

**دانشکده ریاضی و علوم کامپیوتر**

**برنامه‌سازی پیشرفته و کارگاه**

**Garbage Collection & Packages**

استاد درس

دکتر مهدی قطعی

استاد دوم

بهنام یوسفی مهر

نگارش

یونس کاظمی مقدم

زمستان ۱۴۰۳

**فهرست**

[**مقدمه** 3](#_Toc190993440)

[Garbage Collection 4](#_Toc190993441)

[آشنایی با Memory Leak 4](#_Toc190993442)

[Garbage Collection: یه راه‌حل خوب 4](#_Toc190993443)

[Garbage Collector حواسش هست! 5](#_Toc190993444)

[Packages 6](#_Toc190993445)

[نیاز به منظم کردن فایل‌ها 6](#_Toc190993446)

[پکیج چیه؟ 6](#_Toc190993447)

[چطور یک پکیج ایجاد کنیم؟ 7](#_Toc190993448)

[نام گذاری متداول پکیج‌ها (Naming Conventions) 9](#_Toc190993449)

[کلیدواژهImport 10](#_Toc190993450)

[یک مثال عملی 10](#_Toc190993451)

[در اهمیت پکیج ها 10](#_Toc190993452)

[import کردن کل پکیج 11](#_Toc190993453)

[دو مثال دیگه: مرور خاطرات 12](#_Toc190993454)

[java.util.Scanner 12](#_Toc190993455)

[javax.swing.JFrame 12](#_Toc190993456)

[چند نکته در مورد پکیج‌ها 14](#_Toc190993457)

[هر کلاس دقیقا به یک پکیج تعلق داره 14](#_Toc190993458)

[استفاده از یک کلاس بدون import کردن اون 14](#_Toc190993459)

[چه چیزی یاد گرفتیم؟ 15](#_Toc190993460)

مقدمه

توی این داکیومنت ابتدا مفهوم memory leak رو مطرح می‌کنیم. بعدش سازوکاری در جاوا به نام garbage Collection رو معرفی می‌کنیم.

بعدش سراغ package ها که مفهوم مهم و کاربردی در جاوا هستن میریم و اون‌ها رو بررسی می‌کنیم.

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 زبان های برنامه نویسی (مثل C و C++) مسئولیت «آزاد کردن» حافظه بر عهده خود برنامه نویس هست. این یعنی شما باید هر وقت که دیگه به یک شی نیاز نداشتین، خودتون اون حافظه رو آزاد کنین. مثلا 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;  
}

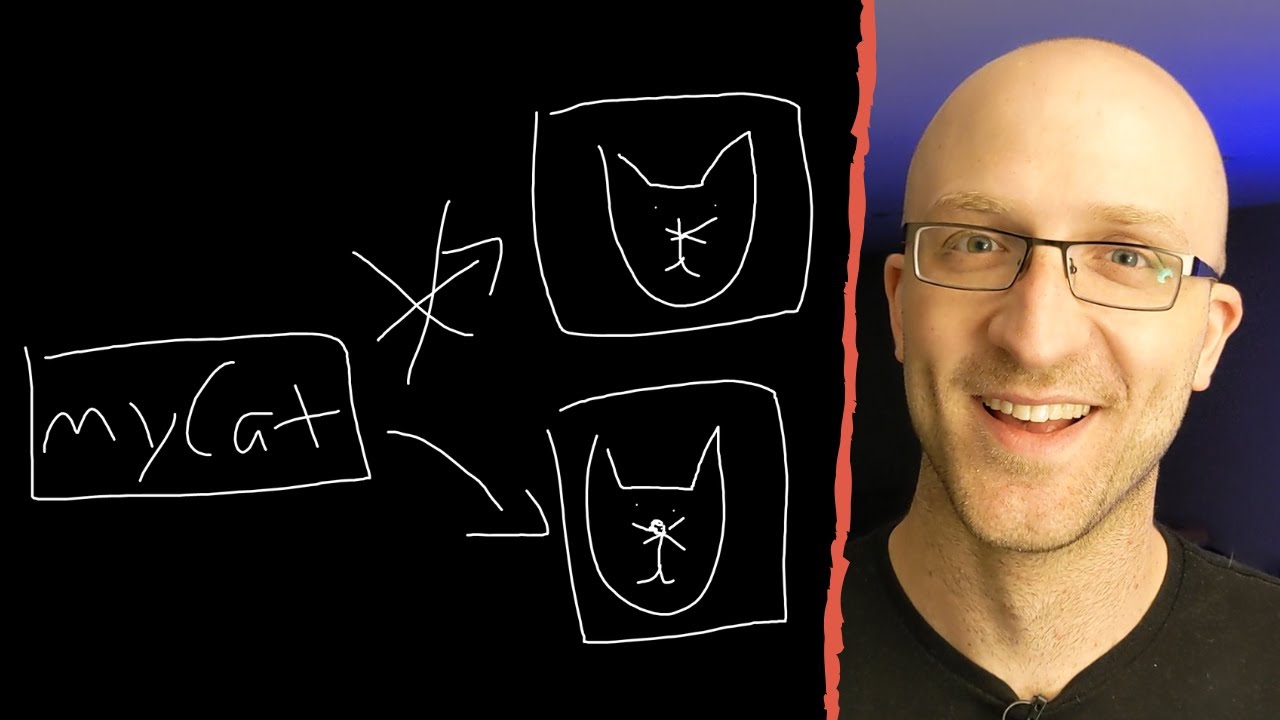
توی این برنامه که به زبان C (!) نوشته شده، اول به اندازه 5 متغیر int حافظه اشغال میکنیم. اشاره گر ptr به اولین خونه از این 20 بایت[[1]](#footnote-1) اشاره میکنه. حالا میایم 5 تا عدد صحیح رو در حافظه دخیره میکنیم. فرض کنین با این اعداد یه سری کار انجام دادیم و الان کارمون باهاشون تموم شده. اما بعد اینکه کارمون تموم شد، فراموش کردیم که این 20 بایت حافظه رو آزاد کنیم! در حقیقت باید با صدا 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 دیگر 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 راه‌حل خوب

