

# بخش چهارم: اشیاء در جاوا

## سرفصل مطالب

در این فصل در مورد اشیاء در جاوا صحبت خواهیم کرد و جزئیاتی بیشتری در مورد آنها خواهیم دید.

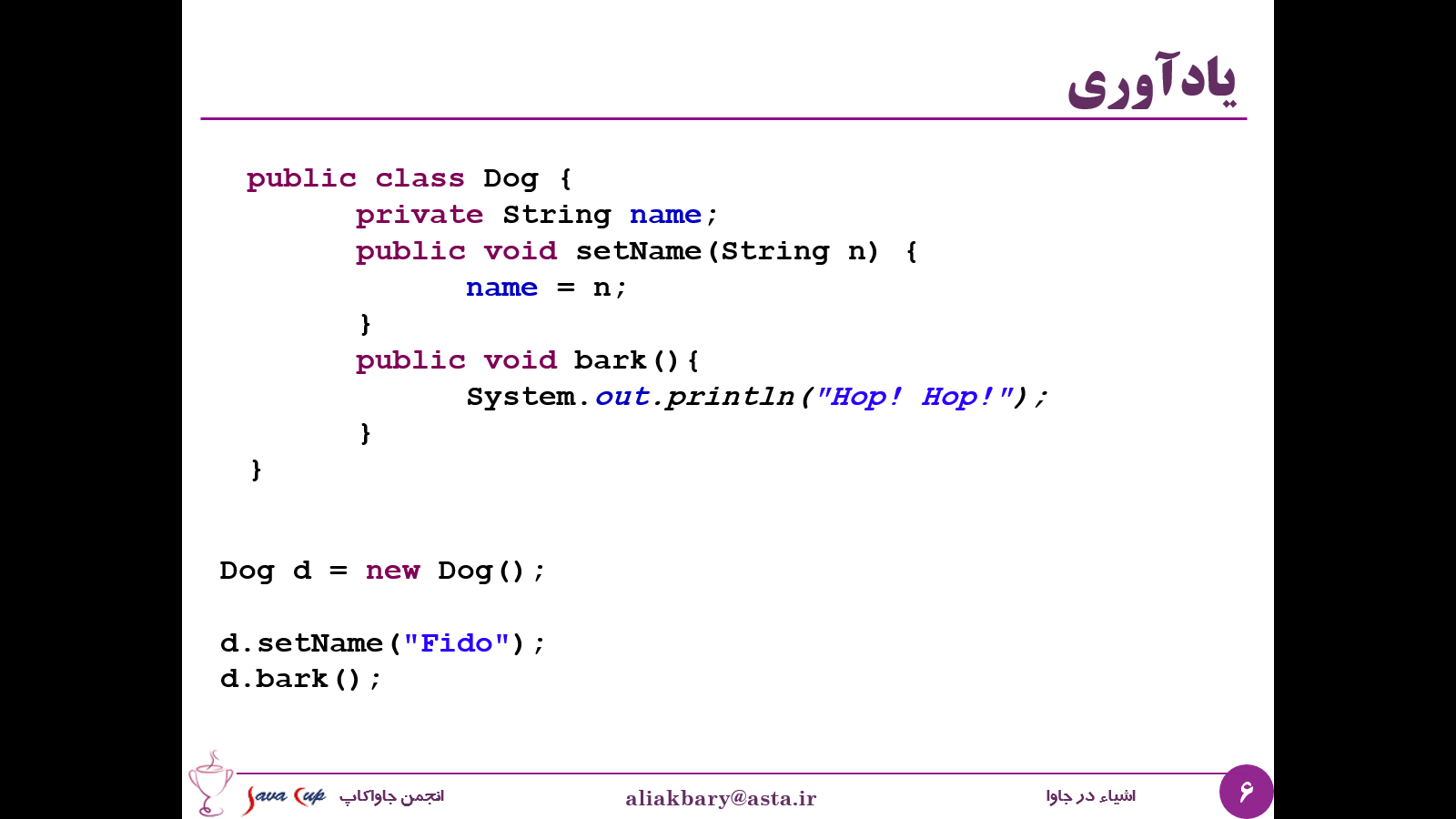
موضوعات مهمی که در این فصل در مورد آنها صحبت خواهیم کرد به شرح زیر است:

* ایجاد کردن اشیاء
* وضعیت اشیاء در حافظه
* نحوه ی آزاد شدن اشیا توسط زباله روب (Garbage Collector)
* ارسال پارامترها به متدها (Parameter Passing)، روش های مرسوم این کار و رویکرد جاوا در آن
* بخش های مختلف حافظه و نحوه ی استفاده یک برنامه در جاوا از آنها

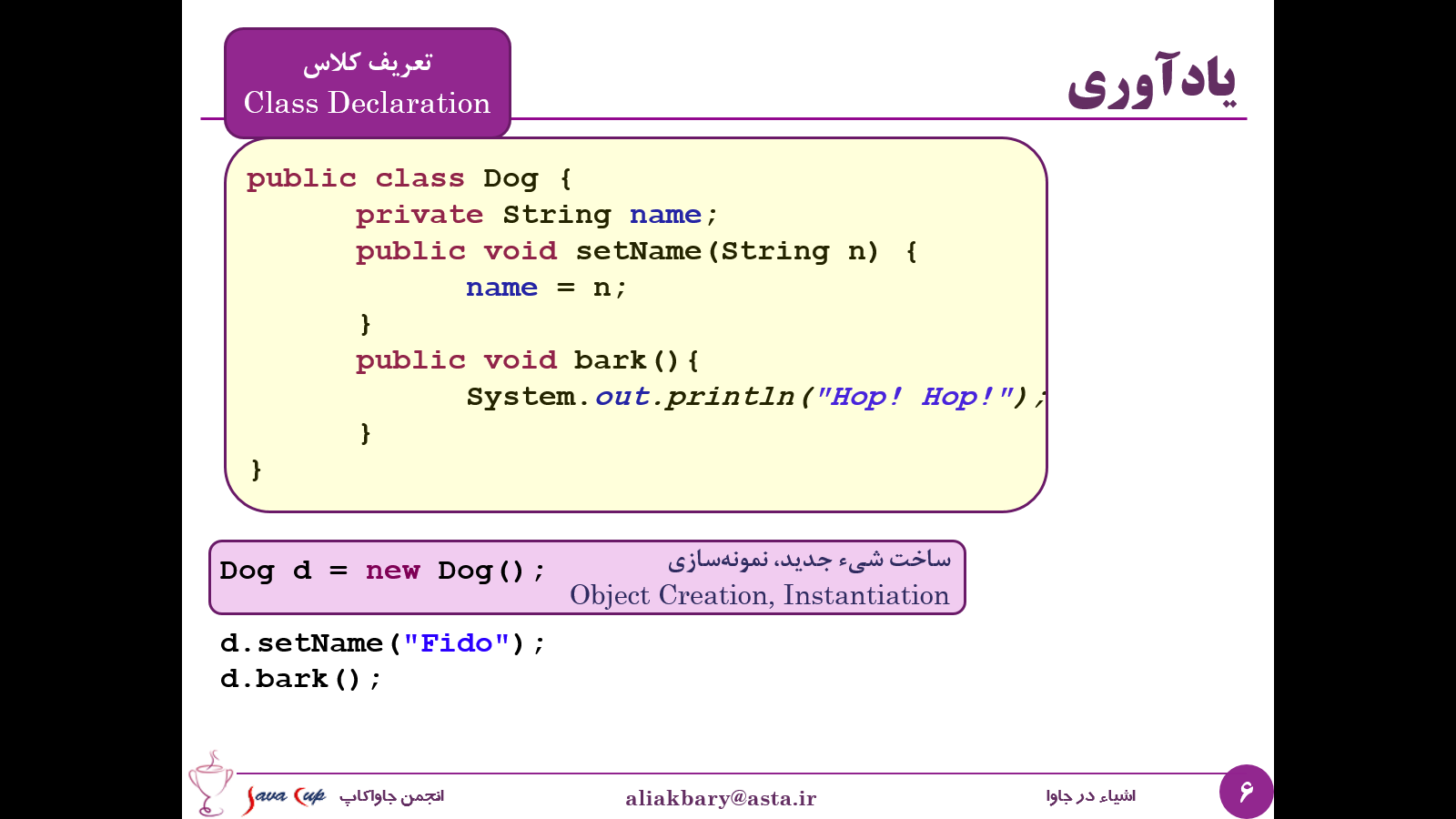
یکی از اهداف این فصل بررسی و ارائه یک شهود نسبتا سطح پایین در مورد وضعیت اشیاء در حافظه است. در واقع ما مرور می کنیم که نحوه ی ذخیره سازی اشیاء و ارجاع ها در جاوا چگونه خواهد بود. این شهود سطح پایین در کنار شهود سطح بالا مهم است. در واقع شهود سطح بالا برای تحلیل و طراحی بسیار مهم است و شهود سطح پایین برای پیاده سازی. به همین دلیل در این فصل اندکی سطح پایین تر فکر می کنیم و در مورد وضعیت اشیاء در حافظه صحبت خواهیم کرد.

## اشیاء در برنامه های جاوا

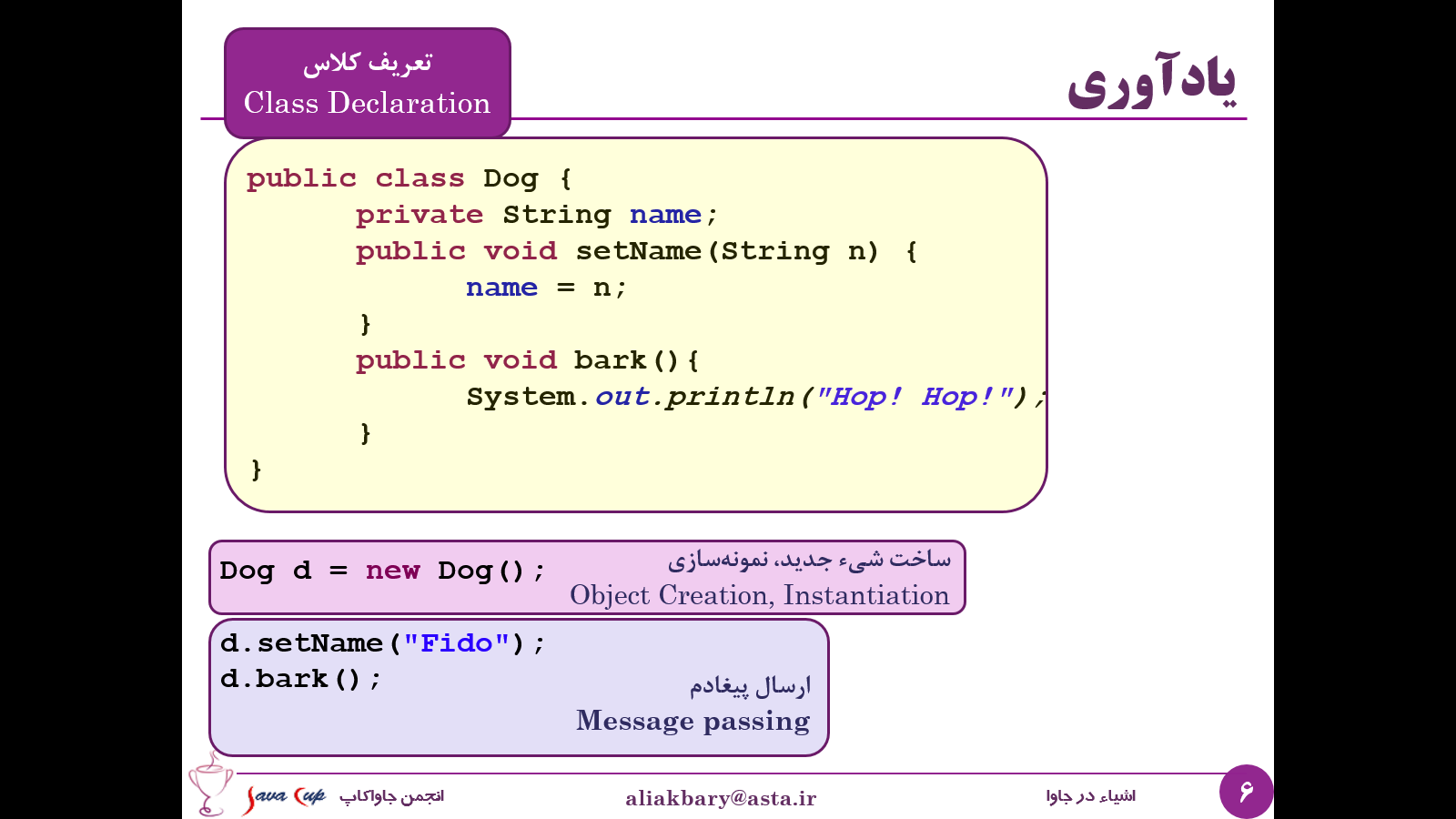
در ابتدا به عنوان یادآوری چیزهایی را مرور کنیم. فرض کنید در فضایی برنامه نویسی می کنیم که سگ ها وجود دارند، پس همانطور که انتظار دارید کلاسی برای سگ ها ایجاد می کنیم، فرض کنید که هر سگ ویژگی مانند اسم دارد و در نتیجه رفتاری برای تعیین این ویژگی خواهد داشت، همچنین رفتاری برای پارس کردن برای آن ها قرار می دهیم:



به کدی که برای تعریف این کلاس نوشته ایم Class Declaration گفته می شود. حال چطور می شود از این کلاس استفاده کرد؟



و وقتی یک شی از این کلاس ساختیم، می توانیم با آن کارهایی انجام بدهیم، و رفتارهای آن را فرخوانی کنیم:

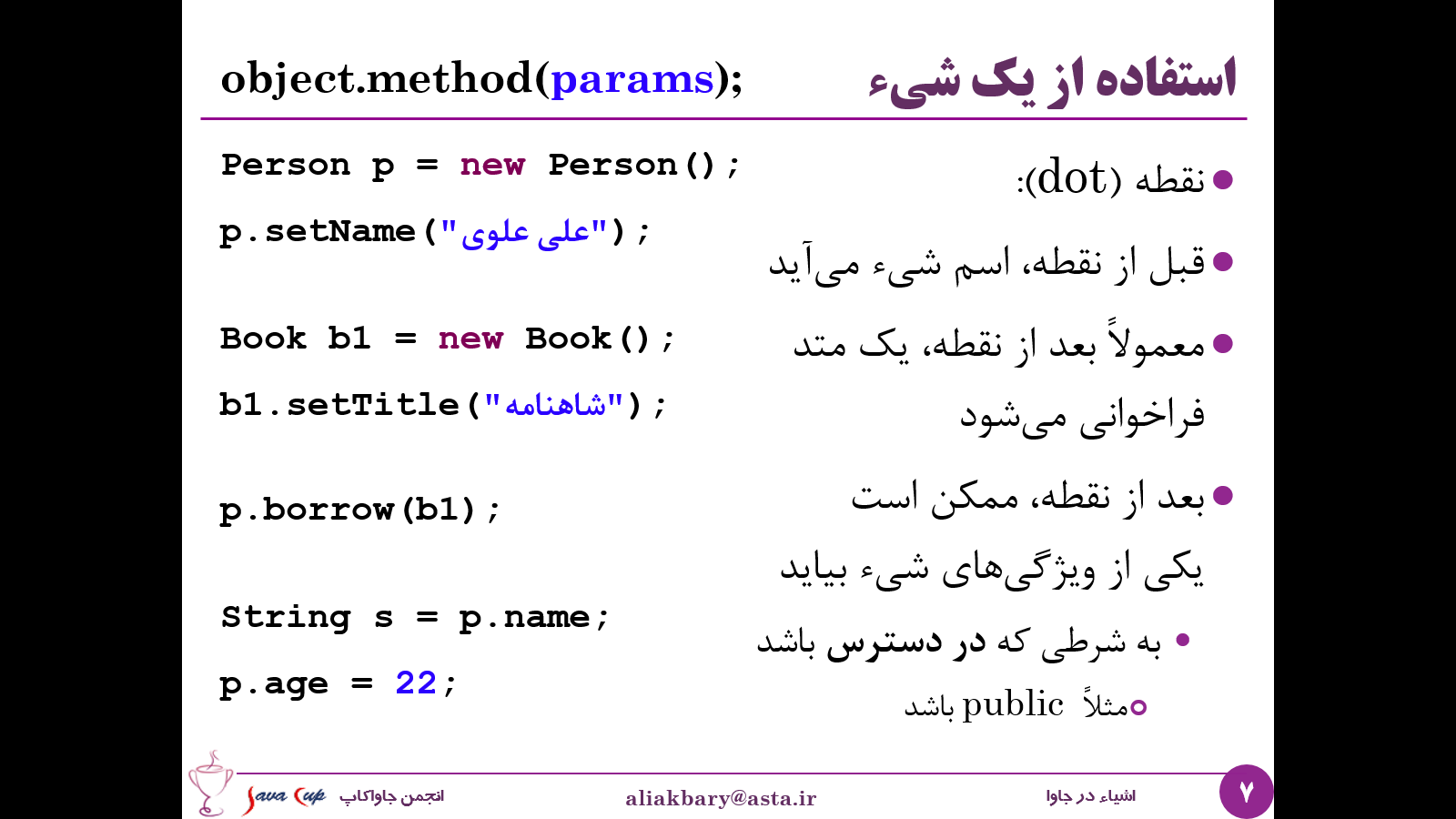


برخی متدها موجب تغییر حالت شی می شوند مانند متد setName در این مثال که ویژگی نام هر سگ را تغییر خواهد داد. به حالت شی اصطلاحا state شی هم گفته می شود.

سوالی که در اینجا می خواهیم به آن بپردازیم این است که d چیست؟ این شی جدید در واقع چگونه متغیری است؟ اگر بخواهیم دقیق صحبت کنیم، d یک ارجاع به یک شی است (Reference). در مورد ارجاع در این فصل بیشتر صحبت خواهیم کرد.

یک مثال دیگر را در نظر بگیرید، فرض کنید که اشیاءی از جنس کتاب و اشخاص داریم و قرار است برای فضای مساله ی کتابخانه و امانت گرفته شدن کتاب برنامه نویسی کنیم. همانطور که قبلا هم دیدیم می توانیم با کمک نقطه بعد از نام یک شی رفتارهای آن را فراخوانی کنیم. پس ابتدا نام شی خواهد آمد، سپس نقطه و در ادامه نام متدی که قصد فراخوانی آن روی شی مورد نظر را داریم.

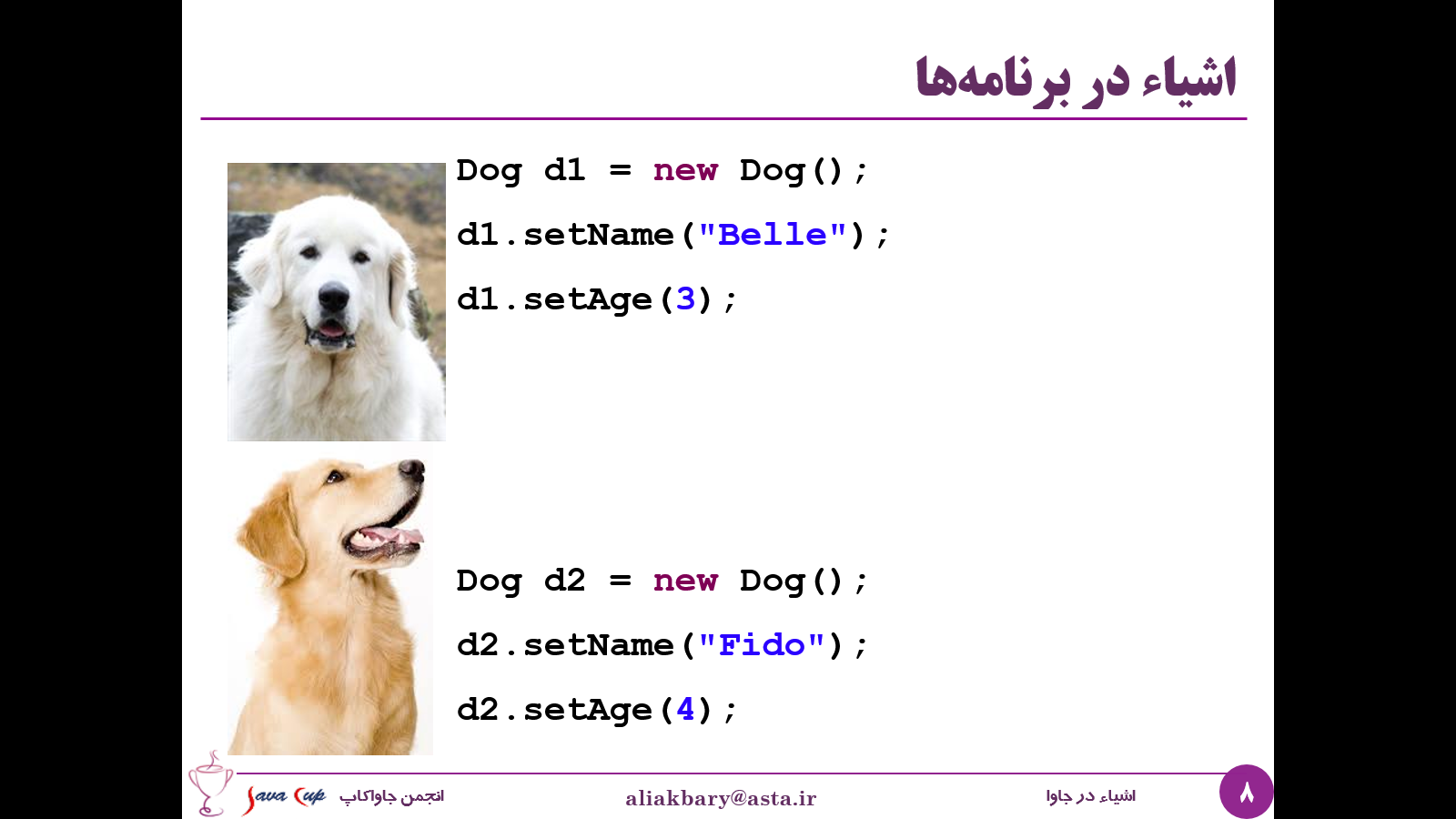
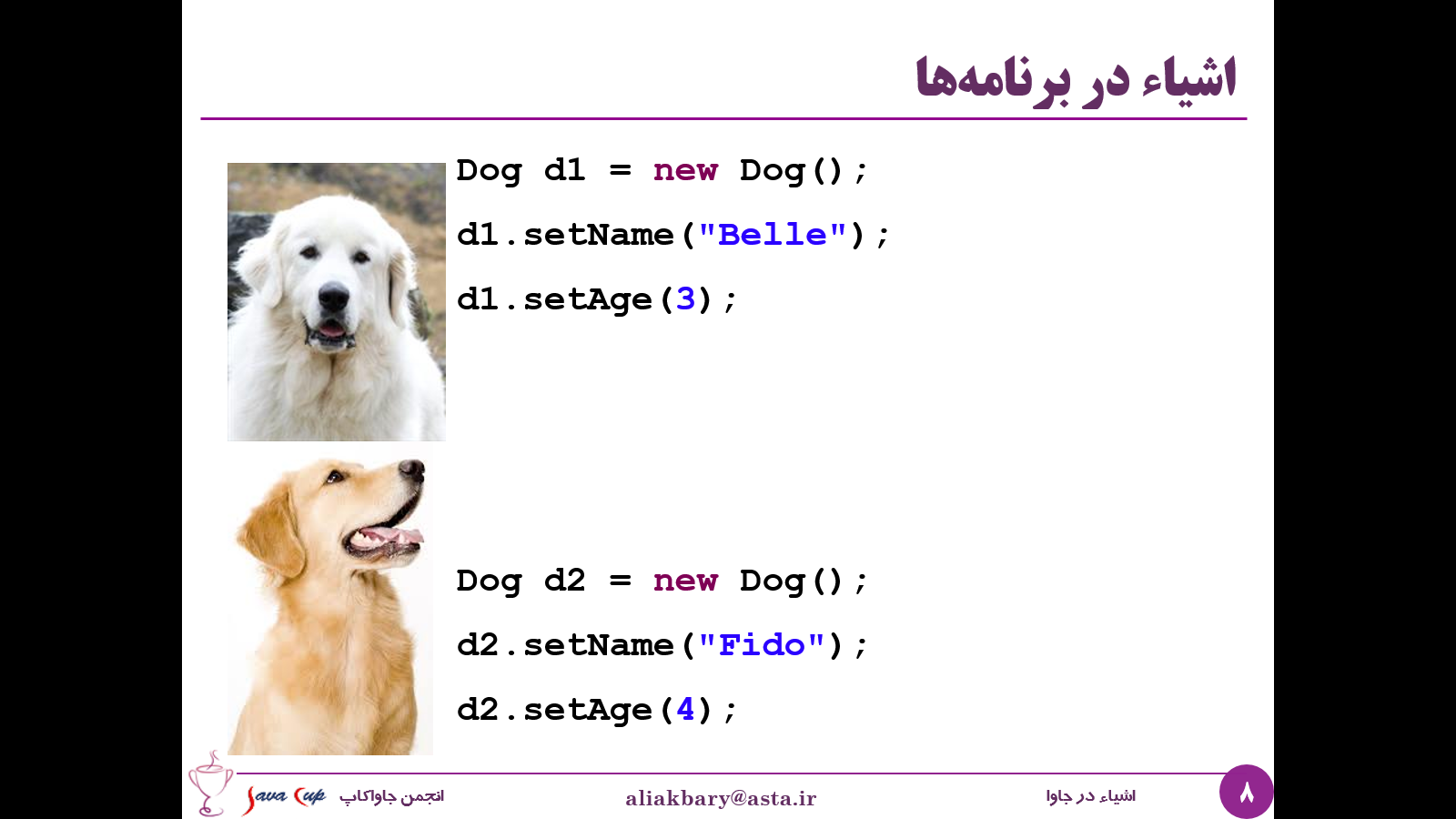
**object.method(params);**



در این مثال فرض کنید که ما یک کتاب و یک عضو داریم و این عضو قصد دارد کتاب را به امانت بگیرد. پس یک عضو با نام علی علوی تعریف می کنیم، سپس یک کتاب با نام شاهنامه و متد امانت گرفتن کتاب را روی علی علوی فراخوانی کرده و کتابی که می خواهد به امانت بگیرد را به آن پاس می کنیم.

البته گاهی بعد از نقطه به جای یک پارامتر یکی از ویژگی های شی قرار می گیرد، مثلا p.name به معنای نام آن عضو کتابخانه است و می توانیم آن را داخل یک متغیر از جنس رشته بریزیم و یا می توانیم مقدار سن این شی را تغییر دهیم اما انجام این کارها یک شرط دارد و آن اینکه ویژگی هایی که بعد از نقطه می آیند باید قابل دسترسی و مثلا public باشند. هر چند بعدا خواهیم دید که این روش کمتر معمول است و برای تغییر ویژگی های یک شی معمولا از متدهایی مانند setName که به آنها setter گفته می شود استفاده می کنیم.

برگردیم به مثال سگ ها، فرض کنید که در دنیای واقعی دو سگ داشته باشیم و بخواهیم برنامه ای بنویسیم که این دو سگ را بازنمایی کند، می توانیم برای هر کدام از آنها یک شی بسازیم:



با نوشتن این برنامه ما حالت که در اینجا نام و سن این اشیاء هست را مشخص کرده ایم. وقتی برنامه ما اجرا می شود، هر کدام از این اشیاء در حافظه در جایی قرار می گیرند. مثلا به d1 بخشی از حافظه اختصاص داده می شود و تمام ویژگی ها و حالات این شی در آن ذخیره می شود و عملا d1 ارجاعی به آن بخش از حافظه خواهد بود.

## اشیاء در حافظه

اگر یادتان باشد قبلا دیدیم که هر شی دارای حالت، رفتار و هویتی است. State شی وضعیت و مقدار ویژگی های شی را نشان می دهد و behavior شی رفتارهایی هست که می توانیم روی آن شی فراخوانی کنیم و هویت یا identity شی چیزی است که آن شی را از اشیاء دیگر متمایز می کند.

محتوای حافظه ای که برای هر شی اختصاص داده می شود عملا حالت آن شی را نشان می دهد. مثلا شی d1 محتوای حافظه ای دارد که شامل اسم آن سگ (Belle) و سن آن (3) است، مجموعه ی این ویژگی ها که خودشان هم می توانند شی ای مستقل باشند، وضعیت شی d1 را نشان می دهد و مجموعه ویژگی های دیگری که در مکان دیگری از حافظه قرار دارد شی d2 را مشخص می کند.

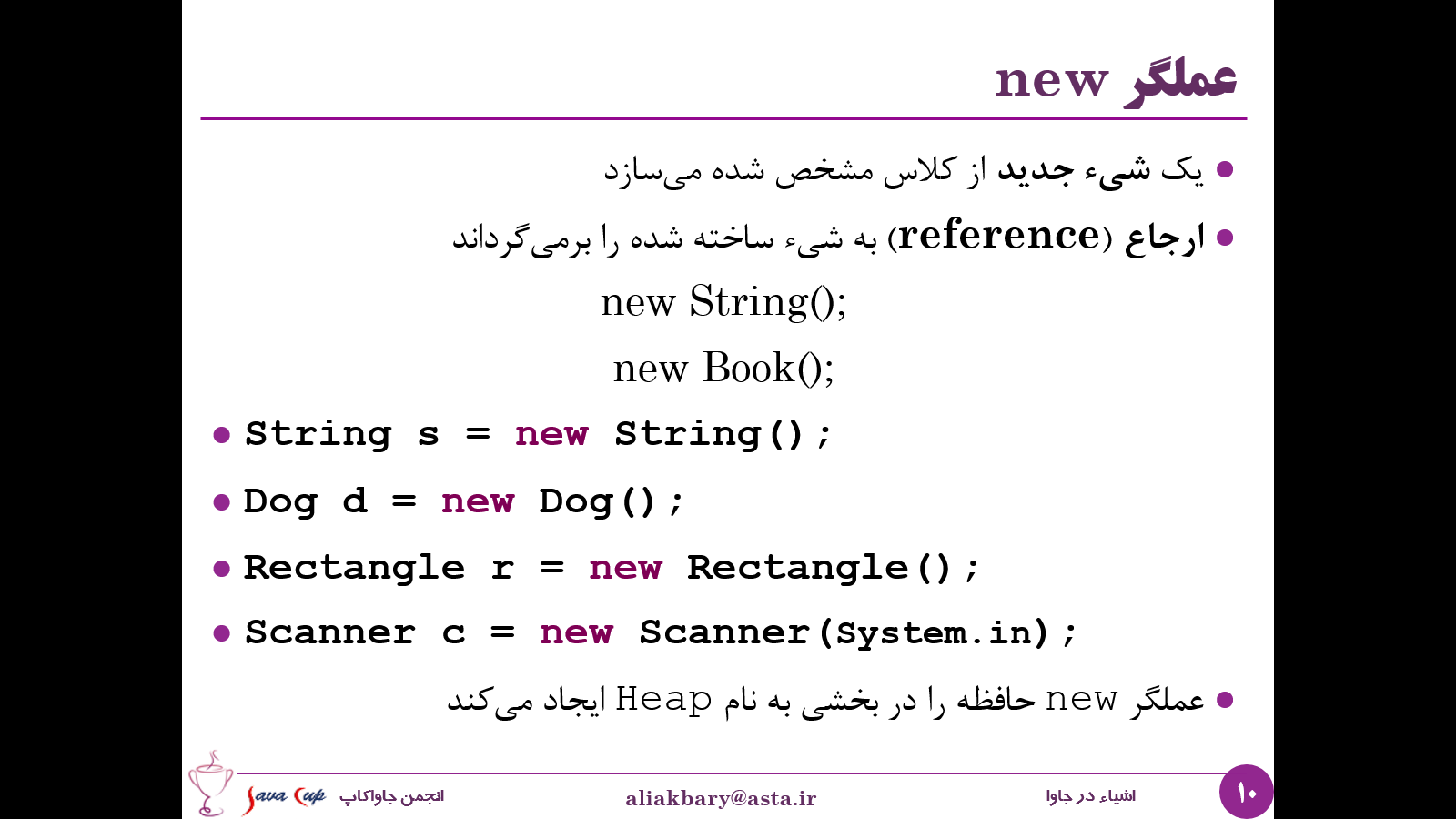
هویت هر شی کجاست و چگونه مشخص می شود؟ می توان با کمی تساهل اینگونه گفت که هویت هر شی تقریبا معادل آدرس آن شی در حافظه است. پس حتی اگر دو شی d1 و d2 دارای مقادیر یکسانی برای ویژگی ها باشند، (اسم و سن یکسان) هویت و مکان قرار گیری آنها در حافظه متفاوت است و دو شی متفاوت هستند. مشابه چیزی که در واقعیت وجود دارد، و اگر ما دو سگ داشته باشیم که اسم و سن آنها یکی باشد، باز هم دو سگ متفاوت با دو هویت متفاوت هستند، پس حتی اگر حالت دو شی مساوی باشد، هویت آنها لزوما یکی نیست.

از بخش های هر شی، بخش رفتار باقی ماند، رفتار هر شی کجا هست؟ رفتار هر شی یک بار وقتی که کلاس آن را تعریف می کردیم مشخص شده است، یعنی ما وقتی کلاس Dog را مشخص ساختیم، مشخص کردیم که تمام اشیایی که از این کلاس ساخته خواهند شد چه رفتاری می توانند داشته باشند. پس لازم نیست برای هر شی به صورت مستقل رفتار در حافظه قرار بگیرد بلکه یک بار هنگام تعریف کلاس رفتارهای آن کلاس تعریف شده است.

## عملگر new

عملگر new چه کار می کند؟ این موضوع را از نمای نزدیک تر ببینیم، این عملگر یک شی جدید از کلاسی که ما مشخص کرده ایم ایجاد می کند، وقتی این شی را ایجاد کرد، یک ارجاع و reference که به نوعی یک اشاره گر هست به این شی جدید برمی گرداند، پس این عملگرد دو کار انجام می دهد، هم یک شی جدید می سازد و هم اشاره گر و یا ارجاع به آن شی را برمی گرداند.

یعنی هنگامی که ما می گوییم new String() یک شی جدید از کلاس String در حافظه ایجاد می شود، یک فضای جدید به آن اختصاص داده می شود و ارجاع به این بخش از حافظه برگشت داده می شود و می توانیم آن را داخل متغیری بریزیم. یا هنگامی که می نویسیم new Book() یک شی Book ایجاد می کند و ارجاع آن را برمی گرداند. پس ما هر وقت چیزی را new می کنیم، ارجاع آن را می توانیم داخل یک متغیر بریزیم.



