

**دانشکده ریاضی و علوم کامپیوتر**

**برنامه‌سازی پیشرفته و کارگاه**

**متد ها**

استاد درس

دکتر مهدی قطعی

استاد دوم

بهنام یوسفی مهر

نگارش

شهاب الدین گریوانی

بهار ۱۴۰۳

**فهرست**

[مقدمه 3](#_Toc191240119)

[متغیر های محلی ( local variables ) 4](#_Toc191240120)

[Shadowing 6](#_Toc191240121)

[کلیدواژه “this” 6](#_Toc191240122)

[متد های استاتیک 8](#_Toc191240123)

[مقدار دهی اولیه به متغیر های محلی 12](#_Toc191240124)

[Method overloading 14](#_Toc191240125)

[چه چیزی یاد گرفتیم؟ 16](#_Toc191240126)

مقدمه

تا اینجای کار، با کلاس‌های نسبتاً ساده ای 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 هروقت 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 void عه (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 لینک زیر راجع بهش بخونید:

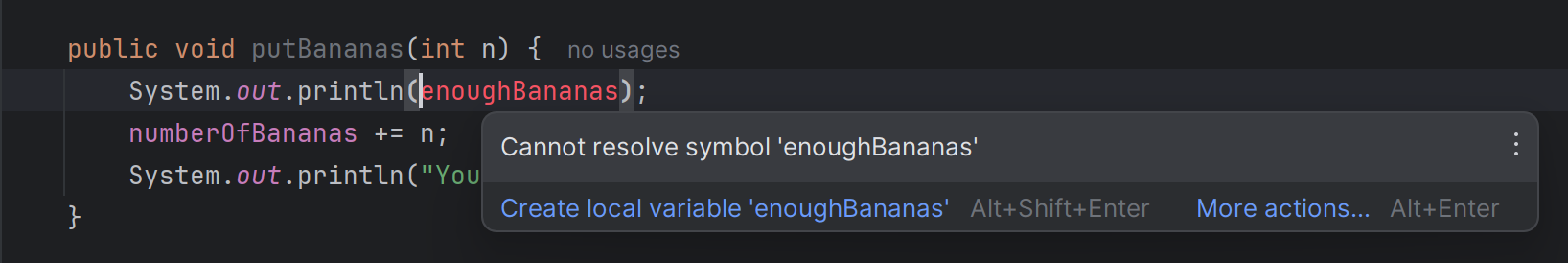
<https://www.geeksforgeeks.org/variable-arguments-varargs-in-java/>

متغیر های 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؛ همچین چیزی مثلا:

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

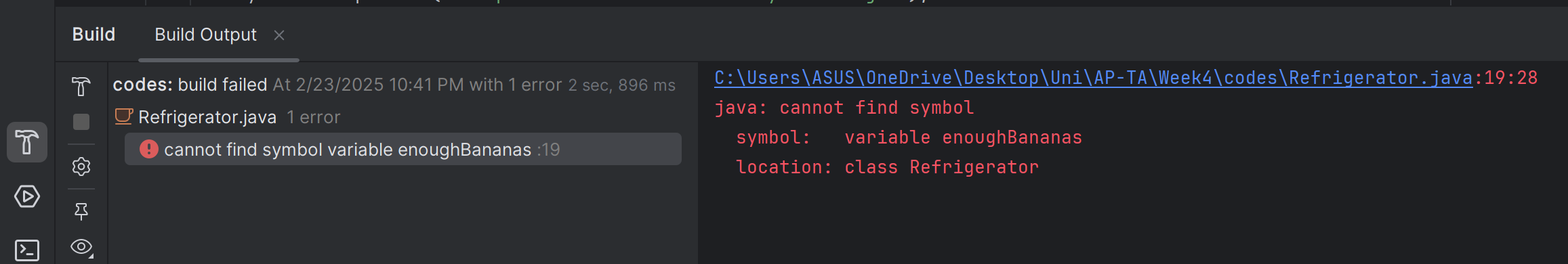
احتمالاً می­­بینید که enoughBananas قرمز شده. موستون رو ببرید روش؛ با همچین چیزی مواجه می­شید:



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 همچین 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 هشدار های 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.

