل، متغیرهای شما رو توی دو بخش متفاوتی از این حافظه، به اسم stack و heap نگه می‌داره. با این دو توی درس‌های ساختمان داده و سیستم عامل بیشتر آشنا می‌شین، ولی برای الآن، بدونید که حافظه stack، از heap سریع‌تره، ولی در مقابل به خورده کم حجم‌تره<sup>1</sup>.

متغیرهایی که از جنس primitive type تعریف می‌کنین، حجم کمی دارن و بین ۱ با ۸ بایت از مموری رو اشغال می‌کنن. به همین خاطر، جاوا اون‌ها رو توی stack نگه می‌داره تا از سرعت بهتر stack

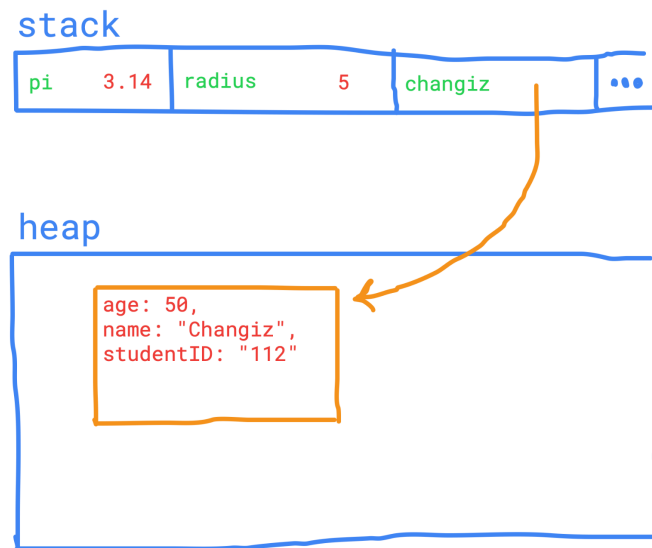
<sup>1</sup> اگر راستش رو بخواید، در واقعیت ممکنه این‌جور نباشه. Heap و Stack به خودی خود از دیگری سریع‌تر یا حجیم‌تر نیستن و ما این‌جا داریم خیلی (خیلی خیلی خیلی) ساده‌سازی می‌کنیم. توی درس‌های بعدی‌تون بهتر می‌فهمید که تفاوت این دو با هم چیه. برای الآن، فرض کنید این چیزی که گفتیم کاملاً درسته.

استفاده بکنه و همزمان، stack سریع پر نشه. از طرفی، ممکنه objectهایی که توی برنامه‌تون تعریف می‌کنین - و همیشه از جنس reference type ان-، حجم بسیار بیشتری داشته باشن. جاوا، اطلاعات این objectها رو توی heap ذخیره می‌کنه و توی stack، صرفاً یه اشاره‌گر (pointer) به اون‌ها نگه می‌داره.

مثلا، برنامه زیر رو در نظر بگیرین:

```
public class Main {
    public static void main(String[] args) {
        int radius = 5;
        double pi = 3.14;
        Student changiz = new Student(50, "Changiz", "112");
    }
}
```

اگر مموری رو حین اجرای این برنامه ببینیم، همچین شکلی داره:



اگر مقدار خود متغیر چنگیز رو چاپ کنید:

```
System.out.println(changiz);
```

همچین خروجی‌ای می‌بینید:

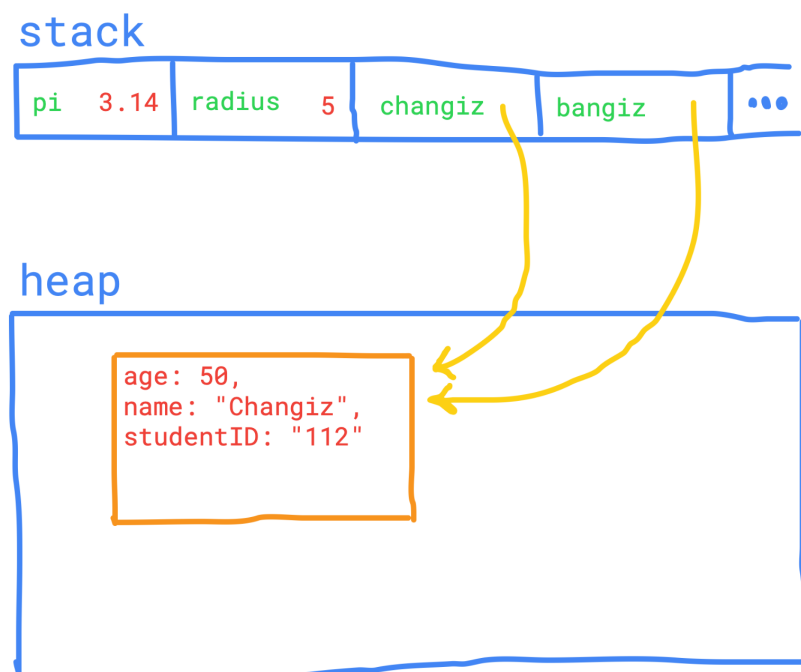
```
Student@6acbcfc0
```

متغیر `changiz`، در واقع صرفاً یک `pointer` یا `reference` به یه آبجکت از جنس `Student` و نه چیزی بیشتر. با استفاده از اپراتور نقطه (`.`)، می‌تونید به `field`ها و `method`های این آبجکت دسترسی داشته باشید.