مثلا در کد بالا s ارجاعی به یک شی از جنس String است. و یا d به شی ای از جنس کلاس Dog اشاره می کند و یا همان طور که قبلا دیده بودیم، می توانیم شی ای از جنس کلاس Scanner بسازیم. الان دیگر دقیق تر می دانیم که کلاسی به نام Scanner وجود دارد که ما می توانیم شی جدیدی از آن بسازیم و ارجاع به آن را داخل متغیری به نام c قرار بدیم.

داخل پرانتر اشاره می کنم که عملگر new از بخشی از حافظه به نام Heap استفاده می کند برای تخصیص حافظه ی جدید به اشیاء، به عبارت دیگر هر برنامه از بخشی از حافظه به نام هیپ می تواند استفاده کند و این عملگر از این بخش استفاده می کند و اشیاء جدید را در این بخش قرار می دهد. در مورد هیپ در ادامه مفصل صحبت می کنیم.

## انواع داده ها

یک نکته ی خیلی مهم این است که در جاوا دو گونه ی نوع داده وجود دارد. اول گونه ی انواع داده ی اولیه یا Primitive Data Types و دوم انواع داده ی ارجاعی و یا Reference Data Types.

* انواع داده ی اولیه:

انواع داده ی اولیه را قبلا با هم دیده ایم، یک تعداد مشخص و محدود از انواع داده هستند که در زبان جاوا تعریف شده اند مانند انواع داده ی عددی، بولین ها و کاراکترها.

* + byte, short, int, long, float, double, boolean, char

که اینها یک بار برای همیشه در جاوا تعریف شده اند و ما نمی توانیم نوع داده ی اولیه ی دیگری برای جاوا تعریف کنیم. نکته ی مهم این است که هر متغیر از این انواع داده ی اولیه حاوی یک مقدار است و نه یک شی، یعنی ارجاع به یک شی نیست. پس تمام متغیرها در جاوا شی نیستند، و انواع داده ی اولیه در واقع یک مقدار را نگه می دارند و نه یک ارجاع به یک شی را. مثلا int number یک متغیر است که یک عدد را نگه می دارد و نه یک ارجاع به یک عدد. همین طور char c یک مقدار کاراکتر را نگه می دارد و نه ارجاع به یک کاراکتر را.

* انواع داده ی ارجاعی:

اما انواع داده ی ارجاعی یا کلاس ها مثل String و یا Dog و یا Scanner، یعنی چه کلاس هایی که در جاوا موجود هستند، و چه کلاس هایی که ما ایجاد می کنیم، انواع داده ای هستند که هر متغیر از آنها ارجاع به یک شی است. مثلا String name یک ارجاع به شی از جنس String خواهد بود و خودش یک مقدار را نگه نمی دارد بلکه ارجاع به یک String است. و یا Book b ارجاعی به شی از جنس Book خواهد بود.

## تفاوت یک متغیر اولیه با یک شی

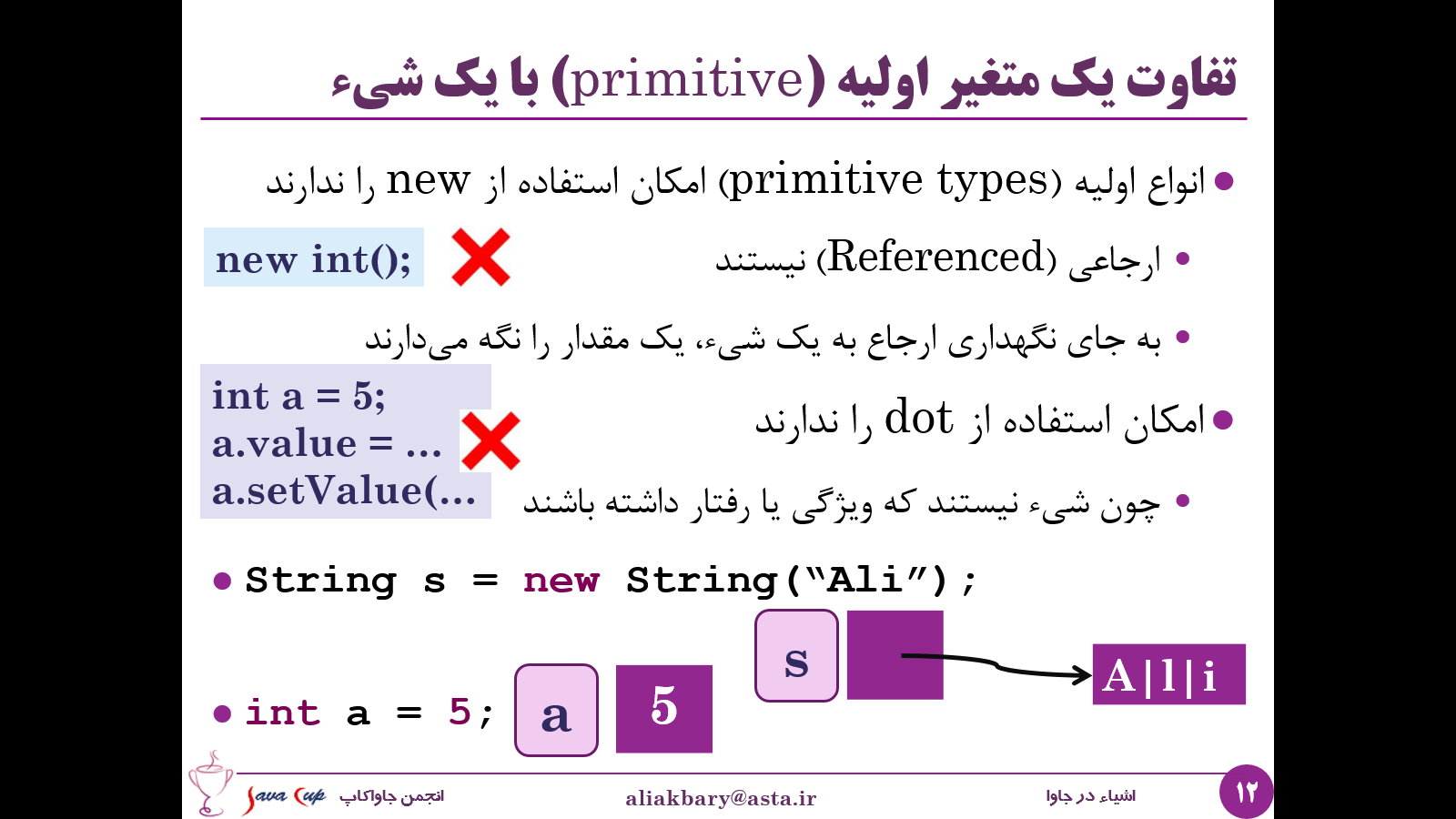
یک تفاوت مهم بین متغیرهای Primitive با اشیاء وجود دارد، اگر متغیر ما از جنس Primitive Data Type باشد، نمی تواند از اپراتور new استفاده کند، چون اصلا ارجاعی نیست که بخواهد به شی حاصل از استفاده از new اشاره بکند، و به جای نگهداری ارجاع به یک شی خود مقدار را نگهداری می کند. پس ما نمی توانیم بگوییم new int() چون int اصلا کلاس نیست که ما بخواهیم آن را new کنیم.

همچنین متغیرهایی از جنس انواع داده ی اولیه، امکان استفاده از dot را ندارند، چون شی نیستند که ویژگی و یا رفتاری داشته باشند که ما بتوانیم بعد از نقطه آنها را فراخوانی کنیم.

پس اگر بعد از انواع داده ی اولیه از dot استفاده کنید، خطای کامپایل از کامپایلر جاوا دریافت خواهید کرد.

حال از نمای نزدیک تر ببینیم هنگامی که می گوییم String s = new String("Ali"); دقیقا چه اتفاقی در حافظه رخ می دهد؟ در این حالت ما در حافظه دو بخش خواهیم داشت، یک بخش که شی جدید ما را نگهداری می کند، همان طور که گفتیم اپراتور new بخشی از حافظه هیپ را به شی جدید ما که در اینجا مقدار Ali دارد اختصاص می دهد، و ارجاع به این شی برمی گردد و ما آن را داخل متغیری به نام s ریخته ایم، پس عملا دو بخش در حافظه هست، یکی s که ارجاع به شی ماست و دیگری محلی از هیپ که خودش ویژگی های شی ما در آن ذخیره شده اند.

اما اگر یک متغیر Primitive تعریف کنیم مثلا int a=5; ، دیگر دو بخش در حافظه نداریم، فقط یک بخش برای متغیر a قرار می گیرد که مستقیما مقدار 5 را ذخیره می کند و ارجاعی به جای دیگری از حافظه نیست.



## مفهوم ارجاع (Reference)

در ادامه می خواهیم در مورد مفهوم ارجاع بیشتر صحبت کنیم. ارجاع مفهومی مانند اشاره گر (Pointer) دارد. در واقع هر متغیری که به شکل ارجاع تعریف می شود قرار است به یک شی اشاره کند، اگر با مفهوم اشاره گر در زبان C یا C++ آشنایی دارید، مفهوم ارجاع در جاوا مشابه مفهوم اشاره گر در زبان C و C++ است. اگر هم آشنا نیستید، اهمیتی ندارد و با هم با آن آشنا خواهیم شد ☺

وقتی که یک متغیر تعریف می کنیم از جنس یک کلاس، در واقع داریم یک ارجاع می سازیم، مثلا String s موجب ساختن یک شی نمی شود، بلکه s تنها یک ارجاع است. همین طور Book b موجب ایجاد یک ارجاع به یک شی از جنس Book می شود و یک شی نیست.

و دیدیم که استثناء در این قضیه، انواع داده ی اولیه هستند که متغیری از جنس آنها نه شی است و نه ارجاع به یک شی و یک مقدار را مستقیما نگهداری می کنند.

## ساخت اشیاء

گفتیم که وقتی کد زیر را می نویسیم str یک شی جدید ایجاد نمی کند:

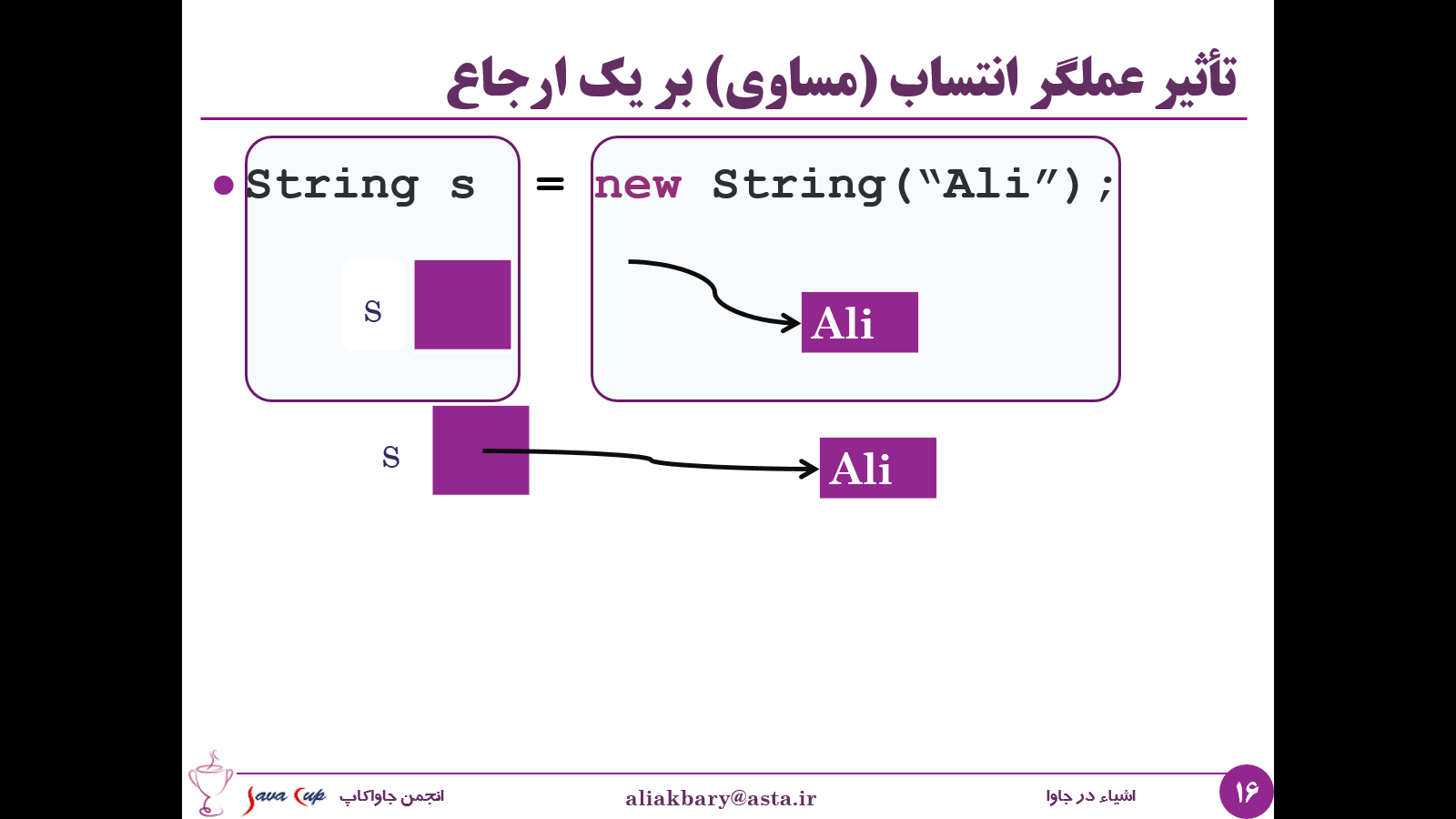
String str;

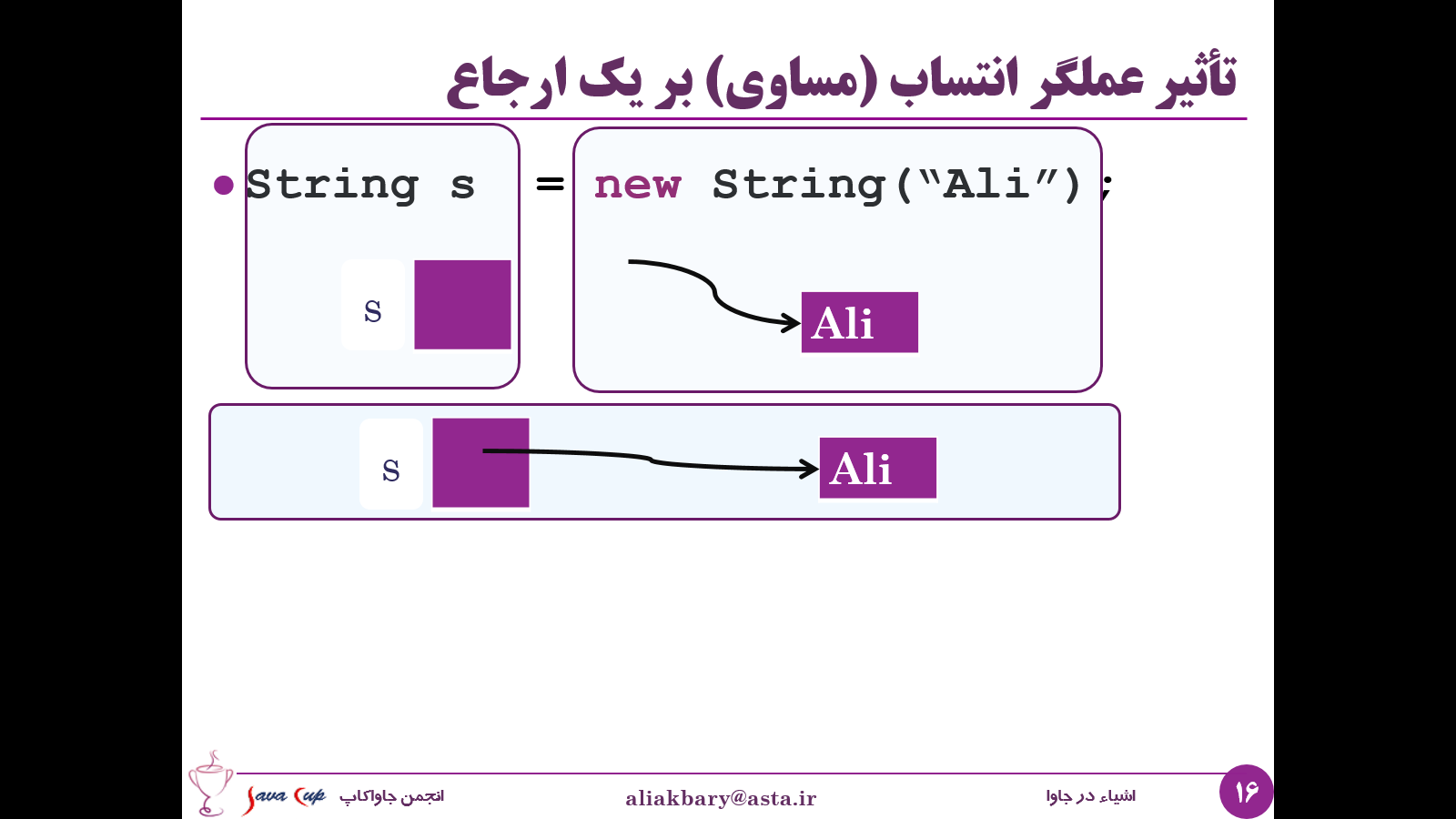
فقط یک ارجاع می سازد که هنوز به شیءی ارجاع نمی دهد، به همین علت هنوز نمی توانیم از آن استفاده کنیم. اصطلاحا str در حال حاضر خالی (null) است و به هیچی اشاره می کند. مقدار null یعنی ارجاعی که به شیءای اشاره نمی کند. هر متغیر باید به یک شی متصل شود مگر اینکه از نوع Primitive باشد. پس هر reference ای که ما ایجاد می کنیم مانند یک Scanner یا یک Book و یا یک String باید به یک شی ارجاع دهد که این کار ممکن است با اپراتور new انجام شود، مثلا str = new String(); یعنی یک شی جدید ساخته شود و str به آن اشاره کند یا اینکه str مساوی یک متغیر دیگر که به یک شی اشاره می کند قرار داده شود، یعنی با اپراتور مساوی هم می توانیم یک ارجاع به یک شی که قبلا ایجاد شده است را مقدار دهی کنیم.

اما دقیق تر ببینیم، عملگر انتساب یا مساوی (assignment) دقیقا چه کاری با یک ارجاع انجام می دهد؟ کد زیر را در نظر بگیرید:

String s = new String("Ali");

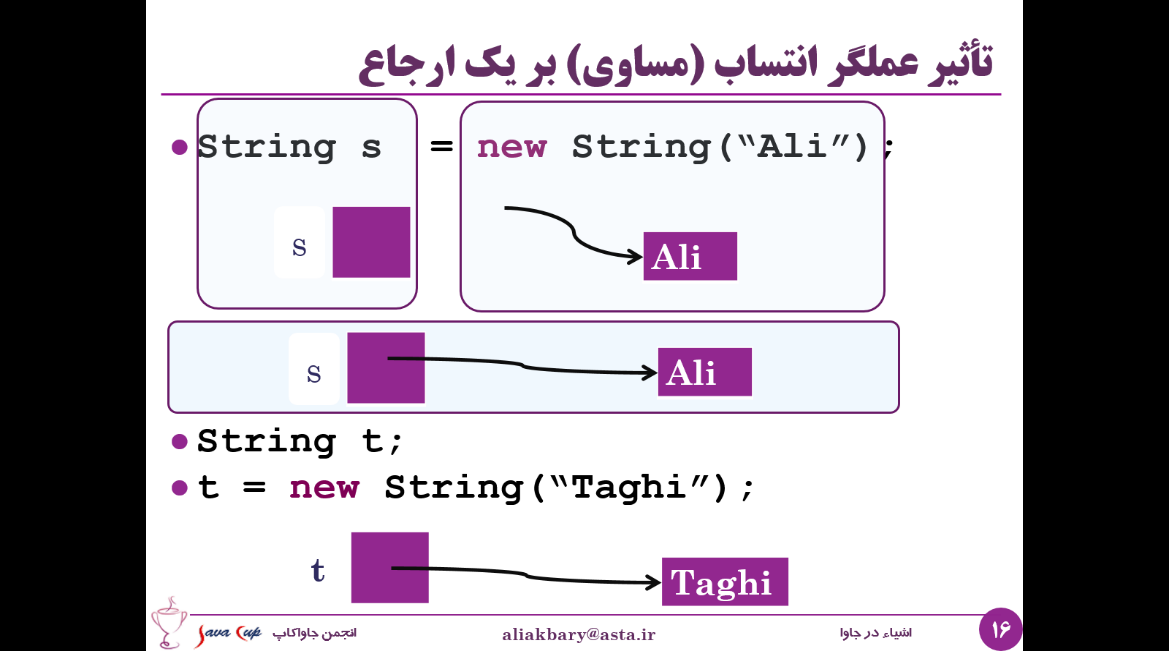
قسمت اول یعنی String s بخشی از حافظه را برای ارجاع s اختصاص می دهد و وقتی که اپراتور new را استفاده می کنیم، یک شی جدید ساخته می شود و ارجاع آن برگردانده می شود، این ارجاعی که برگردانده می شود را در این جا داخل متغیر s ریخته ایم. بنابراین در نهایت s ارجاع می دهد به شی جدیدی که ساخته ایم.



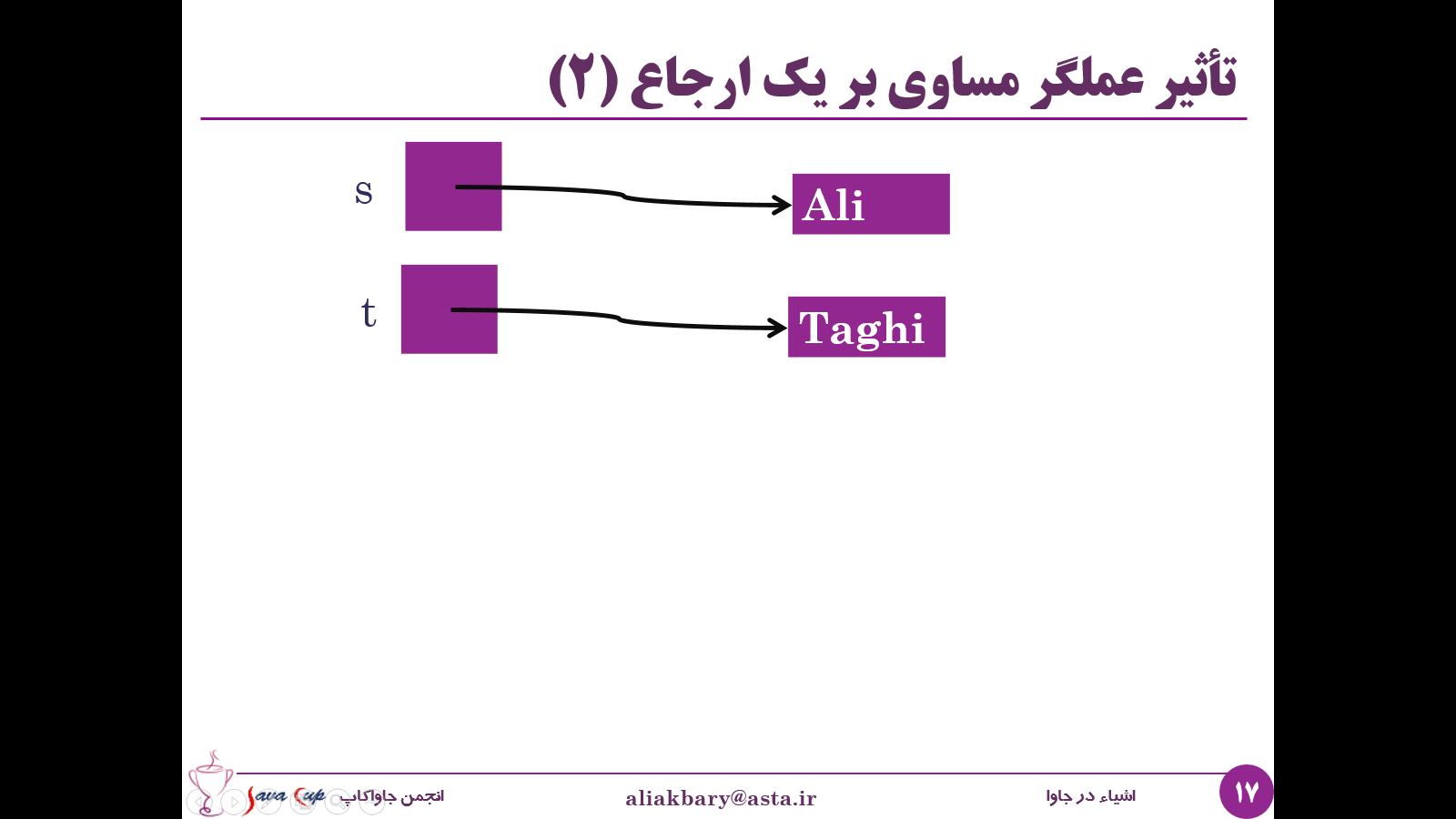


از یک منظر دیگر می توان اینطور نگاه کرد که شی جدیدی که ساخته شده یک آدرسی در حافظه دارد و s داخل خودش چیزی شبیه این آدرس حافظه را نگه می دارد، مثلا آدرس حافظه ی این شی جدید 005 است، پس s داخل خودش مقدار 005 را نگه خواهد داشت. به همین ترتیب اگر ما String دیگری را تعریف کنیم خواهیم داشت:

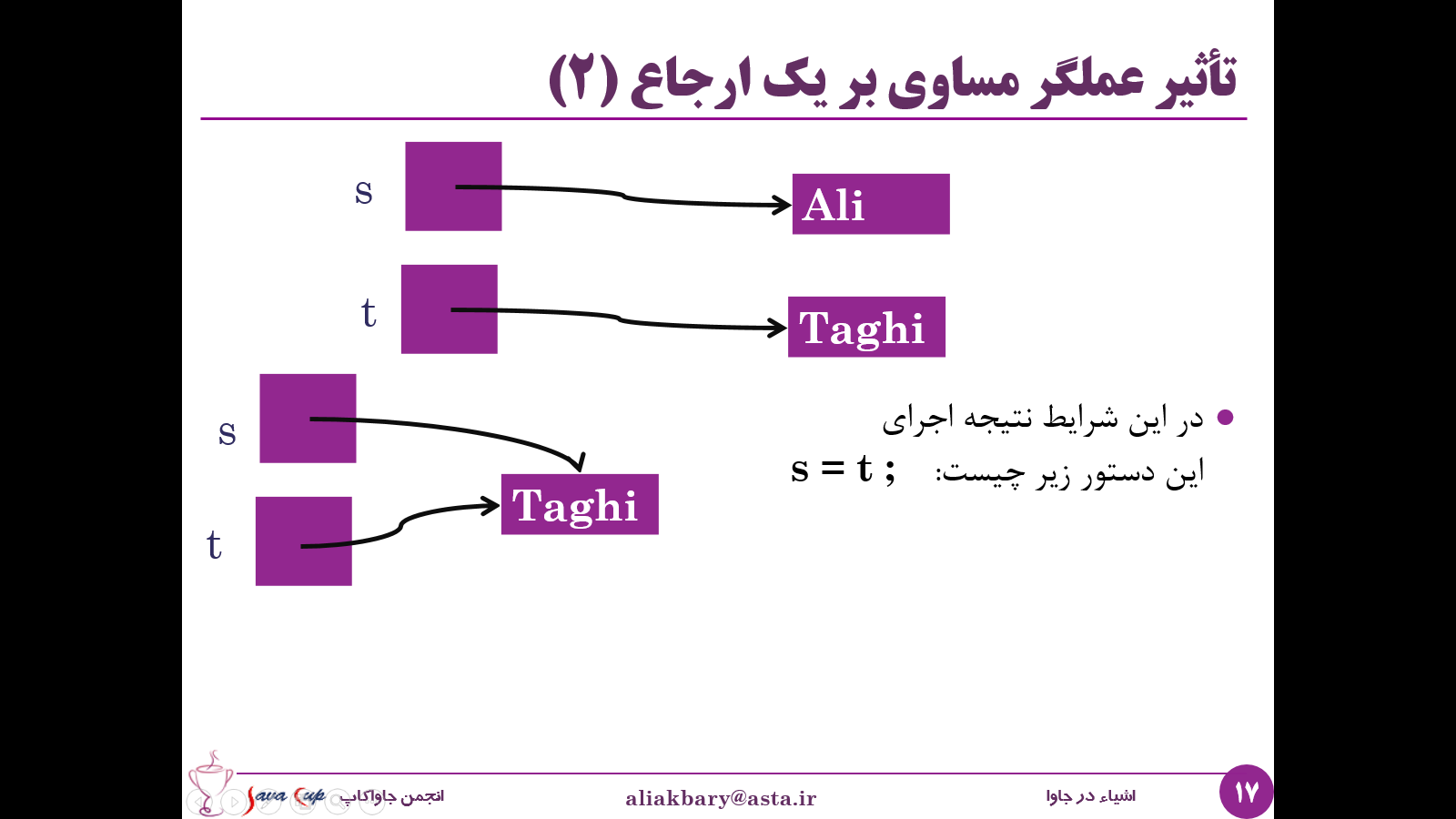
String t;

t = new String("Taghi");

پس به این ترتیب ما دو ارجاع ساختیم، ارجاع s و t. که s به مکانی از حافظه با محتوای Ali اشاره می کند و t به مکانی از حافظه با محتوای Taghi:



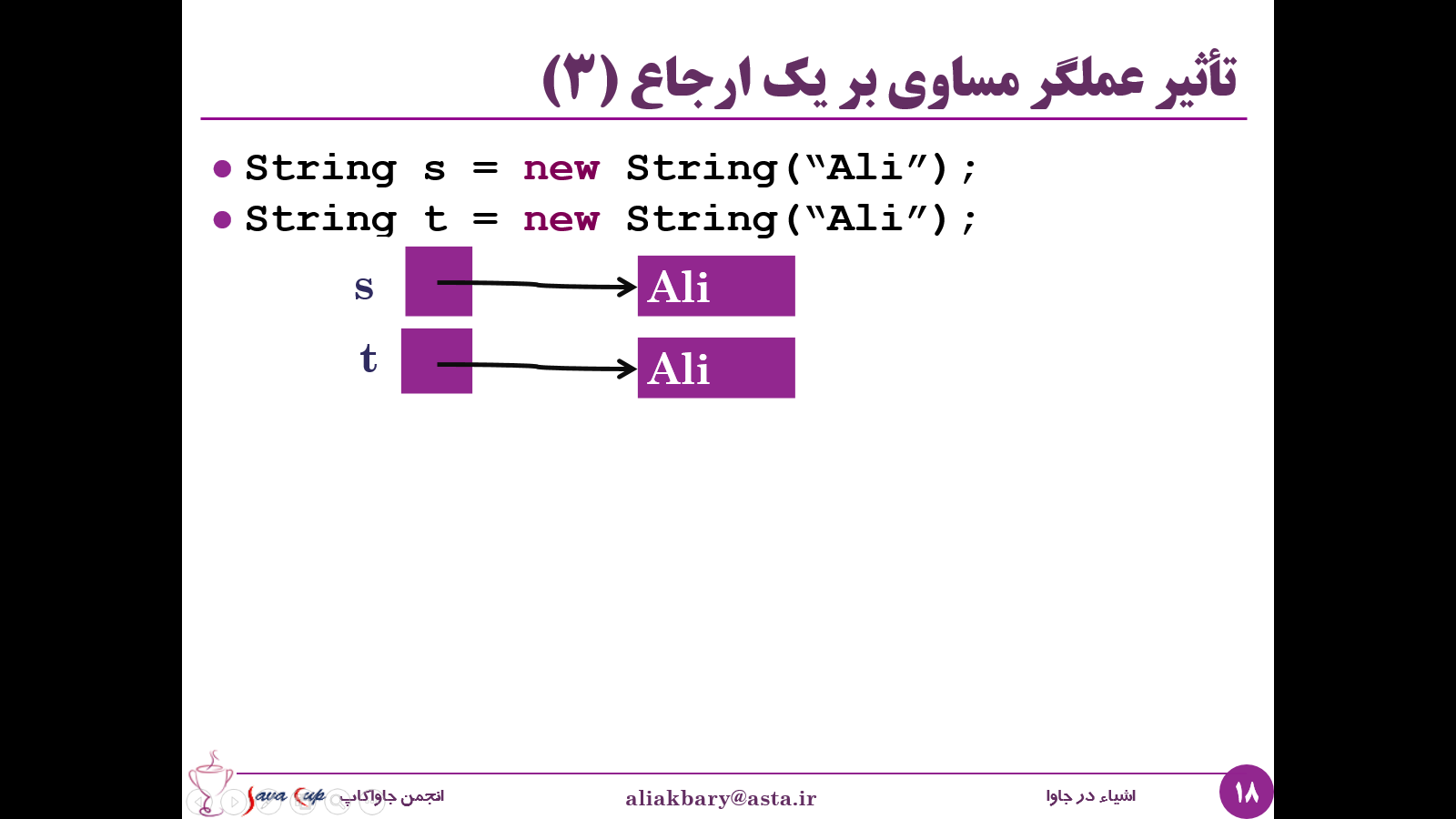
با این شرایط اگر دستور s = t; فراخوانی شود چه اتفاقی می افتد؟ این دستور assignment چه بلایی بر سر ارجاع ها و شی هایی که این دو متغیر به آنها ارجاع می دهند می آورد؟ این دستور باعث می شود s به همان جایی اشاره کند که t به آن اشاره می کرده است، پس بعد از اجرای این دستور، s و t هر دو به شی دارای مقدار Taghi اشاره خواهند کرد:



از منظر دیگر به آن نگاه کنیم، t یک آدرس حافظه داشته، مثلا 456 و s یک آدرس حافظه ی دیگر، مثلا 123 بعد و گفتیم s مساوی t قرار بگیرد، پس آدرسی که در t ذخیره شده بوده الان در s هم کپی می شود، پس s هم از این به بعد به همان 456 اشاره می کند.

