

به نام خدا



برنامه‌سازی ییشوفته

دانشگاه شهید بهشتی · دانشکده مهندسی کامپیووتر

دکتر مجتبی وحیدی اصل

آشنایی با متد ها

آریا شاکو

فهرست مطالب

1. آشنایی با متدها
2. استفاده مجدد از متدها و ماژوله بندی کد
3. سربارگذاری متدها (Method Overloading)
4. حوزه متغیرهای محلی
5. انتزاع متدها
6. متدهای مثلثاتی و توان
7. متدهای روندسازی
8. متدهای ریاضی

```
In [ ]: !pip install jbang
import jbang
jbang.exec("trust add https://github.com/jupyter-java")
jbang.exec("install-kernel@jupyter-java")
#You need this cell in order to run the java Codes :))
```

مقدمه

در این بخش با یکی از مهمترین مفاهیم برنامه‌نویسی در جاوا یعنی **متدها (Methods)** آشنا می‌شویم. متدها بلوک‌هایی از کد هستند که برای انجام کارهای مشخص نوشته می‌شوند و کمک می‌کنند برنامه‌های شما خواناتر، قابل نگهداری‌تر و انعطاف‌پذیرتر شوند. با استفاده از متدها می‌توانید

کدهای تکراری را حذف کنید، برنامه خود را مازوله و ساخت یافته‌تر بنویسید و از قابلیت‌های مهمی مثل سربارگذاری و انتزاع استفاده کنید. در این فصل، از تعریف و انگیزه متدها شروع می‌کنیم، سپس سراغ امضا، پارامترها، نوع مقدار برگشتنی، فراخوانی و حوزه متغیرها در متدها می‌رویم و در ادامه با مفاهیمی مانند سربارگذاری متدها، متدهای آماده جاوا (ریاضی و مثلثاتی)، و مازوله‌بندی کد آشنا خواهید شد. هدف این فصل این است که با کاربرد متدها در برنامه‌های واقعی و نحوه تعریف و استفاده صحیح از آن‌ها، آشنا شوید و بتوانید از این ابزار قدرتمند برای حل مسائل پیچیده‌تر بهره ببرید. تمرین و کدنویسی فعال، کلید موفقیت شما در این بخش است.

نکته: اگر به دنبال حرفه‌ای شدن در برنامه‌نویسی هستید، باید با متدها دوست شوید! متدها به شما اجازه می‌دهند کدهایی تمیز، قابل فهم و توسعه‌پذیر بنویسید.

فهرست مطالب با جزئیات دقیق تر

آشنایی با متدها.

A. چرا به متدها نیاز داریم - انگیزه

B. استفاده از متدها - راه حل

C. تعریف متدها

D. امضای متد

E. پارامترهای فرمال و واقعی

F. نوع مقدار برگشتنی متدها

G. فراخوانی متدها

H. ردگیری فراخوانی متد

استفاده مجدد از متدها و مازوله‌بندی کد.

A. استفاده مجدد از متدها از کلاس‌های دیگر

B. مازوله‌بندی کد

3. سربارگذاری متدها (Method Overloading)

- A. مفهوم سربارگذاری متدها
- B. مثال‌هایی از سربارگذاری متدها
- C. فراخوانی مبهم متدها

4. حوزه متغیرهای محلی

- A. تعریف و محدوده متغیرهای محلی
- B. مثال‌ها و محدودیت‌های حوزه متغیرهای محلی

5. انتزاع متدها

- A. تعریف و کاربردهای انتزاع متدها
- B. متدهای مثلثاتی و توان
- A. متدهای سینوس، کسینوس و تانژانت
- B. مثال‌های exp, log, log10, pow و sqrt

6. متدهای روندساری

- A. استفاده از متدهای ceil, floor, rint و round
- B. مثال‌هایی از این متدها

7. متدهای ریاضی

- A. متدهای max, min و abs
- B. متدهای random

چرا به متدها نیاز داریم؟

متدها ابزاری ضروری در برنامه‌نویسی هستند که به ما این امکان را می‌دهند تا کدهای خود را سازماندهی کرده و بخش‌های مختلفی از برنامه را از یکدیگر جدا به عنوان مثال، فرض کنید می‌خواهید مجموع کنم. این کار به برنامه‌نویس کمک می‌کند تا کد را خواناتر، قابل نگهداری‌تر و استفاده مجدد از آن را ساده‌تر کند اعداد یک بازه خاص را محاسبه کنید. اگر هر بار این کار را به صورت دستی انجام دهید، کد شما به سرعت طولانی و پیچیده می‌شود. اما با استفاده از متدها می‌توانید یکبار این عملیات را تعریف کرده و هر جا که نیاز به محاسبه مجموع داشتید، فقط متدها را فراخوانی کنید.

```
In [ ]:  
int sum = 0;  
for (int i = 1; i <= 10; i++) sum += i;  
System.out.println("Sum from 1 to 10 is " + sum);  
  
sum = 0;  
for (int i = 20; i <= 30; i++) sum += i;  
System.out.println("Sum from 20 to 30 is " + sum);  
  
sum = 0;  
for (int i = 35; i <= 45; i++) sum += i;  
System.out.println("Sum from 35 to 45 is " + sum);
```

همانطور که مشاهده می‌کنید، این کد چندین بار عملیات مشابهی را انجام می‌دهد و تکرار این عملیات نه تنها باعث پیچیده‌تر شدن برنامه می‌شود، بلکه امکان بروز خطا را نیز افزایش می‌دهد. اینجاست که استفاده از متدها به کمک می‌آید تا کدهای تکراری حذف شوند و کارایی برنامه بهبود یابد.

```
In [ ]:  
public static int sum(int i1, int i2) {  
    int sum = 0;  
    for (int i = i1; i <= i2; i++) sum += i;  
    return sum;  
}  
  
public static void main(String[] args) {  
    System.out.println("Sum from 1 to 10 is " + sum(1, 10));  
    System.out.println("Sum from 20 to 30 is " + sum(20, 30));
```

```
System.out.println("Sum from 35 to 45 is " + sum(35, 45));  
}
```

با استفاده از متدها، کدهای مشابهی که قبلاً چندین بار تکرار می‌شد، اکنون تنها با فراخوانی یک متده را اجرا می‌شود. این کار باعث کاهش پیچیدگی برنامه و افزایش قابلیت نگهداری آن می‌شود.

تعريف متدها

در جاوا به مجموعه‌ای از دستورات گفته می‌شود که برای انجام یک کار خاص، با یک نام مشخص و قابل فراخوانی در برنامه نوشته می‌شود. (Method) متدها امکان تکرارپذیری کد، مازوله‌سازی و ساده‌سازی برنامه را فراهم می‌کنند. به کمک متدها، می‌توانید یکبار عملیاتی را بنویسید و بارها با پارامترهای مختلف از آن استفاده کنید؛ به این ترتیب، از تکرار بی‌مورد کد جلوگیری می‌شود و تغییر و نگهداری برنامه نیز ساده‌تر خواهد بود.