این اتفاق، یه ساید افکت جالب روی کدهای شما داره. فرض کنید، یه دانشجوی دیگه به اسم `بنگیز` درست کردیم و اون رو مساوی با `چنگیز` قرار دادیم:

```
Student bangiz = changiz;
```

حالا، `بنگیز` و `چنگیز` هر دو به یک نقطه از `heap` اشاره می‌کنن:



حالا اگر شما، شماره دانشجویی `چنگیز` رو عوض کنید:

```
changiz.studentID = "40113";
```

و بعد، شماره دانشجویی `چنگیز` و `بنگیز` رو چاپ کنید:

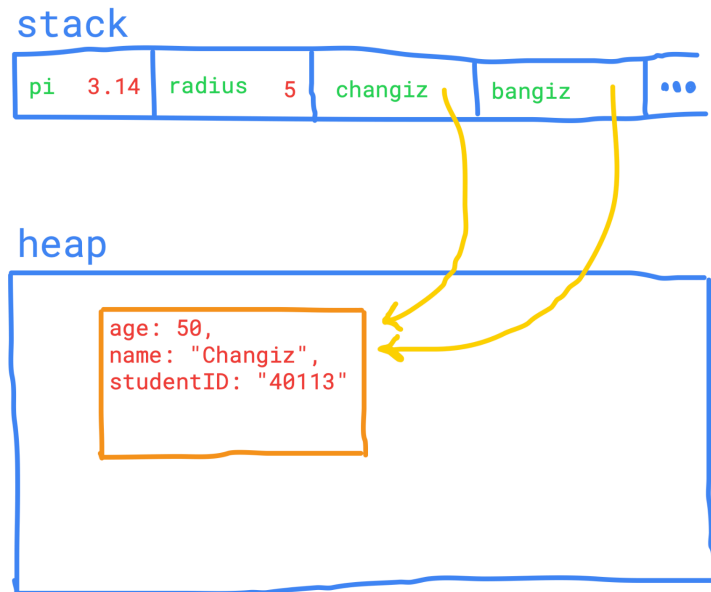
```
System.out.println("Changiz studentID: " + changiz.studentID);
System.out.println("Bangiz studentID: " + bangiz.studentID);
```

می‌بینید که شماره دانشجویی `بنگیز` هم عوض شده!

```
Changiz studentID: 40113
Bangiz studentID: 40113
```



عجیبه، نه؟ راستش نه اونقدران. اگر به مموری نگاه کنید، بعد از عوض شدن شماره دانشجویی چنگیز، همچین وضعیتی داره:



چنگیز، شماره دانشجویی آبجکتی که به اون اشاره می‌کرده رو تغییر داده. دست بر قضا، بنگیز هم به همین آبجکت اشاره می‌کرده و در نتیجه، شماره دانشجویی بنگیز هم واقعا عوض شده. یه جای دیگه هم اتفاق مشابه‌ای می‌افته. برای دیدن اون، متد زیر رو به کدتون اضافه کنین:

```
public static void resetID(Student student) {
    student.studentID = "00000000";
}
```

این متد، یه Student رو ورودی می‌گیره، و شماره دانشجویی‌ش رو دستکاری می‌کنه. حالا با استفاده از کد زیر، چنگیز رو به این متد ورودی بدین و بعدش، شماره دانشجویی چنگیز و بنگیز رو چاپ کنید:

```
resetID(changiz);

System.out.println("Changiz studentID: " + changiz.studentID);
System.out.println("Bangiz studentID: " + bangiz.studentID);
```

خروجی، به همچین شکلیه:

```
Changiz studentID: 00000000
Bangiz studentID: 00000000
```

شاید بتونید حدس بزنید که این‌جا چی شده. با ورودی دادن چنگیز به resetID، در واقع شما اشاره‌گرتون به heap رو به این تابع ورودی دادین. پس متغیر student توی resetID و changiz و

bangiz، هر سه به یک نقطه از heap اشاره می‌کنن و مثل قبل، با تغییر فیلدهای یکی از اون‌ها، هر سه تغییر می‌کنن. به این نوع ورودی دادن به توابع، اصطلاحاً passing by reference می‌گن. مشابه هیچ کدوم از این اتفاق‌ها، برای primitive type نمی‌افته. چون همیشه توی stack نگهداری می‌شن و پوینتری به heap برای اون‌ها نگهداری نمی‌شه.

## کلاس‌های wrapper برای primitive type

هر کدوم از primitive type، یه تایپ مشابه از جنس reference type هم دارن. این تایپ‌ها، توی جدول زیر اومدن:

Primitive	Wrapper
void	java.lang.Void
boolean	java.lang.Boolean
char	java.lang.Character
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double

شما لازم نیست خیلی نگران این تایپ‌های جدید باشین، ولی بدونین که وجود دارن. وقتی که یک آرایه از int‌ها تعریف می‌کنین، از اون‌ها استفاده می‌کنین:

```
var arr = new ArrayList<Integer>();
```

اگر دقت کنید، به جای این که بین دو براکت از int استفاده کنید، از Integer استفاده کردین. اگر روی اون کلیک راست کنید و از Go To به Declaration and Usages برین، می‌تونید ببینید که پشت اون یک کلاسه:

```
74 @jdk.internal.ValueBased
75 public final class Integer extends Number
76     implements Comparable<Integer>, Constable, ConstantDesc {
    A constant holding the minimum value an int can have,  $-2^{31}$ .
81 @Native public static final int MIN_VALUE = 0x80000000;
82
    A constant holding the maximum value an int can have,  $2^{31}-1$ .
87 @Native public static final int MAX_VALUE = 0x7fffffff;
88
    The Class instance representing the primitive type int.
    Since: 1.1
```

این کلاس، معادل reference type گونه‌ای برای `int` آنه، و وقتی با کلاس‌ها سرو کار داریم ازش استفاده می‌شه. اگر خواستین بیشتر راجع به اون بدونین، به [این داکيومنت](#) رجوع کنین.