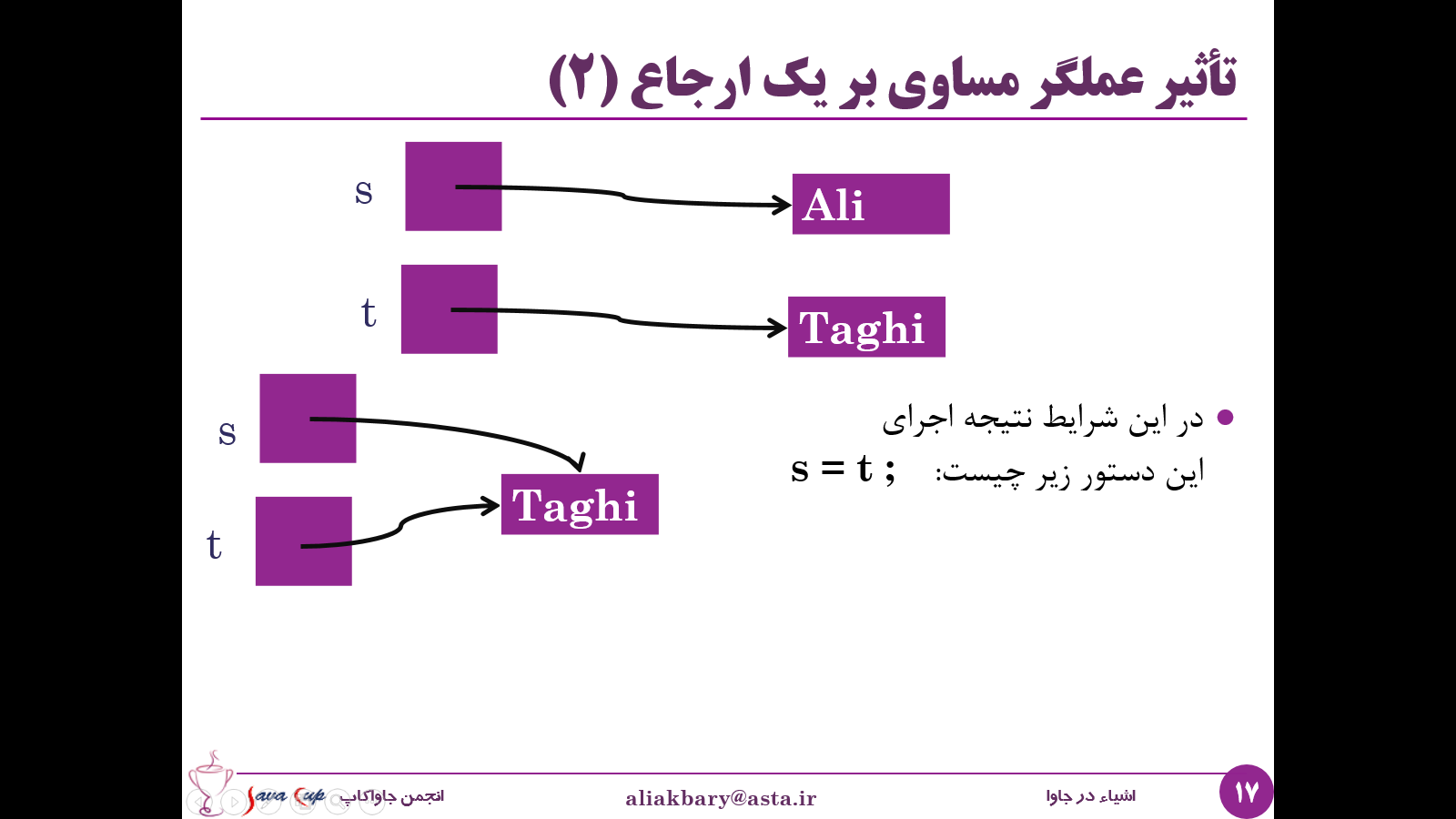
نکته: این خیلی خیلی مهم است که هنگامی که می گوییم s = t، محتوای اشیاء کپی نمی شود، Taghi سر جای خودش هست و Ali هم سر جای خودش، و دستور مساوی تنها ارجاع به این اشیاء را تغییر می دهد و فقط محتوای داخل t داخل s کپی می شود و نه اشیایی که اینها دارند بهشان اشاره می کنند.

از دید دیگری به این مثال نگاه کنید، فرض کنید که s و t رشته هایی با محتوای یکسان باشند:



در این حالت s یک ارجاع به شی با محتوای Ali است و t هم یک ارجاع به شی با محتوای Ali است، در این حالت، s و t به اشیایی اشاره می کنند که وضعیت یکسان دارند، و محتوای آنها برابر است، اما به دو شی متفاوت اشاره می کنند، اصطلاحا هویت(Identity) اشیایی که s و t به آنها اشاره می کنند متفاوت است هر چند محتوای آنها یکسان است.

حال وقتی دستور s = t اجرا می شود، s به همان جایی اشاره خواهد کرد که t به آن اشاره می کند و به این ترتیب هویت s و t یکی می شود، هر دو شی در حافظه خواهند بود اما متغیرهای ما به یکی از آنها اشاره می کنند و شی دیگر در حافظه سرگردان خواهد بود و بعدها خواهیم دید که برای این اشیای سرگردان چه اتفاقی خواهد افتاد.

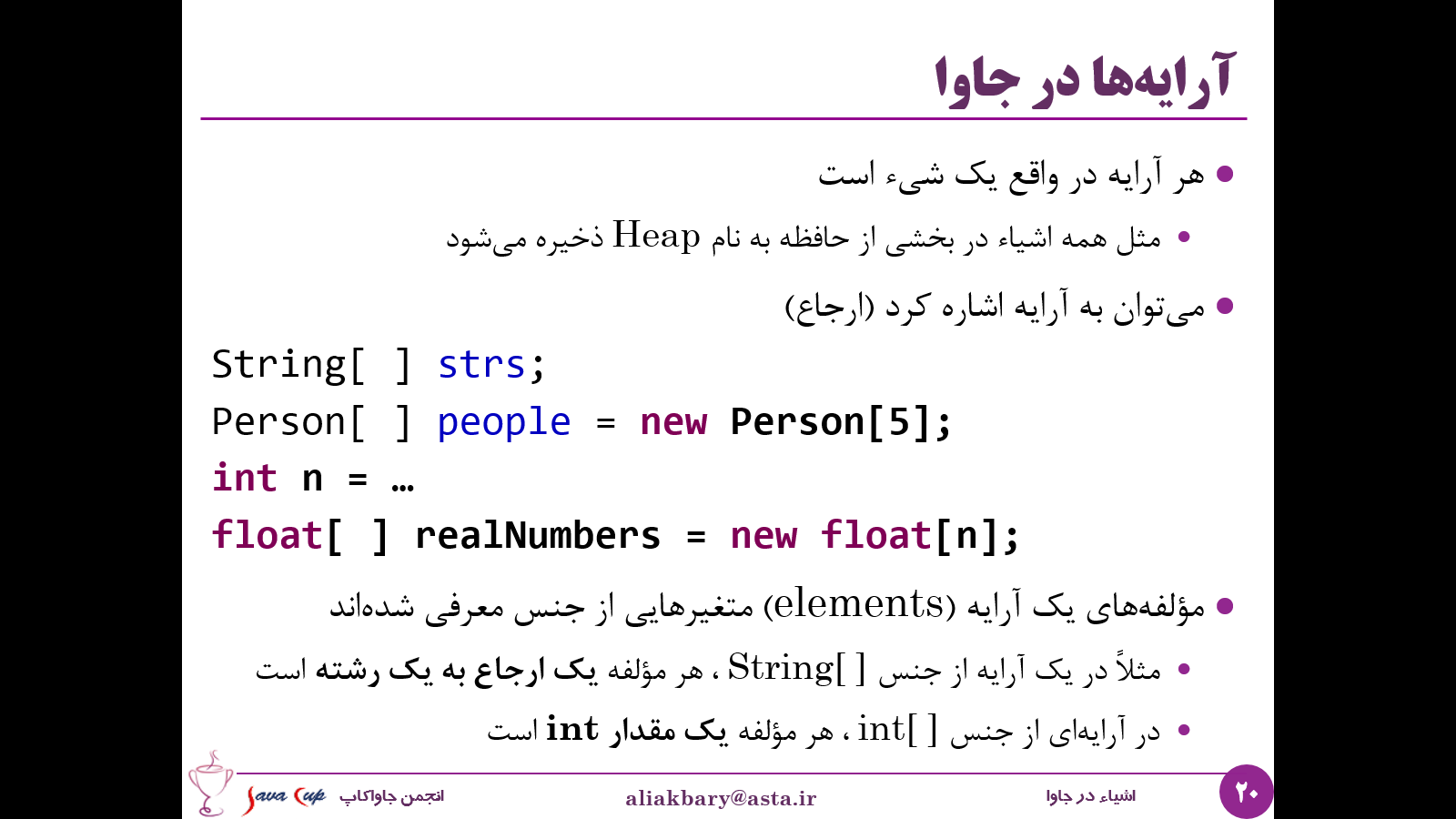


## آرایه ها در جاوا

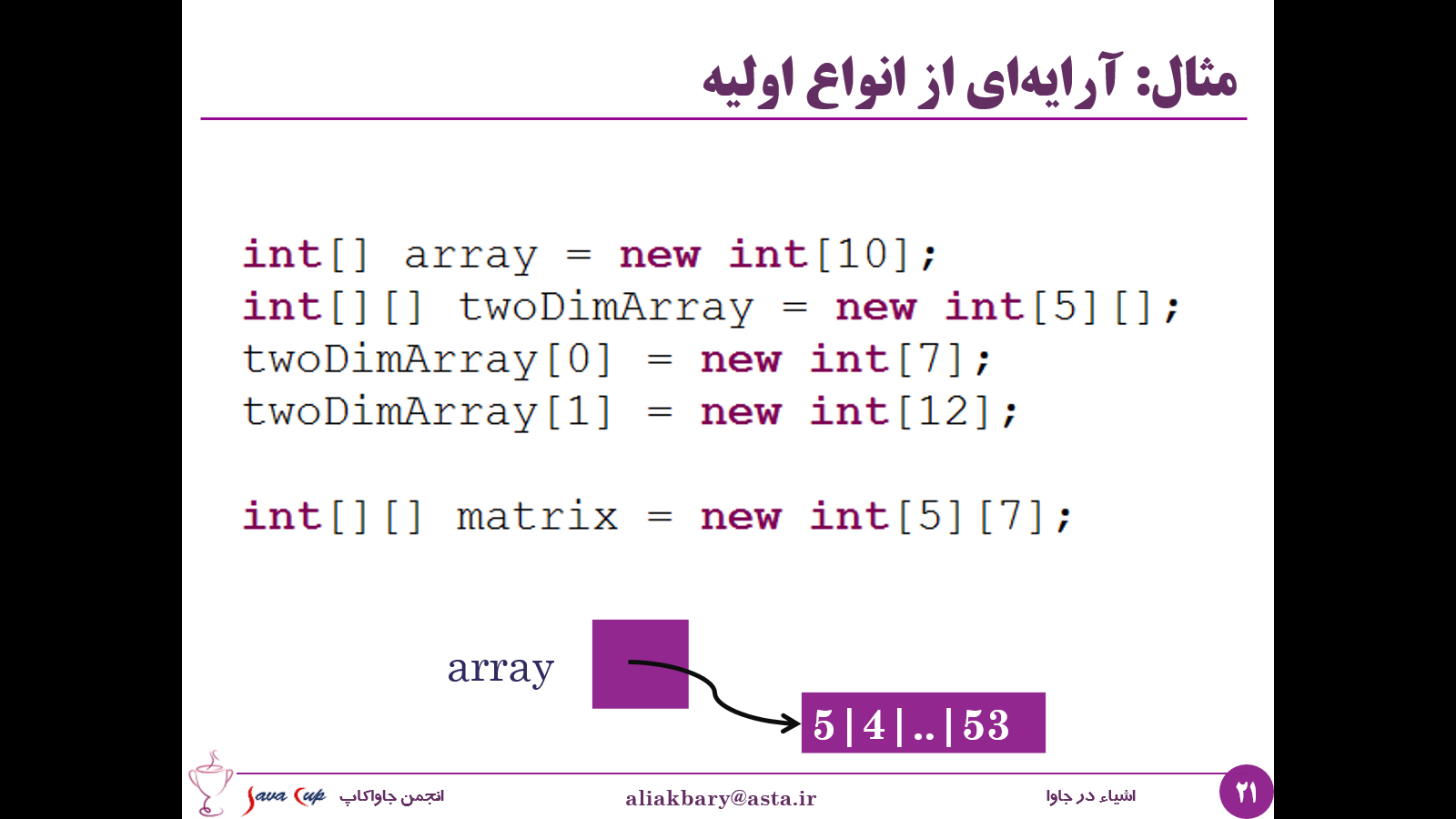
ما قبلا به جز انواع داده ی اولیه (Primitive Data Types)، با دو نوع دیگر از داده ها کار کرده بودیم، آرایه ها و رشته ها. الان می خواهیم ببینیم که آرایه ها و رشته های هم در واقع شی هستند. از آرایه ها شروع می کنیم. هر آرایه در واقع یک شی است و مثل همه ی اشیاء، هر آرایه هم در هیپ (Heap) جای می گیرد و مانند همه ی اشیاء ما می توانیم به آرایه ها هم اشاره کنیم و ارجاع به آنها داشته باشیم. مثلا وقتی می گوییم:

String[ ] strs;

الان دیگر دقیق تر می دانیم که strs یک ارجاع به شی ای از نوع آرایه است و با کمک اپراتور new ما می توانیم این شی را ایجاد کنیم، مثلا new Person[5] یعنی یک شی از جنس آرایه که حاوی 5 Person خواهد بود. به همین ترتیب new float[n] یعنی یک شی از جنس آرایه که حاوی n عدد float خواهد بود.

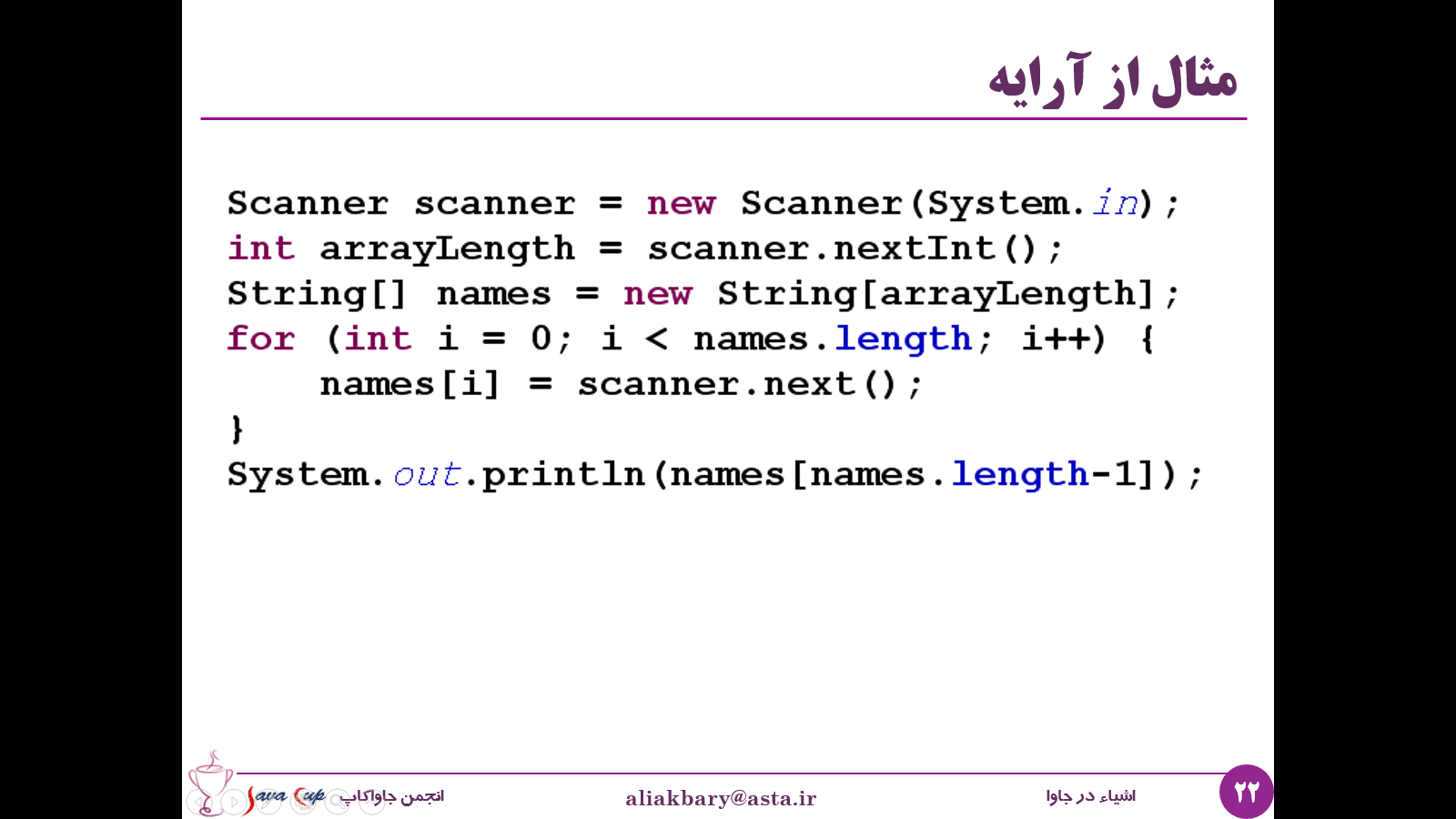


و به این دقت کنید، مولفه های هر آرایه، دقیقا از جنس تعیین شده هستند، مثلا مولفه های آرایه ی People در مثال بالا، از جنس ارجاع به Person هستند، مولفه های آرایه ی realNumbers از جنس یک مقدار float هستند. پس آرایه ای که از جنس reference ایجاد می شود مانند Person یا String، هر مولفه اش یک ارجاع به شی ای از آن جنس است، مثلا String[]، هر مولفه اش ارجاع به یک رشته است. ولی int[]، هر مولفه اش دقیقا یک مقدار int است و ارجاع به جای دیگری نیست. مثلا اگر یک int[] تعریف کنید با عنوان array و آن را مساوی یک شی 10تایی از intها قرار دهید، array یک متغیر در حافظه خواهد بود که اشاره گر یا ارجاع یا reference به یک جای دیگر در حافظه است که در جا یک شی در جنس آرایه قرار دارد که داخل آن عناصر این آرایه که هر کدام یک int هستند قرار می گیرند. و به همین ترتیب عناصر آرایه های چند بعدی هم در حافظه پشت سر هم قرار می گیرند.



پس دقت کنیم که array ارجاع به یک شی است که در این مثال آن شی از جنس آرایه است.

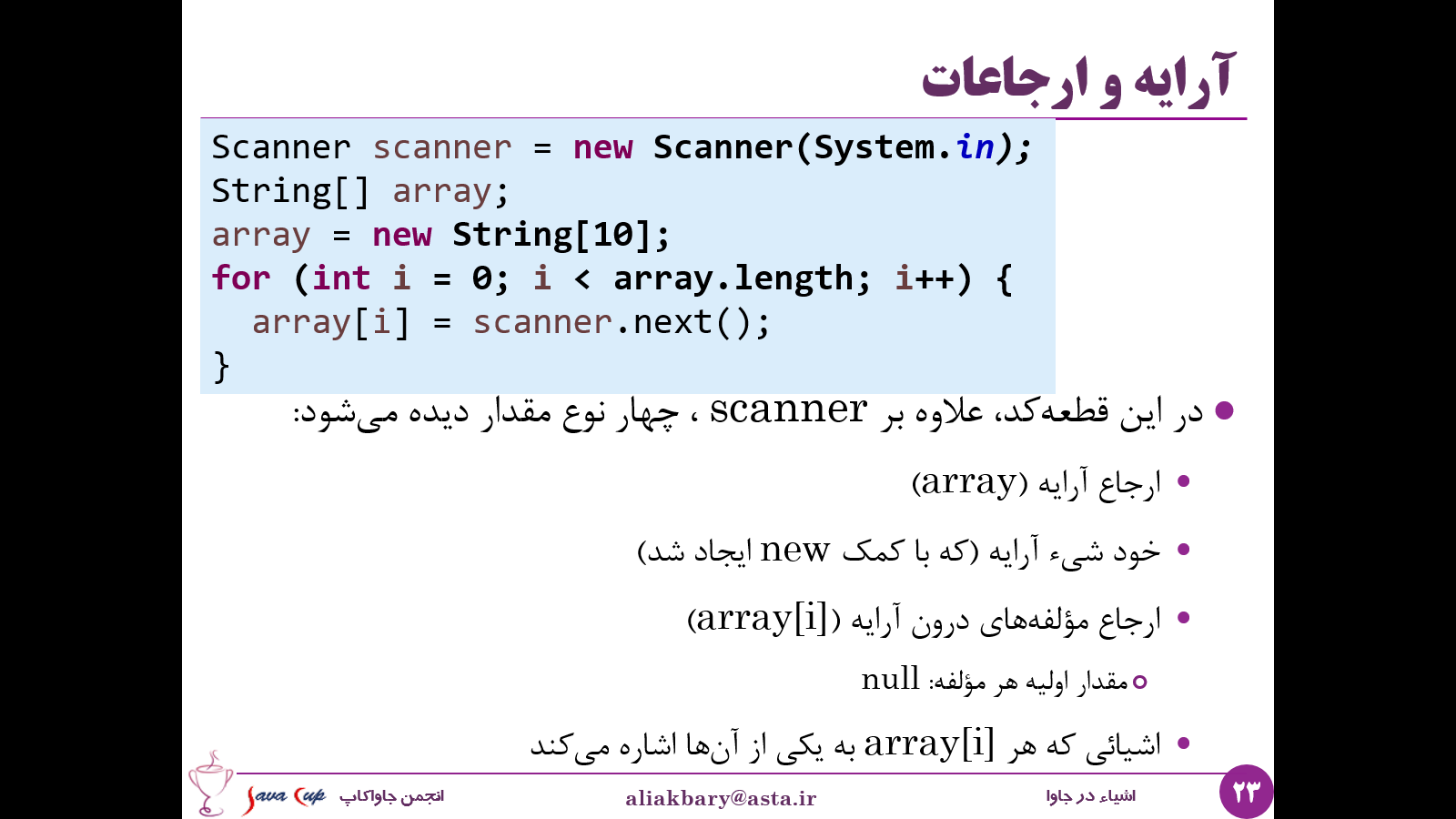
یک مثال دیگر ببینیم، در این مثال یک Scanner ایجاد کرده ایم، با کمک آن یک عدد از ورودی خوانده ایم و داخل متغیری ذخیره کرده ایم. یه آرایه به اسم names ایجاد کرده ایم، آرایه ای از رشته های که طول این آرایه همان عددی است که از ورودی خوانده ایم. و بعد یک حلقه ایجاد کرده ایم به تعداد خانه های این آرایه و هر خانه از این آرایه را مساوی یک مقدار جدید قرار دادیم که از ورودی (Scanner) خوانده شده است. در نهایت هم آخرین عضو این آرایه را چاپ کرده ایم:



نکته ای که اینجا جدید است، استفاده از ویژگی length برای آرایه ها هست. آرایه ها، هر کدام یک شی هستند، و هر شی ویژگی هایی دارد، یکی از ویژگی هایی که برای آرایه های وجود دارد ویژگی length است که طول آرایه را برمی گرداند. این ویژگی public است و ما می توانیم آن را بخوانیم و از آن استفاده کنیم.

## آرایه و ارجاعات

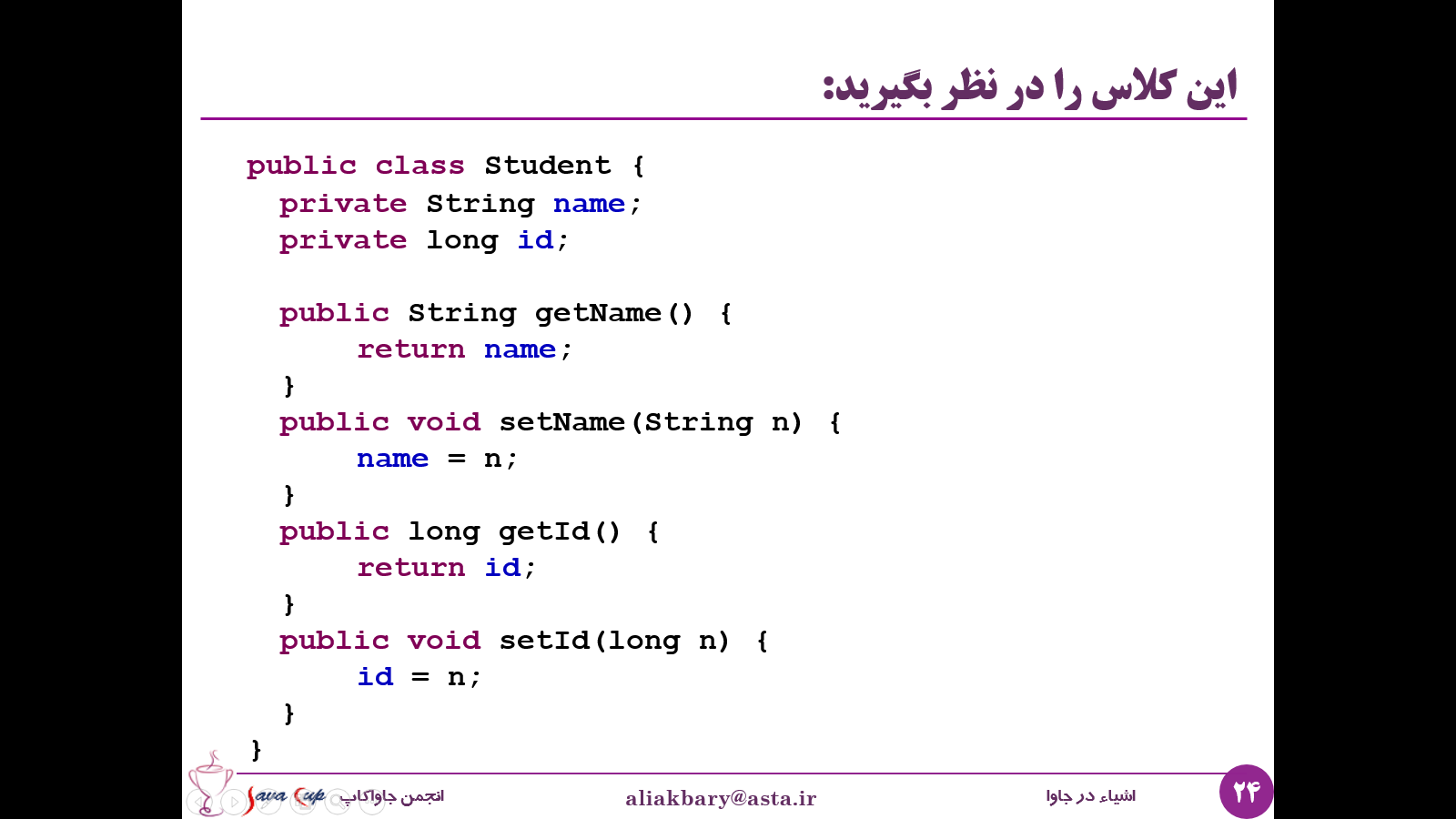
فرض کنید یک ارجاع به آرایه به نام array ایجاد کرده ایم که به آرایه ای به طول 10 از جنس رشته ها اشاره می کند، و باز یک حلقه ایجاد کرده ایم و هر کدام از خانه های این آرایه را از ورودی خوانده ایم:



در این قطعه کد، علاوه بر scanner که خود یک شی است، چهار نوع مقدار دیده می شود:

* ارجاع array که جایی در حافظه است و قرار است آدرس یک آرایه را نگهدارد.
* خود شی آرایه که با کمک new ساخته می شود، و در جای دیگری از حافظه قرار می گیرد.
* ارجاع هایی که داخل این آرایه قرار دارند، آرایه ای که ما ساخته ایم، حاوی 10 عنصر (element) است که هر کدام از آنها مانند array[i]، خودشان یک ارجاع به یک String هستند و در ابتدای کار همه ی آنها null هستند. وقتی ما این آرایه را new می کنیم تک تک elementهای این آرایه null و خالی هستند و هنوز به چیزی اشاره نمی کنند.
* و چهارمین نوع مقداری که در این مثال داریم هر کدام از مقادیری هستند که array[i] دارند به آنها اشاره می کنند، مثلا array[0] به یک رشته اشاره می کند و array[1] به یک رشته ی دیگر اشاره می کند که هر کدام از اینها از ورودی خوانده شده اند و در جایی از حافظه قرار گرفته اند.

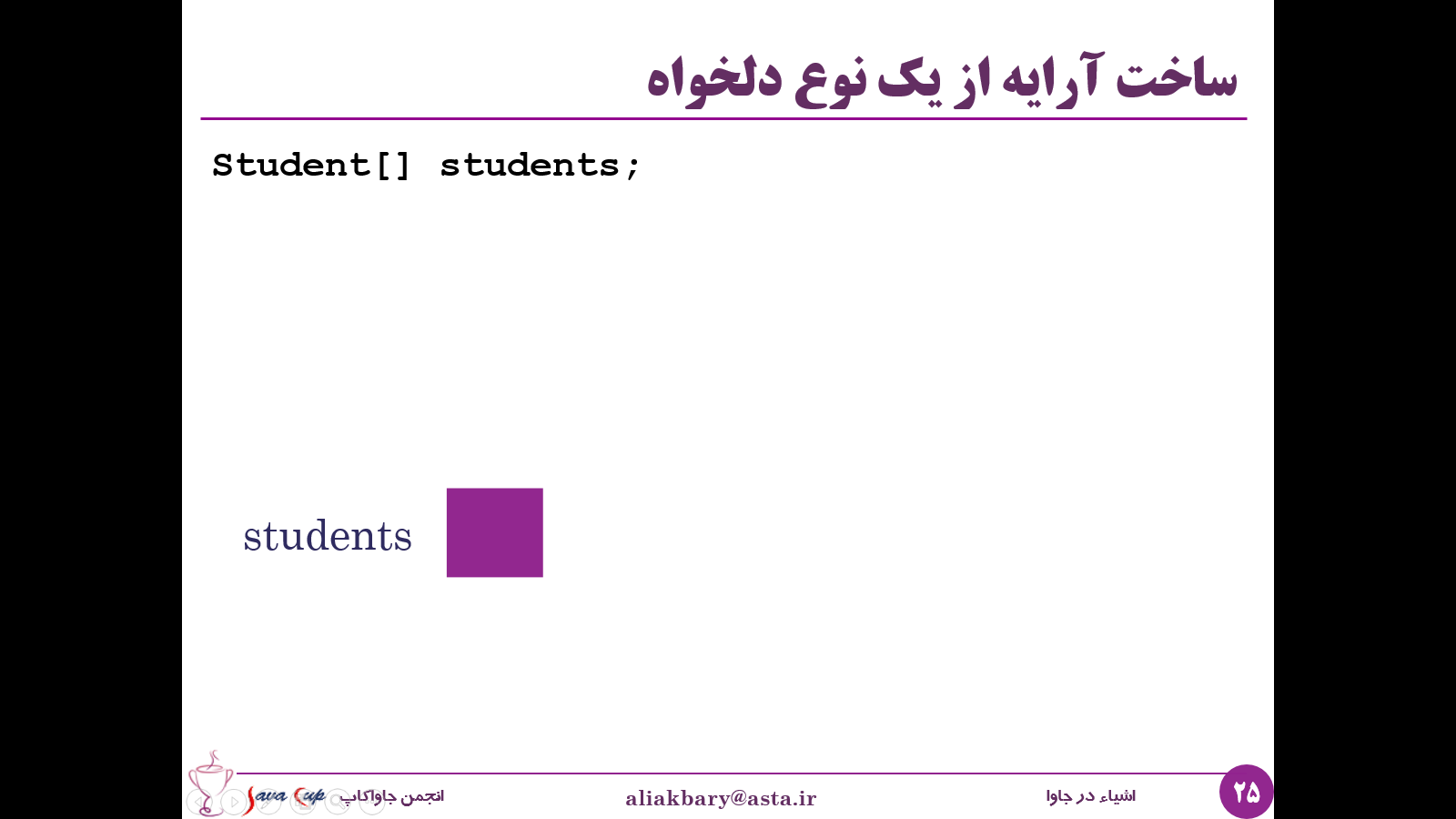
باز برای اینکه شهودمان تکمیل تر شود، کلاس زیر را در نظر بگیرید:



کلاسی برای دانشجو نوشته ایم که ویژگی های نام و شماره دانشجویی دارد و getter و setter هایی برای خواندن و مشخص کردن ویژگی هایش. با این فرض، دستور زیر چه کاری انجام می دهد؟

Student[] students;

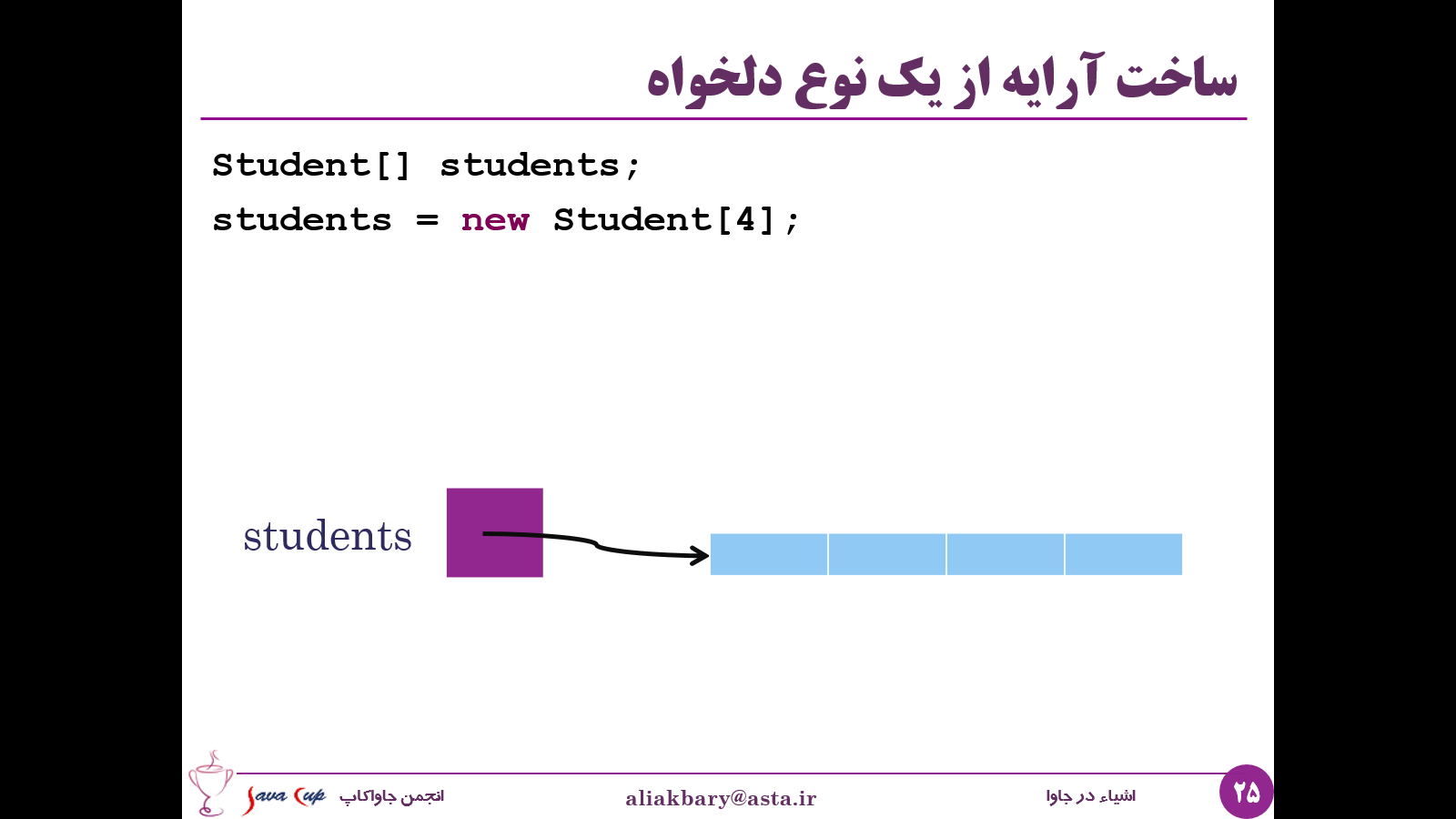
یک ارجاع به آرایه ای از دانشجوها ایجاد می کند، پس تا اینجا ما فقط یک متغیر داریم که قرار است داخل آن ارجاعی به یک آرایه ذخیره شود:



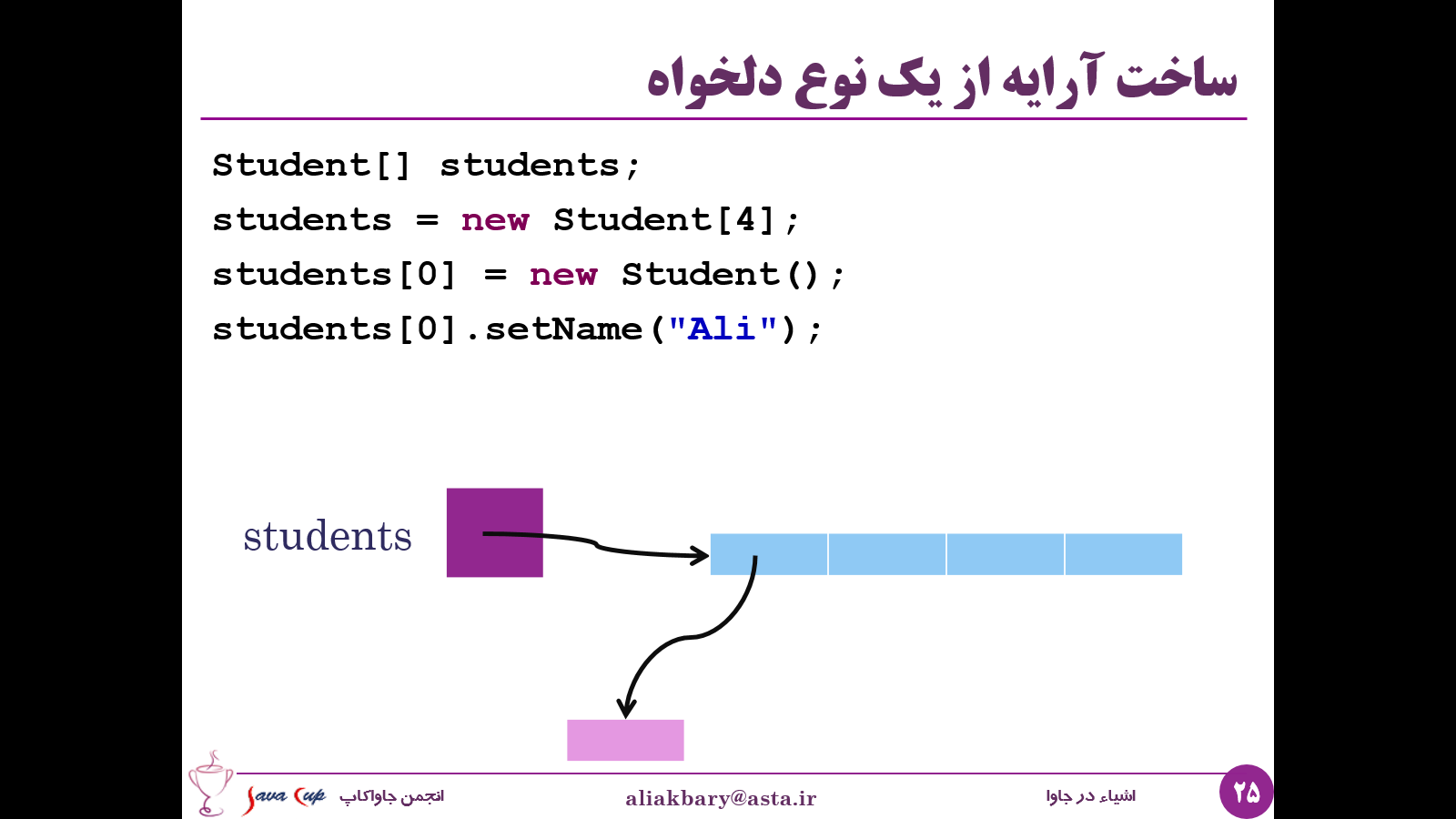
حال اگر بگوییم:

students = new Student[4];

جایی در حافظه حاوی 4 ارجاع برای Student ایجاد می شود، در واقع یک شی جدید از جنس آرایه در حافظه ایجاد می شود و students به آن اشاره می کند. توجه کنید که در این حالت تمام ارجاع های داخل این آرایه خالی هستند و به جایی اشاره نمی کنند:

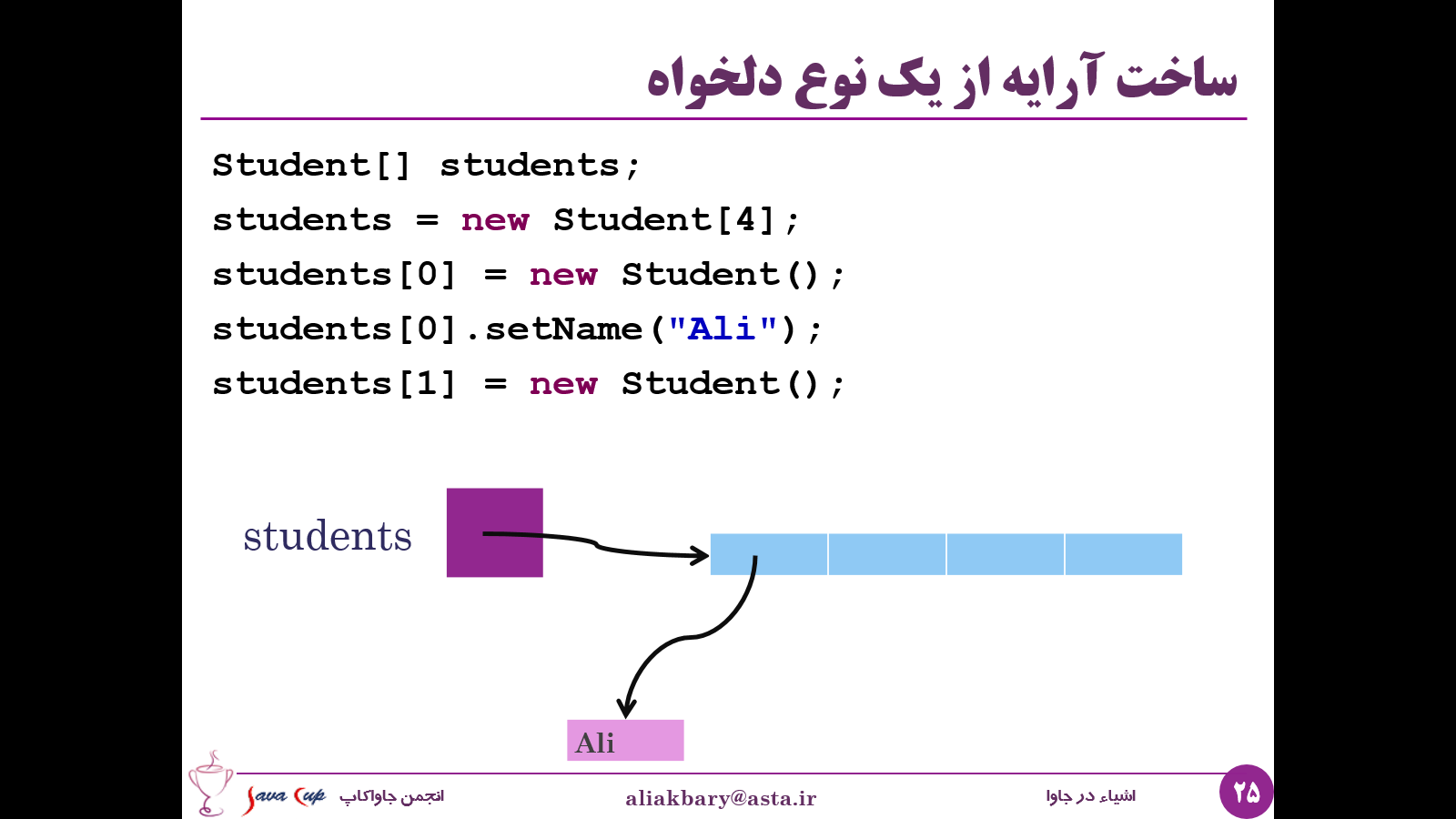


اگر بگوییم students[0] = new student();، اولین خانه ی این آرایه که الان null هست، به یک جایی حاوی یک شی از جنس Student اشاره خواهد کرد:



حال می توانیم برای این خانه ی اول آرایه ویژگی نام را مقدار دهی کنیم:

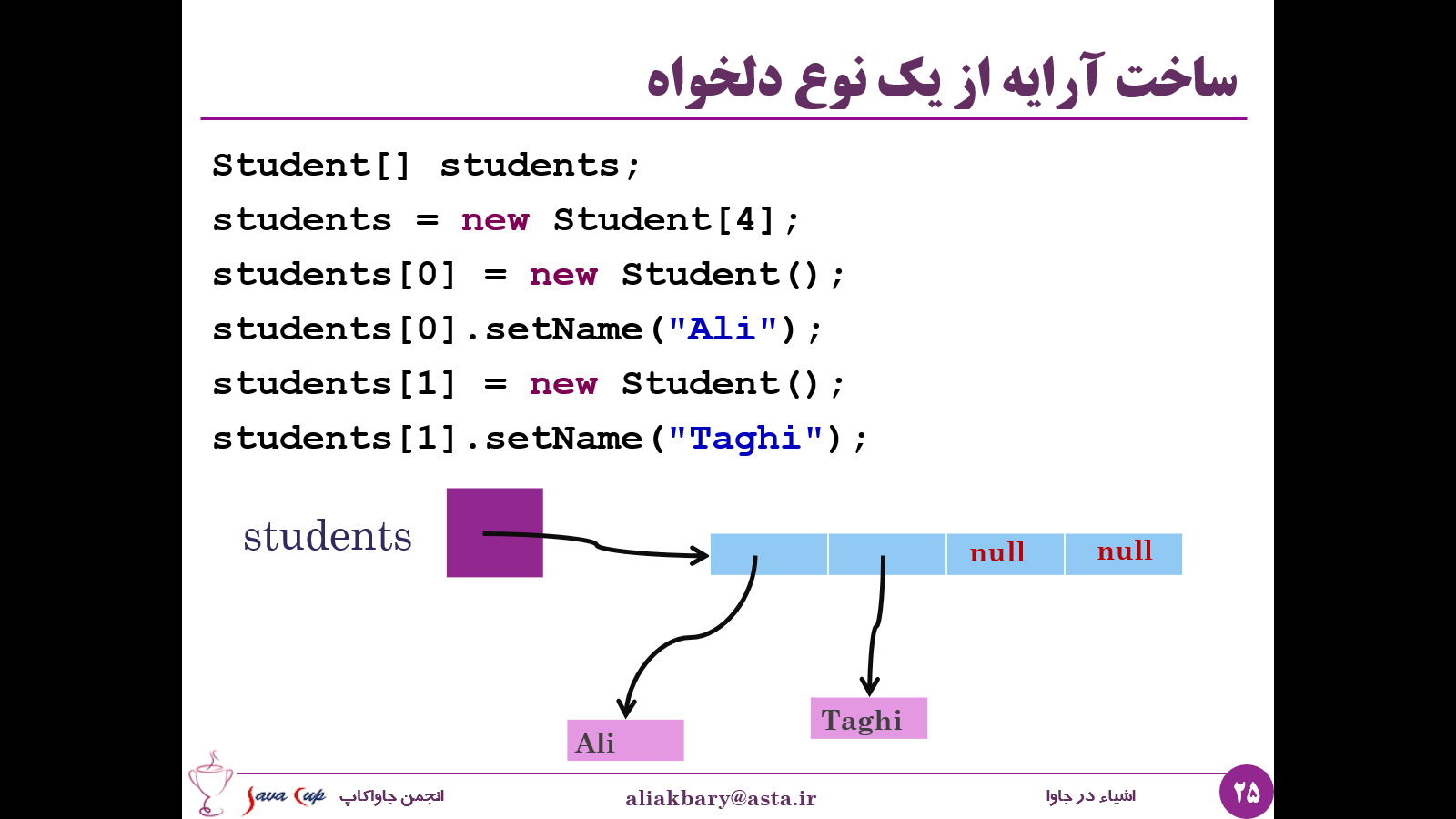
students[0].setName("Ali");



البته اگر بخواهیم دقیق تر نشان بدهیم، داخل خود این شی (عنصر صفرم آرایه) اشاره گر دیگری هست به جای دیگر از حافظه که رشته ی نام Ali در آنجا ذخیره شده است. در ادامه می توانیم عنصر دیگری از آرایه را new کرده و ویژگی نام را برای آن مقدار دهی نماییم:

students[1] = new Student();

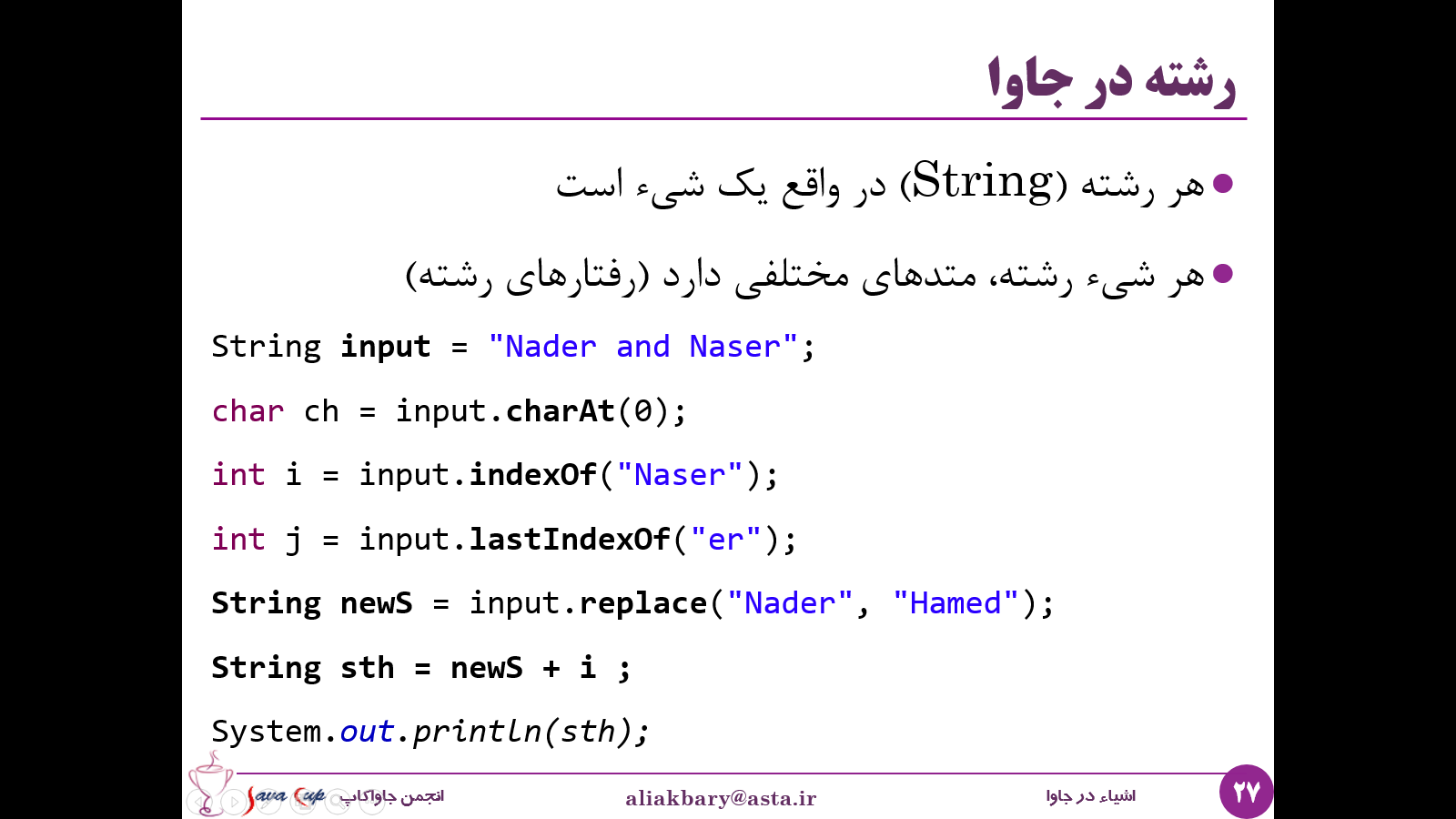
students[1].setName("Taghi");



در این حالت دو خانه ی اول آرایه ما مقدار دارند و ارجاع هایی هستند که به مکان معتبری از حافظه اشاره می کنند و خانه های دیگر آرایه همچنان null باقی مانده اند و نمی توانیم از آنها استفاده ی درستی داشته باشیم.

## رشته در جاوا

هر رشته هم مانند آرایه ها یک شی است. ما قبلا با رشته ها کار کردیم و الان دقیق تر می توانیم ببینیم که هر متغیر از جنس رشته در واقع ارجاعی به یک شی از این جنس است. همان طور که هر شی دیگری رفتارهایی دارد، رشته ها هم متدها و رفتارهای مختلفی دارند، مثال زیر را ببینید:

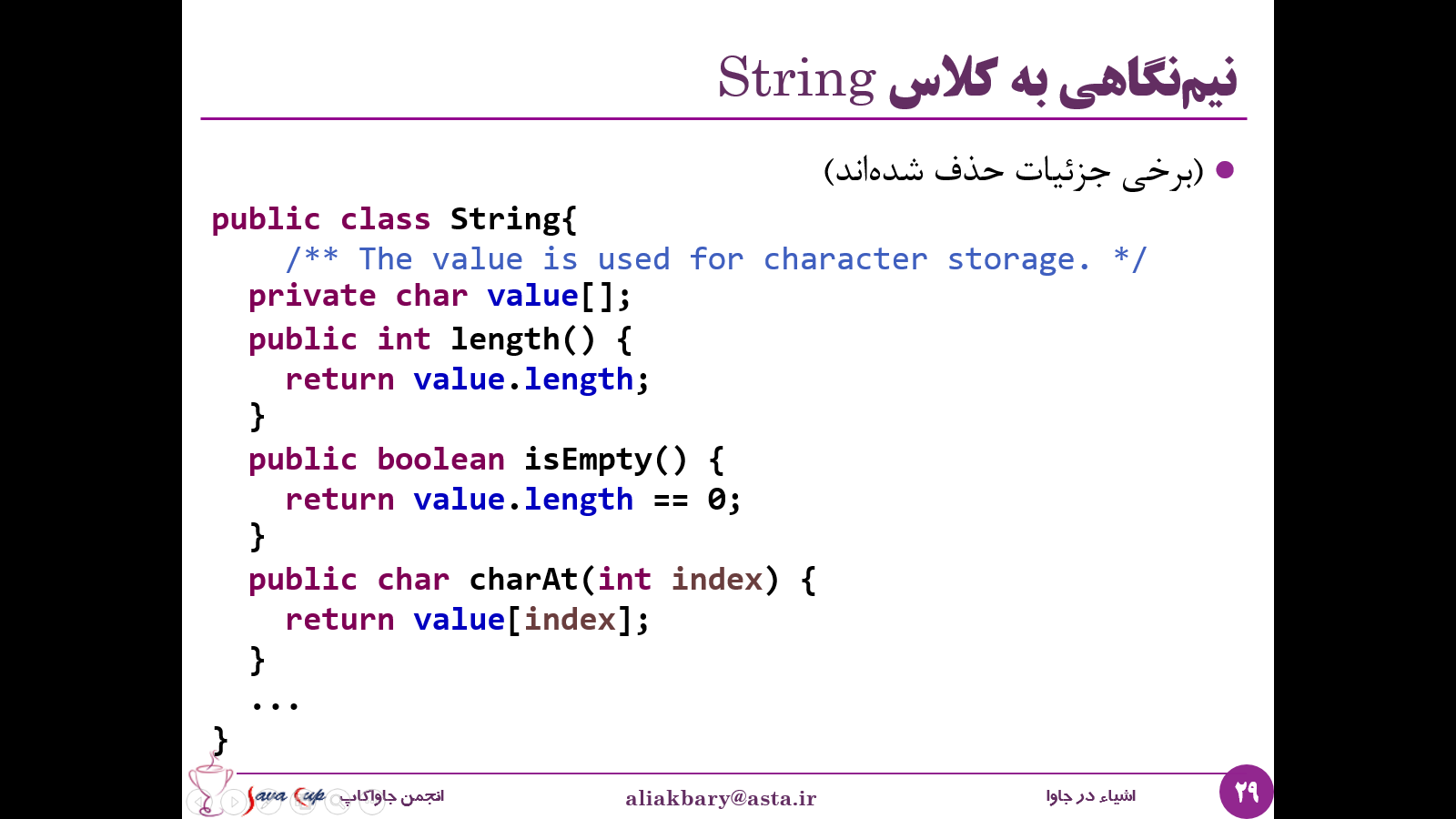


حالا دیگر دقیق تر می دانیم که input.charAt(0) چه معنایی دارد، در این حالت ما رفتار charAt را روی شی input فراخوانی کرده ایم و این رفتار یک کاراکتر برمی گرداند.

و دیدیم که اپراتور خاص برای رشته ها وجود دارد مثل + و مقدار دو رشته را به هم متصل کرده و خروجی می دهد.

## نیم نگاهی به کلاس String

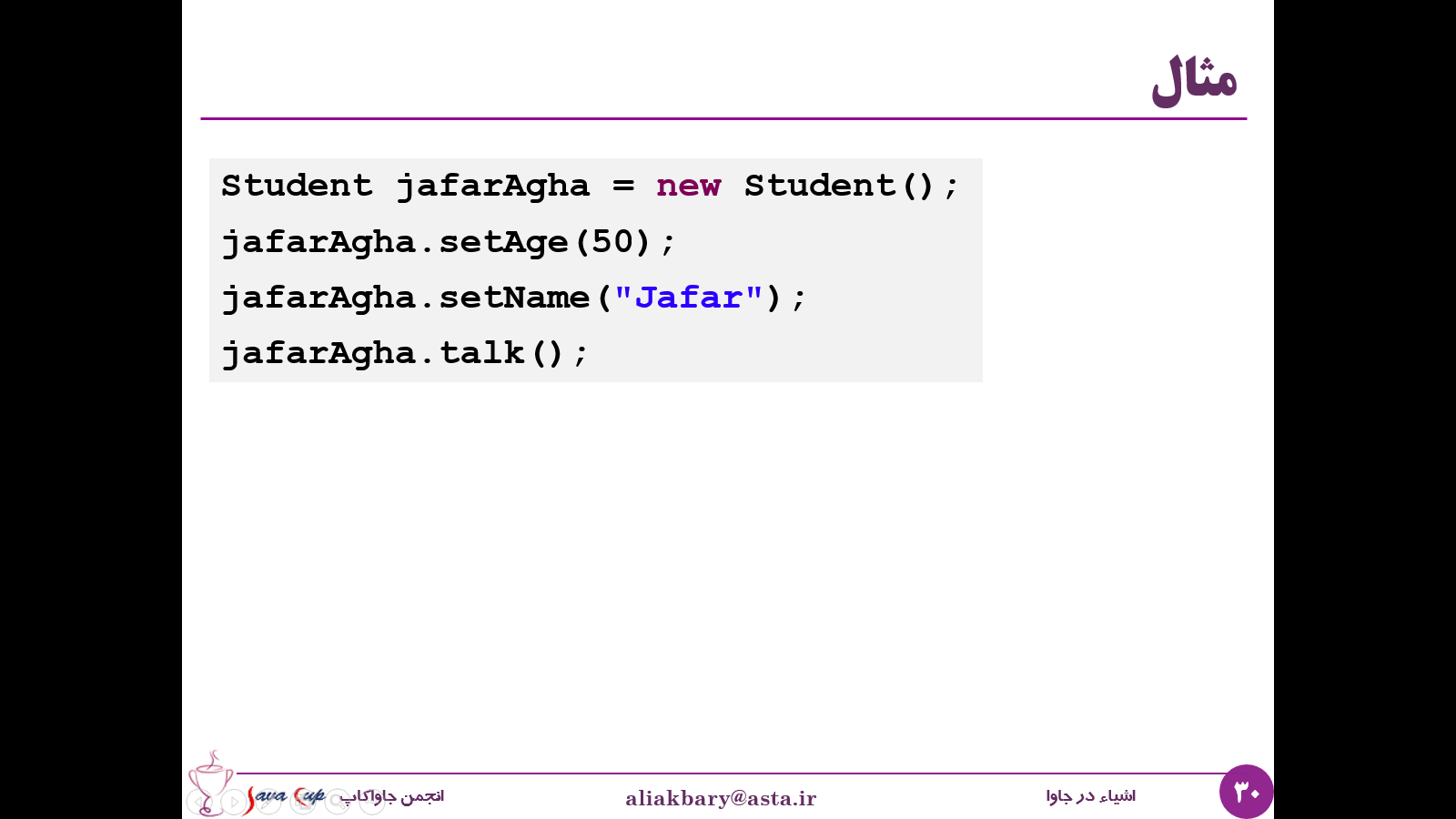
بگذارید نیم نگاهی به کلاس String داشته باشیم. این کلاس در جاوا تعریف شده، من بخش هایی را حذف کرده ام تا در مورد کلیات آن قدری با هم صحبت کنیم. شما هم می توانید متن این کلاس را باز کرده و مشاهده کنید. در متن این کلاس یک ویژگی value وجود دارد که آرایه ای از کاراکترهاست و به صورت خصوصی تعریف شده است.



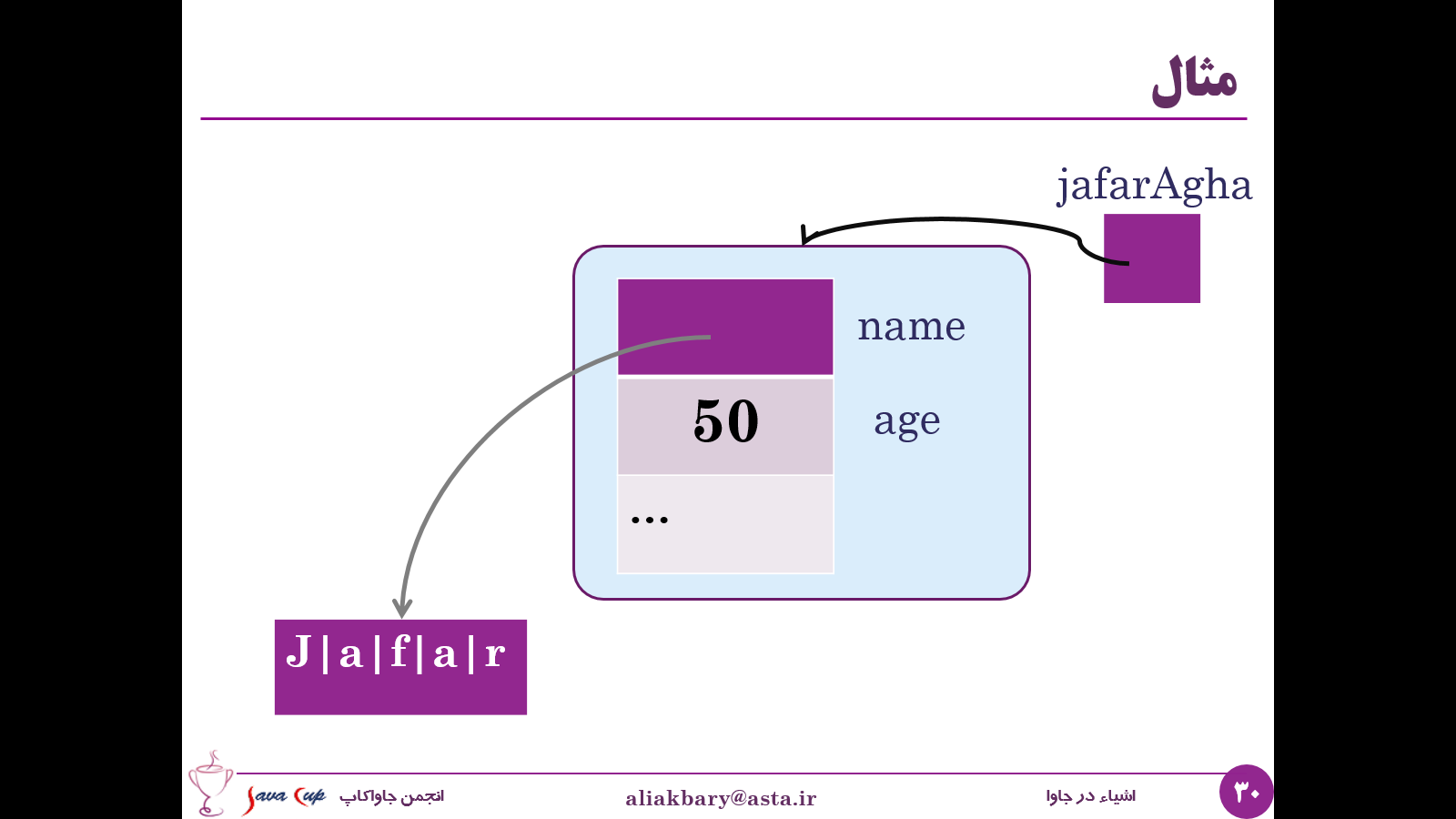
پس داخل هر شی از این کلاس عملا آرایه ای از کاراکترها نگهداری می شود. و متدهایی مانند length()، isEmpty() و charAt() با کمک همین آرایه به سادگی پیاده سازی شده اند، مثلا length() طول همین آرایه را برمی گرداند و ما قبلا دیدیم که این طول چه معنایی برای آرایه ها دارد (یکی از ویژگی های کلاس آرایه است)، یا isEmpty() متدی هست که Boolean برمی گرداند و مشخص می کند این رشته خالی هست یا نه. اگر آرایه ای که داخل این رشته هست طولش صفر باشد این رشته خالی است و کاراکتر خاصی در آن قرار نگرفته است. به همین ترتیب cahrAt(int index) با کمک آرایه ی value و پاس کردن ایندکس به آن، کاراکتری که در آن خانه از آرایه وجود دارد را برمی گرداند.

## مثال

مثال دیگری ببینیم برای اینکه شهود دقیقی داشته باشیم از جایگاه ارجاع ها به حافظه. فرض کنید یک شی از جنس دانشجو داریم به نام jafarAgha که به یک شی جدید از جنس دانشجو اشاره می کند. ویژگی سن این شی 50 سال است، نامش Jafar است و متد talk() را روی آن فراخوانی می کنیم:

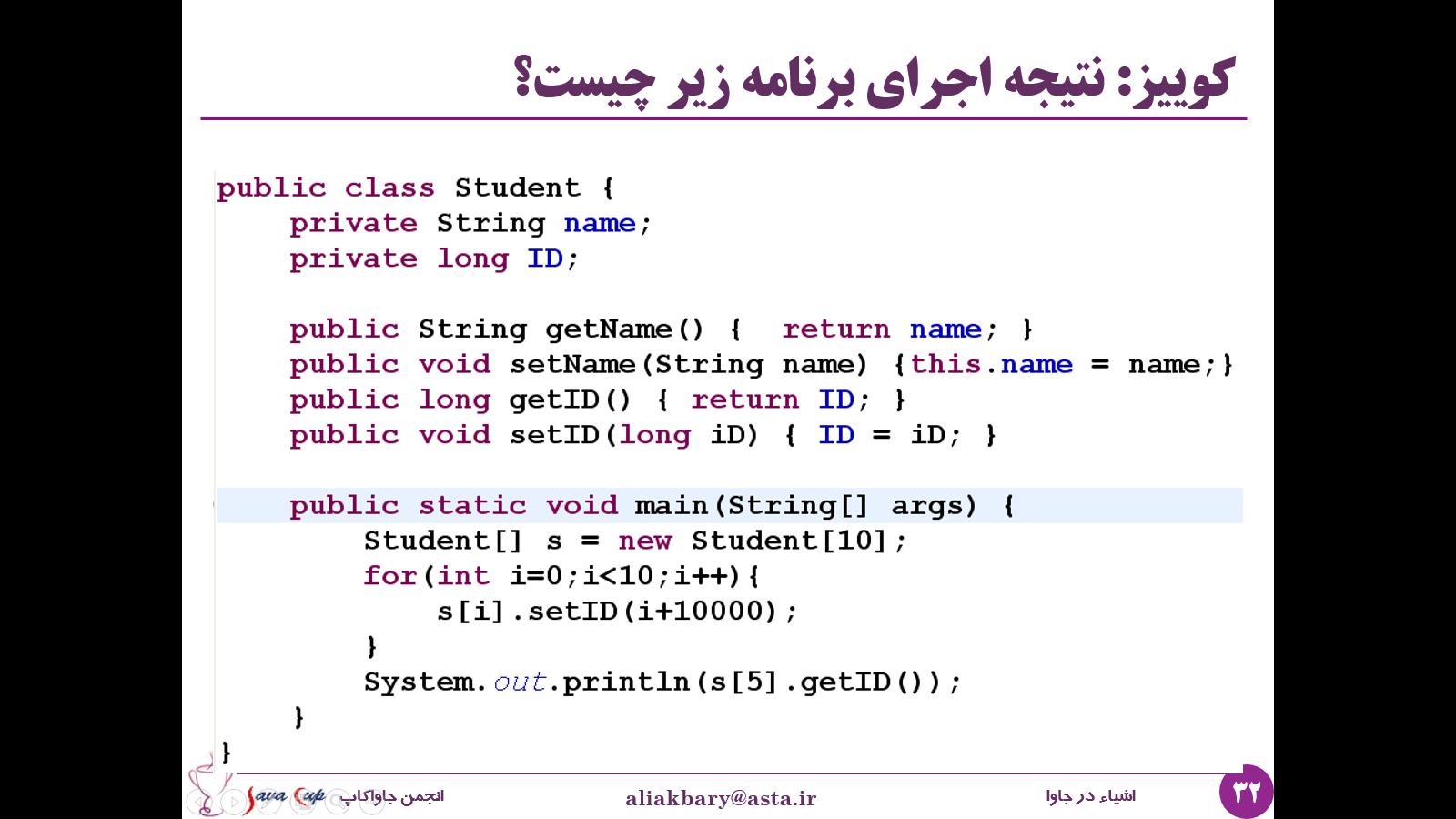


وضعیت حافظه چطور است؟ یک ارجاع در حافظه وجود دارد، یک متغیر به نام jafarAgha که دارد به جایی از حافظه اشاره می کند که در آنجا یک شی از جنس کلاس Strudent قرار دارد، داخل Student، ویژگی های مختلفی وجود دارد، مثلا name که خودش ارجاعی به یک رشته است که باید جای دیگری از حافظه نگهداری شود، یا مثلا age که یک int است (primitive است) و مقدار 50 را نگه می دارد و ارجاع به جای دیگری نیست و همین طور الی آخر ...