- هر متده معمولاً یک هدف مشخص دارد (مثلًا جمع دو عدد، پیدا کردن بزرگ‌ترین مقدار، یا چاپ پیام).
- متدها می‌توانند ورودی داشته باشند (پارامتر) و/یا خروجی (مقدار برگشته) تولید کنند.
- بدنه متده شامل دستورات اجرایی است که با فراخوانی متده اجرا می‌شوند.

```
In [ ]: public static int sum(int start, int end) {  
    int total = 0;  
    for (int i = start; i <= end; i++)  
        total += i;  
    return total;  
}
```

در مثال بالا، متدهای دو عدد به عنوان ورودی می‌گیرد و مجموع تمام اعداد بین آن دو را محاسبه و بازمی‌گرداند. این متدهای می‌توانند هر تعداد بار که خواستید با ورودی‌های متفاوت صدا بزنید.

امضای متدهای متفاوت

در جاوا فقط شامل **نام متدهای متفاوت** و **تعداد و نوع پارامترهای آن** است. نوع مقدار برگشتی، بخش امضا محسوب نمی‌شود. اهمیت امضا متد (Method Signature) امضا متد است که کامپایلر به کمک آن متدهای مشابه را (مثلاً در سربارگذاری) از هم تشخیص می‌دهد.

- متدهایی با نام یکسان اما امضا متفاوت (تعداد/نوع پارامترها) می‌توانند در یک کلاس وجود داشته باشند.
- تغییر فقط نوع مقدار برگشتی باعث تغییر امضا نمی‌شود و کامپایلر آن را به عنوان متدهای شناسایی نمی‌کند.

```
In [ ]: // امضا متد : sum(int, int)
public static int sum(int a, int b) { ... }

// امضا متد : sum(double, double)
public static double sum(double a, double b) { ... }
```

در این مثال، دو متدهای یکسان `sum` داریم، اما امضا آن‌ها به دلیل تفاوت در نوع پارامترها متفاوت است و هر دو می‌توانند در یک کلاس قرار بگیرند.

پارامترهای فرمال و واقعی

پارامترهای فرمال (Formal Parameters) پارامترهایی هستند که هنگام تعریف متدها در داخل پرانتز ذکر می‌شوند و نقش متغیرهای محلی را دارند.

پارامترهای واقعی یا آرگومان‌ها (Actual Parameters) مقادیری هستند که هنگام فراخوانی متدها به آن ارسال می‌شوند.

- هر بار که متدها صدا زده می‌شود، مقادیر آرگومان‌ها به پارامترهای فرمال کپی می‌شوند و متدها با همان مقادیر اجرا می‌شود.
- پارامترهای فرمال فقط در داخل بدنه متدها قابل استفاده هستند و بیرون از آن وجود ندارند.
- تعداد و نوع آرگومان‌ها باید با پارامترهای فرمال مطابق باشد.

```
In [ ]: public class MainClass {

    public static void greet(String name) {
        // name پارامتر فرمال است
        System.out.println("سلام " + name + "!");
    }

    public static void main(String[] args) {
        // فراخوانی متدها با پارامتر واقعی
        مقدار "علی" آرگومان واقعی است // ; ("علی")
    }
}
```

در این مثال، متدهای `greet` یک پارامتر فرمال به نام `name` دارد که هنگام فراخوانی مقدار "علی" به آن اختصاص داده می‌شود. مثالی دیگر از پارامترهای واقعی و فرمال در متدهای ریاضی:

```
int maxNum = max(10, 20); // 10 آرگومان واقعی هستند
public static int max(int a, int b) { // a و b پارامترهای فرمال هستند
```

```
    return (a > b) ? a : b;  
}
```

نوع مقدار برگشتی متدها

هر متدهای مقداری را به برنامه بازگرداند که نوع آن قبل از نام متدها ذکر نمی‌شود. اگر متدهای مقداری بازنمی‌گردانند (مثلًاً فقط عملی را انجام می‌دهند)، نوع برگشتی آن void باید باشد.

- متدهای غیر void باید حتماً با دستور return مقداری از نوع تعیین شده را بازگردانند.
- اگر نوع برگشتی void باشد، وجود دستور return اختیاری است (فقط برای پایان دادن زودهنگام متدها است).

```
In [ ]: public static double circleArea(double radius) {  
    return Math.PI * radius * radius;  
}  
  
public static void printHello() {  
    System.out.println("Hello world!");  
}
```

متدهای add و printHello از نوع void دارند و متدهای printSum و printHello مقدار برگشتی ندارند و فقط پیام چاپ می‌کنند. توجه: اگر متدهای void باشد و برای همه مسیرهای ممکن مقدار برگشتی تعریف نشده باشد، برنامه دچار خطای کامپایل می‌شود.

فراخوانی متدها

برای استفاده از متدها، کافی است نام متد را همراه با آرگومان‌های مناسب بنویسید. اگر متد مقدار برمی‌گرداند، می‌توانید نتیجه را در یک متغیر ذخیره کنید یا مستقیماً آن را استفاده کنید. همچنین می‌توانید متدهای بدون مقدار بازگشتی (`void`) را تنها با نامشان و آرگومان‌های لازم صدا بزنید.

- در صورتی که متد در همان کلاس باشد، کافی است نام متد را بنویسید. در غیر این صورت، باید با `ClassName.methodName` فراخوانی شود.
- در متد `main`، معمولاً متدهای دیگر صدا زده می‌شوند تا کد مرتب‌تر و قابل فهم‌تر شود.

```
public static void main(String[] args) {
    int s = sum(1, 100);
    System.out.println("مجموع از ۱ تا " + s);
```

که `printHello` ذخیره شده است. همچنین متد `s` با پارامترهای ۱ و ۱۰۰ فراخوانی شده و نتیجه‌اش در متغیر `sum` در این مثال، متد `{}` است: می‌توانید به شکل زیر فراخوانی کنید، (`TestMath.sum()` مثلاً) اگر متد در کلاس دیگری قرار دارد. است، بدون مقدار برگشتی فراخوانی شده است

```
int result = Math.max(7, 12);
```

ردگیری فراخوانی متد

وقتی یک متد فراخوانی می‌شود، برنامه کنترل اجرا را به آن متد منتقل می‌کند. پارامترهای واقعی به پارامترهای فرمال متد کپی می‌شوند. پس از اتمام دستورات کنترل برنامه به نقطه‌ای که متد از آن صدا زده شده برمی‌گردد، `void` یا رسیدن به انتهای متد `return` متد و اجرای دستور

- ساخته می‌شود تا متغیرهای محلی و پارامترهای آن جداگانه (stack) در حافظه، هر بار که متد فراخوانی می‌شود، یک " بلاک " مخصوص آن روی پشته نگهداری شوند.
- پس از پایان کار متده، این بلاک حذف می‌شود و مقدار برگشتی (در صورت وجود) جایگزین فراخوانی متده می‌شود.
- روند مقداردهی پارامترها و مقدار برگشتی را دنبال کرد (step by step) می‌توان با ردگیری اجرای برنامه.

```
public static void main(String[] args) {
    int i = 5, j = 8;
    int bigger = max(i, j);
    System.out.println("".است " + " عدد " + " + i + " + j + " + bigger + ").
}
```

گام به گام اجرای مثال بالا:

1. مقداردهی می‌شوند `j` و `i` شروع می‌شود. متغیرهای `main` برنامه از.