Shadowing

وقتی که یک متغیر محلی یا یک ورودی متد اسمش با اسم یکی از فیلد های کلاسمون یکی باشه، اون متغیر محلی، اصطلاحا 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 مخصوصا وقتی که متغیرهامون اسم های 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 بخواهیم ماشین رو حرکت بدیم، باید بتونیم مختصاتش رو تغییر بدیم، ولی چطور می­تونیم به فیلد های x و y که مربوط به آبجکتمون هستن دسترسی پیدا کنیم؟

کلید­واژه “this”

هروقت 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.

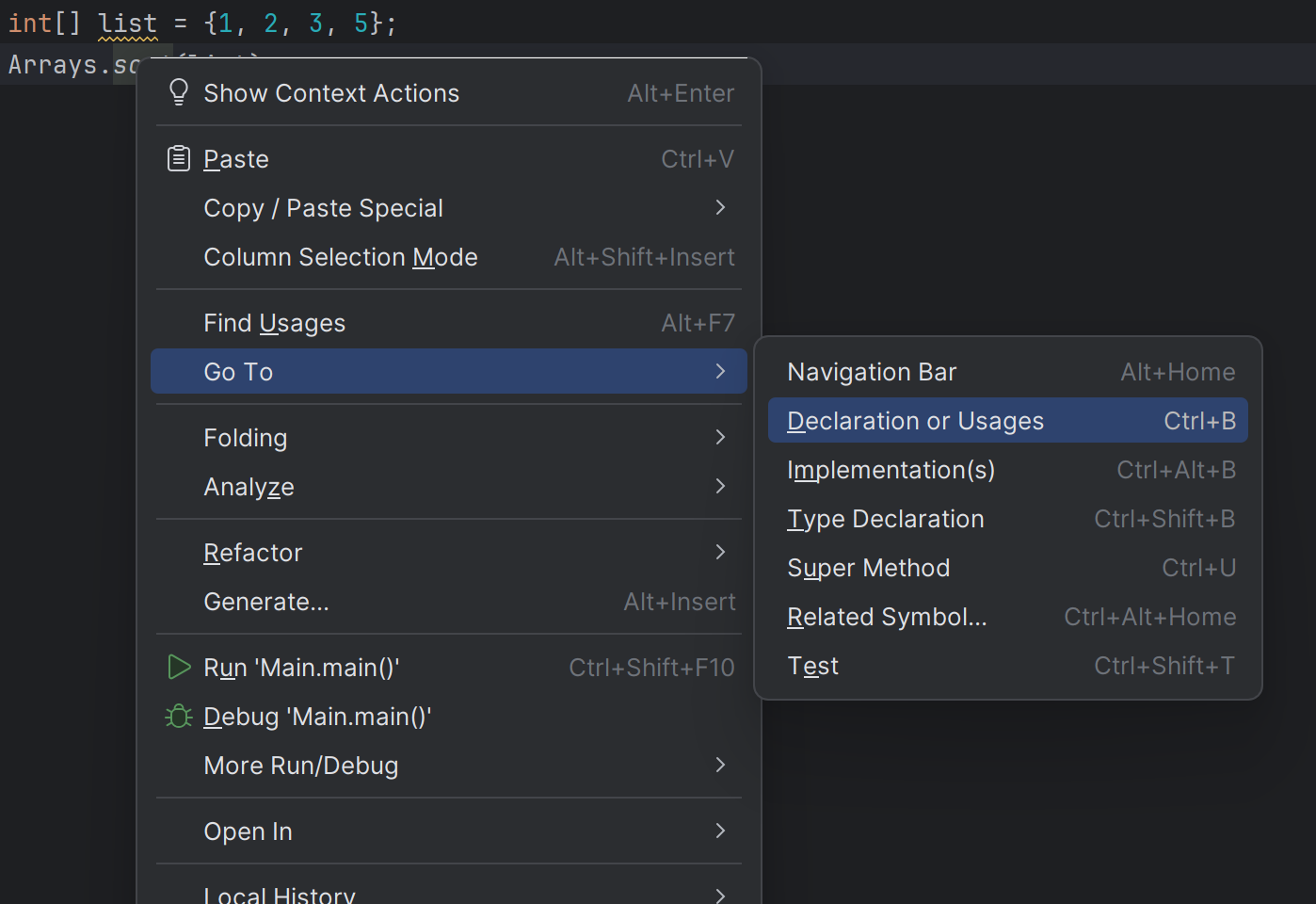
اینجا یک فیلد gas هم به Car اضافه کردیم که قراره مقدار بنزین ماشین رو نشون بده. در ادامه ی متد اول مطمئن می­شیم که ماشین به اندازه کافی بنزین داره و بعد اگه بنزین داشت ماشین رو حرکت می­دیم. می­بینید که برای دسترسی به gas از this استفاده نکردیم؛ این به این دلیله که اشاره به آبجکتی که توش هستیم به طور 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 بخواهیم 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 روشون 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 به‌عنوان ورودی به یه متد بفرستید.

