

**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](#_Toc190737068)

[متغیر های محلی ( local variables ) 5](#_Toc190737069)

[Shadowing 6](#_Toc190737070)

[اشاره گر “this” 6](#_Toc190737071)

[متد های استاتیک 8](#_Toc190737072)

[مقدار دهی اولیه به متغیر های محلی 10](#_Toc190737073)

[ورودی دادن و Reference ( ارجاع ) ها 12](#_Toc190737074)

[کلاس های wrapper برای primitive type ها 14](#_Toc190737075)

[Method overloading 16](#_Toc190737076)

[چه چیزی یاد گرفتیم؟ 18](#_Toc190737077)

مقدمه

تا اینجای کار، با کلاس های 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 متد ها وارد عمل میشن: به طور کلی هروقت بخواهید توی کدتون تصمیمی بگیرید یا عملیات منطقی انجام بدید یا کلا کاری انجام بدید، باید از متد ها استفاده کنید. متد ها به قدری اساسی هستن که حتی توی اولین مواجهه­تون با جاوا از متد main استفاده کردید و توی اون کدتون رو نوشتید! توی این بخش قراره دقیق تر و کامل تر با متد ها آشنا بشید.

متد ها رو تا الان زیاد دیدید، هم توی جاوا، و هم توی درس مبانی برنامه­نویسی: Function هایی که توی C داشتید خیلی شبیه به متد های جاوا هستند. متد ها همیشه باید نوع خروجیشون مشخص باشه که می­تونه primitive type، reference type، یا void باشه ( که یعنی تابع خروجی نمیده). همچنین متد ها می­تونن ورودی (arguments) داشته باشن. 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 می­تونیم از توی یخچالمون موز برداریم!

اینجا، متدمون تعداد مشخص و ثابتی ورودی داره ( یکی )؛ اما می­تونید متد هایی تعریف کنید که تعداد ورودی هاشون ثابت نباشه. اگه دوست دارین بیشتر راجع به این موضوع 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. متغیر های محلی، موقتی هستند و فقط درون بلوک متدی که توی اون تعریف شدند قابل استفاده­ اند. متغیر های محلی هنگامی که متد صدا زده می­شه به وجود میان و بعد از تموم شدن متد معمولاً از بین میرن و همچنین نمیشه بهشون از خارج از متد دسترسی داشت. ورودی های یک متد هم جزو متغیر های محلی اون متد هستند با این تفاوت که مقدار اولیه­شون هنگام صدا زدن متد و از طرف کسی که متد رو صدا زده معلوم میشه.

یک آبجکت که توی یک متد ساخته میشه و توی یک متغیر محلی ریخته میشه، ممکنه بعد از تموم شدن متد، توی حافظه باقی بمونه یا نمونه؛ همونطور که توی بخش های بعدی می­بینید، این بستگی داره به این که آیا reference ای به اون متغیر باقی می­مونه یا نه: اگه یه آبجکت ساخته بشه، توی یک متغیر محلی ریخته بشه و هیچ جای دیگه ای استفاده نشه، بعد از تموم شدن متد دیگه هیچ چیزی به اون آبجکت اشاره نمی­کنه و بنابرین garbage collection اون آبجکت رو از بین می­بره. اما، اگه ما توی متد یک آبجکت رو توی یک فیلد کلاسمون بریزیم، یا به عنوان ورودی به یک متد دیگه بدیمش یا returnش کنیم، ممکنه از بین نره و باقی بمونه.

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 متغیر کلاسمون سایه می­اندازه و جلوی دسترسی ما به اون متغیر رو می­گیره. شاید فکر کنید این مشکل به ندرت پیش میاد ، ولی shadowing اتفاق نسبتاً متداولیه مخصوصا وقتی که متغیر هامون 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 متغیر 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؟

اشاره گر “this”

هروقت نیاز دارید که صریحاً به آبجکتی که توش هستیم یا یکی از اعضای اون اشاره کنید، می­تونید از اشاره­گر 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 مربوط به آبجکت ( مختصات فعلی ماشین )، از this.x استفاده کردیم. this در واقع به همون آبجکتی که توش هستیم اشاره می­کنه.

اینجا یک متغیر gas هم به Car اضافه کردیم که قراره مقدار بنزین ماشین رو نشون بده. در ادامه ی متد ابتدا مطمئن شدیم که ماشین به اندازه کافی بنزین داره و بعد اگر بنزین داشت ماشین رو حرکت دادیم. 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 متغیر های محلیمون با اسم یکی از متغیر های کلاسمون یکی باشه. اون وقت اگر می­خواهیم از متغیر کلاس استفاده کنیم، باید صریحاً این رو مشخص کنیم وگرنه پیش­فرض جاوا استفاده از متغیر محلیه.