وقتی داریم راجع به garbage collection در جاوا صحبت می‌کنیم، به زبان ساده یعنی جاوا خودش می‌ره و حافظه‌ای که دیگه به هیچ کار نمی‌آد رو آزاد می‌کنه. مثلا 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 اصلا 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](https://youtu.be/Mlbyft_MFYM?si=jhAdnlq12houCo1F)  و این [داک اوراکل](https://docs.oracle.com/en/java/javase/21/gctuning/introduction-garbage-collection-tuning.html) 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 با آبجکت cuteCat رو بگین؟

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

Garbage Collector حواسش هست!

Garbage Collector آبجکت هایی که هنوز بهشون نیاز داریم (بهشون رفرنس داریم) رو پاک نمیکنه. مثلا بیاین 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() {  
 Duck localDuck = new Duck();  
 localDuck.name = "A White Duck";  
 return localDuck;  
 }  
}

توی این کد اومدیم اول یه کلاس خیلی ساده به اسم Duck تعریف کردیم. یه متد هم به اسم createDuck تعریف کردیم که اول میاد یه 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؟ احتمالا توی درس های قبلیتون خوندین که *«متغیر هایی 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 آبجکت ها 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

Packages

نیاز به منظم کردن فایل‌ها

در برنامه های که توی جاوا می نویسیم، همیشه از کلاس ها یا اینترفیس ها[[2]](#footnote-2) استفاده می کنیم. مثلا برنامه ساده زیر که در کلاس Sample نوشته شده رو ببینین:

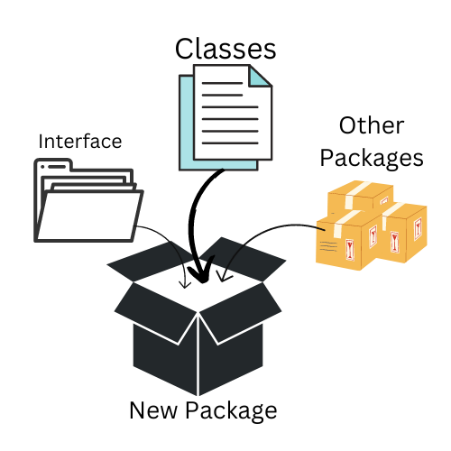
public class Sample {  
 public static void main(String[] args) {  
 System.*out*.println("Hello World!");  
 }  
}

ولی برنامه های پیچیده تر ممکنه از صد ها کلاس تشکیل شده باشن. اگه همه این کلاس ها رو بدون هیچ نظمی کنار هم قرار بدیم باعث میشه برنامه مون ناخوانا باشه و خودمون هم گیج می شیم.

احتمالا یکی از اولین چیز هایی که برای منظم کردن فایل ها به ذهنمون میرسه استفاده از پوشه هاست. و این دقیقا همون امکانیه که جاوا برای منظم کردن کلاس های برنامه مون واسه ما فراهم کرده: ایجاد package های مختلف.