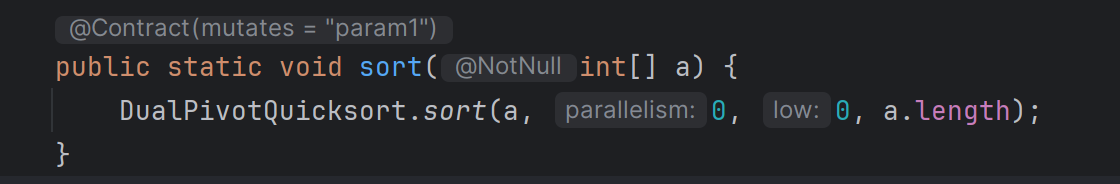
متدهای استاتیک

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:



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، مدل های مختلف ماشین رو به صورت فیلد­های static final تعریف کردیم. حالا فرض کنید به اسم این مدل ها 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 به طور 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);  
}

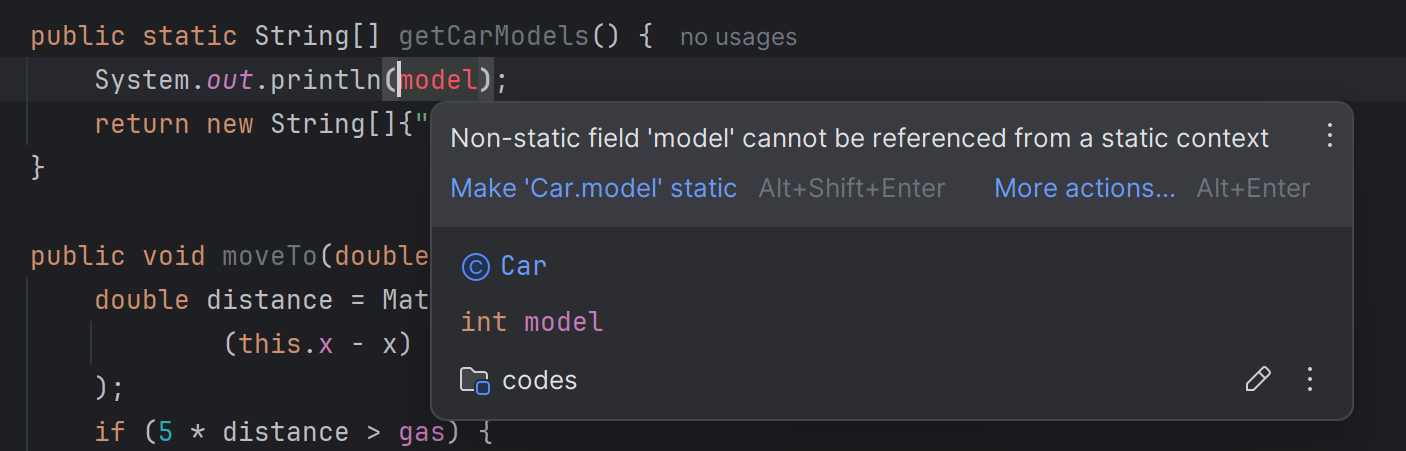
این متد، یک ArrayList از ماشین ها می­گیره، تعداد مدل های مختلف ماشین ها رو می­شمره و چاپ می­کنه. همونطور که می­بینید کارکرد این متد هیچ ربطی به یک instance خاص از Car نداره و به همین دلیل استاتیک تعریفش می­کنیم.