2. کپی می‌شوند `b` و `a` فراخوانی می‌شود؛ مقادیر `0` و `8` به پارامترهای `(i, j)` متده.

3. بازگردانده می‌شود (یعنی `b`) بررسی می‌شود (در اینجا `0 < 8` که غلط است)، پس مقدار `max(a > b)` در داخل متده.

4. قرار می‌گیرد `bigger` برمی‌گردد و مقدار برگشتی متده در متغیر `main` کنترل برنامه به.

5. در نهایت خروجی چاپ می‌شود.

این مکانیزم باعث می‌شود هر متده کاملاً مستقل و با متغیرهای محلی خودش کار کند و تداخلی با دیگر بخش‌های برنامه نداشته باشد.

متدهای جاوا و فراخوانی :

Java Methods

Magic in Your Code!

sum ()

printHello()

max(a, b)

method()

```
for (int i = 0;  
     i < n; i++)  
    return x + y;
```





استفاده مجدد از متدها و مازوله‌بندی کد

یکی از مزیت‌های مهم متدها، **امکان استفاده‌ی مجدد** از کد است. کافیست متدها را یکبار بنویسید و در بخش‌های مختلف برنامه، یا حتی برنامه‌های دیگر، هر زمان نیاز داشتید فقط آن را فراخوانی کنید. این ویژگی باعث می‌شود برنامه‌ها مازوله (قطعه قطعه) و ساختارمند باشند، رفع اشکال و گسترش آن‌ها هم بسیار راحت‌تر می‌شود.

استفاده مجدد از متدها از کلاس‌های دیگر

متدهایی که به صورت `public static` در یک کلاس تعریف شده‌اند، از هر کلاس دیگری هم قابل فراخوانی هستند. فقط کافیست نام کلاس را قبل از نام متدها بنویسید.
به مثال زیر توجه کنید:

```
public class MathUtil {  
    public static int square(int n) {  
        return n * n;  
    }  
}
```

فقط یکبار نوشته شده اما می‌تواند هرچا در `square` در این مثال، متدهای `int result = MathUtil.square(5); // 25;` فراخوانی متدهای `square` از کلاس دیگر // برنامه یا حتی پروژه‌های دیگر فراخوانی شود. این همان مازوله‌بندی است: کدها را به بلوک‌های قابل استفاده و مستقل تقسیم کنیم.

ماژوله‌بندی کد

هدف اصلی ماژوله‌بندی، تقسیم یک مسئله بزرگ به بخش‌های کوچکتر (ماژول) است. هر ماژول معمولاً یک کار خاص انجام می‌دهد و هر زمان نیاز باشد، می‌توانیم آن را جداگانه تغییر یا تست کیم.

```
public class StringUtil {  
    public static String reverse(String s) {  
        String result = "";  
        for (int i = s.length() - 1; i >= 0; i--) {  
            result += s.charAt(i);  
        }  
        return result;  
    }  
}
```

را در هرجای برنامه فقط با یک `reverse` اینجا هم ماژوله‌بندی باعث شده متده `TPGtahC` خروجی // خواهد بود. خط صدا بزنیم و نیاز نباشد که معکوس کردن رشته را هر بار بنویسیم.

نکته: با ماژوله‌کردن، مدیریت پروژه‌های بزرگ آسان‌تر، رفع باگ‌ها سریع‌تر و توسعه آینده بسیار راحت‌تر خواهد شد.

Modular Java Code Structure



هر ماژول یک قطعه قابل استفاده مجدد از برنامه است.

سربارگذاری متدها (Method Overloading)

یکی از قابلیت‌های مهم و کاربردی در جاوا **سربارگذاری متدها** (Overloading) است. سربارگذاری به این معنی است که می‌توان چندین متده با نام یکسان اما با **امضای متفاوت** (تعداد، نوع یا ترتیب پارامترها) در یک کلاس داشت. این قابلیت باعث می‌شود بتوانید یک عملیات را با ورودی‌های مختلف پیاده‌سازی کنید و برنامه‌تان تمیزتر و خواناتر باشد.

مفهوم سربارگذاری متدها

- امضاهای متد باید متفاوت باشند (نوع/تعداد/ترتیب پارامترها).
- تغییر فقط نوع مقدار برگشتی باعث سربارگذاری نمی‌شود و خطای کامپایل می‌دهد.
- انتخاب متد مناسب هنگام فراخوانی، به صورت خودکار توسط کامپایلر براساس ورودی‌ها انجام می‌شود.

```
public class Calculator {  
    public static int add(int a, int b) {  
        return a + b;  
    }  
    public static double add(double a, double b) {  
        return a + b;  
    }  
    public static int add(int a, int b, int c) {  
        return a + b + c;  
    }  
}
```

// استفاده از متدهای سربارگذاری شده
// خروجی: int x = Calculator.add(2, 3); // 5
// خروجی: double y = Calculator.add(2.5, 3.1); // 5.6
int z =
Calculator.add(1, 2, 3); // 6

مثال‌هایی از سربارگذاری متدها

```
public static void print(String msg) {  
    System.out.println(msg);  
}  
public static void print(int num) {  
    System.out.println(num);  
}  
public static void print(double val) {  
    System.out.println(val);  
}
```

```
// با ورودی‌های مختلف print فراخوانی متدها: print("Hello!"); print(100); print(3.14);
```