## Garbage Collection

قبل اینکه بریم سراغ Garbage Collection بیاین اول مفهوم نشت حافظه (Memory Leak) رو بررسی کنیم.

### آشنایی با Memory Leak

توی بعضی از زبان‌های برنامه‌نویسی (مثل C و C++) مسئولیت «آزاد کردن» حافظه بر عهده خود برنامه‌نویس هست. این یعنی شما باید هر وقت که دیگه به یک شی نیاز نداشتین، خودتون اون حافظه رو آزاد کنین. مثلاً اگه با استفاده از تابع malloc یه مقداری از حافظه رو allocate کردین، وقتی که دیگه این حافظه رو نیاز نداشتین، باید خودتون با استفاده از تابع free اون حافظه رو آزاد کنین. بیاین یه مثال ببینیم:

```
#include <stdlib.h>

int main() {
    // Allocate memory dynamically
    int *ptr = (int *)malloc(sizeof(int) * 5); // Allocating memory for 5
    integers

    // Use the allocated memory
    for (int i = 0; i < 5; i++) {
        ptr[i] = i + 1;
    }
    /*
     * Doing some stuff with these numbers
     */

    // Forgetting to free the allocated memory causes a Memory Leak
    // free(ptr); // If we uncomment this line, the Memory Leak will be
    avoided.

    return 0;
}
```

توی این برنامه که به زبان C (!) نوشته شده، اول به اندازه 5 متغیر int حافظه اشغال می‌کنیم. اشاره‌گر ptr به اولین خونه از این 20 بایت<sup>1</sup> اشاره می‌کنه. حالا میایم 5 تا عدد صحیح رو در حافظه ذخیره می‌کنیم. فرض کنین با این اعداد یه سری کار انجام دادیم و الان کارمون باهاشون تموم شده. اما بعد اینکه کارمون تموم شد، فراموش کردیم که این 20 بایت حافظه رو آزاد کنیم! در حقیقت باید با صدا

<sup>1</sup> با این فرض که هر متغیر از جنس int، حافظه‌ای به اندازه 4 بایت رو اشغال کنه.

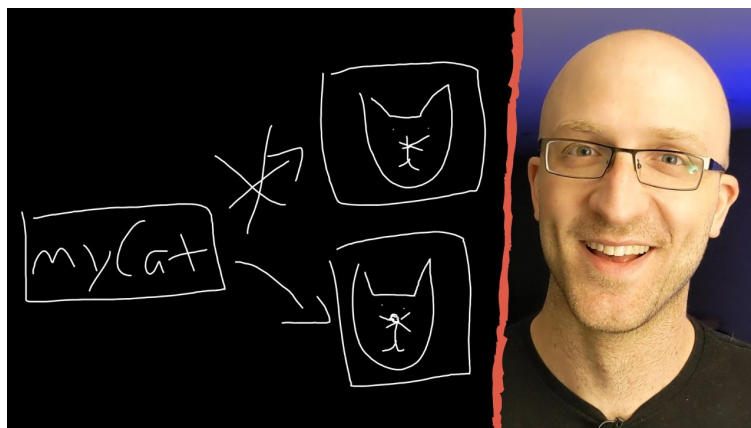
زدن تابع free اعلام می‌کردیم که ما دیگه به این 20 بایت نیازی نداریم و در نتیجه این حافظه آزاد میشد.

اگه آزادش نکنین چی میشه؟ در این صورت اون حافظه همچنان در اختیار برنامه قرار داره و به اصطلاح Memory Leak رخ می‌ده. این یعنی حافظه‌ای که دیگه به کار نمیاد، همچنان در اختیار برنامه باقی می‌مونه و هیچوقت آزاد نمی‌شه. این موضوع می‌تونه باعث بشه که برنامه به مرور زمان حافظه زیادی مصرف کنه و سیستم دچار مشکلاتی مثل کندی عملکرد یا حتی crash بشه.

## Garbage Collection: یه راه حل خوب

وقتی داریم راجع به Garbage Collection در جاوا صحبت می‌کنیم، به زبان ساده یعنی جاوا خودش می‌ره و حافظه‌ای که دیگه به هیچ‌کار نمی‌آد رو آزاد می‌کنه. مثلاً وقتی که شما یه شی رو توی برنامه می‌سازید و دیگه به اون نیاز ندارین، جاوا خود به خود این شی رو پاک می‌کنه. شما اصلاً نیازی نیست که خودتون حافظه رو آزاد کنین، همه چی به صورت خودکار اتفاق می‌افته!

شاید بپرسین که چطور این کار انجام میشه؟ خوب، جاوا از یه سری الگوریتم‌ها برای این کار استفاده می‌کنه. ولی ما اون‌ها رو اینجا بررسی نمی‌کنیم. توی این [ویدیوی یوتیوب](#) و این [داک اوراکل](#) می‌تونین مطالب بیشتری در رابطه با این موضوع ببینین.