## کویز

برای مرور بخشی از مطالبی که با هم دیدیم، این کویز یا سوال کوتاه را ببینید. برنامه ی زیر شامل یک کلاس Student کامل و نیز متد main است. نتیجه ی اجرای آن چه خواهد بود؟



پاسخ:

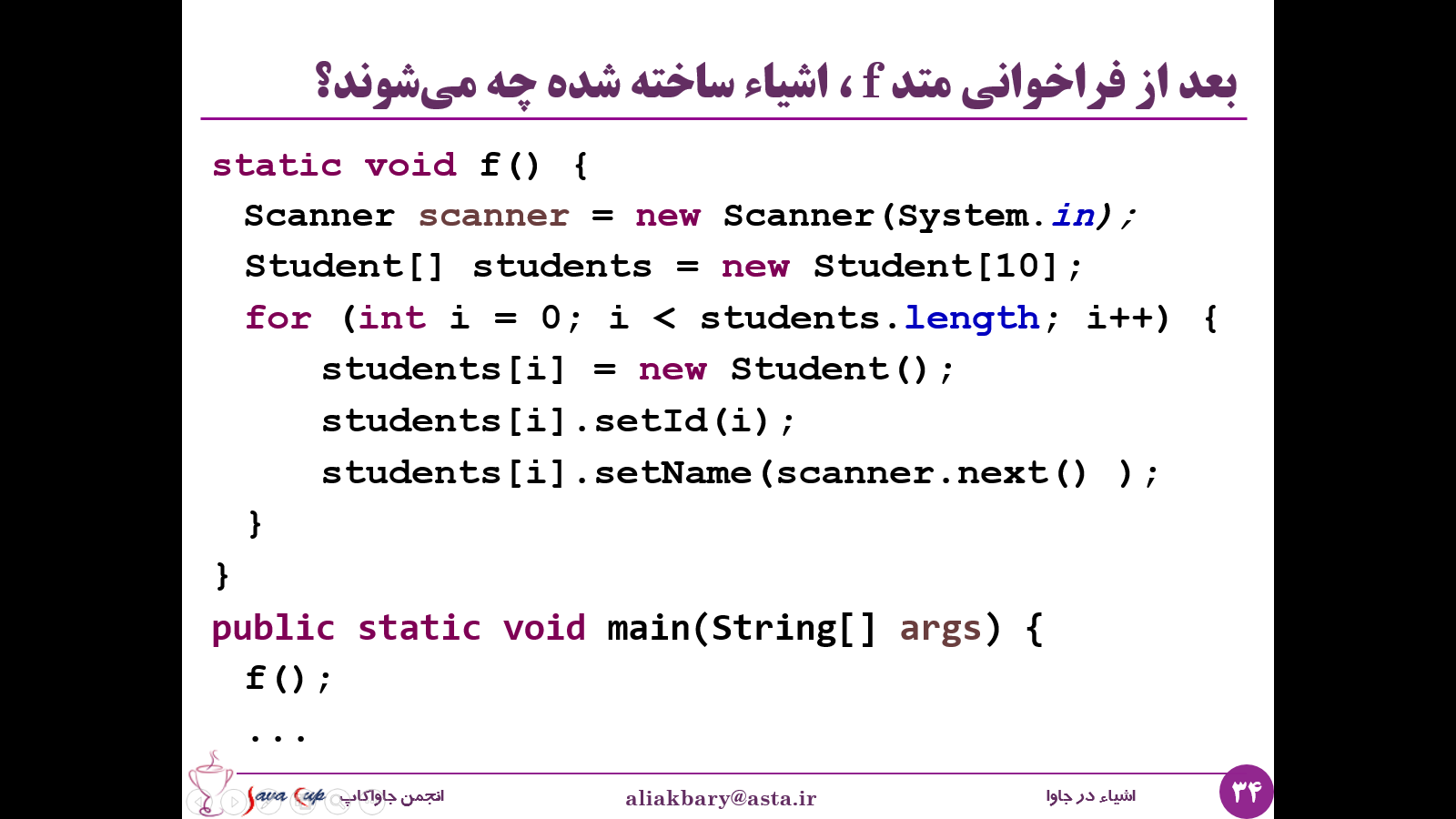
نتیجه ی اجرای این برنامه یک خطا در زمان اجراست. آن خطا در داخل بدنه ی حلقه ی for اتفاق می افتد. ما در واقع خیلی باید دقت بکنیم که s، یک آرایه ی 10تایی از student هاست که شامل 10 عنصر است که هر یک از این عناصر ارجاعی به یک Student هستند اما در ابتدا همگی null هستند. در واقع هر کدام از خانه های این آرایه قرار است ارجاع به یک شی باید اما الان ارجاع به هیچ شی ای نیست. بنابراین اگر ما از یکی از خانه های این آرایه استفاده کنیم و ویژگی یا متدی را روی آن فراخوانی کنیم، بلافاصله بعد از اجرا شدن این خط ما با یک خطا در زمان اجرا مواجه می شویم چون s[i] حتما null است و فراخوانی کردن یک ویژگی یا یک متد روی مقدار null باعث اصطلاحا NullPointerException می شود. پس باید دقت کنیم اگر می خواهیم آرایه ای ایجاد کنیم و این آرایه از جنس انواع داده ی اولیه (Primitive Data Types) نیست، تک تک خانه های این آرایه باید به نحوی new شود، به نحوی ایجاد شوند تا تازه بتوانیم از آنها استفاده کنیم.

چرا قبلا در مثال هایی که داشتیم، تک تک خانه های مثلا String را new نمی کردیم؟ به این خاطر که ما هر کدام از آن ارجاعات را مساوی خروجی خوانده شده از Scanner قرار می دادیم و داخل آن متد next() از کلاس اسکنر عملا اپراتور new فراخوانی می شود و ما در مثال هایی که قبلا داشتیم روی چیزهایی که هنوز new نشده اند متد فراخوانی نمی کردیم.

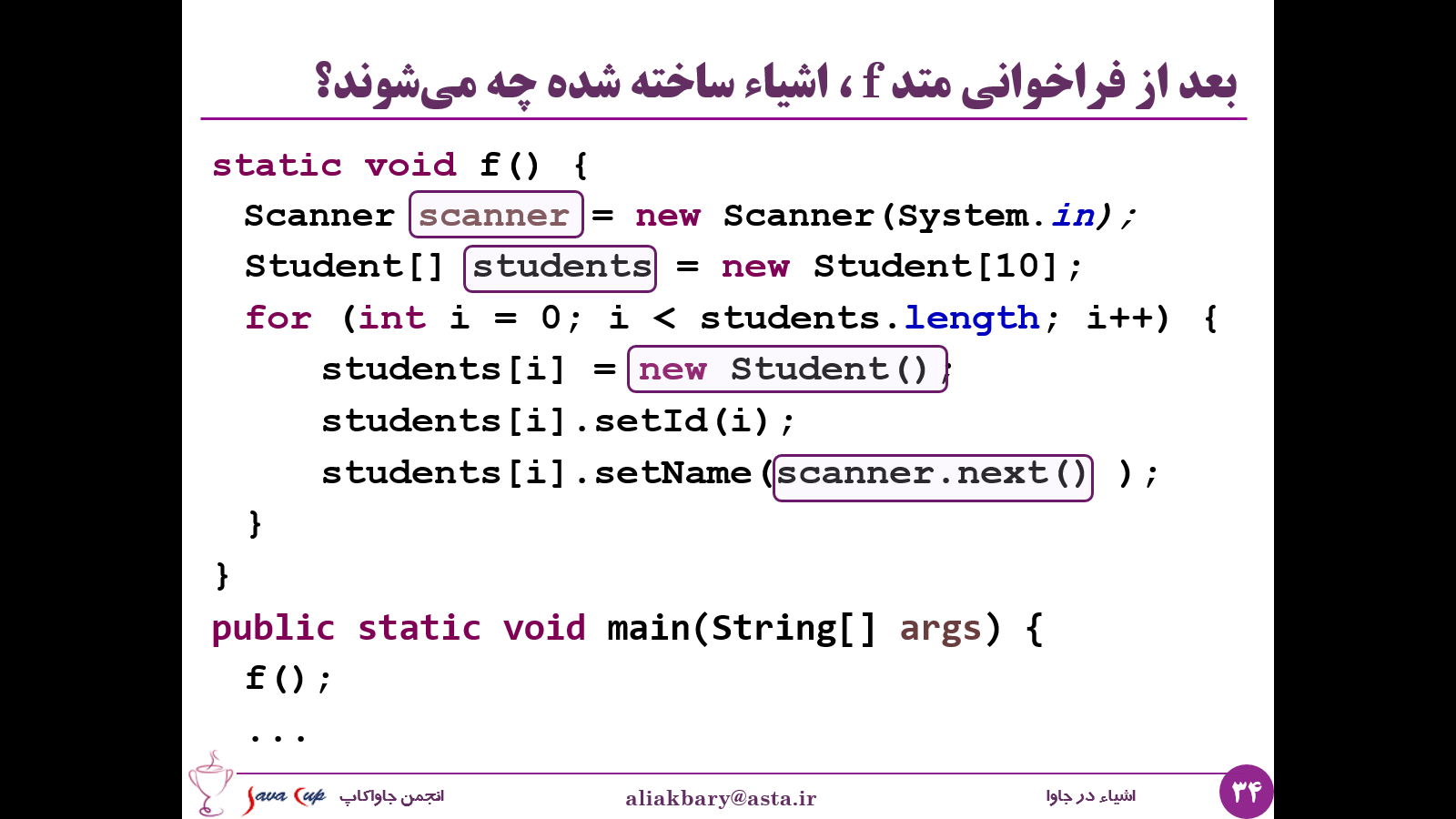
پس خیلی مهم است که این نکته را بفهمیم که چرا این مثال با خطا مواجه می شود.

## زباله روب (Garbage Collector)

در این بخش می خواهیم درباره ی آزادسازی حافظه ی اشیاء صحبت کنیم. مثال زیر را در نظر بگیرید، فرض کنید متدی مثل f وجود دارد که داخل آن یک اسکنر new شده و یک آرایه از دانشجویان و هر کدام از خانه های این آرایه مساوی یک شی جدید قرار گرفته است و ویژگی های آن شی جدید مقدار دهی شده است. فرض کنید در متدی مانند main از این متد استفاده کنیم.



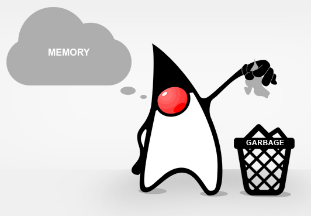
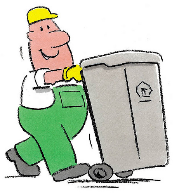
وقتی که فراخونی متد f() تمام می شود و از آن خارج می شویم، چه اتفاقی برای اشیایی که داخل این متد ساخته شده اند می افتد؟ ما اشیاء مختلفی داخل متد f() ساخته ایم که بعد فراخوانی متد f() دیگر قابل استفاده نیستند، اصطلاحا این اشیاء مردند و دیگر به آنها دسترسی نخواهیم داشت. اشیاءی مانند شی scanner، آرایه ی students، هر کدام از عناصر آرایه student که به یک شی جدید اشاره می کنند و هر کدام از آنها که یک ویژگی name از جنس شی String دارند و به شی دیگری اشاره می کند. این اشیاء بعد از فراخوانی متد f() چه می شوند؟ چه کاری باید با آنها انجام بدهیم؟



وقتی که دیگر نیازی به یک شی نیست، باید حافظه ی آن آزاد شود، وگرنه بعد از مدتی، حافظه ی هیپ پر از اشیایی می شود که دیگر قابل استفاده نیستند، پس باید اشیایی که دیگر قابل دست رسی نیستند و دیگر نیازی به آنها نیست، از حافظه ی هیپ حذف شوند.

برخی از زبان های برنامه نویسی، برنامه نویس را مجبور می کنند که خودش عملیات حذف کردن اشیا از حافظه را انجام دهد، مثلا در زبان C++ اپراتوری به نام delete وجود دارد که هر چیزی که new می شود، وقتی که استفاده از آن تمام شود باید توسط برنامه نویس به صورت دستی delete شود. به عبارت دیگر برنامه نویس مشخص می کند که چه زمانی حافظه ی یک شی باید آزاد شود.

اما آزاد کردن حافظه توسط برنامه نویس، کاری بسیار پر خطا و پیچیده است. هم برنامه نویسی را سخت و پیچیده می کند و هم اشتباهات برنامه نویسی را زیاد می کند، چون ممکن است برنامه نویس به اشتباه چیزی که باید delete می کرده را delete نکند و یا شی ای که هنوز مورد استفاده است را به اشتباه delete کند. یا فراموش کند که حافظه ی تخصیص داده شده را آزاد کند. اگر برنامه ی ما پیچیده شود و اشیاء به هم ارجاعات متعددی داشته باشند، این فرآیند حذف کردن اشیاء از حافظه بسیار سخت و پرخطا خواهد بود.

خبر خوب این است که در جاوا، آزاد سازی حافظه ی اشیاء بر عهده ی برنامه نویس نیست و این کار به صورت خودکار انجام می شود و نیازی به دخالت برنامه نویس نیست. این کار تحت عنوان فرآیند زباله روبی (Garbage Collection) انجام می شود. موجودی وجود دارد به نام Garbage Collector یا زباله روب که اشیای مرده را در حافظه پیدا می کند و حافظه ی آنها را آزاد می کند. زباله روب بخشی از JVM است و تمام برنامه های ما به زبان جاوا بر روی JVM اجرا می شود و بخشی از ماشین مجازی جاوا به صورت خودکار اشیایی که در حافظه هستند اما دیگر در برنامه قرار نیست استفاده شوند را پیدا می کند و حافظه ی مربوط به آنها را آزاد می کند.

Garbage Collector هر از چندگاهی به حافظه ی هیپ سر می زند و آن را بررسی می کند، اشیاء مرده را پیدا می کند، یعنی اشیایی که مثلا دیگر کسی به آنها اشاره نمی کند و در واقع دیگر قابل دست رسی و استفاده نیستند، و حافظه ی آنها را آزاد می کند و به این ترتیب حافظه برای ساخت اشیاء جدید باز می شود.

## فراخوانی متد

در این بخش می خواهیم درباره ی نحوه ی ارسال پارامترها به متدها در زبان جاوا صحبت کنیم. همان طور که می دانید فراخوانی یک متد ممکن است شامل ارسال پارمترهایی باشد. ما وقتی یک متد را روی یک شی فراخوانی می کنیم، ممکن است داخل این متد پارامترهایی تعریف شده باشند و ما آنها را برای آن متد ارسال کنیم. اما ارسال پارامترها به متدها در جاوا چطور است؟

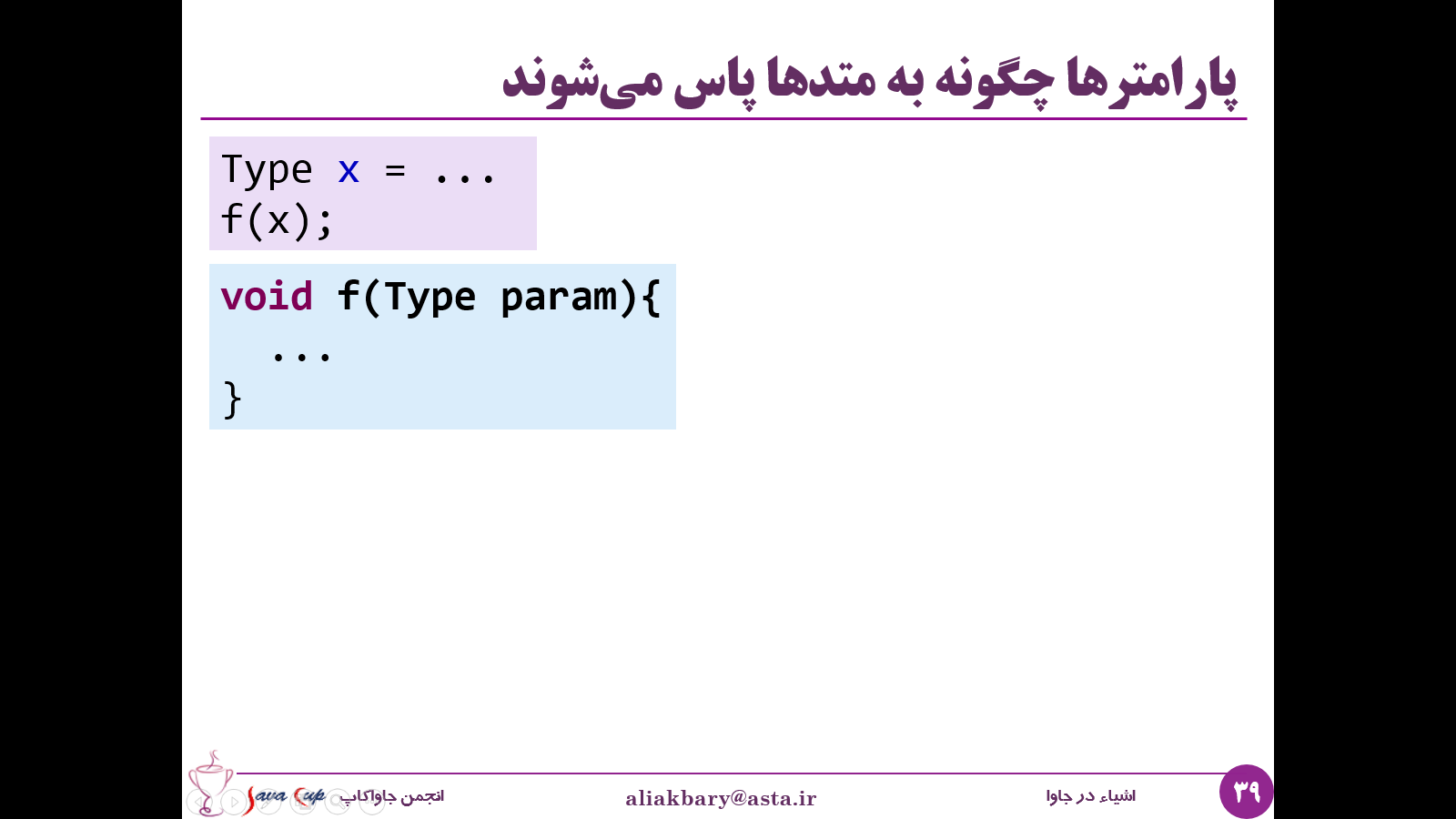
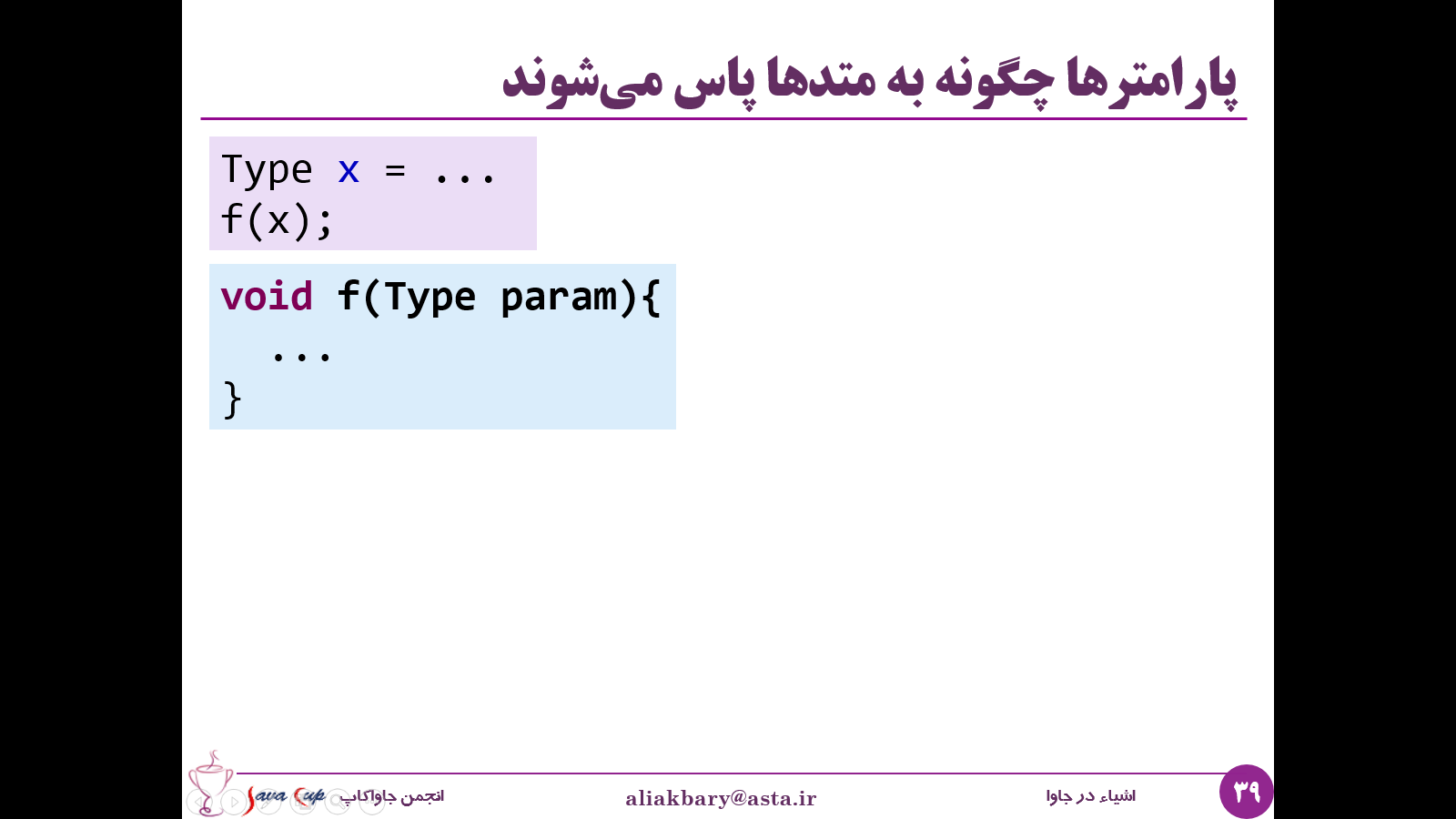
شاید اصطلاحات زیر را شنیده باشید:

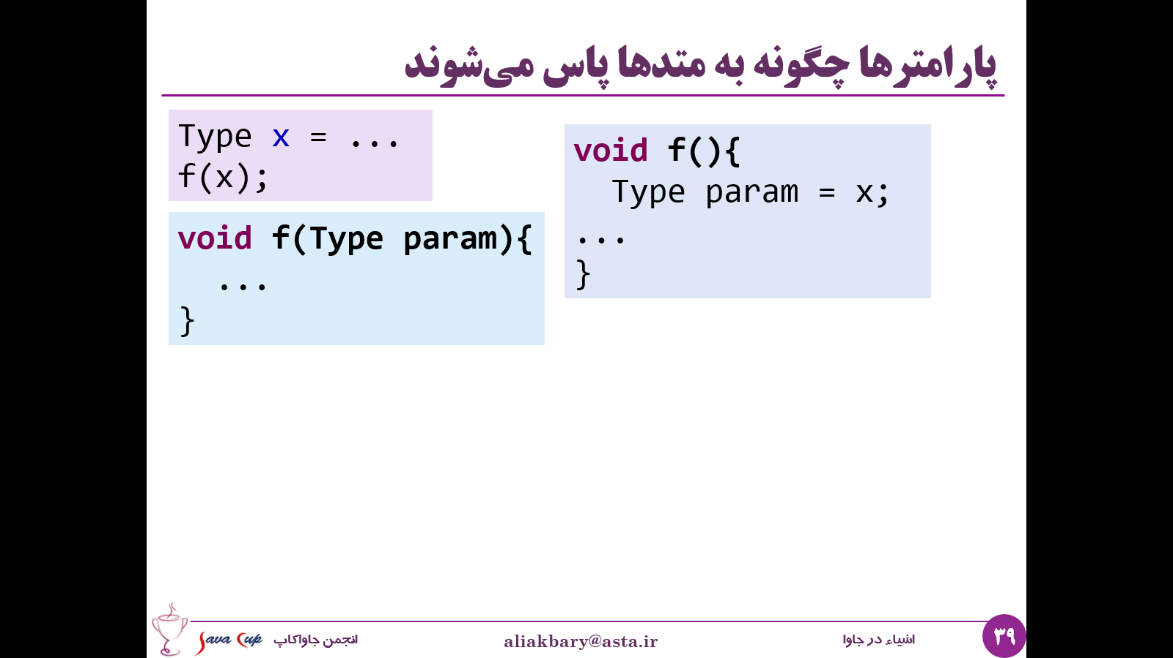
* Call by value
* Call by reference
* Call by pointer

این اصطلاحات برای توصیف کردن نحوه ی ارسال پارامترها به متدها به کار می روند و در بعضی از زبان های برنامه نویسی معنا دارند. در جاوا اگر ما یک متغیر از جنس primitive یا انواع داده ی اولیه را به یک متد پاس کنیم، مثلا یک int را، در واقع به شیوه ی call by value عمل کرده ایم. یعنی متد فقط از مقدار این متغیر primitive استفاده می کند. اگر یک شی را به یک متد ارسال کنیم، در واقع ارجاع آن شی به آن متد پاس شده است و داخل آن متد از ارجاع شی پاس شده استفاده می کنیم و در واقع کاری که جاوا با اشیاء می کند شبیه call by pointer است. من به عمد اصرار دارم که از اصطلاح call by reference برای ارسال پارامترها به متدها در جاوا استفاده نکنم چون در واقع call by reference معنی خاص تری می دهد و در زبان هایی مثل C++ یا C# این روش ارسال پارامتر هم پیاده سازی شده است.

## پارمترها چگونه به متدها پاس می شوند؟

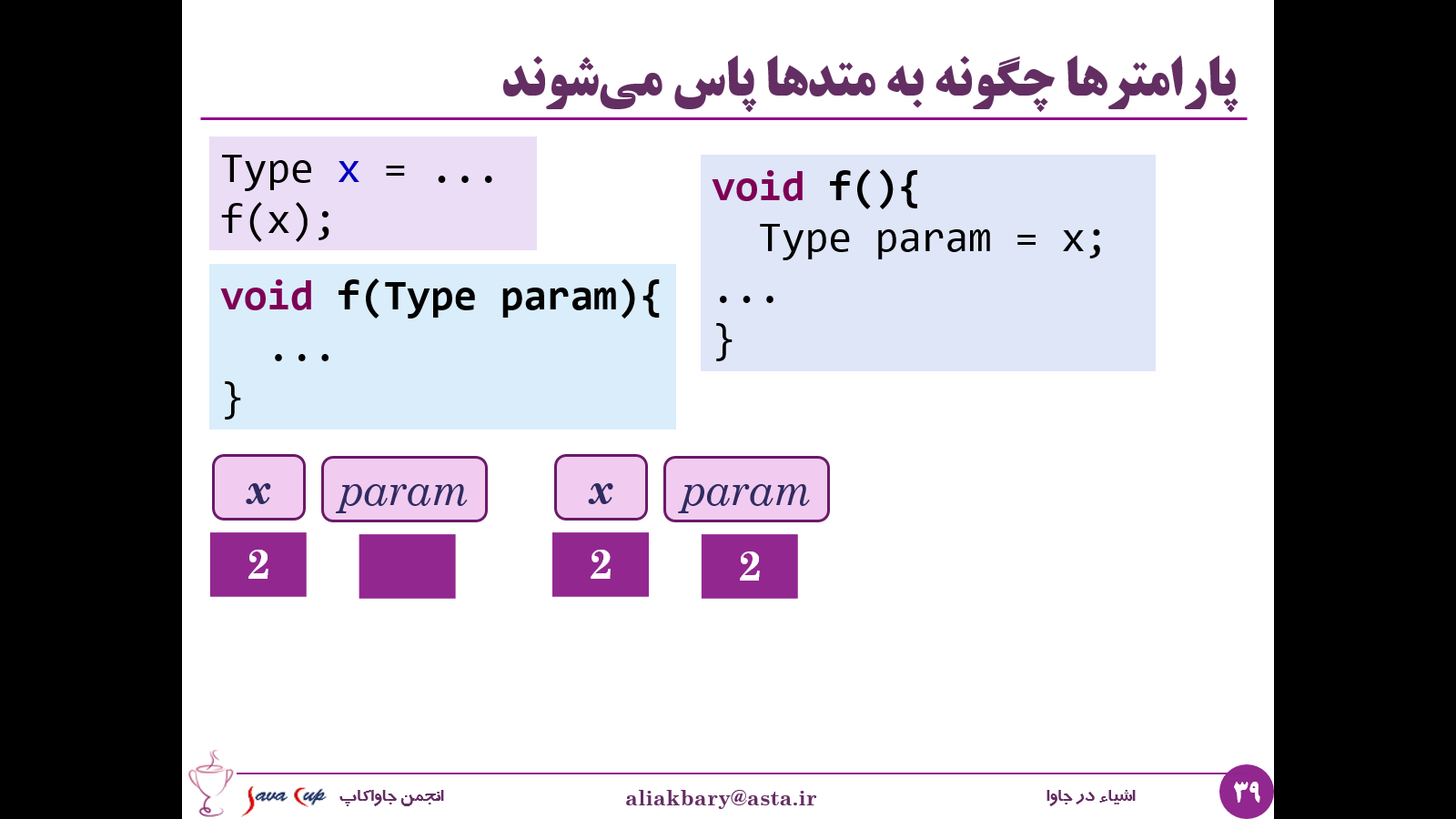
حالا مقدار دقیق تر ببینیم که جاوا با پارامترها می کند؟ فرض کنید یک متغیر داریم به نام x از که از هر جنسی می تواند باشد و فرض کنید که x را به یک متد مانند f() پاس کرده ایم، پس f() متدی است که پارامتری از جنس x می گیرد.



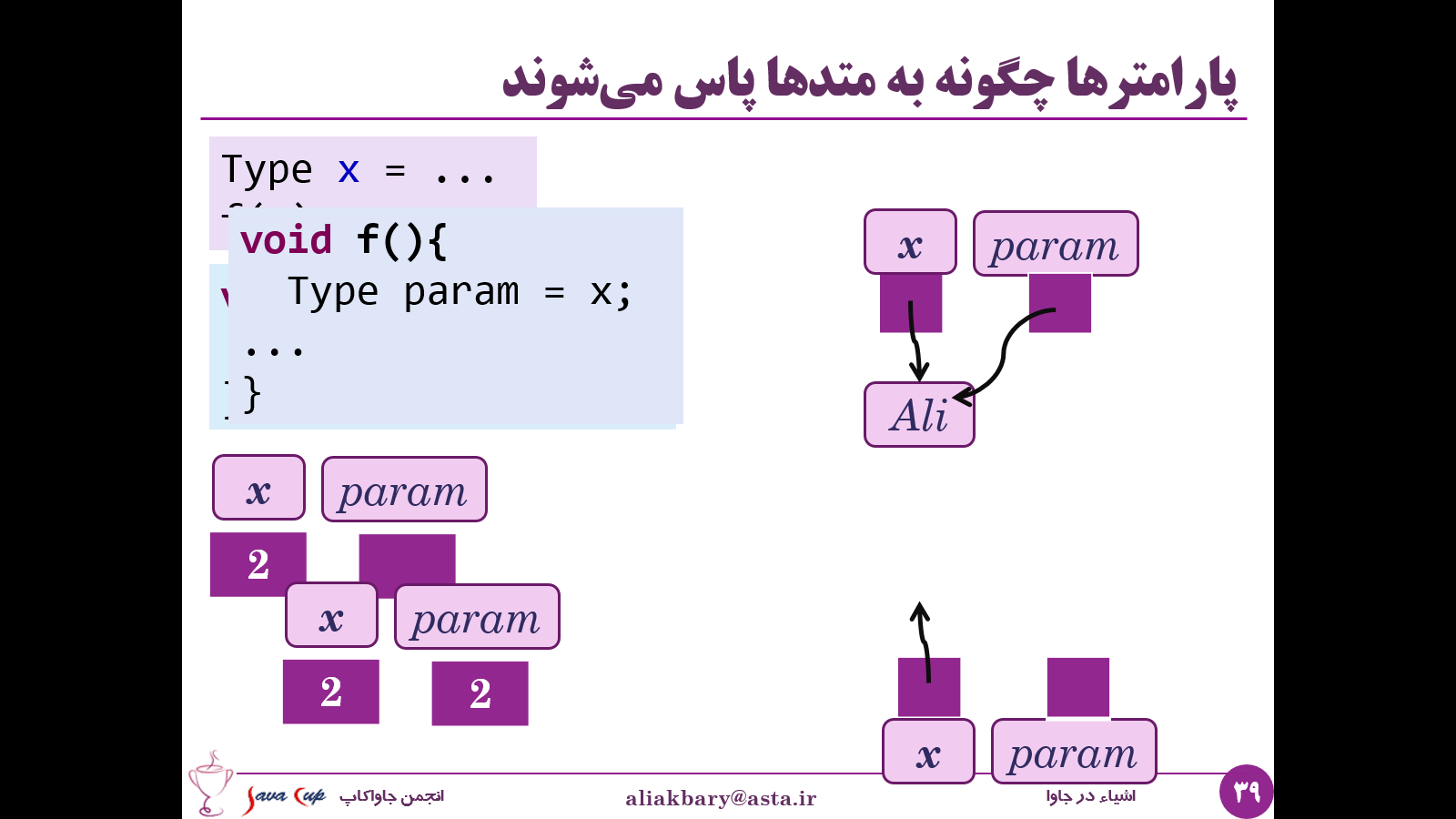
نحوه ی رفتار این متد مثل این است که انگار متد f() پارامتری ندارد بلکه یک متغیر محلی دارد به نام param که این متغیر مساوی همان x که به این متد پاس کرده ایم قرار گرفته است. این فرض از نظر نحوی غلط است اما می خواهیم شهود بهتر داشته باشیم.