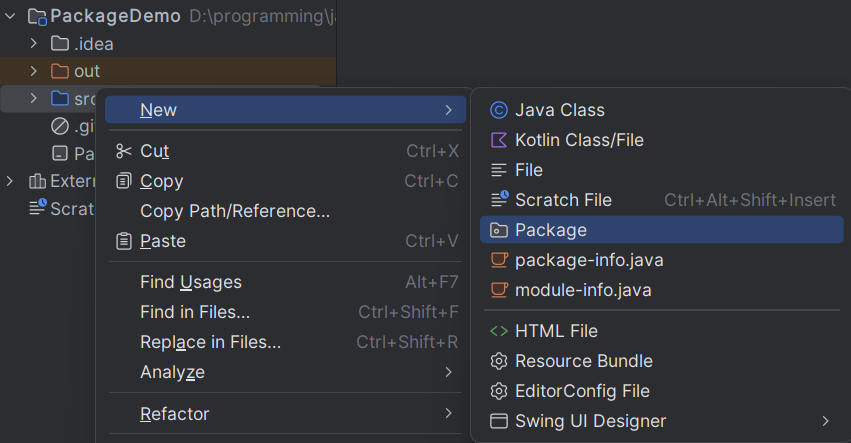
پکیج چیه؟

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



چطور یک پکیج ایجاد کنیم؟

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



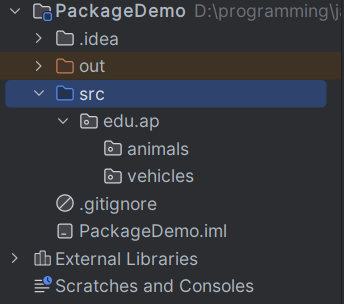
حالا اسم پکیج رو وارد کنین. مثلا در اینجا edu.ap.animals

A black screen with white text

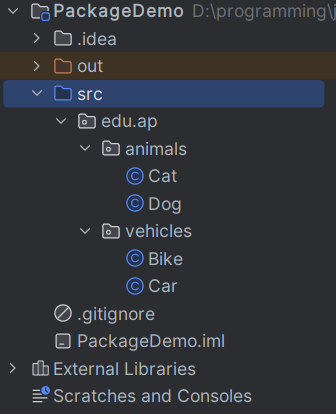
Description automatically generated

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

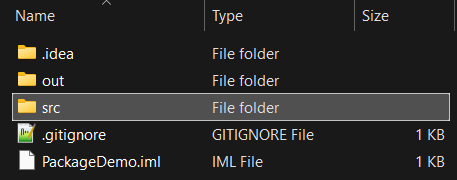
الان باید چیزی شبیه به تصویر زیر رو توی پوشه src داشته باشین:

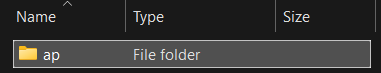


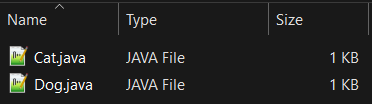
حالا بیاین چند تا کلاس به هر کدوم از این پکیج ها اضافه کنیم. مثلا کلاس های Dog و Cat رو به پکیج animals و کلاس های Bike و Car رو به پکیج vehicles اضافه کنیم. در نهایت باید چیزی شبیه به تصویر زیر رو داشته باشین:



پکیج‌هامون رو ساختیم! حالا ببینیم واقعا چه فایل هایی ایجاد شده. اگه از توی explorer به محل ایجاد پروژه‌تون برین و به ترتیب وارد پوشه های src، edu، ap و animals بشین چیزی شبیه به این ها می بینین:

A screen shot of a computer

Description automatically generatedA screen shot of a computer

Description automatically generated

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

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.

نام گذاری متداول پکیج‌ها (Naming Conventions)

1. پکیج ها 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 کاراکتر های خاص (مثل @،$،&) استفاده نکنین. در نامگذاری پکیج ها فقط مجازیم از dot (.) استفاده کنیم.
   1. نادرست: com.github.my-awesome-project
   2. درست: com.github.myawesomeproject
4. یک قاعده دیگه در نامگذاری پکیج ها، reverse domain name هست (برعکس نوشتن نام دامنه). اگه یه پروژه ای 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 پکیج ها اسمش org.mozilla.javascript هست (البته این قاعده در مورد پکیج های استاندارد خود جاوا صدق نمیکنه).

کلیدواژهImport

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 کلاس هایی 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 یه آبجکت از کلاس Cat بسازیم.

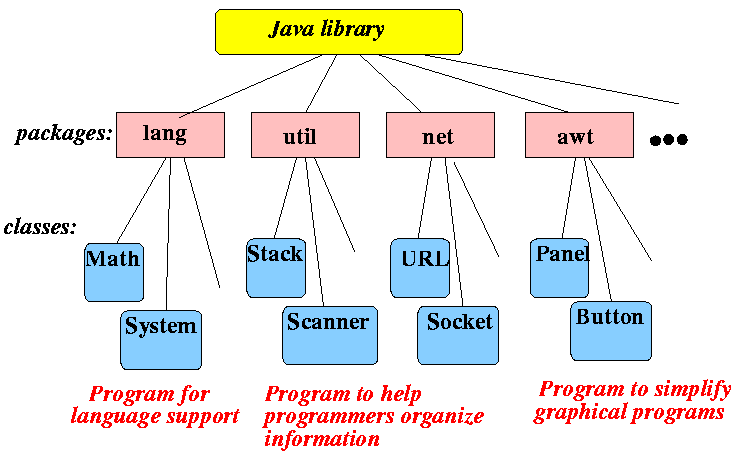
در اهمیت پکیج ها

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

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

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

تو شکل زیر میتونین یه نمای کلی از پکیج های جاوا ببینین:



import کردن کل پکیج

شما میتونین تمام کلاس های موجود توی یک پکیج رو یکجا import کنین. این کار رو با عبارت \* میتونین انجام بدین. مثلا کد زیر تمام کلاس های 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 کلاس ها علاوه بر خوانایی بیشتر کدمون، کمی هم زمان کامپایل‌مون رو کمتر میکنه.

دقت کنین که پکیج ها خودشون میتونن شامل پکیج باشن؛ همونطوری که پوشه ها میتونن داخل خودشون پوشه داشته باشن. اما عبارت \* فقط کلاس های متعلق به پکیج رو import میکنه و نه sub-package ها رو (import کردن recursive نداریم).

مثلا در نظر بگیرین 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 کلاس ها نیاز دارین باید هر کدوم رو جدا import کنین. 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 برنامه هایی 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 قبلا براتون سوال شده باشه که اون import توی خط اول چکار میکنه. خوب الان احتمالا میتونیم به راحتی به این سوال جواب بدیم. در واقع کلاس Scanner متعلق به پکیج java.util هست و ما چون میخوایم 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 می بینین:

package java.util;

javax.swing.JFrame

توی گرافیک، وقتی میخواستیم یه frame ایجاد کنیم، از کد زیر استفاده می کردیم:

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 میخوایم از این کلاس توی کدمون استفاده کنیم، اون رو import کردیم.

می تونیم خیلی راحت این رو بررسی کنیم. هر موقع خواستین کد یک کلاس رو ببین (سورس کد جاوا)، می تونین 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.

A screenshot of a computer

Description automatically generated

اگه 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. اگه 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 کنین که الان کلاس Student رو نمیشه در پروژه ها یا کلاس های دیگه import و استفاده کرد؛ چون به پکیج نام‌گذاری شده ای تعلق نداره (default package واقعا اسم یه پکیج نیست).

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

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

استفاده از یک کلاس بدون import کردن اون

گاهی اوقات که فقط یک بار میخواین از یک کلاس توی کدتون استفاده کنین، میتونین اون کلاس رو به طور مستقیم import نکنین و بجاش به طور کامل به اسم پکیج توی کد اشاره کنین. 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 کلاس هایی که تعداد زیادی import دارن و حتی ممکنه کلاس هایی با اسم یکسان بخوان import بشن، به ما کمک میکنه این مشکلات رو حل کنیم.

چه چیزی یاد گرفتیم؟

Memory Leak یکی از مشکلاتی هست که توی زبان‌های برنامه‌نویسی مثل C و C++ اتفاق می‌افته. یعنی وقتی حافظه‌ای که دیگه استفاده نمی‌شه، آزاد نمی‌شه. توی جاوا، این مشکل به کمک GarbageCollection حل شده،به طوری که حافظه‌ای که دیگه به اون نیازی نیست رو به شکل خودکار پاک می‌کنه.

Garbage Collector در جاوا به‌طور هوشمند آبجکت‌هایی که دیگه به اون‌ها رفرنس نداریم ( به‌عبارت ‌دیگه به اون‌ها نیاز نداریم) رو از حافظه پاک می‌کنه. این ویژگی به ما کمک می‌کنه تا بدون نگرانی از مدیریت حافظه، تمرکز بیشتری روی منطق برنامه مون داشته باشیم.

Package‌ ها در جاوا ابزاری برای سازمان‌دهی کدها و منظم کردن پروژه‌ها هستن. این ابزار به ما کمک می‌کنه تا کلاس‌ها و اینترفیس‌های مختلف رو در پکیج‌های مختلف قرار بدیم و کدهای خودمون رو خواناتر کنیم.

همچنین با استفاده از import می‌تونیم کلاس‌ها یا پکیج‌ها رو به پروژه مون وارد کنیم و از اون‌ها استفاده کنیم. این ویژگی امکان استفاده از کدهای موجود در بقیه پکیج‌ها رو فراهم می‌کنه.

1. با این فرض که هر متغیر از جنس int، حافظه ای به اندازه 4 بایت رو اشغال کنه. [↑](#footnote-ref-1)
2. با اینترفیس‌ها بعدا آشنا میشین. فعلا کاری بهشون نداریم. [↑](#footnote-ref-2)