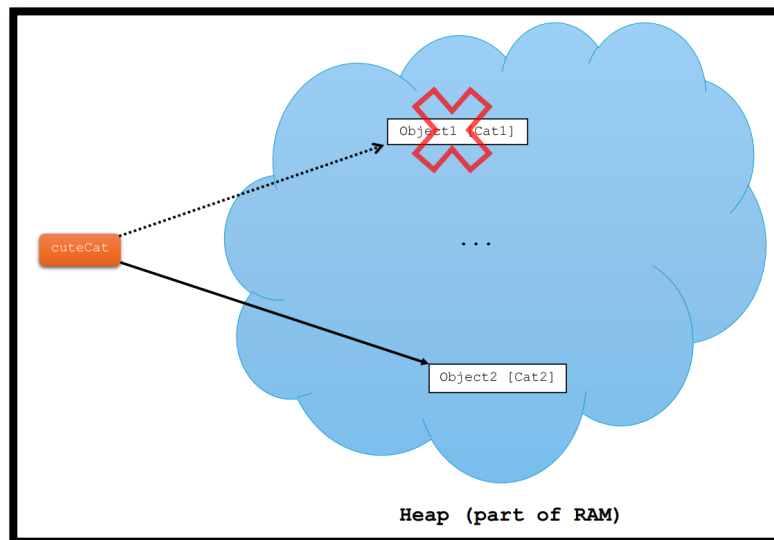
Thumbnail: *Java's Garbage Collection Explained – How It Saves your Lazy Programmer Bu\*\**

به طور کلی این ویژگی جاوا باعث میشه که شما تمرکز بیشتری روی منطق برنامه‌نویسی داشته باشین و دیگه نگران مدیریت دستی حافظه نباشین.

حالا تو کد زیر، می‌تونین رفتار Garbage Collector رو در ارتباط با آبجکت cuteCat رو ببینین؟

```
Cat cuteCat = new Cat("Cat 1");
cuteCat = new Cat("Cat 2");
```

شکل زیر می‌تونه نمایش خوبی از اتفاقات باشه. در اینجا cuteCat به جایی از حافظه اشاره<sup>۱</sup> می‌کنه که توی اون یه آبجکت از جنس Cat (که فیلد اسمش Cat1 هست) ذخیره شده. توی خط بعدی cuteCat به جایی از حافظه اشاره می‌کنه که توی اون یه آبجکت از جنس Cat (که فیلد اسمش Cat2 هست) ذخیره شده. اما الان هیچ فلشی به Cat1 وارد نمیشه، پس Garbage Collector اون را آزاد می‌کنه.



## Garbage Collector حواسش هست!

Garbage Collector آبجکت‌هایی که هنوز بهشون نیاز داریم (بهشون رفرنس داریم) رو پاک نمی‌کنه. مثلا بیا این کد زیر رو ببینیم:

```
class Duck {
    String name;
}

public class Main {
    public static void main(String[] args) {
        Duck duck = createDuck(); // a duck object will be created
        System.out.println(duck.name);
    }

    public static Duck createDuck() {
```

<sup>۱</sup> دقت کنین که لفظ «اشاره کردن» در جاوا خیلی درست نیست. به خاطر اینکه ما در جاوا pointer نداریم و مدیریت حافظه رو خود جاوا برامون انجام میده. در جاوا حتی دسترسی مستقیم به حافظه هم نداریم (برخلاف C). اما برای مشابهت با زبان C، در اینجا هم از لفظ اشاره کردن استفاده کردیم. در جاوا به متغیرهایی مثل cuteCat، می‌گن reference variable.

```
Duck localDuck = new Duck();  
localDuck.name = "A White Duck";  
return localDuck;  
}  
}
```

توی این کد اومدیم اول به کلاس خیلی ساده به اسم Duck تعریف کردیم. به متد هم به اسم createDuck تعریف کردیم که اول میاد به آبجکت از جنس Duck میسازه، بعد متغیر name رو توی این آبجکت مقداردهی می‌کنه و در نهایت این آبجکت رو به عنوان خروجی متد بر میگردونه. بعد توی متد main، سعی می‌کنیم به متغیر name توی این آبجکت دسترسی داشته باشیم.

اما نکته کجاست؟ احتمالاً توی درس‌های قبلیتون خوندین که «متغیرهایی که توی یک تابع تعریف میشن عمرشون به اندازه اجرای همون تابع هست و پس از اتمام اجرای تابع، اون متغیر هم از بین میره». پس شاید انتظار داشته باشیم Garbage Collector آبجکت Duck رو از بین ببره! اما واقعیت اینه که Garbage Collector حواسش هست که ما کدوم آبجکت‌ها رو هنوز نیاز داریم و نباید پاکشون کنه. اینجا هم ما چون Duck رو به عنوان خروجی برگردوندیم، یعنی لابد نیازش داریم، پس پاکش نمی‌کنه.

خروجی کد بالا به صورت زیر هست.

```
A White Duck
```

## یک نکته در مورد کلاس‌ها

در هر فایل جاوا (فایل با پسوند java)، متونیم حداکثر یک کلاس public داشته باشیم و همچنین اسم این کلاس public باید حتماً با اسم فایل یکی باشد. مثال درست زیر رو ببینین:

```
// MyClass.java
public class MyClass {
    public void sayHello() {
        System.out.println("Hello, Java!");
    }
}
```

کدهای زیر نادرست هستن:

```
// MyFile.java
public class MyClass { } // Error! The class name does not match the file
name.
```

```
// MyFile.java
public class MyClass { }

public class AnotherClass { } // Error! Only one public class is allowed.
```