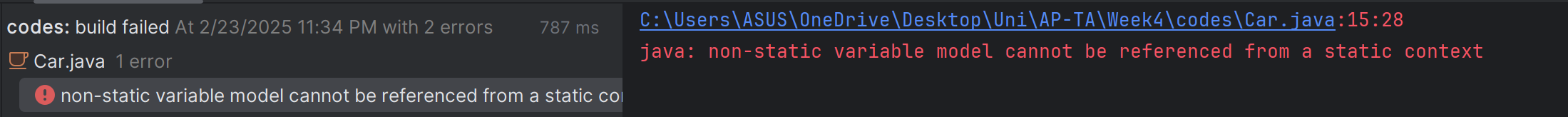
مثال خوب دیگه­ای برای کاربرد متدهای استاتیک، کلاس Math هست. این کلاس قراره مجموعه­ای از عملیات­های ریاضی باشه؛ به همین دلیل تمام متد­های کلاس Math استاتیک هستند. البته Math یک مرحله فراتر میره، شما نمی­تونید اصلا آبجکتی از Math بسازید! اصلاً این که یک آبجکت از روی Math بسازید، معنی نداره و نیازی بهش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

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 توی یک متد استاتیک استفاده کنید. شما می­تونید متد getCarModels رو بدون داشتن هیچ آبجکتی صدا بزنید؛ پس این modelی که سعی دارید ازش استفاده کنید، مربوط به کدوم آبجکته؟ از اون جایی که متدهای استاتیک مربوط به کلاسن 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.

مقدار دهی اولیه به متغیر های محلی

بر خلاف فیلدهای آبجکت که اگه مقداردهیشون نکنید، جاوا براشون مقدار پیش­فرضی قرار میده[[1]](#footnote-1)، متغیر های 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 ازش 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 متداول ترین مشکلاتی که توی زبان هایی مثل C و C++ به وجود میاد اینه که یادتون میره متغیری رو مقداردهی کنید. توی این زبان­ها، متغیر های مقداردهی نشده، مقادیر ظاهراً رندومی اختیار می­کنند و این می­تونه دردسرساز باشه و باعث بشه دیباگ کردن برنامه­ها سخت تر بشه. جاوا با مجبور کردن شما به مقداردهی به متغیر ها، باعث جلوگیری از این مشکلات میشه.

Method overloading

Method overloading، این قابلیته 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 پرکاربردیه. 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 StudentId(String name, int roll\_no) {  
 System.*out*.println("Name :" + name + " "  
 + "Roll-No :" + roll\_no);  
 }  
  
 // Method 2  
 public void StudentId(int roll\_no, String name) {  
 // Again printing name and id of person  
 System.*out*.println("Roll-No :" + roll\_no + " "  
 + "Name :" + name);  
 }  
}

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

چه چیزی یاد گرفتیم؟

توی این داک، با متد ها بهتر و دقیق تر آشنا شدید؛ الان باید بدونید که:

* فرق بین متغیرهای محلی و فیلدهای کلاس چیه.
* مسأله shadowing چیه، this چیه و چجوری shadowing رو با this حل کنید.
* متد استاتیک چیه، کجاها کاربرد داره، با متد عادی چه فرقی می­کنه و این که چجوری تعریفش کنید و چجوری صداش بزنید.
* متغیرهای محلی برعکس فیلدهای کلاس مقدار پیش­فرض نمی­گیرند و این که این موضوع چجوری می­تونه دردسر ساز باشه و چجوری مشکلش رو حل کنید.
* Method overloading چیه، چرا ابزار قدرتمندیه، چه حالت­هایی داره و چطور ازش استفاده کنید.

1. جاوا به طور پیش­فرض برای فیلد های عددی مقدار 0، برای char مقدار \0، برای Boolean ها مقدار false و برای آبجکت ها null رو قرار میده. [↑](#footnote-ref-1)