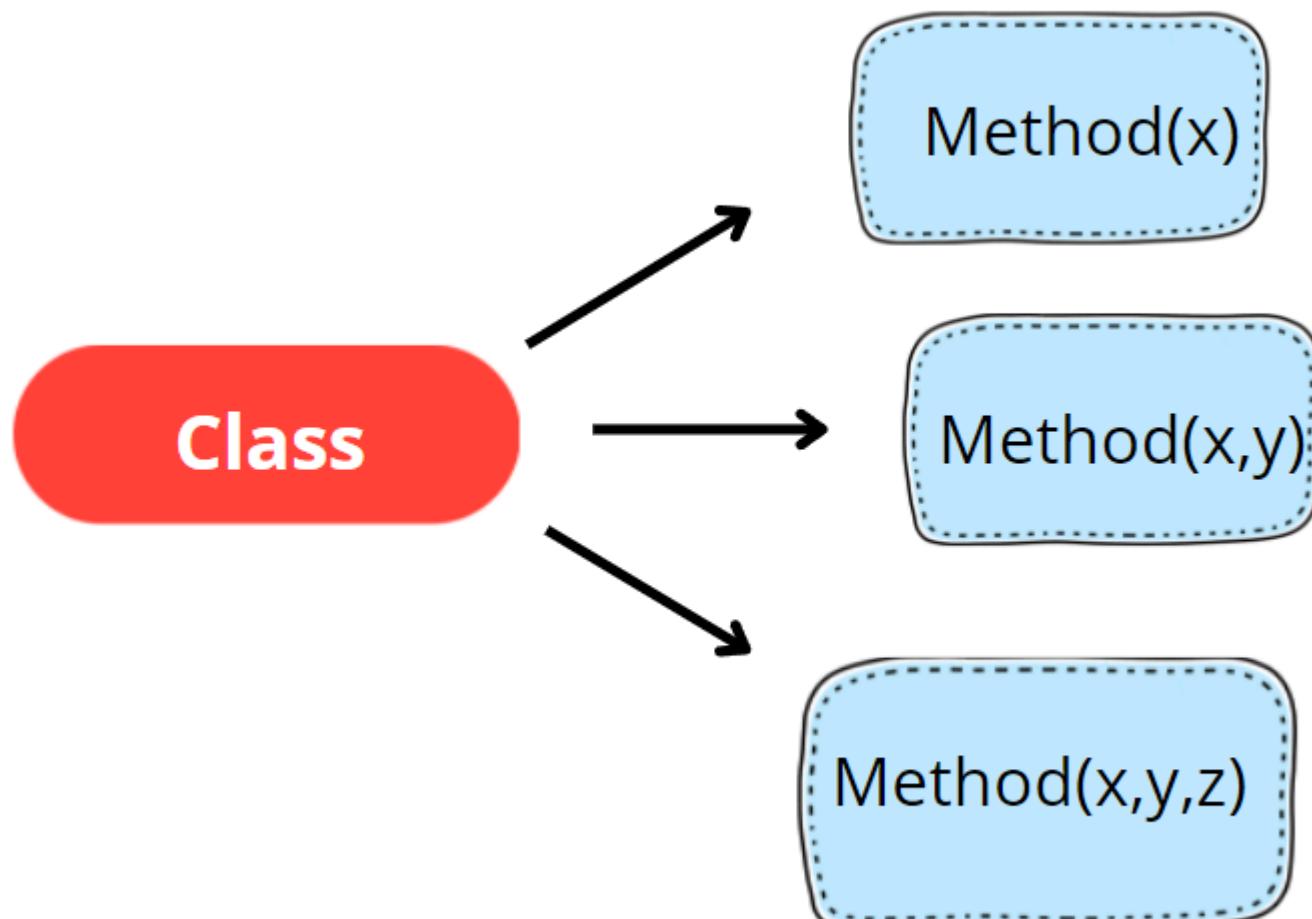
فراخوانی مبهم متدها (Ambiguous Calls)

رخدادهای کامپایلر به دلیل شباهت زیاد امضاها نمی‌تواند متدهای مناسب را انتخاب کند و خطای **فراخوانی مبهم** (Ambiguous Call) گاهی رخ می‌دهد.

```
public static void test(int a, double b) { ... }
public static void test(double a, int b) { ... }
```

در مثال بالا کامپایلر نمی‌داند کدام متدهای `test(10, 20);` را انتخاب کند //: این فراخوانی مبهم است // `test(10, 20)` می‌تواند هم به صورت `(int, double)` تفسیر شود و کامپایلر خطای دهنده `(double, int)` را هم `(double, int)` می‌داند.

نکته: سربارگذاری متدها باعث می‌شود که شما منعطفتر و تمیزتر شوید، ولی مراقب باشید امضاهای متدهای واضح و غیرمبهم باشند!



یک نام، عملکردهای مختلف؛ بستگی به نوع و تعداد پارامترها دارد.

حوزه متغیرهای محلی (Local Variable Scope)

هر متغیر در برنامه، یک **حوزه اعتبار (Scope)** مشخص دارد. **متغیرهای محلی** فقط در همان بخشی از برنامه (معمولاً داخل متد یا بلوک) که تعریف شده‌اند معتبر هستند و بیرون از آن بخش قابل دسترسی نیستند.

تعریف و محدوده متغیرهای محلی

- متغیر محلی معمولاً در ابتدای یک متد یا بلوک (مثلاً یک حلقه یا دستور شرطی) تعریف می‌شود.
- این متغیر فقط تا پایان آن متد یا بلوک معتبر است.
- پس از خروج از بلوک، فضای حافظه متغیر آزاد می‌شود و دسترسی به آن دیگر ممکن نیست.

```
public static void example() {  
    int x = 10; // فقط در این متد معتبر است x  
    if (x > 5) {  
        int y = 20; // معتبر است if فقط در همین y  
        System.out.println(x + y); // OK  
    }  
    // System.out.println(y); // خطای ناشناخته است y!  
}
```

مثال‌ها و محدودیت‌های حوزه متغیرهای محلی

```
public static void main(String[] args) {  
    int num = 42;  
    for (int i = 0; i < 3; i++) {  
        int temp = num + i;  
        System.out.println(temp);  
    }  
    // System.out.println(temp); // خطای ناشناخته است temp!
```