## Packages

### نیاز به منظم کردن فایل‌ها

در برنامه‌های که توی جاوا می‌نویسیم، همیشه از کلاس‌ها یا اینترفیس‌ها<sup>۴</sup> استفاده می‌کنیم. مثلاً برنامه ساده زیر که در کلاس Sample نوشته شده رو ببینین:

```
public class Sample {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

ولی برنامه‌های پیچیده‌تر ممکنه از صد ها کلاس تشکیل شده باشن. اگه همه این کلاس‌ها رو بدون هیچ نظم کناری هم قرار بدیم باعث میشه برنامه‌مون ناخوانا باشه و خودمون هم گیج می‌شیم. احتمالاً یکی از اولین چیزهایی که برای منظم کردن فایل‌ها به ذهنمون میرسه استفاده از پوشه هاست. و این دقیقاً همون امکانیه که جاوا برای منظم کردن کلاس‌های برنامه‌مون واسه ما فراهم کرده: ایجاد package‌های مختلف.

### پکیج چیه؟

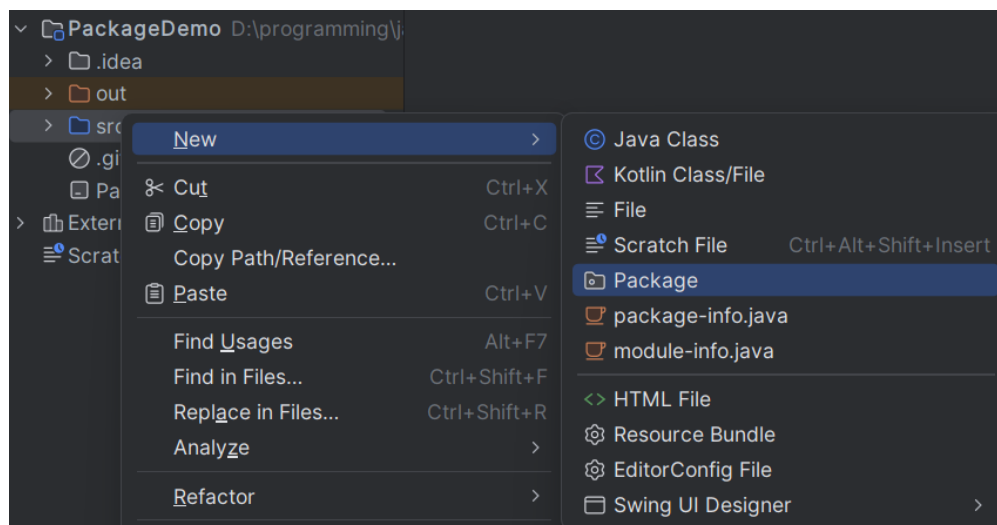
پکیج‌ها در جاوا مثل پوشه‌هایی هستن که کدهای برنامه‌نویسی (شامل کلاس‌ها و اینترفیس‌ها) رو داخلشون قرار می‌دیم تا همه چیز منظم و مرتب باشه. وقتی که برنامه‌های پیچیده‌تر رو می‌نویسیم، تعداد کلاس‌ها زیاد می‌شه و اینجاست که پکیج‌ها به کمکمون میاد تا بتونیم این کلاس‌ها رو دسته‌بندی کنیم.

<sup>۴</sup> با اینترفیس‌ها بعداً آشنا میشین. فعلاً کاری بهشون نداریم.

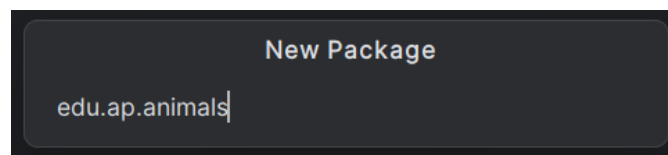


## چطور یک پکیج ایجاد کنیم؟

اول به پروژه به اسم PackageDemo ایجاد کنیم. حالا روی پوشه src راست کلیک کنیم و از نوار New، گزینه Package رو انتخاب کنیم.

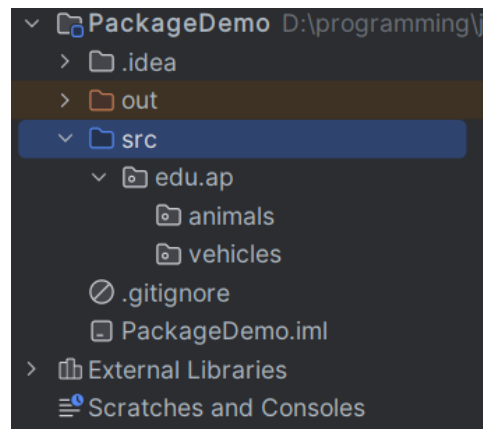


حالا اسم پکیج رو وارد کنیم. مثلا در اینجا edu.ap.animals

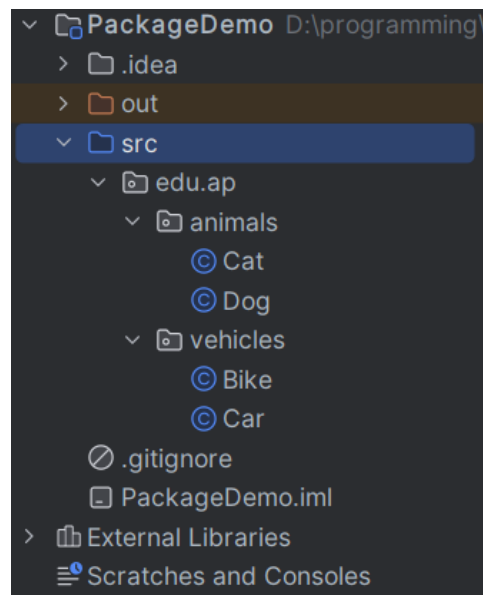


به همین شکل پکیج edu.ap.vehicles رو هم بسازین.

الان باید چیزی شبیه به تصویر زیر رو توی پوشه src داشته باشیم:



حالا بیاین چند تا کلاس به هر کدام از این پکیج‌ها اضافه کنیم. مثلاً کلاس‌های Dog و Cat رو به پکیج animals و کلاس‌های Bike و Car رو به پکیج vehicles اضافه کنیم. در نهایت باید چیزی شبیه به تصویر زیر رو داشته باشیم:



پکیج‌هامون رو ساختیم! حالا ببینیم واقعا چه فایل‌هایی ایجاد شده. اگه از توی explorer به محل ایجاد پروژه‌تون برین و به ترتیب وارد پوشه‌های src، edu، ap، animals بشین چیزی شبیه به این‌ها می‌بینین:

Name	Type	Size
idea	File folder	
out	File folder	
src	File folder	
.gitignore	GITIGNORE File	1 KB
PackageDemo.iml	IML File	1 KB

Name	Type	Size
edu	File folder	

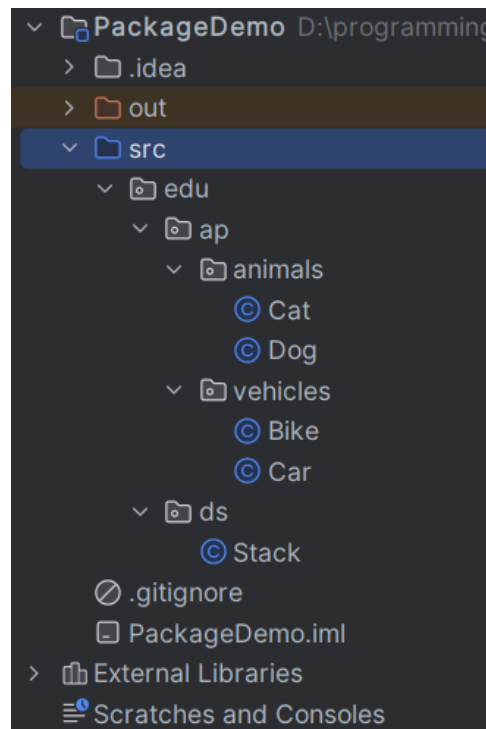
Name	Type	Size
ap	File folder	

Name	Type	Size
animals	File folder	
vehicles	File folder	

Name	Type	Size
Cat.java	JAVA File	1 KB
Dog.java	JAVA File	1 KB

پس دیدیم که وقتی ما داریم به پکیج تعریف می‌کنیم، واقعا پوشه ایجاد می‌شه!

در نهایت پکیج edu.ds شامل کلاس Stack رو درست کنین. در نهایت باید چیزی شبیه به این ببینین:



اما الان بریم کد Cat.java رو ببینیم:

```
package edu.ap.animals;

public class Cat {
}
```

همونطور که میبینین، عبارت package edu.ap.animals; به ابتدای این کد اضافه شده. این خط رو نباید پاک کنین، چون اون موقع جاوا متوجه نمیشه Cat متعلق به کدوم پکیجه و خطای کامپایل میخورین.

## نام‌گذاری متداول پکیج‌ها (Naming Conventions)

1. پکیج‌ها همواره با حروف کوچک نامگذاری میشن. مثلا java.util
2. از کلمات رزرو شده جاوا استفاده نکنین. کلماتی مثل class، public، static و...
3. از underscore (\_)، dash (-)، space و کاراکترهای خاص (مثل @، \$، &) استفاده نکنین. در نامگذاری پکیج‌ها فقط مجازیم از dot (.) استفاده کنیم.

a. **نادرست:** com.github.my-awesome-project

b. درست: com.github.myawesomeproject

4. یک قاعده دیگه در نامگذاری پکیج‌ها، reverse domain name هست (برعکس نوشتن نام دامنه). اگه به پروژه‌ای متعلق به شرکت یا سازمانی هست که دامنه (domain) خودش رو داره، اون دامنه رو به شکل برعکس می‌نویسیم. مثلا شرکت Mozilla (که نام دامنه‌ش mozilla.org هست) چند تا پکیج توی جاوا داره. یکی از این پکیج‌ها اسمش org.mozilla.javascript هست (البته این قاعده در مورد پکیج‌های استاندارد خود جاوا صدق نمی‌کنه).

## کلیدواژه Import

### یک مثال عملی

بیاین توی کلاس Bike یه آبجکت از کلاس Cat بسازیم (دقت کنین که این دو کلاس متعلق به دو پکیج متفاوت هستن):

```
package edu.ap.vehicles;

public class Bike {
    public static void main(String[] args) {
        Cat cat = new Cat();
    }
}
```

اگه سعی کنیم این کد رو اجرا کنیم، موقع کامپایل به مشکل می‌خوریم:

```
java: cannot find symbol
symbol:   class Cat
location: class edu.ap.vehicles.Bike
```

مشکل چیه؟ جاوا نمیتونه کلاسی به اسم Cat رو پیدا کنه! دلیلش هم اینه که جاوا فقط کلاس‌هایی رو می‌بینه که توی همین پکیج هستن.

پس باید یجوری کلاس Cat رو به کدمون اضافه کنیم. کلیدواژه import دقیقا برای همین کار هست. با استفاده از این کلیدواژه، ما به جاوا اعلام می‌کنیم که می‌خوایم این کلاس رو به کدمون اضافه کنیم:

```
package edu.ap.vehicles;
import edu.ap.animals.Cat;

public class Bike {
    public static void main(String[] args) {
        Cat cat = new Cat();
    }
}
```