پس هر پارامتر در واقع مانند یک متغیر محلی است و این متغیر محلی با کمک اپراتور مساوی (assignment)، مساوی مقدار که به آن متد پاس شده است قرار می گیرد. اگر چند پارامتر داشته باشیم هم این اتفاق برای تک تک آنها خواهد افتاد.

حال اگر Type یک نوع داده ی اولیه باشد مثلا int و مقدار x مثلا 2 باشد، اگر x به f() پاس شود، ما داخل متد f() متغیر محلی param را خواهیم داشت که یک متغیر جداگانه از x است ولی محتوای x داخل آن کپی شده است. پس ما دو متغیر x و param داریم که هر کدام خانه های حافظه ی خودشان را دارند اما مقدارشان با همدیگر مساوی است. در این حالت داخل متد f() ما دیگر هیچ دسترسی به x نداریم و تنها کپی مقدار آن در param در اختیار ماست.

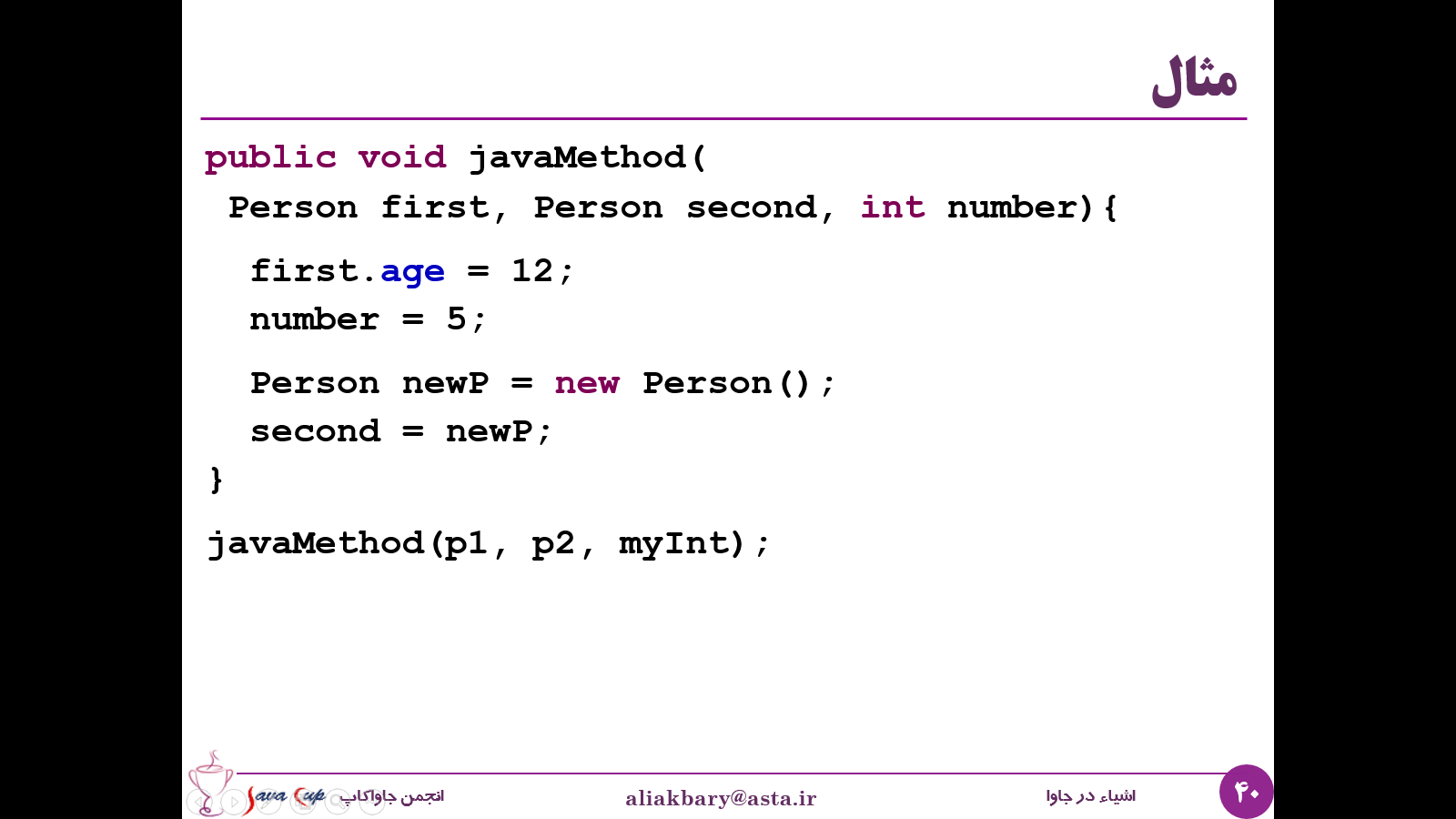


اما اگر x و param از جنس reference باشند، یعنی Type مثلا String باشد یا یک شی که خودمان ایجاد کرده ایم، در این صورت x یک ارجاع به یک شی خواهد بود و param هم یک ارجاع به یک شی است و وقتی که می نویسیم param = x، الان می دانیم که محتوای param که ارجاع به یک شی است به همان جایی اشاره می کند که x به آن اشاره می کرده. این بار شی ای کپی نشده، و شی ای که x به آن اشاره می کرده سر جای خودش هست، فقط یک اشاره گر و ارجاع جدید به همان شی ایجاد شده است.

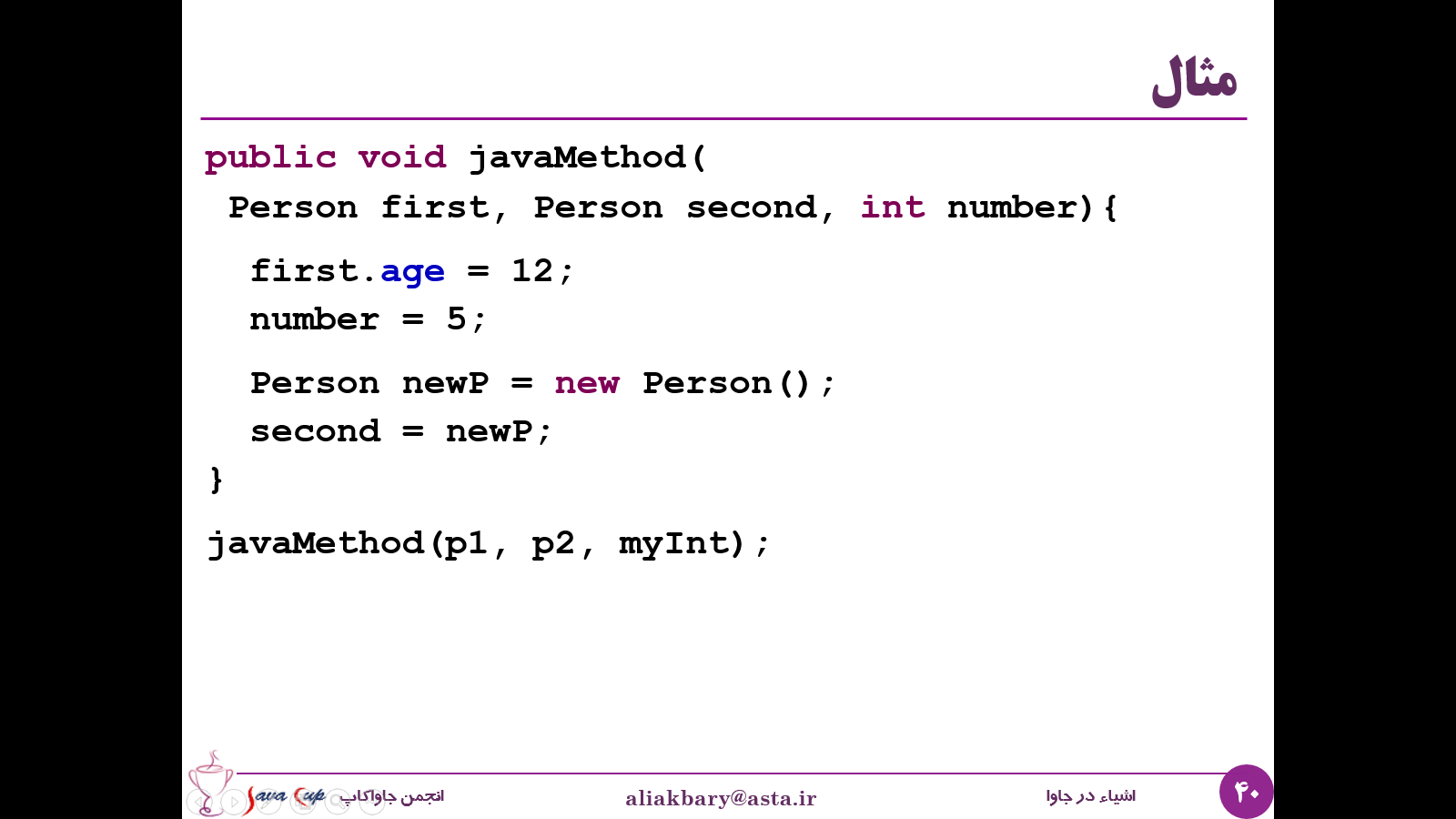


در واقع در حالت اول که پارامتر تابع از نوع داده ی اولیه بود مقدار را کپی می کردیم، اما در حالت دوم مقدار شی کپی نمی شود، بلکه از ارجاع به شی کپی ایجاد می شود.

مثال: این مفاهیم را یک بار در قالب یک مثال با هم دوره کنیم. فرض کنید متدی داریم به نام javaMethod که سه پارامتر دارد، دو پارامتر اول از جنس Person و در واقع ارجاع هستند و پارامتر سوم از جنس انواع داده ی اولیه است و یک مقدار رو نشان می دهد.



فرض کنید داخل این متد first.age را تغییر دهیم، و مقدار number را هم عوض کنیم و خود اشاره گر second را هم که یک ارجاع به Person است، مساوی یک شی جدید قرار دهیم، یعنی یک شی جدید بسازیم و second به این شی جدید اشاره کند. با این اوصاف فرض کنید javaMethod را با ورودی های زیر فراخوانی کنیم، بدیهی است که فرض کرده ایم اشیای p1 و p2 وجود دارند و متغیری به نام myInt هم مقدار دهی شده است و نیز ویژگی age در کلاس Person وجود دارد و public است و می توانیم از آن استفاده کنیم.



سوال اول اینجاست، آیا بعد از فراخوانی این متد، مقدار p1.age تغییر کرده است؟ بله، چون first ارجاع و اشاره گری هست که به همان جایی اشاره می کند که p1 به آن اشاره می کند، پس اگر با کمک first ویژگی age را تغییر دهیم سراغ همان Object ای رفتیم که p1 به آن اشاره می کرده، پس ویژگی age از شی p1 تغییر می کند.

سوال بعد، آیا myInt بعد از فراخوانی این متد تغییر می کند؟ مثلا اگر قبل از این فراخوانی مقدار 3 داشته، بعد از فراخوانی مقدار آن 5 می شود؟ خیر، myInt یک primitive value است و وقتی آن را پاس می کنیم و تحت عنوان number در متد از آن استفاده می کنیم، مقدار myInt کپی می شود داخل متغیر number. پس number مقدارش 3 می شود چون مقدار myInt 3 بوده است، حال اگر number = 5 را اجرا کنیم، متغیر محلی ما که داخل تابع هست مقدارش عوض می شود و متغیر myInt سر جایش هست و بدون هیچ تغییری باقی می ماند.

و سوال آخر، آیا مقدار p2 تغییر می کند؟ p2 که به یک شی اشاره می کرده آیا به خاطر دو دستور زیر:

**Person newP = new Person();**

**second = newP;**

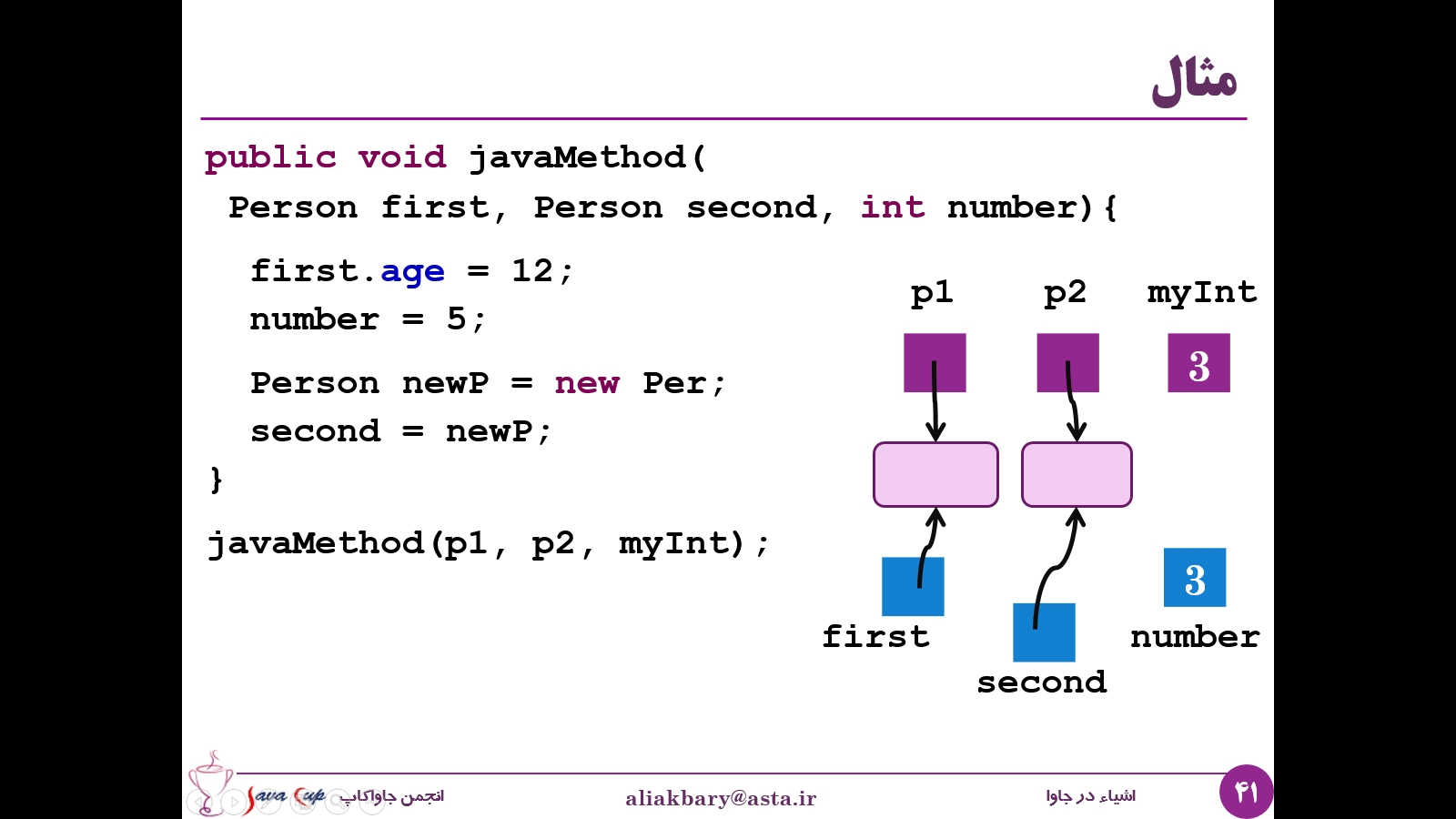
به یک شی دیگر اشاره خواهد کرد؟ سوال مهم و دقیقی هست و پاسخ این است که نه! P2 به شی جدیدی اشاره نمی کند، به همان شی ای که قبلا اشاره می کرده اشاره می کند و محتوا و هویت آن اصلا تغییری نمی کند. به این دقت کنید p2 یک ارجاع بوده و در ابتدا داخل متد second به همان جایی که p2 اشاره می کرده، اشاره می کند، حال ما داخل متد مقدار second را تغییر داده ایم و گفته ایم که این متغیر محلی second به یک جای دیگر اشاره بکند و به یک شی جدید ارجاع داشته باشد اما تغییر دادن این متغیر محلی باعث تغییر دادن p2 نمی شود و p2 همچنان به همان شی قبلی که اشاره می کرده، اشاره می کند.

یک بار به صورت گرافیکی نحوه ی ارسال پارامترها و تغییرات آنها در این متد را ببینیم:

همان طور که می دانید، p1، p2 و myInt سه متغیر هستند که دوتای اول به اشیایی اشاره می کنند و myInt یک مقدار مثلا 3 را نگهداری می کند.

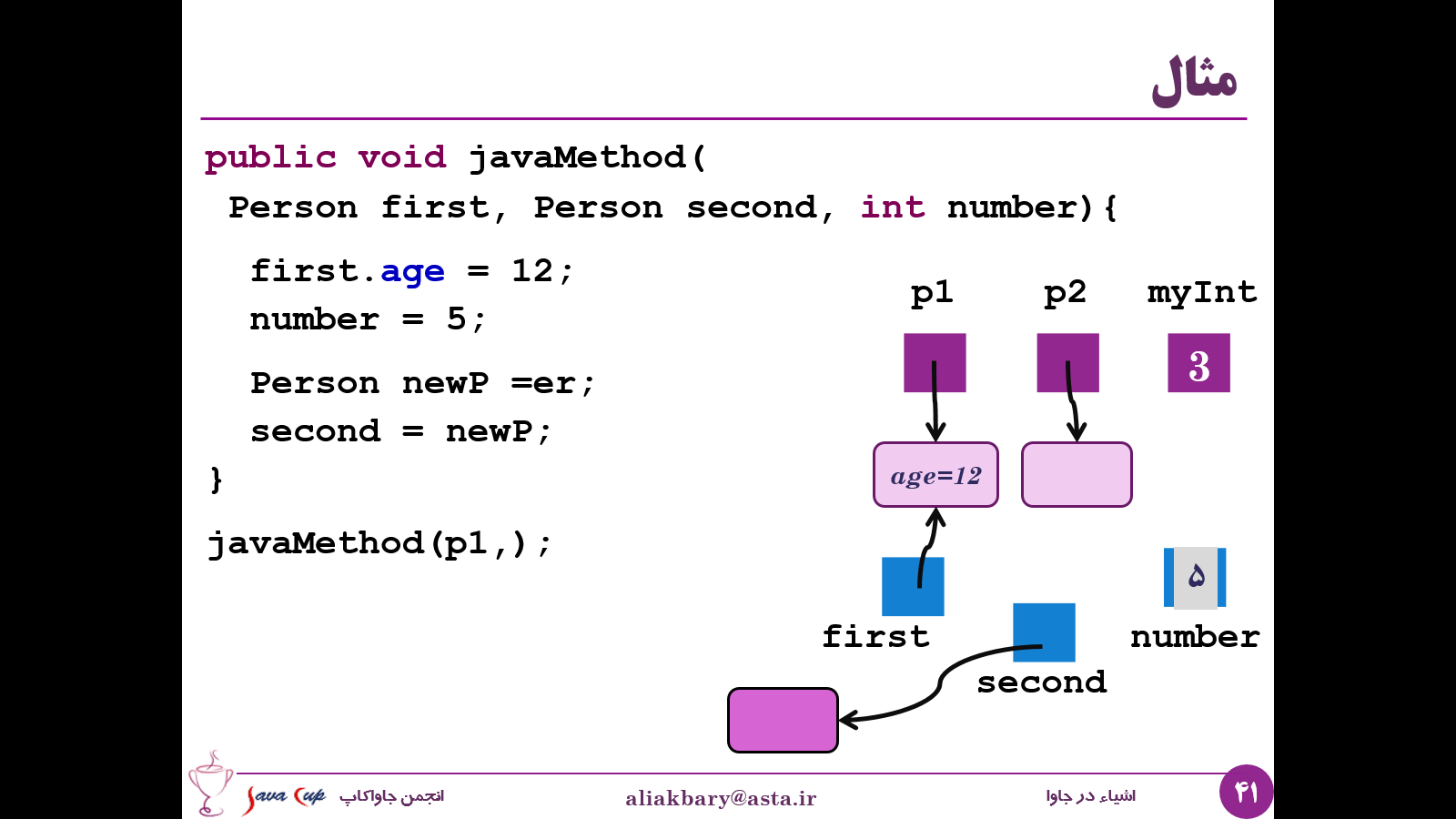


وقتی که این پارامترها پاس می شود به متد javaMethod، متغیرهای محلی به نام first، second و number داریم که هر کدام متناظر یکی از این پارامترها قرار می گیرند، مثلا first به همان جایی اشاره می کند که p1 به آن اشاره می کند، second به همان جایی اشاره می کند که p2 به آن اشاره می کند و number هم همان مقدار را نگهداری می کند که myInt نگهداری می کرده.



پس تک تک مقادیر ارسال شده به تابع، در متغیرهای محلی خود تابع کپی می شوند، کپی شدن متغیرهایی از نوع داده ی اولیه به یک شکل است و کپی شدن اشیاء هم به صورت کپی شدن ارجاعات است. نکته ی مهم این است که این متغیرها (متغیرهای ارسال شده به تابع با متغیرهای محلی خود تابع) از هم مجزا هستند و هر کدام جای مستقلی در حافظه برای خودشان دارند.

حال وقتی می گوییم first.age = 12، پس ویژگی سن از شی ای که first به آن اشاره می کرده مساوی 12 می شود. وقتی می گوییم number = 5، مقدار این متغیر برابر 5 می شود و وقتی می گوییم second مساوی یک شی جدید باشد، پس second دیگر به شی قبلی که اشاره می کرده اشاره نخواهد کرد و به شی جدید ساخته شده اشاره می کند.

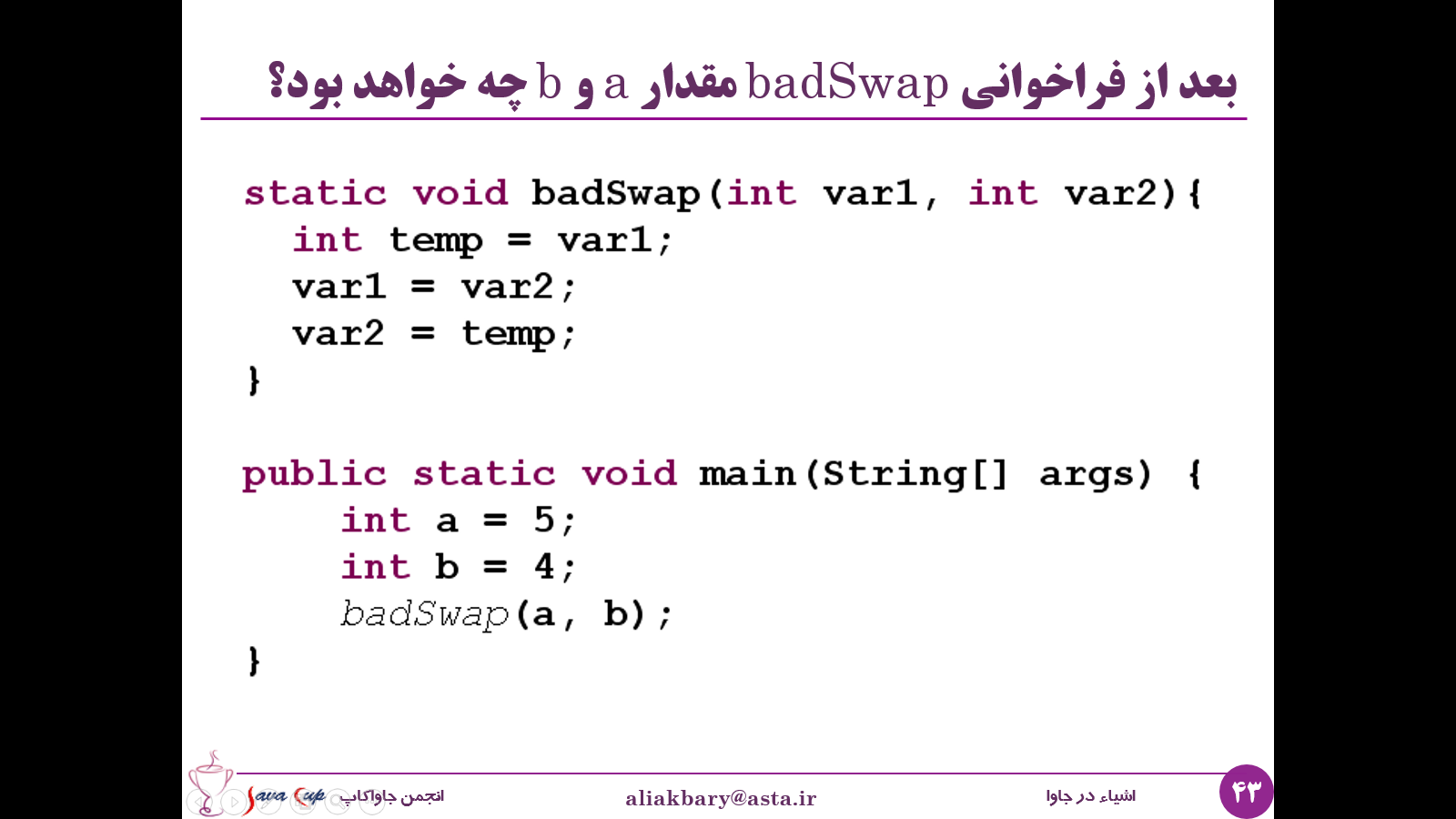


حال برمی گردیم به آن 3 سوال، آیا p1.age بعد از فراخوانی این متد تغییر کرد؟ بله. آیا myInt تغییر کرده؟ نه، کپی آن تغییر کرده و عمر آن کپی هم تمام شده است بعد از بازگشت از تابع. آیا p2 تغییر کرد؟ نه، p2 همچنان به همان شی ای که اشاره می کرد اشاره می کند و این second بوده که به یک شی جدید اشاره کرده و second هم متغیر محلی بوده که طول عمرش تمام شده اما p2 سر جاش باقی مانده.

## کویز

بگذارید مفهوم ارجاعات و مدل های پاس شدن به متدها را با چند سوال کوچک مرور کنیم.

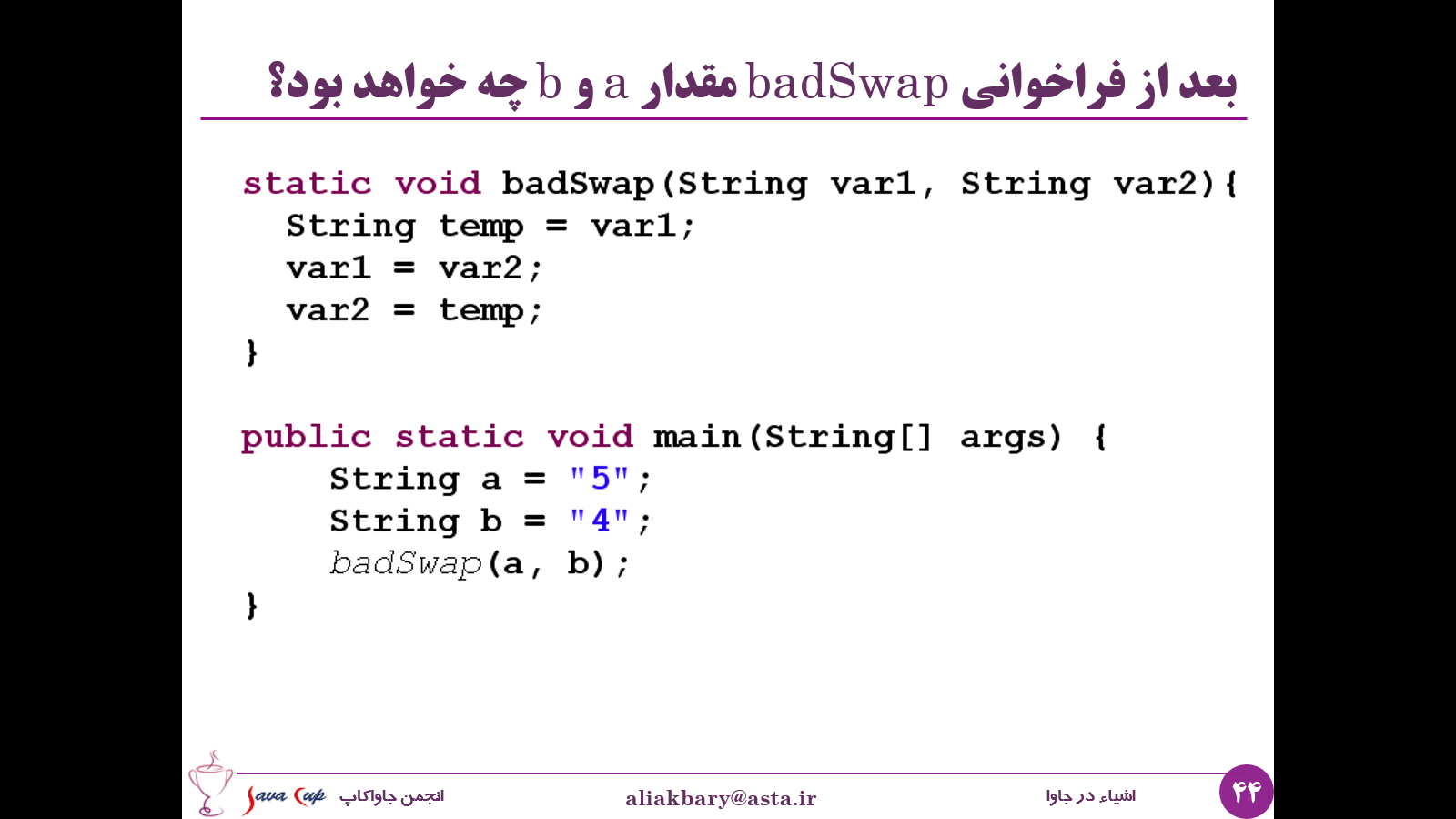
* فرض کنید متدی به نام badSwap به شکل زیر تعریف شده باشد، به صورت زیر در main فراخوانی شود، بعد از فراخوانی badSwap متغیرهای a و b چه مقداری خواهند داشت؟



پاسخ:

مقدار a و b بعد از فراخوانی این متد تغییری نمی کند، در نتیجه مقدار a همچنان 5 و مقدار b همچنان 4 خواهد بود. چون یک کپی از مقدار a و یک کپی از مقدار b تحت عنوان متغیرهای var1 و var2 در تابع استفاده شده اند و مقدار آنها با هم عوض شده است و اصل متغیر a و b هرگز تغییر نخواهند کرد.

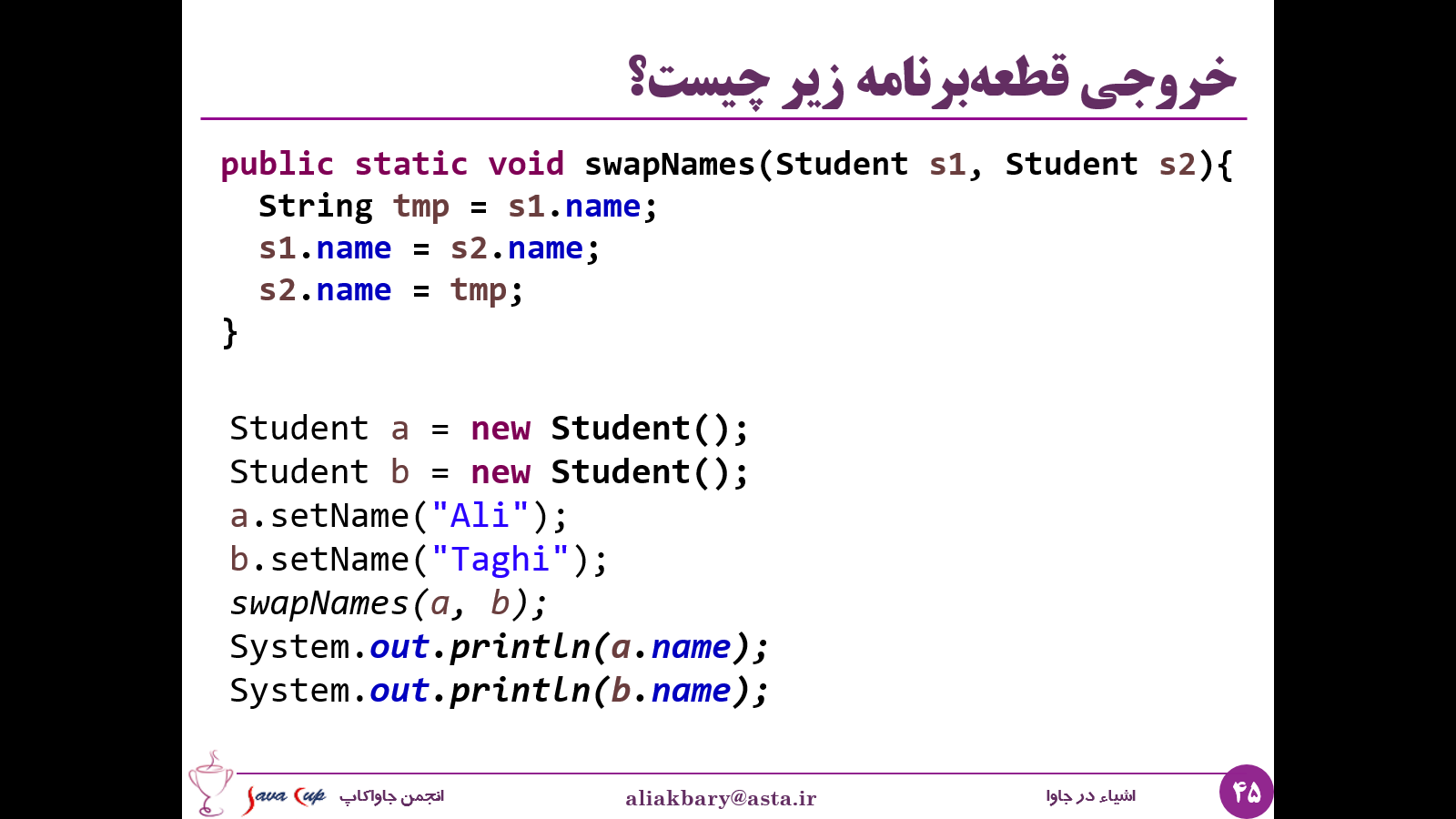
* فرض کنید که این بار متد badSwap پارامترهایی از جنس ارجاع مثلا String داشته باشد به جای int. در این حالت بعد از فراخوانی این متد مقدار a و b چه خواهد بود؟



پاسخ:

باز هم مقدار a و b در اثر فراخوانی چنین متدی تغییر نخواهد کرد. a همچنان مقدار 5 و b همچنان مقدار 4 خواهد داشت و تنها ارجاعاتی که تحت عنوان var1 و var2 به صورت متغیرهای محلی در این تابع وجود داشته اند جا به جا شده اند و روی مقادیر خود a و b تاثیری ندارند. به عبارت دیگر هویت ارجاع a و هویت ارجاع b هرگز با پاس شدن به متد تغییر نخواهد کرد. این نکته ی مهمی هست که هیچ متدی در جاوا نمی تواند هویت پارامترهایی که به آن پاس می شوند را تغییر بدهد.