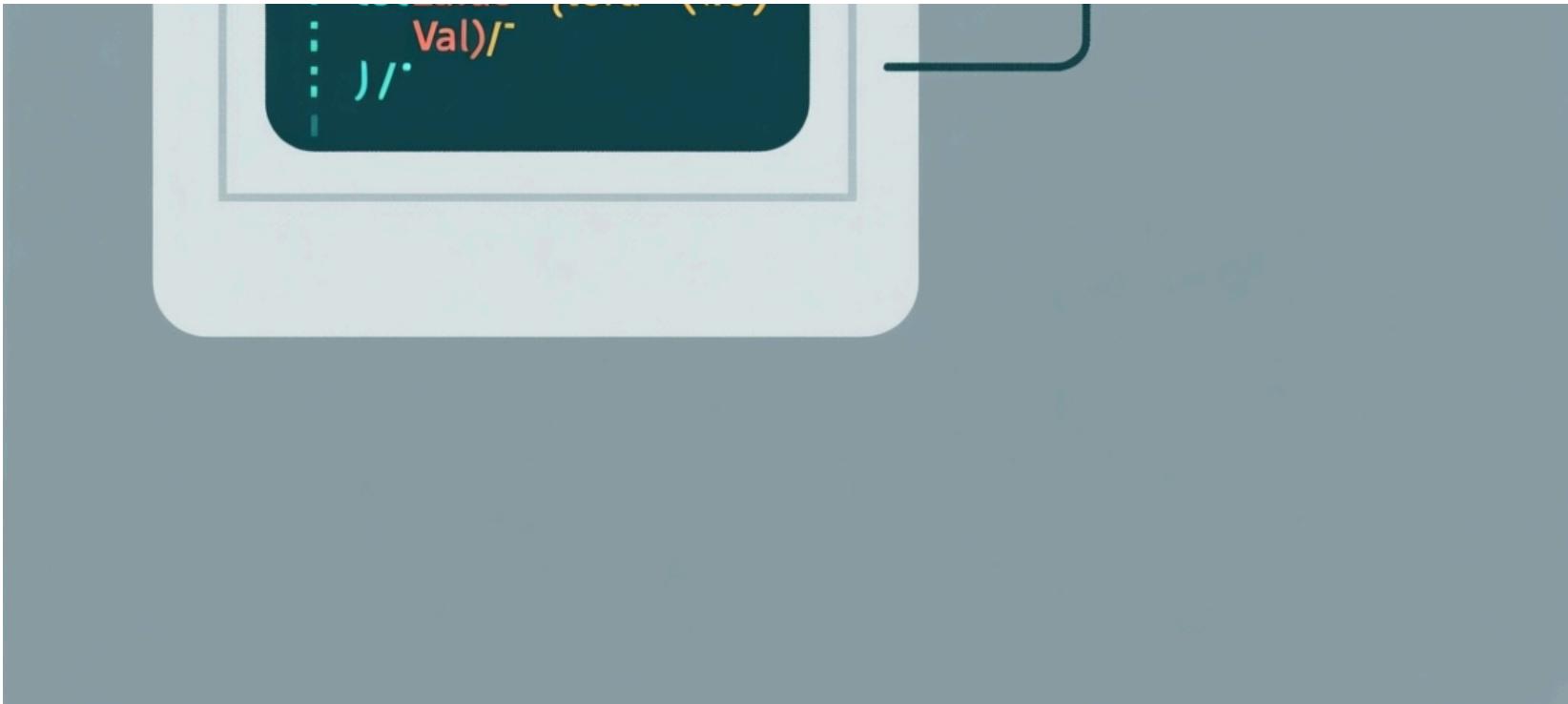
```
}
```

متغیر `temp` فقط در محدوده حلقه `for` معتبر است و بیرون از آن قابل استفاده نیست. تعریف دوباره یک متغیر با همان نام در بلوک‌های مختلف مجاز است:

```
public static void test() {  
    int a = 5;  
    {  
        int a = 10; // همان بلوک متغیر همنام داشت  
    }  
}
```

نکته: هر چه حوزه متغیرها کوچک‌تر باشد، برنامه ایمن‌تر و رفع اشکال آن آسان‌تر خواهد بود.





متغیرهای محلی فقط در بلوک یا متد خودشان وجود دارند.

انتزاع متدها (Method Abstraction)

انتزاع متدها به معنی پنهان کردن جزئیات پیاده‌سازی و نمایش فقط بخش مهم و مورد نیاز برای کاربر است. شما هنگام استفاده از یک متد، فقط به نام و ورودی/خروجی آن توجه می‌کنید و لازم نیست از جزئیات داخلی آن باخبر باشید. این اصل، برنامه را خواناتر، ایمن‌تر و قابل نگهداری‌تر می‌کند.

تعریف و کاربردهای انتزاع متدها

- کاربر فقط با **امضای متد** کار دارد (نام متد، پارامترها و نوع مقدار بازگشتی).
- جزئیات اینکه متد چطور خروجی را محاسبه می‌کند، مخفی است.

- این رویکرد باعث می‌شود بتوانیم متدها را بدون نگرانی از تاثیر روی بخش‌های دیگر برنامه تغییر دهیم.
- در استفاده از کتابخانه‌ها و کدهای آماده، انتزاع بسیار مهم است (مثلًاً متدهای کتابخانه `Math` در جاوا).

```
public static int square(int n) {  
    return n * n;  
}
```

خروجی: 49
public static void main(String[] args) { int num = 7; int result = square(num); // 49 System.out.println(result); }
چه ورودی می‌گیرد و چه خروجی می‌دهد. نحوه محاسبه داخل متدهای استفاده کننده فقط باید بداند متدهای `square` در مثال بالا، استفاده کننده فقط باید بداند متدهای `System.out.println` و `int`.

نکته: انتزاع، اصل کلیدی در برنامه‌نویسی شیء‌گرا و طراحی نرم‌افزارهای قابل توسعه و نگهداری است.

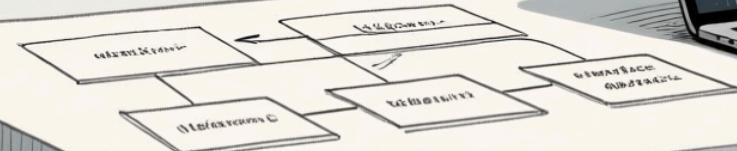
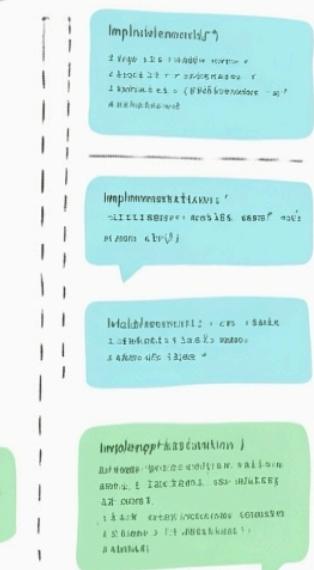
Method abstraction)

> . := (Java abstract) <= (/'-->)
 & (implements)
 > - & abstract <= (/'< == \$)
 & (implements .)
 > - / (subtract) <= void (= ==)
 & (void .)
 > - / _ (append) <= (==)
 & (?)
 > - / _ (init (- exeterat)) <= (==)
 & (Unordered . ,)

```

    graph TD
      JI[Java interface] --> UI[UML interface]
  
```

UNI implementieren
 1. SAGT DASS EINE
 UML-KLASSE IMPLIMENTIEREN
 MUSS, UM EINE KLASSE ERSTELLEN



!شما از متدهای آن استفاده می‌کنید، نه از جزئیات آن

متدهای مثلثاتی و توان (Trigonometric and Power Methods)

در جاوا، بسیاری از محاسبات ریاضی به کمک متدهای آماده کلاس `Math` انجام می‌شود. متدهای مثلثاتی و توانی از جمله پرکاربردترین این متدها هستند که محاسبات سینوس، کسینوس، تانژانت، لگاریتم، توان و جذر را به سادگی انجام می‌دهند.