استفاده از this برای دسترسی به متغیر هایی که روشون سایه افتاده، روش مرسومیه و باعث می­شه که بتونیم از اسم های مرسومی که برای متغیر ها وجود داره ( مثلاً x و y برای مختصات ) بتونیم استفاده کنیم و درگیر این که برای متغیر هامون اسم های جدید اختراع کنیم نشیم. همچنین، هرجای دیگه ای که بخواهید به آبجکتی که توش هستید اشاره کنید می­تونید از this استفاده کنید؛ مثلاً وقتی که خود آبجکتی که توش هستید رو می­خواهید به عنوان ورودی به یک متد بدید.

متد های استاتیک

متد های استاتیک (static methods)، مثل متغیر های استاتیک، به خود کلاس تعلق دارند، نه به آبجکت های مستقلی که ما از روی اون کلاس می­سازیم. اما این یعنی چی؟ اول از همه، متد های استاتیک خارج از آبجکت ها وجود دارند و برای صدا زدنشون لازم نیست آبجکتی وجود داشته باشه؛ شما می­تونید اسم کلاس رو بنویسید و با عملگر نقطه متد های استاتیک رو صدا بزنید. قبلاً از متد های استاتیک زیاد استفاده کردید، مثلاً برای مرتب کردن آرایه ها از Arrays.sort() استفاده می­کردید؛ 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.

اصلی ترین استفاده متد های استاتیک، برای تعریف کردن متد های کمکی هست که یا مستقل از آبجکت ها فعالیت می­کنن، یا روی آبجکت هایی که از اون کلاس می­سازیم کاری انجام میدن و منطقشون به یک instance خاص تعلق نداره و به طور کلی عمل می­کنند. به عنوان مثال، کلاس Math رو در نظر بگیرید. این کلاس قراره مجموعه ای از عملیات های ریاضی باشه؛ به همین دلیل تمام متد های کلاس Math استاتیک هستن. اصلاً این که یک آبجکت از روی Math بخواهیم بسازیم، معنی نداره و نیازی بهش نیست. این کلاس صرفاً قراره که مجموعه ای از متد ها و متغیر ها برای انجام عملیات ریاضی باشه. شما چند "ریاضی" مختلف ندارید که بخواهید instance های مختلفی از Math بسازید!

از اون جایی که متد های استاتیک مربوط به کلاسند و از آبجکت ها جدا هستن، طبیعیه که نمی­تونند به متغیر ها و متد های عادی که مربوط به هر آبجکت هستن دسترسی داشته باشن و فقط به متد ها و متغیر های استاتیک دسترسی دارن؛ اگه اینطوری نبود و متد های استاتیک به متغیر/متد های غیر استاتیک دسترسی داشتن، سوالی که پیش میومد این بود که این متغیر/متد مربوط به کدوم آبجکته؟

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 متغیر جدید به ماشین هامون اضافه کردیم به اسم model که مدل ماشینمون رو نشون میده: ماشینمون می­تونه شاسی بلند (model = 0)، سواری (model = 1) یا هاچ بک (model = 2) باشه. برای راحتی، مدل های مختلف ماشین رو به صورت متغیر های static final تعریف کردیم. حالا فرض کنید به اسم مدل های مختلف ماشین ها به صورت String نیاز داشته باشیم، می­تونیم مثل بالا یک متد استاتیک تعریف کنیم که این اطلاعات رو بهمون بده. توجه کنید که مدل های مختلف موجود برای ماشین ها هیچ ربطی به یک ماشین خاص یا یک آبجکت خاص از نوع Car نداره و جدا از همه ی ماشین هاست، به همین دلیل استفاده از متغیر ها و متد های استاتیک بهترین انتخابه.

مقدار دهی اولیه به متغیر های محلی

بر خلاف متغیر های آبجکت که اگه مقداردهیشون نکنید، جاوا براشون مقدار پیش فرضی قرار میده، 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 پیچیده تر میشه که مقدار دهی 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 مقدار دهی کنید، یا استفاده ای که از متغیر می­کنید رو هم داخل شرط ببرید، یا می­تونید با توجه به برنامه ای 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 مقدار دهی 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 زبان ها، متغیر های مقداردهی نشده، مقادیر ظاهراً رندومی اختیار می­کنند و این می­تونه دردسرساز باشه و باعث بشه دیباگ کردن برنامه ها سخت تر بشه. جاوا با مجبور کردن شما به مقداردهی به متغیر ها، باعث جلوگیری از این مشکلات میشه.