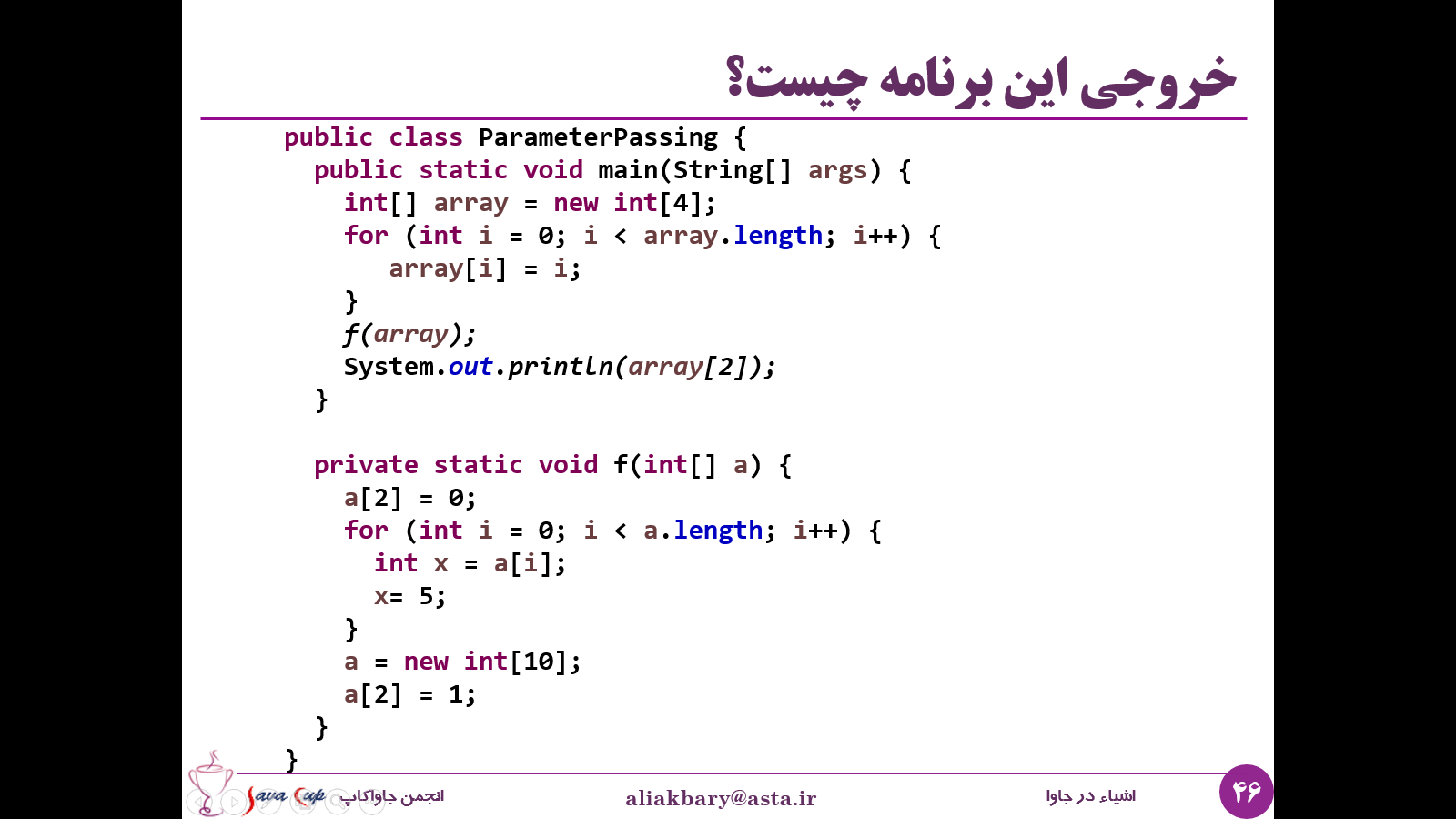
* حال این برنامه را در نظر بگیرید. متدی به نام swapNames داریم که دو شی از جنس Student به عنوان ورودی می گیرد. فرض کنید که هر Student ویژگی name دارد و متد setName هم دارد، اگر دو شی a و b تعریف بکنیم و ویژگی نام آنها را مقدار دهی کنیم، سپس متد swapNames رو برای آنها فراخوانی کنیم و بعد از بازگشت از متد، آنها را چاپ نماییم، مقدار آنها چه خواهد بود؟ بدون تغییر خواهد بود؟ یا جا به جا می شود؟ و یا اتفاق دیگری رخ خواهد داد؟



پاسخ:

در این مثال a.name مقدار Tsghi خواهد داشت و b.name مقدار Ali. به عبارت دیگر ویژگی name این دو شی با همدیگر swap شده اند. چرا؟ چون در این مثال، s1 و s2 به همان جایی اشاره می کنند که a و b اشاره می کنند. بنابراین اگر با کمک s1 و S2 محتوای اشیای را تغییر دهیم، محتوای a و b هم تغییر می کند. پس در جاوا می توانیم با کمک ارجاع هایی که به صورت پارامتر به متدها پاس می شوند محتوای اشیاء را تغییر دهیم، می توانیم وضعیت و state آنها را تغییر دهیم اما هویت آنها را نمی توانیم. به عبارت دیگر هویت پارامترهای ارسال به تابع همواره حفظ می شود و پس از فراخوانی متد بدون تغییر می ماند، و به همان جایی که قبلا اشاره می کردند اشاره می کنند اما محتوا و حالت آنها می تواند تغییر کند.

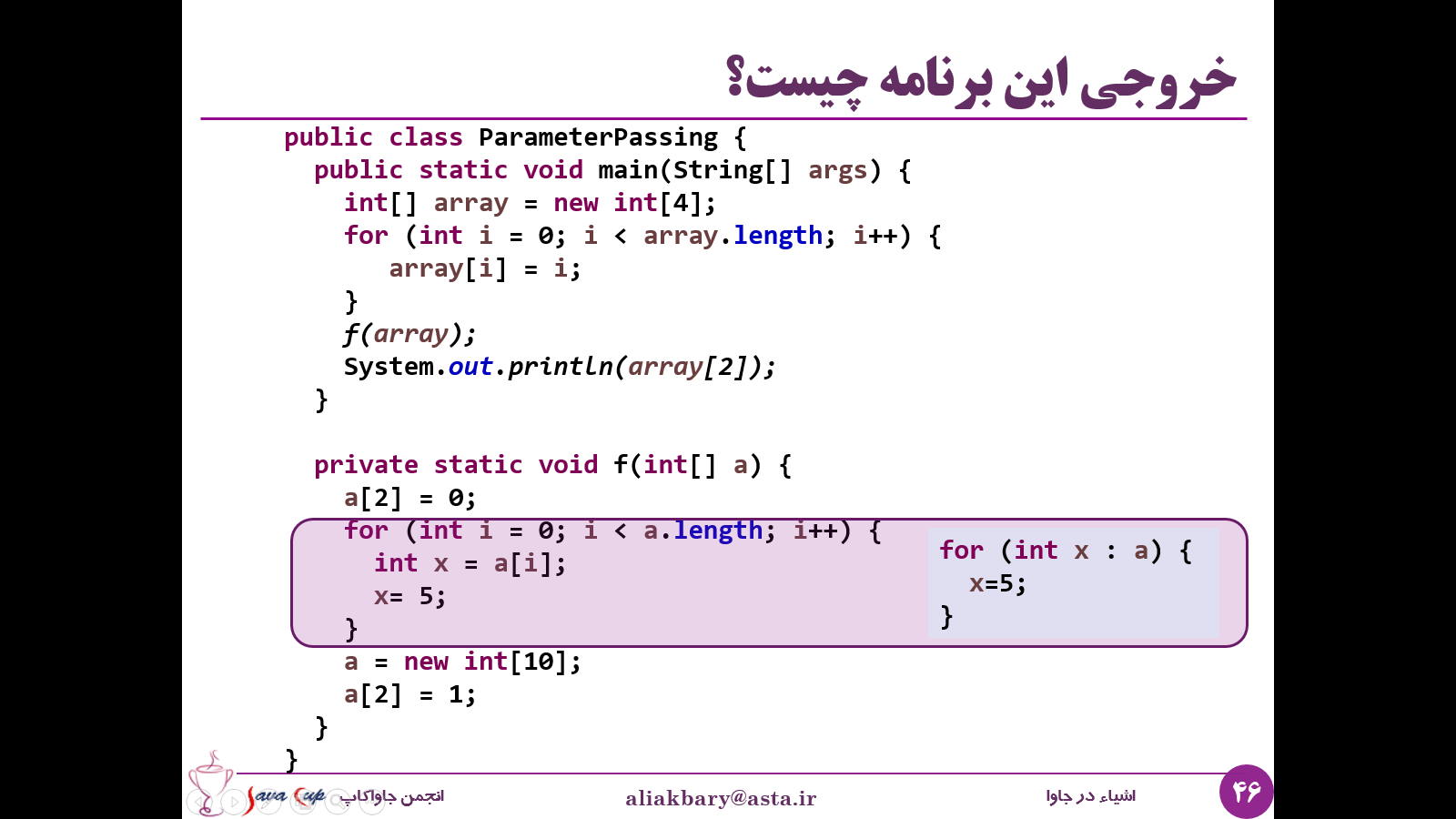
* فرض کنید که متدی داریم مانند f() که آرایه ای به عنوان پارامتر می گیرد و طبق کدی که در زیر می بینید با آن کارهایی انجام می دهد، اگر یک آرایه به آن پاس شود و پس از بازگشت، خانه ی دوم از این آرایه چاپ شود، خروجی این برنامه چه خواهد بود؟



پاسخ:

پاسخ این سوال عدد صفر است. اگر دقت کنید در ابتدا خانه های آرایه مقادیر 0,1,2,3 دارند، اما هنگامی که این آرایه به تابع پاس می شود، a به همان جایی ارجاع می دهد که array به آن اشاره می کند، پس هنگامی که a[2] = 0 می شود، خانه ی دوم array هم تغییر می کند و مساوی صفر می شود. پس تا اینجا می دانیم که array[2] مساوی صفر است. در ادامه یک حلقه ایجاد شده و در این حلقه یک متغیر محلی ساخته شده و هر بار مساوی یکی از عناصر این آرایه قرار گرفته و تغییر کرده، تغییر این متغیر محلی، هیچ بلایی سر عناصر این آرایه نمی آورد و element های این آرایه سر جای خود می مانند. پس از آن ما این متغیر محلی a را مساوی یک شی جدید از جنس آرایه قرار داده ایم، پس از این به بعد a به یک آرایه ی دیگر اشاره می کند، بنابراین اگر خانه ی دوم این شی جدید را تغییر دهیم، شی قبلی که به این متد پاس کرده بودیم، تغییری نمی کند چرا که ما آرایه ی جدیدی را تغییر داده ایم و آرایه ی قبلی سر جای خود هست و تنها با کمک reference خارج از تابع ما یعنی array بعد از این به آن دسترسی داریم. پس در تمام طول این متد به جز خط اول آرایه ای که به متد پاس شده بدون تغییر می ماند و خط اول هم خانه ی دوم آرایه را صفر می کند و خروجی برنامه صفر خواهد بود.

سوال: اگر حلقه ی for در این متد با for each بازنویسی شود، و به شکل زیر نوشته شود، خروجی برنامه چه خواهد بود؟



پاسخ این است که هیچ فرقی نمی کند، چرا که این بازنویسی دقیقا معادل کد قبلی است و در اینجا هم یک متغیر محلی به نام x وجود دارد که با تغییر آن عناصر آرایه تغییر نمی کنند.

## بخش های مختلف آرایه

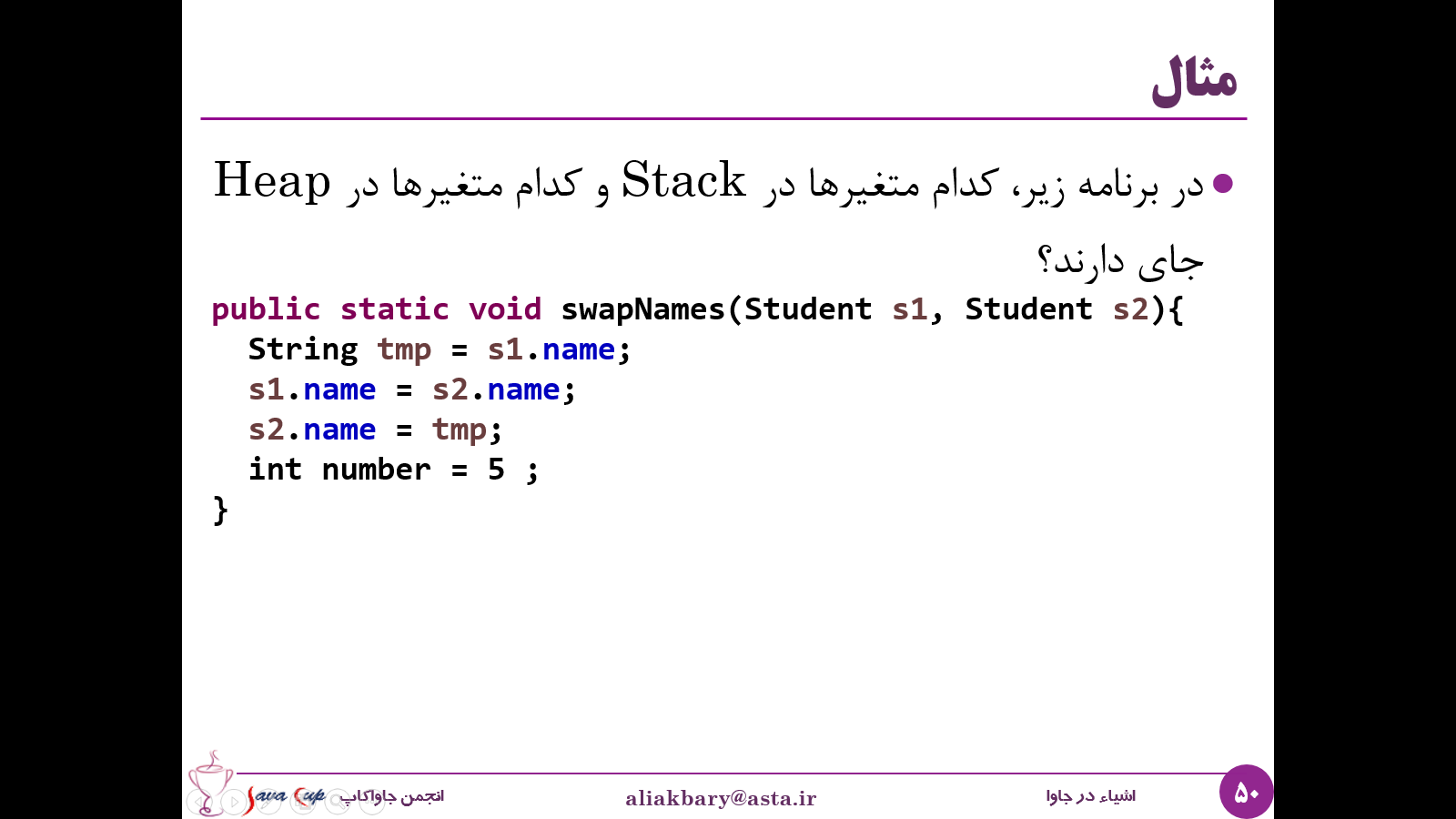
در این قسمت می خواهیم بخش های مختلف یک برنامه ی جاوا را بررسی کنیم. همانطور که می دانید داده های یک برنامه، یعنی متغیرها و مقادیری که در برنامه استفاده می شوند، در حافظه قرار می گیرند و به صورت عادی این داده ها در حافظه ی RAM هستند. اینکه متغیرها و اشیاء می توانند در حافظه ی ثانویه مانند دیسک سخت قرار بگیرند یا برخی از داده ها می توانند در حافظه های سریع تر از RAM هم قرار بگیرند امکان دارد، اما بحث فعلی ما نیست.

یک برنامه جاوا بخشی از حافظه را استفاده می کند که شامل دو بخش مهم است: Stack و Heap. اشیاء در هیپ قرار می گیرند و متغیرهای محلی هر متد در پشته (Stack). البته در هیپ و پشته اتفاقات دیگری هم رخ می دهد که فعلا تمرکز ما روی این دو بحث است. پس هر شی جدیدی که ایجاد می شود جاش در بخش هیپ است. پس استفاده از اپراتور new موجب قرار گرفتن شی جدید روی هیپ خواهد شد و اشیایی که بعد از مدتی می میرند و دیگر کسی از آنها استفاده نمی کند و کسی به آنها دسترسی ندارد توسط زباله روب جمع آوری و آزاد می شوند.

متغیرهای محلی هر متد روی پشته قرار می گیرند و وقتی فراخوانی متد تمام می شود، تمام آنها از روی پشته پاک می شوند، پس نحوه ی عملکرد به این ترتیب است که وقتی یک متد فراخوانی می شود، به تعداد متغیرهای محلی اش روی فضا به آن اختصاص داده می شود و اگر آن متد، متد دیگری را فراخوانی کند، باز روی Stack برای این متد جدید (برای متغیرهایش) فضا روی پشته اختصاص داده می شود و وقتی که یک متد تمام می شود، همه ی متغیرهایش از روی پشته پاک می شوند اما این کار هیچ نیازی به زباله روب ندارد و Garbage Collector تنها وظیفه اش تمیز کردن هیپ است. تمیز کردن روی Stack یک فرآیند اتوماتیک است که در همه ی زبان های برنامه نویسی که فراخوانی متد یا تابع دارند انجام می شود و اتفاقی ساده و سطح پایین محسوب می شود. این تمیز کردن هیپ است که یک اتفاق خاص و نسبتا پیچیده است و جاوا برای آن امکانی فراهم کرده که زبان هایی مانند C و C++ از آن بی بهره اند. پس حتی در زبان های بدون زباله روب هم پاک سازی پشته در پایان فراخوانی یک متد به صورت اتوماتیک انجام می شود.

مثال:

در برنامه زیر اگر فرض کنیم متد swapNames فراخونی شود، متغیرهای مختلفی در این متد وجود دارند که برخی روی پشته و برخی روی هیپ قرار می گیرند و می خواهیم در این باره بیشتر صحبت کنیم:

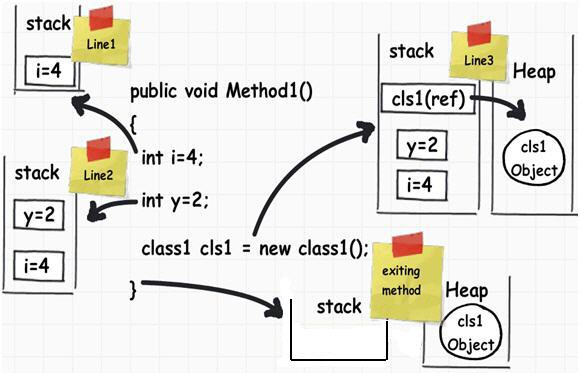


کدام یک از متغیرهای موجود در این تابع روی پشته و کدام یک روی هیپ قرار می گیرند؟ اگر به صورت گرافیکی بخواهیم به این فضا نگاه کنیم این برنامه بخشی به نام Stack و بخشی به نام Heap دارد. هنگامی که این متد فراخوانی می شود به ازای همه ی متغیرهای محلی، روی پشته حافظه ایجاد می شود، مثلا این متد متغیر محلی به نام s1 دارد که جزو پارامترهای این متد است، و یا متغیر محلی temp که اینها از جنس ارجاع هستند و یک متغیر number هم داریم که از جنس primitive است. پس number در اینجا یک مقدار نگه می دارد و 3 متغیر دیگر که ارجاع هستند به جایی در هیپ اشاره می کنند.



مثال:

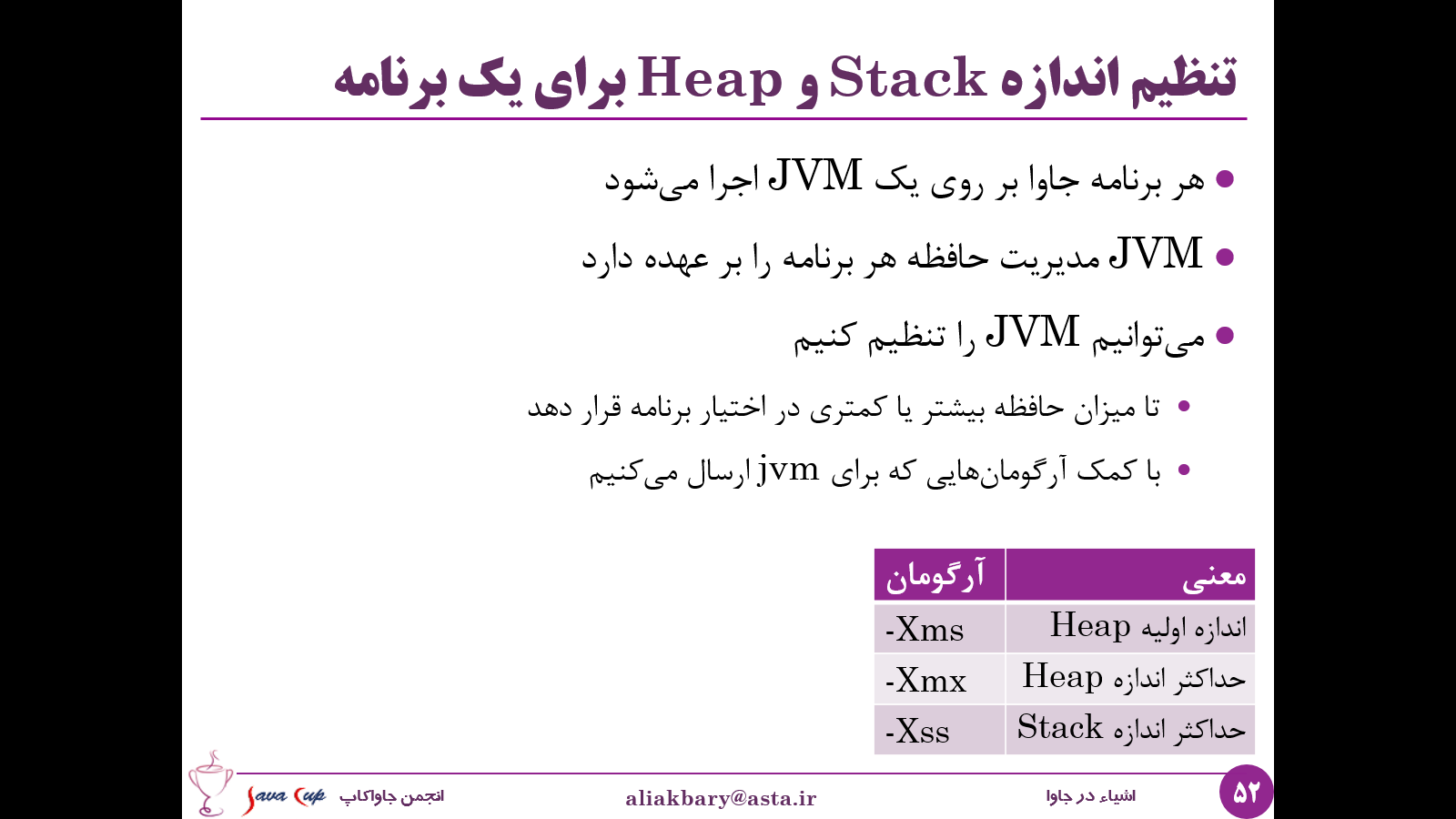
یک متد دیگر را ببینیم. در این مثال اگر Mthod1() فراخوانی شود، چه اتفاقی می افتد؟ قدم اول، یک متغیر محلی به نام i با مقدار 4 ساخته شده است، پس روی پشته یک جا برای این متغیر باز می شود، قدم دوم، یک متغیر primitive دیگر با مقدار 2 و نام y که باز هم روی پشته جای می گیرد. قدم سوم، یک ارجاع با نام cls1 ساخته شده است، پس باز روی Stack یک reference ایجاد می شود که قرار است به جایی در هیپ اشاره کند. و چون در همین خط ما new هم انجام داده ایم، این اپراتور یک جا در هیپ ایجاد می کند، و مرجع آن را برمی گرداند و cls1 به آن محل اشاره می کند. حال وقتی این متد تمام می شود چه اتفاقی می افتد؟ هر آنچه روی پشته مربوط به این متد است آزاد می شود و این کار به سادگی و با سرعت انجام می شود اما شی ای که تا پیش از cls1 به آن اشاره می کرد و روی هیپ ایجاد شده بود به صورت بلااستفاده روی هیپ باقی می ماند تا بعدها زباله روب سر فرصت بیاید و آن را حذف کند. پس لزوما این شی بلافاصله بعد از اتمام متد آزاد نمی شود و هر از چند گاهی Garbage Collector به هیپ مراجعه می کند و اشیاء مرده را پیدا می کند و آنها را پاک می کند.



## تنظیم اندازه Stack وHeap برای یک برنامه

ما در برنامه های جاوا می توانیم اندازه پشته و هیپ را هم تنظیم نماییم. گاهی لازم است که اندازه ی بزرگتری برای مثلا پشته به برنامه مان بدهیم. اگر در برنامه مان از اشیاء زیاد و بزرگی استفاده می کنیم، ممکن است به هیپ بزرگتری نیاز داشته باشیم و اگر فراخوانی متدهای بیشتری داشته باشیم، شاید به پشته ی بزرگتری نیاز داشته باشیم.

هر برنامه جاوا بر روی JVM اجرا می شود، و ماشین مجازی جاوا مدیریت حافظه ی برنامه ها را بر عهده دارد. ما می توانیم JVM را تنظیم کنیم تا میزان حافظه ی کمتر یا بیشتری به برنامه ای که دارد اجرا می کند اختصاص دهد، و این کار را می توانیم با کمک آرگومان هایی که برای ماشین مجازی جاوا ارسال می کنیم انجام دهیم. آرگومان هایی که در این زمینه می توان برای JVM تنظیم کرد به شرح زیر است:



Xms حداقل و مقدار اولیه ی حافظه ی هیپ که برنامه ی ما از آن استفاده خواهد کرد را مشخص می کند، Xmx حداکثر این مقدار را مشخص می کند، یعنی در طول اجرای برنامه که ممکن است اشیاء جدیدی مرتبا ساخته شوند، حداکثر فضایی که برنامه ما می تواند استفاده کند برای هیپ با این آرگومان مشخص می شود. و حداکثر اندازه ای که برای Stack می توان اندازه گرفت با کمک Xss تنظیم می شود.

چند مثال ببینیم. ما می دانیم که برای اجرای کلاسی مانند Person می توانیم دستور زیر را فراخوانی کنیم:

* java Person

به این ترتیب JVM اجرا می شود و کلاس Person را فراخوانی می کند. اما گفتیم که می توانیم پارامترهایی را به JVM پاس کنیم، مثلا:

* java -Xms512m -Xmx3750m Person

این دستور یعنی قرار است جاوا کلاس Person را اجرا کند و حداقل 512 مگابایت و حداکثر 3750 مگابایت حافظه هیپ برای این برنامه در نظر بگیرد. پس دقت کنید بلافاصله بعد از عبارت -Xms یک عدد و بعد واحد آن می آید و نشان دهنده ی میزان حافظه است. یا:

* java -Xss4m Test

یعنی برنامه Test فراخوانی و اجرا شود و حداکثر 4 مگابایت برای پشته ی آن در نظر گرفته شود. و یا ترکیبات مختلفی از این تنظیمات مثلا:

* java -Xmx3750m -Xss4m Main

که هم حداکثر اندازه هیپ و هم حداکثر اندازه پشته را مشخص می کند. و اگر هر کدام از اینها را مشخص نکنیم یک مقدار پیش فرض در نظر گرفته می شود.