متدهای سینوس، کسینوس و تانژانت

- `Math.sin(x)` : مقدار سینوس زاویه x (بر حسب رادیان)
- `Math.cos(x)` : مقدار کسینوس زاویه x (بر حسب رادیان)
- `Math.tan(x)` : مقدار تانژانت زاویه x (بر حسب رادیان)
- تبدیل درجه به رادیان: `Math.toRadians(degree)`

```
double angle = 30; // درجه
double radian = Math.toRadians(angle);
double s = Math.sin(radian); // سینوس ۳۰ درجه
double c = Math.cos(radian); // کسینوس ۳۰ درجه
double t = Math.tan(radian); // تانژانت ۳۰ درجه
```

متدهای `sqrt`, `exp`, `log`, `log10`, `pow`

- `Math.exp(x)` : مقدار e به توان x
- `Math.log(x)` : لگاریتم طبیعی (بر مبنای e)
- `Math.log10(x)` : لگاریتم بر مبنای 10
- `Math.pow(a, b)` : a به توان b
- `Math.sqrt(x)` : جذر عدد x

```
double power = Math.pow(2, 3);      // 8.0
double root = Math.sqrt(16);        // 4.0
double exponential = Math.exp(2);   // e^2
double ln = Math.log(10);          // ln(10)
double log10 = Math.log10(100);    // 2.0
```

استفاده از این متدها نیاز به هیچ کتابخانه اضافه‌ای ندارد و فقط کافی است `Math` را صدا بزنید!

نکته: ورودی متدهای مثلثاتی `Math` باید بر حسب رادیان باشد.



متداهای ریاضی برای سینوس، کسینوس، توان، جذر و غیره!

متدهای روندسازی (Rounding Methods)

متدهای روندسازی برای گرد کردن اعداد اعشاری به کار می‌روند. در جاوا این متدها همگی در کلاس `Math` قرار دارند و شامل `ceil`، `floor` و `rint` هستند.

استفاده از متدهای `round`، `ceil`، `floor` و `rint`

- `Math.ceil(x)` : کوچکترین عدد صحیح بزرگ‌تر یا مساوی x (گرد کردن به بالا)
- `Math.floor(x)` : بزرگ‌ترین عدد صحیح کوچک‌تر یا مساوی x (گرد کردن به پایین)
- `Math.rint(x)` : نزدیک‌ترین عدد صحیح به x به صورت عدد اعشاری (`double`)
- `Math.round(x)` : نزدیک‌ترین عدد صحیح به x (نتیجه `long` برای `double`، و `int` برای `float`)

```
double a = 5.3;
double b = -5.7;
```

```
System.out.println(Math.ceil(a)); // 6.0 System.out.println(Math.floor(a)); // 5.0 System.out.println(Math.rint(a)); // 5.0
System.out.println(Math.round(a)); // 5 System.out.println(Math.ceil(b)); // -5.0 System.out.println(Math.floor(b)); // -6.0
System.out.println(Math.rint(b)); // -6.0 System.out.println(Math.round(b)); // -6
```

همیشه به بالا گرد می‌کند (حتی برای اعداد منفی) **نکته:** `ceil` و `floor` می‌کند (int/long) به نزدیک‌ترین عدد صحیح به صورت `round`، به پایین `rint`.

مثال‌هایی از این متدها

```
System.out.println(Math.round(3.5));    // 4
System.out.println(Math.round(3.49));    // 3
System.out.println(Math.ceil(-3.1));    // -3.0
```

```
System.out.println(Math.floor(-3.1)); // -4.0
```

Rounding Methods in Java

Math.round()

2.3

18.34

2.7

45380

Math.ceil()

2.7

12.892

-2.7

42.350

Math.floor() ↑ ↓

-2.3

41.250

Math.rint() ↓

-2.7

20248

BigDecimal.ROUND_HALF_UP

BigDecimal.ROUND_HALF_EVEN

2.7

401841

-2.7

144_EVEN

ai

به بالا، باس: با نزدیکت بـ. عدد صحیح Math گـد کـدن. اعداد با متدهـای

متدهـای ریاضـی (Math Methods)

کلاس Math در جاوا مجموعهـای از متدهـای آماده برای انجام محاسبـات رایج ریاضـی فراهم کـرده است کـه استفادـه از آنـها بـسیار سـاده و سـریع است. از جملـه این متدهـا مـیتوان به random و max، min، abs اشارـه کـرد.

- b : بـزرگـترین مـقدار بـین a و Math.max(a, b)

- b : کـوچـکـترین مـقدار بـین a و Math.min(a, b)

- x : قـدر مـطلق عـدد Math.abs(x)

- () : یـک عـدد اـعـشارـی تـصادـفـی بـین ۰ تـا ۱ Math.random

```
int m1 = Math.max(5, 10);      // 10
int m2 = Math.min(5, 10);      // 5
int abs1 = Math.abs(-7);       // 7
double rand = Math.random();   // 0.253489
مثال:
```

مثالـهـای تـكمـيلـی

```
System.out.println(Math.max(-5, 3));      // 3
System.out.println(Math.min(0, -2));        // -2
System.out.println(Math.abs(-3.8));         // 3.8
```

// تولید یک عدد صحیح تصادفی بین 1 تا 6 (مثل تاس): int dice = 1 + (int)(Math.random() * 6);

مثال‌های بیشتر (سربرگذاری متدها)

مثال ساده:

```
public static int multiply(int a, int b) {
    return a * b;
}
public static double multiply(double a, double b) {
    return a * b;
}
// استفاده:
int x = multiply(2, 3);      // 6
double y = multiply(2.5, 4); // 10.0
```

در اینجا ضرب اعداد صحیح و اعشاری هر دو با یک نام متدازن جام شده است.

مثال متوسط:

```
public static void showInfo(String name) {
    System.out.println("Name: " + name);
}
```

```
public static void showInfo(String name, int age) {  
    System.out.println("Name: " + name + ", Age: " + age);  
}  
public static void showInfo(String name, int age, String city) {  
    System.out.println("Name: " + name + ", Age: " + age + ", City: " + city);  
}  
// استفاده:  
showInfo("Ali");  
showInfo("Sara", 20);  
showInfo("Mehdi", 23, "Tehran");
```

نمایش اطلاعات کاربر با تعداد ورودی‌های متفاوت به کمک سربارگذاری متدهای استatic.

مثال سخت:

```
public static int sum(int[] arr) {  
    int s = 0;  
    for(int i : arr) s += i;  
    return s;  
}  
public static double sum(double[] arr) {  
    double s = 0;  
    for(double d : arr) s += d;  
    return s;  
}  
public static int sum(int a, int b, int c) {  
    return a + b + c;  
}  
// استفاده:  
int[] nums = {1, 2, 3, 4};  
double[] scores = {2.5, 3.5, 4.5};  
System.out.println(sum(nums));      // 10  
System.out.println(sum(scores));    // 10.5  
System.out.println(sum(1, 2, 3));   // 6
```

در این مثال جمع آرایه اعداد صحیح، آرایه اعشاری و سه عدد جدا با سربارگذاری متدها انجام شده است.