```
}
}
```

الان دیگه کدمون کار می‌کنه و می‌تونیم به آبجکت از کلاس Cat بسازیم.

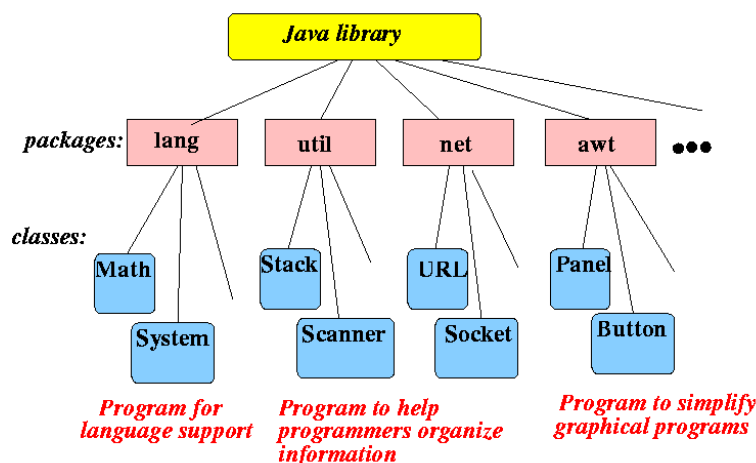
## در اهمیت پکیج‌ها

یکی از بزرگترین مزایای جاوا داشتن کتابخانه (library) ها بسیار متنوع و کاربردی هست.

می‌خواهین به PDF درست کنین؟ کتابخانه مربوط به اون رو import کنین. می‌خواهین با دیتابیس کار کنین؟ کتابخانه مربوط بهش رو import کنین ...

مثلا یکی از پکیج‌های مهم جاوا، java.lang هست. این پکیج شامل کلاس‌های پایه ای مثل System، Integer، Math و String هست.

تو شکل زیر میتونین به نمای کلی از پکیج‌های جاوا ببینین:



## import کردن کل پکیج

شما میتونین تمام کلاس‌های موجود توی یک پکیج رو یکجا import کنین. این کار رو با عبارت \* میتونین انجام بدین. مثلا کد زیر تمام کلاس‌های موجود در پکیج edu.ap.animals رو import می‌کنه.

```
import edu.ap.animals.*;
```

این قابلیت کار ما رو خیلی اوقات آسون می‌کنه. ولی انجام این کار همیشه هم مناسب نیست. import کردن دقیق کلاس‌ها علاوه بر خوانایی بیشتر کدمون، کمی هم زمان کامپایل مون رو کمتر می‌کنه.

دقت کنید که پکیج‌ها خودشان میتونن شامل پکیج باشن؛ همونطوری که پوشه‌ها میتونن داخل خودشان پوشه داشته باشن. اما عبارت `*` فقط کلاس‌های متعلق به پکیج رو `import` می‌کنه و نه sub-package ها رو (`import` کردن recursive نداریم).

مثلا در نظر بگیرید که پکیج `java.awt` یک کتابخونه استاندارد جاوا هست که شامل sub-package `java.awt.event` هست. کلاس `Color` متعلق به پکیج `java.awt` و کلاس `ActionEvent` متعلق به زیرپکیج `java.awt.event` هست.

حالا شما اگه به هر دوی این کلاس‌ها نیاز دارید باید هر کدوم رو جدا `import` کنید. کد زیر اشتباه هست:

```
import java.awt.*;
```

دلیل اشتباه بودنش هم اینه که در نتیجه این کد کلاس `Color`، `import` میشه ولی کلاس `ActionEvent` نه.

کد زیر درست هست:

```
import java.awt.Color;
import java.awt.event.ActionEvent;
```

## دو مثال دیگه: مرور خاطرات

### `java.util.Scanner`

احتمالا یکی از اولین برنامه‌هایی که توی جاوا نوشتین گرفتن ورودی از کاربر بوده. مثلا کد ساده زیر رو ببینین:

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        int x;
        Scanner scanner = new Scanner(System.in);
        x = scanner.nextInt();
    }
}
```



شاید قبلاً براتون سوال شده باشه که اون `import` توی خط اول چکار می‌کنه. خوب الان احتمالاً می‌تونیم به راحتی به این سوال جواب بدیم. در واقع کلاس `Scanner` متعلق به پکیج `java.util` هست و ما چون می‌خواهیم از این کلاس توی کدمون استفاده کنیم، اون رو `import` کردیم.

حتی اگه وارد سورس کد این کلاس بشین، عبارت زیر رو در خطوط ابتدایی می‌بینین:

```
package java.util;
```