ورودی دادن و Reference ( ارجاع ) ها

وقتی که به متد ها ورودی می­دیم، اگه جنس ورودیمون primitive type باشه، مقدار ورودی کپی میشه ولی اگه جنس ورودیمون آبجکت باشه، اون آبجکت کپی نمیشه، بلکه یک ارجاع ( reference ) به اون آبجکت توی متد کپی میشه و استفاده میشه. بیاید با یک مثال پیش بریم:

public class Warrior {  
 double power;  
 double hp;  
  
 public void hit(double multiplier, Warrior enemy) {  
 if (multiplier >= 100) {  
 multiplier = 100;  
 }  
 double finalPower = power \* multiplier;  
 enemy.hp -= finalPower;  
 }  
  
 public static void main(String[] args) {  
 Warrior warrior1 = new Warrior();  
 warrior1.hp = 100;  
 warrior1.power = 100;  
  
 Warrior warrior2 = new Warrior();  
 warrior2.hp = 200;  
 warrior2.power = 50;  
  
 double m = 1.5;  
 warrior1.hit(m, warrior2);  
 }  
}

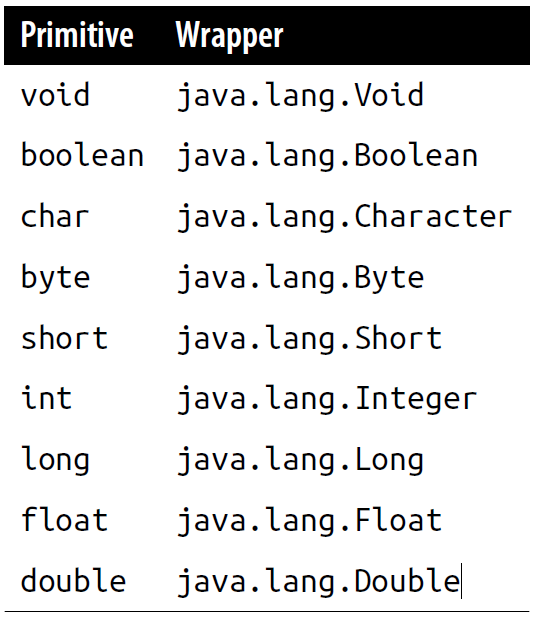
اینجا یک کلاس Warrior ( جنگجو ) داریم که قدرت ( power ) و جون ( hp ) داره. یک متد hit هم تعریف کردیم براش که بتونه به دشمنش حمله کنه! در ادامه، دو تا جنگجو ساختیم و متد hit رو روی warrior1 صدا زدیم و m و warrior2 رو بهش ورودی دادیم. وقتی متد رو صدا زدیم، مقدار m توی یک متغیر محلی توی متد به اسم multiplier کپی میشه و ازش استفاده میشه. متد ما هر تغییری توی multiplier ایجاد کنه، این تغییرات محلی می­مونن و m تغییری نمی­کنه. خودتون امتحان کنید! توی متد اگر multiplier بیشتر از 100 باشه، مقدارش به 100 کاهش پیدا می­کنه ( جنگجو ها نباید زیادی قوی بشن! ). m رو بیشتر از 100 بذارید و ببینید چی میشه.

در مورد ورودی دوم، جاوا یک "ارجاع" ( reference ) به آبجکتی که توی warrior2 قرار داره رو توی متغیر محلی enemy کپی می­کنه و ازش استفاده می­کنه، ولی چون این فقط یک ارجاعه، enemy و warrior2 هر دو در واقع به یک آبجکت اشاره می­کنند و هر تغییری که از طریق enemy یا warrior2 ایجاد بشه، آبجکت اصلی تغییر می­کنه. خودتون امتحان کنید و ببینید! صدا زدن متد باعث میشه جون warrior2 واقعاً کم بشه. ولی، اگر متد hit، توی متغیر enemy ارجاع به یک آبجکت دیگه رو بریزه، warrior2 تغییری نمی­کنه. یعنی مثلاً اگر خط زیر رو اول متد hit بنویسید، warrior2 دیگه هیچ تغییری نمی­کنه.

enemy = new Warrior();

کلاس های wrapper برای primitive type ها

همونطور که تا الان دیدید، توی جاوا به نوعی تفرقه بین جنس متغیر ها وجود داره: بین primitive type ها و آبجکت ها ( reference type ها ). جاوا این تفرقه و دوگانگی رو می­پذیره تا در مقابل، بهینه باشه.وقتی شما با اعداد سر و کار دارید و روی اونا محاسبات انجام میدید، دلتون می­خواد که محاسباتتون سبک باشه و سریع اجرا بشه. اگر قرار بود برای primitive type ها از آبجکت استفاده بشه، بهینه کردن محاسبات کار سختی می­شد. ولی، گاهی اوقات ما نیاز داریم تا اعداد و مقدار های primitive دیگه رو توی آبجکت ها ذخیره کنیم ( مثال هاش رو توی داک آرایه دیده بودید! ). برای این کار، جاوا برای هر primitive type، یک کلاس wrapper استاندارد داره که می­تونید توی این جدول ببینید:



یک آبجکت از جنس یک wrapper class، یک آبجکت غیر قابل تغییره که مثل یک جور جعبه می­مونه که مقداری که بهش می­دیم رو توی خودش نگه می­داره و بهمون اجازه می­ده که ازش استفاده کنیم. ساخت یک آبجکت wrapper، خیلی ساده­ست؛ دقیقا مثل مقدار دهی primitive type هاست:

Double pi = 3.14;  
Boolean someBoolean = false;

همه ی wrapper های عددی، متد های “value” دارن که با استفاده ازشون می­تونید مقداری که توی آبجکت ذخیره کردید رو به هرکدوم از primitive type های عددی که می­خواهید تبدیل کنید. این متد ها doubleValue()، floatValue()، longValue()، intValue()، shortValue() و byteValue() هستند:

Double size = 32.76;  
double d = size.doubleValue(); // 32.76  
float f = size.floatValue(); // 32.76f  
long l = size.longValue(); // 32L  
int i = size.intValue(); // 32

این کد دقیقاً معادل با اینه که یک متغیر primitive از نوع double رو به انواع دیگه cast کنید.

متداول ترین استفاده برای کلاس های wrapper، وقتیه که می­خواهید یک مقدار primitive رو به یک متدی به عنوان ورودی بدید در صورتی که اون متد فقط آبجکت می­پذیره. برای مثال، متد add مربوط به ArrayList که قبلاً باهاش آشنا شدید، فقط با آبجکت ها کار می­کند و اگر بخواهیم از این متد استفاده کنیم، باید مقدار های primitiveی که داریم رو به آبجکت تبدیل کنیم:

// Manually wrapping an integer  
ArrayList myNumbers = new ArrayList();  
Integer thirtyThree = 33;  
myNumbers.add(thirtyThree);

اینجا ما مقدار 33 رو توی یک آبجکت از نوع Integer قرار دادیم تا بتونیم اون رو به متد add بدیم. از اون طرف، وقتی می­خواهیم مقدار 33 رو از لیست دربیاریم، باید اون رو از Integer به int تبدیل کنیم:

// Manually unwrapping an integer  
Integer theNumber = (Integer)myNumbers.get(0);  
int n = theNumber.intValue(); // 33

البته، همونطور که توی داک آرایه هم دیدید، در واقعیت نیازی نیست که ما این کار ها رو انجام بدیم و می­تونیم مستقیماً 33 رو به متد add بدیم و همینطور مستقیماً 33 رو به صورت int از لیست در بیاریم. دلیل این اتفاق اینه که تمام تبدیل های بالا رو جاوا خودش به صورت اتوماتیک برامون انجام میده. جاوا به این کار autoboxing میگه. این قابلیت جاوا باعث میشه که کد ما کوتاه تر و امن تر بشه. استفاده از کلاس های wrapper تا بخش خیلی زیادی از ما توسط کامپایلر پنهان میشه، اما همچنان داخل کامپایلر داره از این کلاس ها استفاده میشه. اینجا یک مثال دیگه داریم که از autoboxing و generics استفاده می­کنه ( با generics توی هفته های آینده بیشتر آشنا می­شید. )

// Using autoboxing and generics  
ArrayList<Integer> myNumbers = new ArrayList<Integer>();  
myNumbers.add(33);  
int n = myNumbers.get(0);

همونطور که می­بینید، اینجا متغیری از نوع Integer نداریم و همه ی تبدیل ها داره به صورت ضمنی انجام میشه. البته توجه کنید که موقع تعریف کردن ArrayList، داریم بهش می­گیم که مقادیرمون قراره از نوع Integer باشند.

Method overloading

Method overloading، این قابلیته که شما چند متد 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 متد هایی درست کنیم که روی ورودی های مختلف، کار های یکسانی انجام میدند. 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 سخت تر میشه. مثلاً برای چاپ چیز های مختلف باید متد های مختلف با اسم های 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 تابع با نام یکسان sum داریم، ولی ورودی هاشون فرق می­کنه.

به غیر از نوع ورودی ها و تعدادشون، با تغییر دادن ترتیب ورودی ها هم میشه متد ها 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 بیشتر آشنا شدید، به method overloading دوباره برمیگردیم.

چه چیزی یاد گرفتیم؟