مثال‌های بیشتر (ماژوله‌بندی و استفاده مجدد از متدها)

مثال ساده:

```
// یک متده برای چاپ خط جداکننده
public static void printLine() {
    System.out.println("-----");
}

// استفاده چندباره
printLine();
System.out.println("Hello Java!");
printLine();
```

یک قطعه کد ساده که چند بار استفاده می‌شود، به کمک ماژوله‌بندی مدیریت می‌شود.

مثال متوسط:

```
// کلاس مجزا برای عملیات ریاضی
public class MathUtil {
    public static int cube(int n) {
        return n * n * n;
    }
}

// استفاده در کلاس دیگر
int x = MathUtil.cube(4); // 64
```

در کلاس جدا تعریف شده و از هرجای پروژه می‌توان آن را صدا زد cube اینجا متد

مثال سخت:

```
// کلاس برای کار با رشته ها
public class StringUtil {
    public static boolean isPalindrome(String s) {
        StringBuilder rev = new StringBuilder(s).reverse();
        return s.equals(rev.toString());
    }
}
// استفاده در پروژه :
if(StringUtil.isPalindrome("level"))
    System.out.println("Palindrome!");
else
    System.out.println("Not palindrome!");
```

بررسی پالیندروم بودن رشته به صورت مازوله شده و قابل استفاده مجدد در پروژه.

مثال‌های بیشتر (حوزه متغیرهای محلی)

مثال ساده:

```
public static void printNumber() {
    int num = 5;
    System.out.println(num); // این متد معتبر است num فقط در این متد
}
```

اعتبار دارد و بیرون از آن ناشناخته است printNumber فقط در محدوده متغیر num محدود است.

مثال متوسط:

```
public static void main(String[] args) {  
    for(int i = 0; i < 3; i++) {  
        int temp = i * 2;  
        System.out.println(temp);  
    }  
    // System.out.println(temp); // خط !  
}
```

فقط داخل حلقه معتبر است temp !

در هر بار اجرای حلقه تعریف و فقط داخل حلقه اعتبار دارد temp ، در این مثال.

مثال سخت:

```
public static void main(String[] args) {  
    int a = 10;  
    if(a > 5) {  
        int b = a + 2;  
        System.out.println(b);  
    }  
    // System.out.println(b); // خط !  
}
```

معتبر است if فقط در b : خط !

وجود ندارد if معتبر است؛ بیرون از بلوک if فقط در محدوده b

مثال‌های بیشتر (انتزاع متدها)

مثال ساده:

```
// فقط امضای متد مهم است
public static int triple(int n) {
    return n * 3;
}
// استفاده:
System.out.println(triple(4)); // 12
```

یک عدد می‌گیرد و سه برابر را برمی‌گرداند؛ جزئیات پیاده‌سازی پنهان است triple استفاده کننده فقط باید بداند.

مثال متوسط:

```
متد انتزاعی در یک کتابخانه //
public static double circleArea(double r) {
    return Math.PI * r * r;
}
// استفاده:
double area = circleArea(2.5);
```

شعاع می‌گیرد و مساحت دایره را می‌دهد؛ فرمول داخلی را لازم نیست بداند circleArea کاربر فقط می‌داند متد.

مثال سخت:

```
استفاده از انتزاع متدها برای تغییر راحتتر منطق برنامه //
public static boolean isEven(int n) {
    return n % 2 == 0;
}
public static void main(String[] args) {
    for(int i=1; i<=5; i++)
        if(isEven(i))
            System.out.println(i + " even");
        else
            System.out.println(i + " odd");
```

}

تصمیم می‌گیرد؛ منطق داخلی هر زمان قابل تغییر است `isEven` برنامه بدون آگاهی از جزییات

مثال‌های بیشتر (انتزاع متدها)

مثال ساده:

```
// فقط امضای متدها مهم است
public static int triple(int n) {
    return n * 3;
}
// استفاده:
System.out.println(triple(4)); // 12
```

یک عدد می‌گیرد و سه برابرش را برمی‌گرداند؛ جزییات پیاده‌سازی پنهان است `triple` استفاده کننده فقط باید بداند

مثال متوسط:

```
// متدهای انتزاعی در یک کتابخانه
public static double circleArea(double r) {
    return Math.PI * r * r;
}
// استفاده:
double area = circleArea(2.5);
```

شعاع می‌گیرد و مساحت دایره را می‌دهد؛ فرمول داخلی را لازم نیست بداند `circleArea` کاربر فقط می‌داند متند

مثال سخت:

```
// استفاده از انتزاع متدها برای تغییر راحتتر منطق برنامه
public static boolean isEven(int n) {
    return n % 2 == 0;
}
public static void main(String[] args) {
    for(int i=1; i<=5; i++)
        if(isEven(i))
            System.out.println(i + " even");
        else
            System.out.println(i + " odd");
}
```

تصمیم می‌گیرد؛ منطق داخلی هر زمان قابل تغییر است isEven برنامه بدون آگاهی از جزئیات

مثال‌های بیشتر (متدهای مثلثاتی و توان)

مثال ساده:

```
// محاسبه سینوس ۹۰ درجه
double deg = 90;
double rad = Math.toRadians(deg);
double s = Math.sin(rad); // 1.0
```

ورودی باید بر حسب رادیان باشد. خروجی سینوس ۹۰ درجه، ۱ است.

مثال متوسط:

```
// محاسبه جذر و توان
double r = Math.sqrt(25);           // 5.0
double p = Math.pow(3, 4);          // 81.0
```

برای جذر و توان عدد `sqrt` و `pow` مثال استفاده همزمان از متدهای `Math` است.

مثال سخت:

```
// محاسبه لگاریتم و نمایی
double exp1 = Math.exp(1);          // e^1 ≈ 2.718
double ln100 = Math.log(100);        // ln(100) ≈ 4.605
double log10_1000 = Math.log10(1000); // 3.0
```

نمونه‌ای از استفاده ترکیبی از توابع نمایی و لگاریتم در جاوا.

مثال‌های بیشتر (متدهای روندساری)

مثال ساده:

```
double x = 4.6;
System.out.println(Math.round(x)); // 5
```

به ۰ گرد می‌شود `round` عدد ۴.۶ با.

مثال متوسط:

```
double a = -2.8;
System.out.println(Math.ceil(a)); // -2.0
```

```
System.out.println(Math.floor(a)); // -3.0
```

همیشه به پایین floor همیشه به بالا گرد می‌کند، حتی برای اعداد منفی: ceil.

مثال سخت:

```
double[] arr = {1.2, 2.5, 3.8, 4.49};  
for(double num : arr) {  
    System.out.print(Math.rint(num) + " ");  
}  
// 4.0 4.0 2.0 1.0
```

گرد می‌کند double هر عدد را به نزدیکترین عدد صحیح به صورت rint.