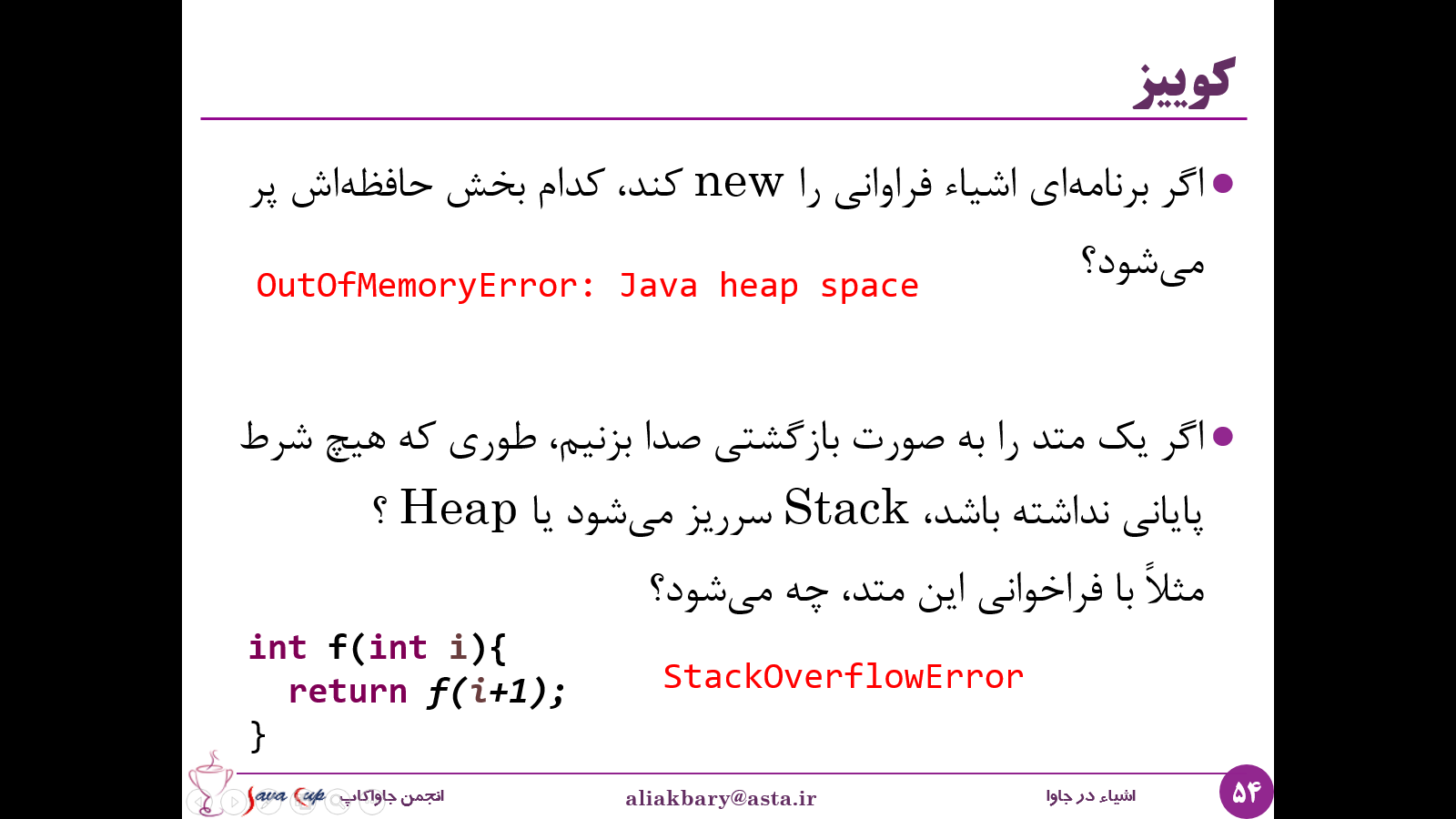
## کویز

چند سوال کوتاه را به عنوان کویز با هم مرور کنیم:

* اگر برنامه ای اشیاء فراوانی را new کند، کدام بخش حافظه اش پر می شود؟ Stack آن سرریز می کند و یا هیپ؟

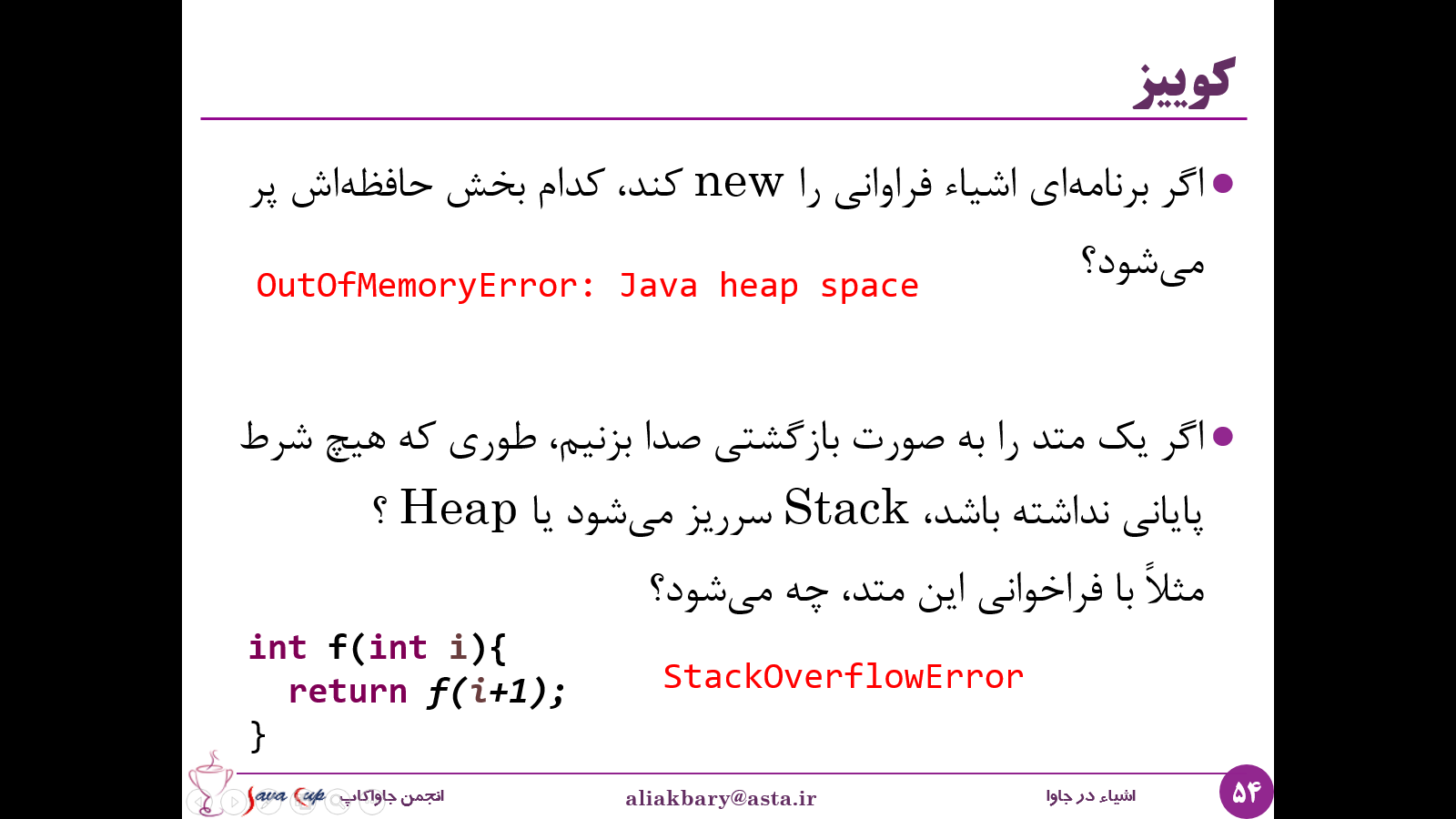
پاسخ:

در چنین مواقعی خطای زیر به وجود می آید:

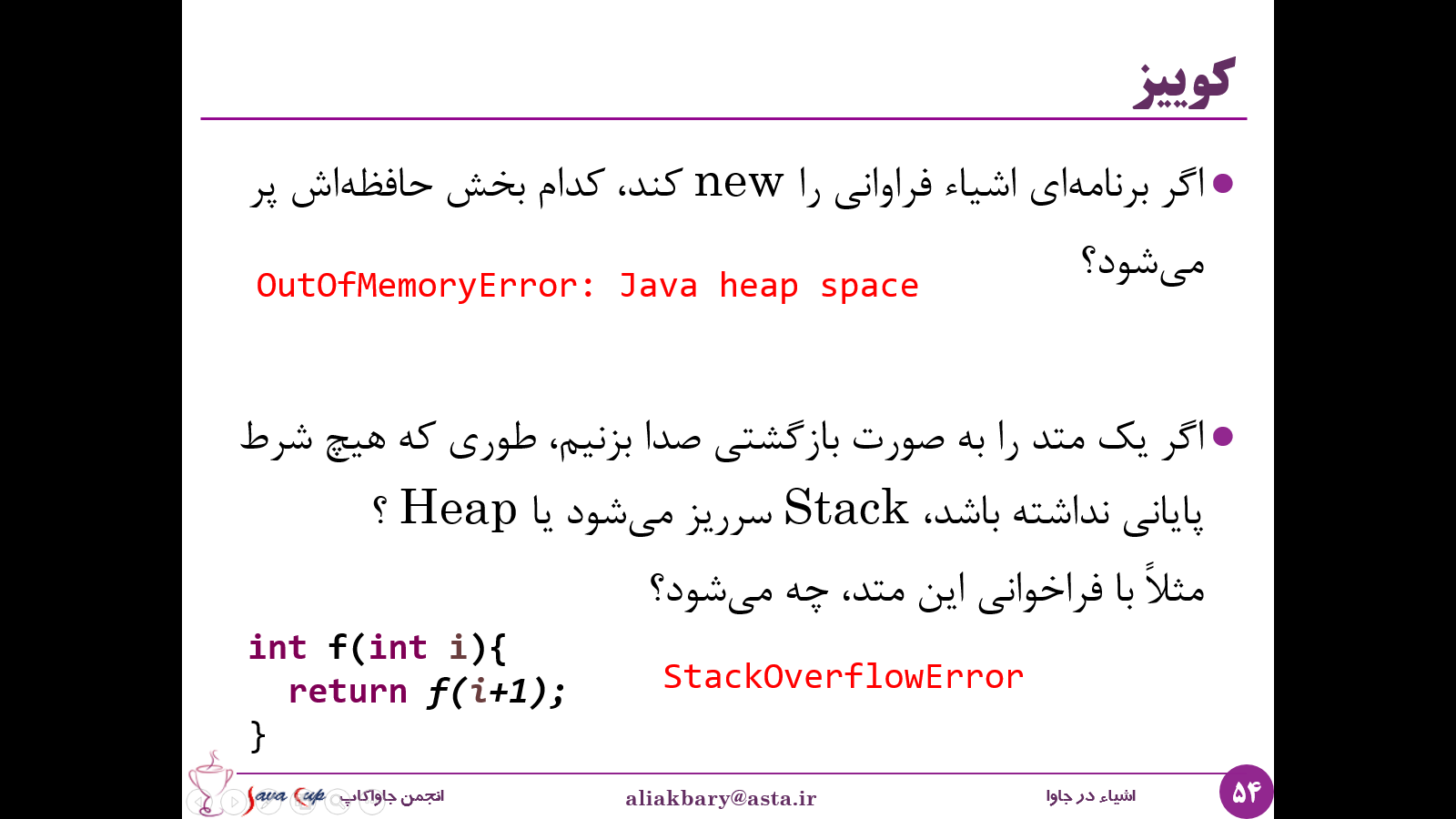


یعنی در واقع بخش هیپ از حافظه ی برنامه ی ما پر می شود و سرریز می کند.

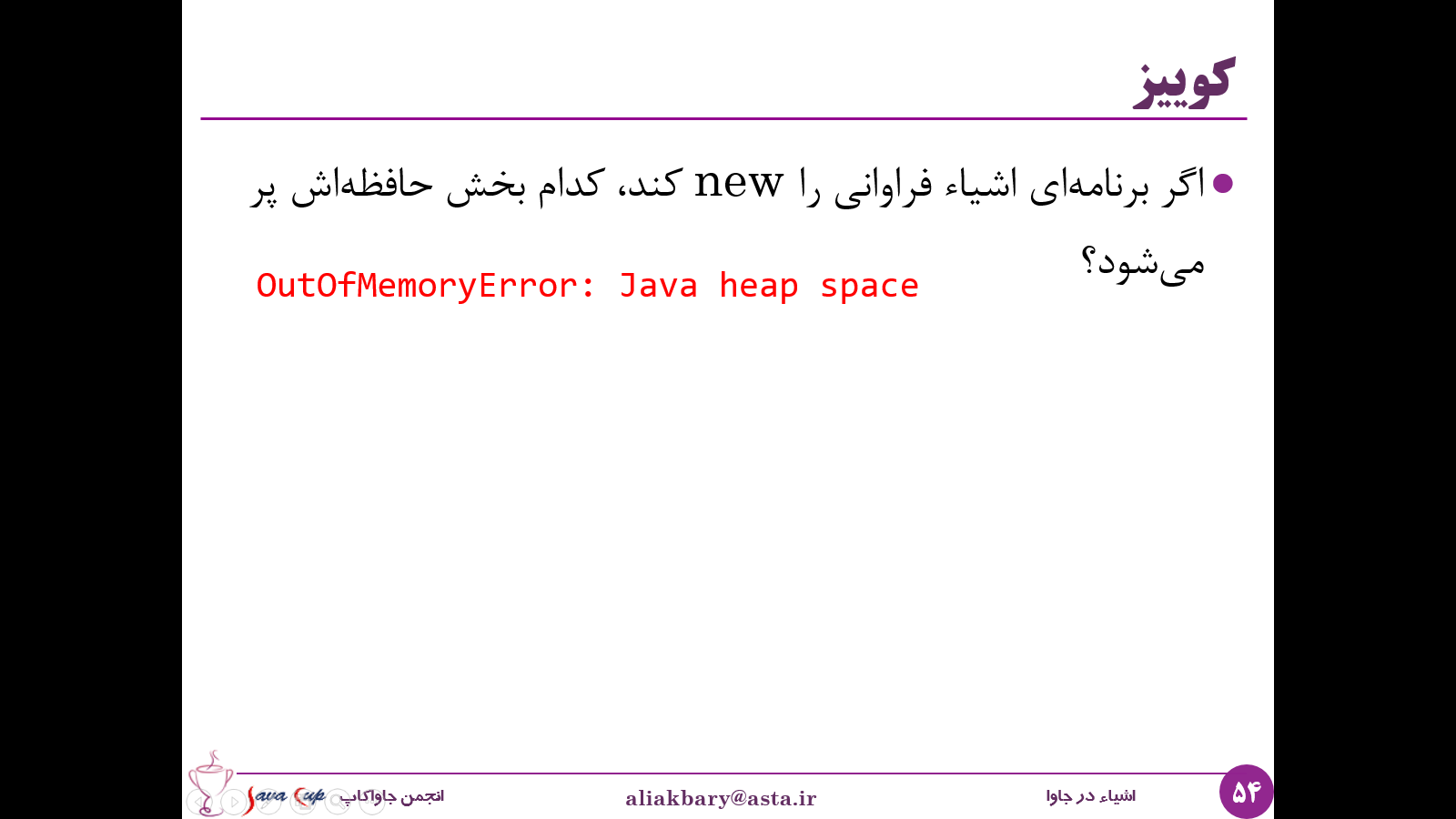
* اگر یک متد را به صورت بازگشتی فراخوانی کنیم، طوری که هیچ شرط پایانی نداشته باشد، Stack سرریز می شود یا هیپ؟ مثلا با فراخوانی متد زیر چه می شود؟



پاسخ:

در این پشته ی برنامه ما پر می شود و خطای StackOverFlowError پرتاب خواهد شد چرا که به ازای فراخوانی هر متد یک بخش روی پشته ایجاد می شود برای نگهداری متغیرهای محلی و برخی اطلاعات دیگر و بعد دوباره همان تابع فراخوانی می شود و باز بخش دیگری برای تابع جدید از پشته گرفته می شود. پس همین طور بخش هایی روی پشته اختصاص داده می شود تا اینکه در نهایت فضای پشته پی و اصطلاحا سرریز می شود و این خطا توسط JVM گزارش خواهد شد.

* فرض کنید برنامه ای به نام HeapGame نوشته ایم، اگر در زمان اجرا با این خطا مواجه شویم:



و اگر مطمئن باشیم که برنامه ما اشکالی ندارد و کامپیوتر ما فضای کافی برای اجرای این برنامه دارد. در چنین شرایطی چه باید بکنیم؟ چطور می توانیم مشکل حافظه را برطرف کنیم؟

پاسخ:

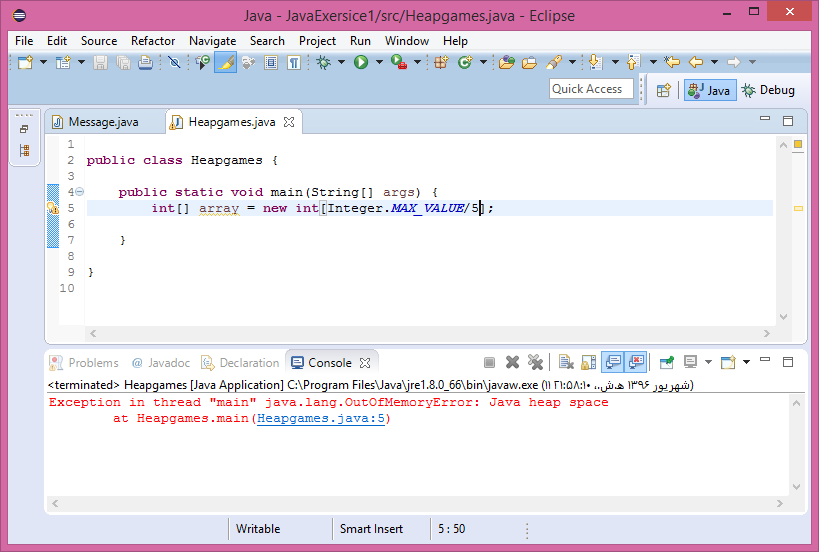
ما در چنین شرایطی باید از دستوراتی که حافظه ی هیپ را تنظیم می کنند استفاده کنیم، مثلا از –Xmx استفاده کنیم تا حداکثر مقداری که برنامه ی ما از هیپ می خواهد استفاده کند بیشتر شود. مثلا برنامه را با دستور زیر اجرا کنیم:

java –Xmx4096m HeapGames

به این ترتیب پس به شرطی که برنامه ما اشکال خاصی نداشته باشد و کامپیوتر ما هم حافظه ی کافی بیش از 4 گیگابایت داشته باشد، با کمک این دستور می توانیم حافظه ی هیپی که برنامه دارد از آن استفاده می کند را افزایش دهیم تا دیگر با خطای بالا مواجه نشویم.

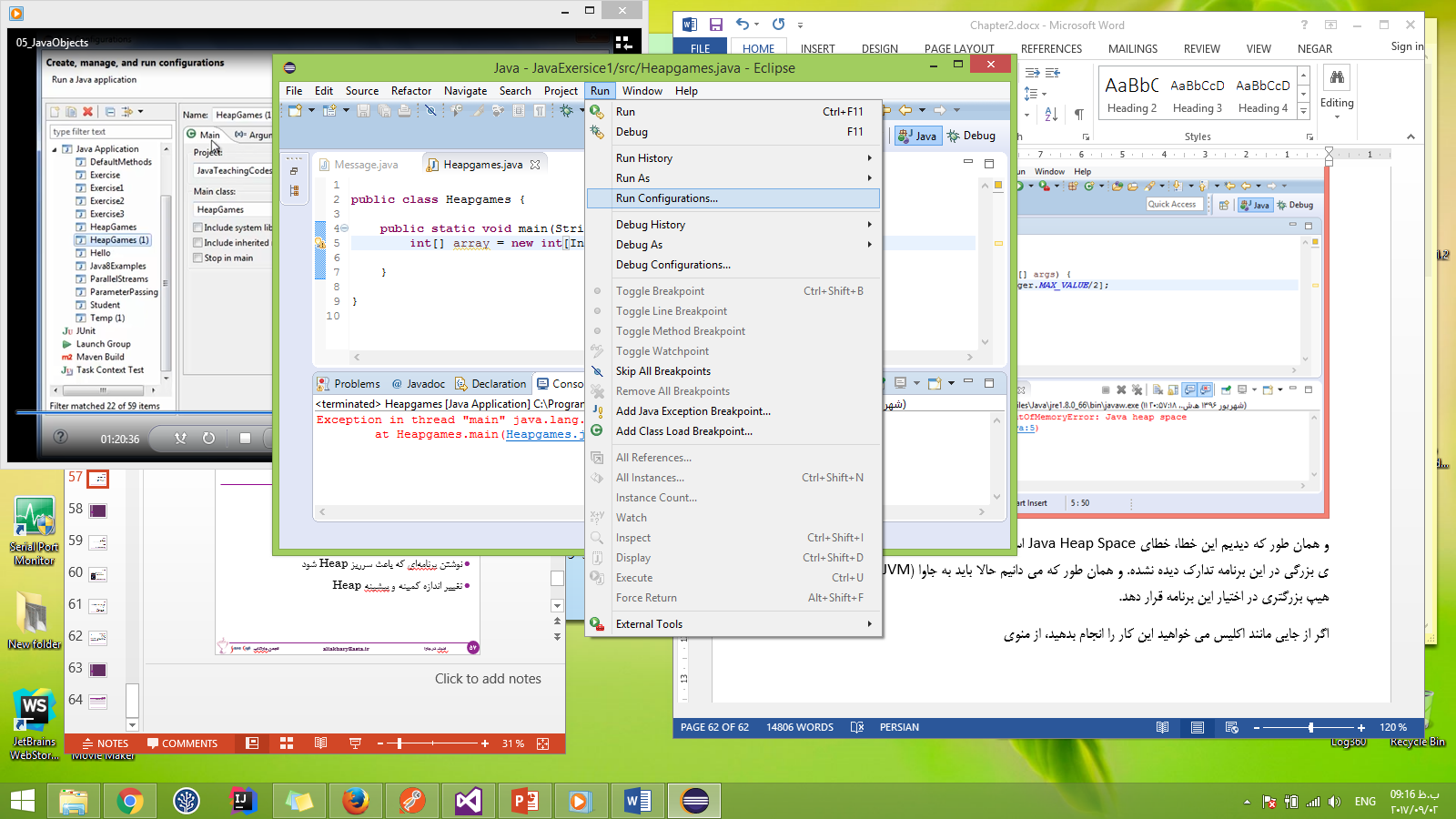
## تمرین عملی

می خواهیم تنظیمات هیپ و پشته را یک بار دیگر مرور و تمرین کنیم. فرض کنید کلاسی به اسم HeapGames داریم که متد main دارد، فرض کنید می خواهیم یک آرایه بسیار بزرگ در آن تعریف کنیم، مثلا به طول یک پنجم حداکثر Integer موجود، اگر این برنامه را اجرا کنیم، با یک خطا مواجه خواهیم شد:

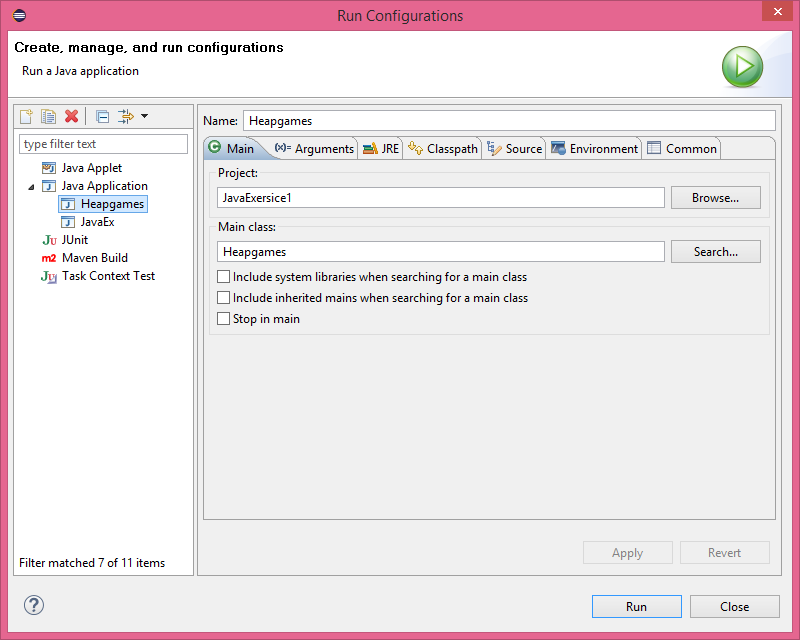


و همان طور که دیدیم این خطا، خطای Java Heap Space است. یعنی که هیپ کافی برای نگهداری چنین آرایه ی بزرگی در این برنامه تدارک دیده نشده. و همان طور که می دانیم حالا باید به جاوا (JVM) آرگومان پاس کنیم تا هیپ بزرگتری در اختیار این برنامه قرار دهد.

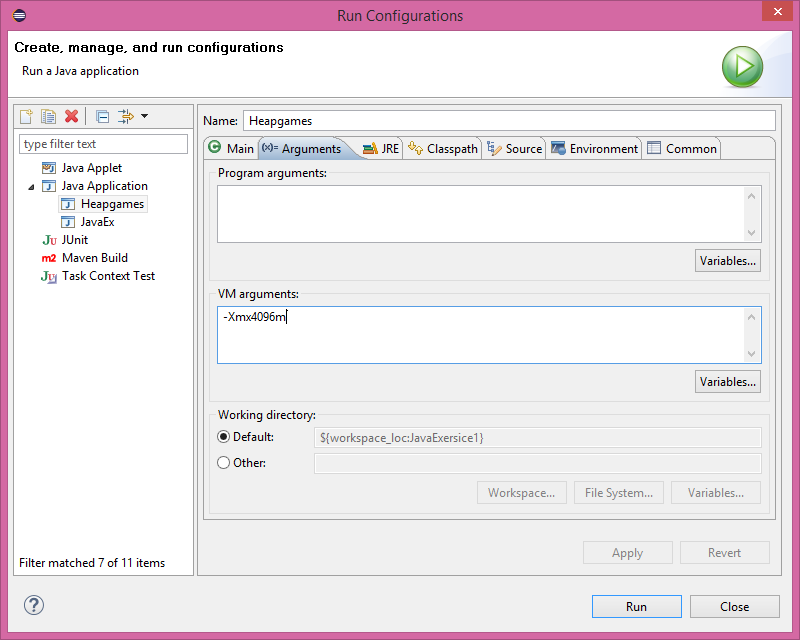
اگر از جایی مانند اکلیس می خواهید این کار را انجام بدهید، از منوی Run گزینه ی Run Configurations را انتخاب کنید:



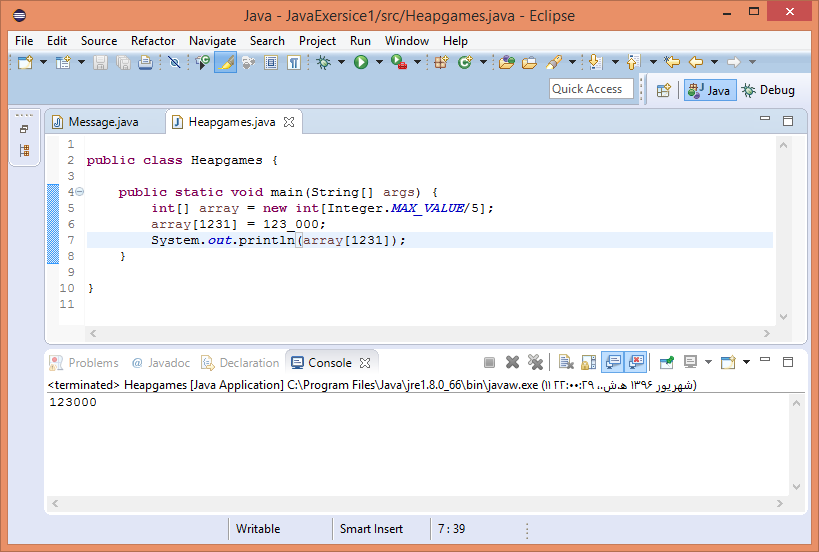
سپس برنامه خود را در بین برنامه های فهرست شده در ستون سمت چپ پیدا و انتخاب کنید:



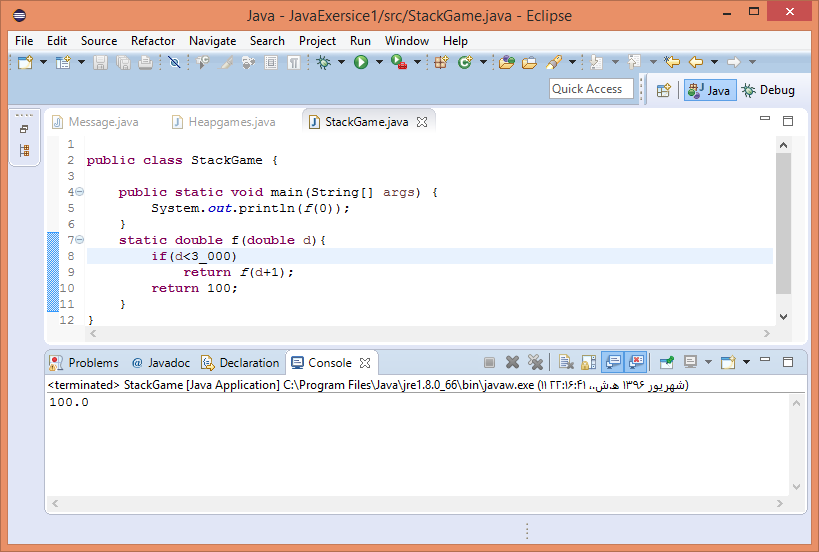
در این پنجره می توانید تنظیمات مختلفی برای برنامه ی خود انجام دهید، برای مثال در منوی Arguments می توانید آرگومان هایی که می خواهید به برنامه تان و یا ماشین مجازی جاوا پاس کنید را تنظیم نمایید. ما می خواهیم آرگومان –Xmx به آن پاس کنیم تا حداکثر حافظه ای که به هیپ ما اختصاص می دهد بیشتر شود:



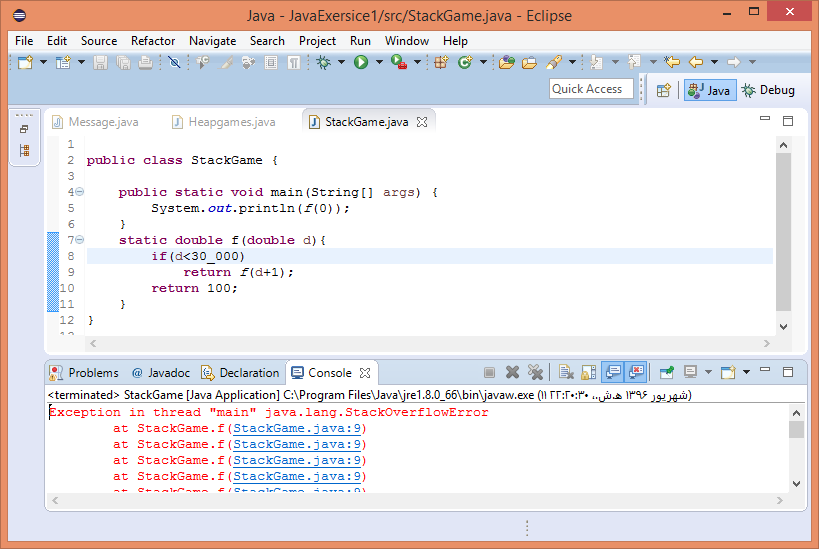
با این کار 4 گیگابایت حافظه به هیپ برنامه ما اختصاص داده می شود. حالا اگر برنامه را اجرا کنیم، دیگر به خطای قبلی برخورد نمی کنیم و می توانیم از این آرایه به راحتی استفاده کنیم. مثلا یکی از خانه های آن را مقدار دهی کنیم، و یا همان خانه را چاپ کنیم، و دیگر به آن خطای هیپ برخورد نمی کنیم:



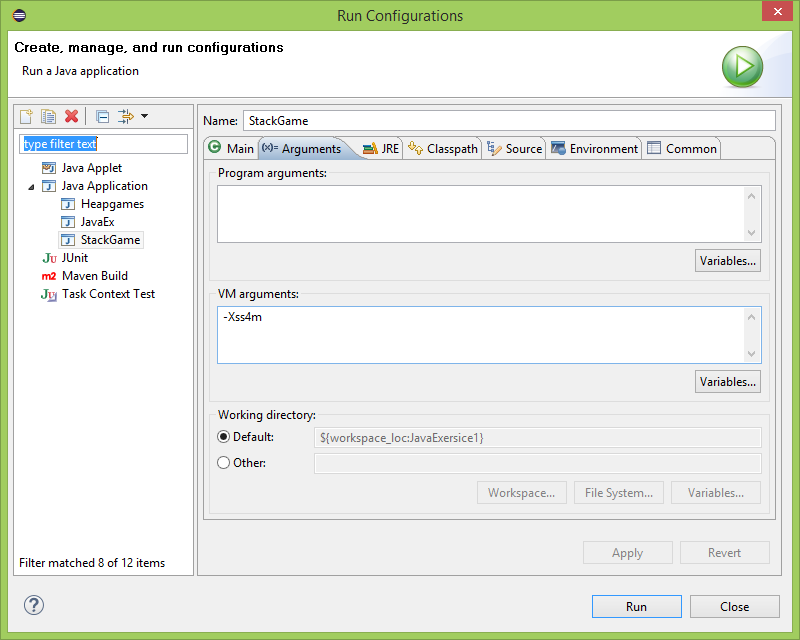
حال می خواهیم با استک کار کنیم، فرض کنید که یک متد داریم به نام f() که یک پارامتر به عنوان آرگومان دریافت می کند. و این متد static است و خروجی از جنس double دارد و بدنه ای دارد که این متد را به صورت بازگشتی فراخوانی می کند، مثلا اگر ورودی متد از مقدار 3000 کمتر بود، متد را با یک مقدار بیشتر صدا می زند، و در غیر این صورت مقداری مانند 100 را برمی گرداند. در متد main این متد f() را با پارامتر صفر فراخوانی کرده ایم و این متد قرار است به صورت بازگشتی بارها و بارها خودش را فراخوانی کند و احتمالا پشته ی بسیار بزرگی برای اجرا شدن لازم خواهد داشت. هنگامی که این برنامه را اجرا می کنیم، بدون هیچ مشکلی اجرا می شود:



حال فرض کنید که با افزایش مقدار داخل شرط، کاری کنیم که برنامه به مقدار Stack بیشتری نیاز داشته باشد، یعنی این تابع به تعداد بیشتری خودش را صدا کند، مثلا 30 هزار، اگر این برنامه را اجرا کنیم، با خطای سرریز پشته مواجه خواهیم شد:



حال با این خطا چه کنیم؟ Stack ما سرریز شده است، در واقع این تابع با هر بار فراخوانی خودش مقدار از حافظه ی پشته را برای ذخیره متغیرهای محلی و برخی اطلاعات دیگر گرفته است و به علت زیاد شدن تعداد فراخوانی ها، پشته سرریز شده است. چه کنیم؟ مجددا بایست آرگومان هایی را برای ارسال به ماشین مجازی جاوا تنظیم نماییم، پس به همان پنجره ی Run -> Run Configuration می رویم و پس از انتخاب نام برنامه از منوی سمت چپ، در بخش Arguments مقدار پشته را تنظیم می کنیم. این بار به جای هیپ، به Stack بزرگ نیاز داریم:



مثلا پشته ای با 4 مگابایت حجم کافی خواهد بود تا این برنامه اجرا شود.

## جمع بندی

در پایان به جمع بندی مطالب این فصل بپردازیم. ما در این فصل درباره ی مفهوم ارجاع به اشیاء، وضعیت اشیاء در حافظه، محتوای اشیاء و ارجاع هایی که به آنها ساخته می شود صحبت کردیم. درباره ی تفاوت های متغیرهایی که از جنس Primitive types هستند با Reference Types چیزهایی یاد گرفتیم. دیدیم متغیرهایی مانند آرایه ها و رشته ها که قبلا با آنها کار می کردیم هم شی و از جنس ارجاعات هستند.

درباره زباله روب یا Garbage Collector صحبت کردیم و دیدیم که چطور هر از چندگاهی اجرا می شود و هیپ را بررسی می کند و اشیایی که مرده هستند و دیگر استفاده نمی شوند را پیدا می کند و حافظه ی آنها را آزاد می کند.

درباره نحوه ی ارسال پارامترها به متدها صحبت کردیم و دیدیم که شیوه ی جاوا در این زمینه چگونه است.

و در نهایت درباره بخش های مختلف حافظه هم صحبت کردیم، مخصوصا درباره ی Stack و Heap و اینکه چه متغیرهایی روی هر کدام از آنها ذخیره می شوند.

برای مطالعه فصل 3 و 6 از کتاب دایتل مرتبط با مباحث این فصل ما هست. که خوب است آنها را مطالعه کنید و تمرین های همین فصل ها را هم حل کنید.

## تمرین

* سعی کنید دست به کار (دست به کد) شوید و برنامه‌هایی بنویسید که Stack را سرریز کنند. سعی کنید این کار را به شکل متفاوت با کاری که در این فصل انجام داده ایم انجام دهید.
* با ایجاد تعداد زیادی شیء کوچک، برنامه‌ای بنویسید که Heap را سرریز کند.
* با ایجاد تعداد کمی شیء بسيار بزرگ، برنامه‌ای بنویسید که Heap را سرریز کند.
* در موارد بالا سعی کنید تنظيمات حافظه JVM را تمرين کنید و اندازه هیپ و پشته را تغییر دهید:
  + به صورت command line
  + در محیط توسعه نرم‌افزار (مثلاً Eclipse)

## جستجو کنید و بخوانید

مثل همیشه موضوعات بسیار مختلف و متنوعی هست که ما تنها به آنها اشاره کردیم و جا دارد که درباره ی آنها بیشتر جستجو و مطالعه کنید. مثلا:

* Heap, Stack

در مورد نحوه ی مدیریت هیپ و یا نحوه ی ایجاد و حذف پشته.

* Heap generations

هیپ بخش های مختلفی دارد که به آنها generations گفته می شود، در مورد چیستی آنها و اینکه هر کدام چه داده هایی را نگه می دارند بخوانید.

* Garbage Collection Algorithms

یا در مورد الگوریتم های زباله روبی مطالعه کنید. اگر کمی در آن دقیق شوید خواهید دید که اصلا کار ساده و بدیهی نیست و نیازمند الگوریتم های بسیار کارآمدی است که با سرعت و دقت این کار را انجام بدهند.

* Call by value, Call by pointer, Call by reference

در مورد مکانیزم های ارسال پارمترها به توابع و متدها مطالعه کنید، حتی در زبان هایی غیر از جاوا، با این کار دید بهتری نسبت به این مفهوم پیدا خواهید کرد.

* مثلا بررسی کنید گونه‌های ارسال پارامتر به متد در C++ چطور است:

**void cppMethod(Person byValue,   
 Person\* byPointer, Person& byReference){...**

* و یا گونه‌های ارسال پارامتر به متد در زبان C# : (تفاوت حالتی که ref وجود دارد با حالتی که وجود ندارد)

**void cSharpMethod(ref int a, ref Person p){...**