## `javax.swing.JFrame`

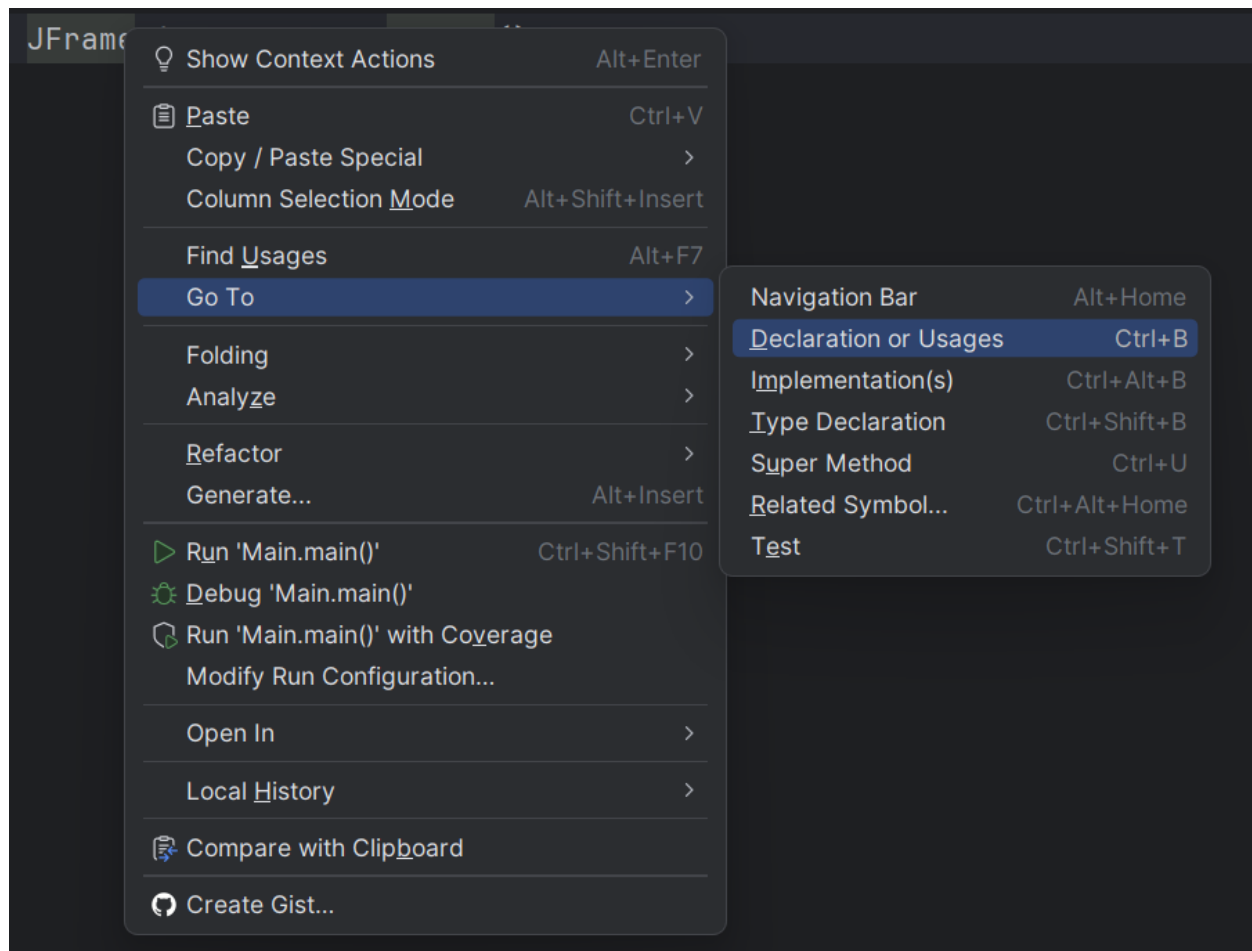
توی گرافیک، وقتی می‌خواستیم یه `frame` ایجاد کنیم، از کد زیر استفاده می‌کردیم:

```
import javax.swing.JFrame;

public class Main {
    public static void main(String[] args) {
        JFrame jFrame = new JFrame();
    }
}
```

همونطور که می‌تونیم حدس بزنیم، کلاس `JFrame` متعلق به پکیج `javax.swing` هست و چون ما می‌خواهیم از این کلاس توی کدمون استفاده کنیم، اون رو `import` کردیم.

می‌تونیم خیلی راحت این رو بررسی کنیم. هر موقع خواستین کد یک کلاس رو ببین (سورس کد جاوا)، می‌تونین روی اسم اون کلاس راست کلیک کنین و از نوار `Go To` قسمت `Declaration or Usages` رو انتخاب کنین. می‌بینین که به سورس کد اون کلاس منتقل میشین.



اگر این کار رو انجام بدین، توی فایل JFrame.java که بهش منتقل شدین، توی خطوط ابتدایی عبارت package javax.swing; رو میبینین.

## چند نکته در مورد پکیج‌ها

### هر کلاس دقیقا به یک پکیج تعلق داره

هر کلاس دقیقا به یک پکیج تعلق داره. اگر کلاسی که نوشتین رو توی یه پکیج قرار ندادین، به صورت پیش‌فرض این کلاس متعلق به default package خواهد بود. مثلا کلاس زیر رو در نظر بگیرین:

```
public class Student {
    private String name;
    private int age;

    // Some other variables and methods
}
```

دقت کنید که الان کلاس Student رو همیشه در پروژه‌ها یا کلاس‌های دیگه import و استفاده کرد؛ چون به پکیج نام‌گذاری شده‌ای تعلق نداره (default package واقعا اسم یه پکیج نیست).

تعریف نکردن پکیج برای کلاس هامون در پروژه‌های کوچک ایرادی نداره؛ ولی در پروژه‌های بزرگ حتما باید سعی کنیم که پکیج‌های مناسبی ایجاد کنیم.

همچنین یک کلاس نمیتونه به بیش از یک پکیج تعلق داشته باشه (در غیر این صورت خطای کامپایل میخوریم).

## استفاده از یک کلاس بدون import کردن اون

گاهی اوقات که فقط یک بار می‌خواهین از یک کلاس توی کدتون استفاده کنید، میتونین اون کلاس رو به طور مستقیم import نکنین و بجاش به طور کامل به اسم پکیج توی کد اشاره کنید. مثال زیر رو ببینین:

```
public class Main {  
    public static void main(String[] args) {  
        javax.swing.JFrame jFrame = new javax.swing.JFrame();  
    }  
}
```

در اینجا ما بدون استفاده از عبارت `import javax.swing.JFrame;` تونستیم از کلاس JFrame استفاده کنیم. اینجور نوشتن شاید یکم طولانی بنظر بیاد، ولی توی کلاس‌هایی که تعداد زیادی import دارن و حتی ممکنه کلاس‌هایی با اسم یکسان بخوان import بشن، به ما کمک می‌کنه این مشکلات رو حل کنیم.