مثال‌های بیشتر (متدهای ریاضی)

مثال ساده:

```
int max = Math.max(11, 5); // 11  
int min = Math.min(-2, 7); // -2
```

max و min بدست آوردن بیشینه و کمینه دو عدد به کمک متدهای max و min.

مثال متوسط:

```
System.out.println(Math.abs(-12)); // 12  
System.out.println(Math.abs(3.5)); // 3.5
```

abs محاسبه قدرمطلق یک عدد صحیح و اعشاری با

:مثال سخت

```
// تولید عدد صحیح تصادفی بین 1 تا 100
int rnd = 1 + (int)(Math.random() * 100);
System.out.println(rnd);
```

در ۱۰۰ و جمع کردن ا، عدد تصادفی بین ۱ تا ۱۰۰ تولید می‌شود Math.random با ضرب

مثال چالشی (ماژوله‌بندی و استفاده مجدد از متدها)

یک متده بنویسید که یک آرایه از اعداد صحیح بگیرد و دو متده مجزا برای جمع اعداد مثبت و جمع اعداد منفی آرایه بنویسد. در نهایت، حاصل جمع اعداد مثبت و جمع اعداد منفی را در خروجی چاپ کنید.

مثال چالشی (حوزه متغیرهای محلی)

آرایه را در خروجی چاپ کند. حتماً بررسی اول بودن عدد را با متغیر محلی (prime) یک متده بنویسید که یک آرایه از اعداد صحیح دریافت کند و فقط اعداد اول انجام دهید که فقط در همان محدوده حلقه معتبر باشد.

مثال چالشی (انتزاع متدها)

چند متده شرطی (مثلاً بررسی زوج بودن، بزرگتر از ۵ بودن و غیره) بنویسید و یک متده بنویسید که با دریافت آرایه و یک پارامتر شرطی (مثلاً یک رشته) فقط اعدادی را چاپ کند که شرط مربوطه را داشته باشند.

مثال چالشی (متدهای مثلثاتی و توان)

برنامه‌ای بنویسید که زاویه‌های 0 تا 180 را با گام 30 درجه به رادیان تبدیل کند و مقدار سینوس هر زاویه را محاسبه و به صورت جدولی نمایش دهد.

مثال چالشی (متدهای روندسازی)

گرد کند و نتیجه هر روش را به صورت round و ceil برنامه‌ای بنویسید که 0 عدد اعشاری تصادفی بین 0 تا 100 تولید کند و هر عدد را به سه روش ستونی چاپ کند.

مثال چالشی (متدهای ریاضی)

بار عدد تصادفی بین -100 تا 100 تولید کند و بیشینه، کمینه و میانگین قدر مطلق این اعداد را محاسبه و چاپ کند n برنامه‌ای بنویسید که

راه حل (ماژوله‌بندی و استفاده مجدد از متدها)

```
public static int sumPositives(int[] arr) {  
    int sum = 0;  
    for(int x : arr)  
        if(x > 0) sum += x;  
    return sum;  
}  
public static int sumNegatives(int[] arr) {  
    int sum = 0;  
    for(int x : arr)  
        if(x < 0) sum += x;  
    return sum;  
}  
public static void main(String[] args) {  
    int[] nums = {4, -2, 7, -5, 3};
```

```
        System.out.println("Sum of positives: " + sumPositives(nums));
        System.out.println("Sum of negatives: " + sumNegatives(nums));
    }
```

راه حل (حوزه متغیرهای محلی)

```
public static boolean isPrime(int n) {
    if(n < 2) return false;
    for(int i = 2; i <= Math.sqrt(n); i++)
        if(n % i == 0) return false;
    return true;
}
public static void printPrimes(int[] arr) {
    for(int num : arr) {
        boolean primeHere = isPrime(num); // متغیر محلی
        if(primeHere)
            System.out.print(num + " ");
    }
}
public static void main(String[] args) {
    int[] arr = {2, 4, 5, 9, 13};
    printPrimes(arr); // خروجی: 13 5 2
}
```

راه حل (انتزاع متدها)

```
public static boolean isEven(int n) { return n % 2 == 0; }
public static boolean isGreaterThanFive(int n) { return n > 5; }
```

```
public static void printByCondition(int[] arr, String cond) { for(int n : arr) { if(cond.equals("even") && isEven(n)) System.out.print(n + " ");
else if(cond.equals("gt5") && isGreaterThanFive(n)) System.out.print(n + " "); } }
public static void main(String[] args) { int[] arr = {2, 7, 4, 9}; printByCondition(arr, "even"); // خروجی: 4 2 System.out.println(); printByCondition(arr, "gt5"); // خروجی: 9 7 }
```

راه حل (متدهای مثلثاتی و توان)

```
public static void main(String[] args) {  
    System.out.println("Angle\tSin");  
    for(int angle = 0; angle <= 180; angle += 30) {  
        double rad = Math.toRadians(angle);  
        double sinVal = Math.sin(rad);  
        System.out.printf("%3d\t%.2f\n", angle, sinVal);  
    }  
}
```

راه حل (متدهای روندسازی)

```
public static void main(String[] args) {  
    System.out.println("Num\tRound\tCeil\tFloor");  
    for(int i = 0; i < 10; i++) {  
        double num = Math.random() * 100;  
        System.out.printf("%.2f\t%d\t%.0f\t%.0f\n",  
            num, Math.round(num), Math.ceil(num), Math.floor(num));  
    }  
}
```

راه حل (متدهای ریاضی)

```
public static void main(String[] args) {  
    int n = 10, max = -51, min = 51, sumAbs = 0;  
    for(int i = 0; i < n; i++) {  
        int rnd = -50 + (int)(Math.random() * 101);  
        max = Math.max(max, rnd);  
        min = Math.min(min, rnd);  
        sumAbs += Math.abs(rnd);  
    }  
    System.out.println("Max: " + max + ", Min: " + min +  
        ", Avg Abs: " + (sumAbs / (double)n));
```

خودمون رو بسنجیم

این بخش برای این طراحی شده که در پایان مطالعه این اسلاید، بتونی خودت رو محک بزنی و ببینی آیا مفاهیم رو به خوبی یاد گرفتی یا نه. سوالات زیر رو مرور کن و سعی کن بدون نگاه کردن به متن درس، به اون ها پاسخ بدی.

- چه تفاوتی با هم دارند؟ **پارامتر (parameter)** و **آرگومان (argument)**
- را با یک مثال توضیح دهید **Method Overloading** (سریارگذاری متدها).
- در تعریف یک متدها به چه معناست؟ **void** کلمه کلیدی
- متدهای **ماژوله** چه مزایایی در توسعه نرم افزار دارند؟

پایان

(: در صورت وجود هرگونه سوال میتوانید با من در ارتباط باشید
gmail: ariashakoo1@gmail.com **telegram:** @Ariashakoo