



کارگاه برنامه نویسی پیشرفته

دستور کار شماره پنج

اهداف

- آشنا شدن با انواع مدل حافظه در جاوا
- آشنایی با داده‌های اولیه و غیر اولیه
- آشنایی با تفاوت میان **identity** و **equality**
- آشنایی با کلاس‌های **wrapper**
- آشنایی با فرآیند تست و دیباگ^۱ پروژه

^۱ Debug



فهرست مطالب

۴	مدل حافظه در جاوا
۵	نوع های اولیه و نوع های غیر اولیه
۵	Identity vs Equality
۶	زباله روب
۷	تمرین
۹	Wrapper Classes
۱۰	اتو باکسینگ و آنباکسینگ
۱۱	آموزش کار با اشکال زدا
۱۳	معرفی انواع نقطه توقف ها
۱۵	معرفی منوی اصلی اشکال زدا
۱۵	معرفی execution toolbar
۱۶	Step over
۱۶	Step into
۱۶	Force step into
۱۷	Step out
۱۷	Run to cursor
۱۸	معرفی status toolbar
۱۸	Rerun
۱۸	Edit configuration
۱۸	Resume program
۱۹	Stop
۱۹	Show breakpoint



۱۹

۱۹

۲۰

۲۳

۲۴

۲۴

۲۴

Mute breakpoints

معرفی قسمت متغیرها

معرفی watcherها

معرفی قسمت کنسول

تمرین استفاده از اشکال زدا

آماده سازی پروژه

توضیح پروژه



مدل حافظه در جاوا

در کلاس با مدل حافظه در جاوا و نحوه عملکرد آن آشنا شدید. در این بخش قصد داریم به مرور و توضیح برخی نکات پیرامون مدل حافظه بپردازیم:

(۱) در جاوا امکان دسترسی به آدرس یک شیء در حافظه وجود ندارد؛ به همین خاطر تمام آدرس‌های حافظه که ما در کشیدن مدل حافظه هیپ^۲ و استک^۳ استفاده می‌کنیم، تنها آدرس‌های فرضی هستند و نمی‌توانیم این آدرس‌ها را به صورت دقیق مشخص کنیم.

(۲) برای نمایش آدرس یک خانه حافظه از اعداد در مبنای ۱۶ (هگزا دسیمال^۴) استفاده می‌کنیم و همچنین، برای راحتی کار، آدرس‌دهی از ۰ شروع می‌شود.

(۳) تقسیم بندی خانه‌های حافظه می‌تواند براساس بایت باشد اما معمولاً استفاده از اندازه یک عدد صحیح در حافظه برای این کار مناسب‌تر می‌باشد که این مقدار در جاوا برابر با ۴ بایت است.

توجه: همواره بخشی از حافظه مربوط به ذخیره متغیرهای ثابت و همچنین متغیرهای مربوط به کلاس‌ها است، که به این بخش داده‌های استاتیک^۵ گفته می‌شود.

هنگامی که یک شیء ساخته می‌شود، بخشی از حافظه هیپ به آن اختصاص داده می‌شود. همچنین هنگامی که یک متد فراخوانی می‌شود، بخشی از حافظه تحت عنوان استک فریم^۶ در نظر گرفته خواهد شد که مربوط به نگهداری متغیرهای محلی آن متد است. نکته قابل توجه این است که ورودی‌های یک متد نیز جزو متغیرهای محلی محسوب می‌شود. این استک فریم در حافظه استک ذخیره می‌شود.

^۲ Heap

^۳ Stack

^۴ Hexadecimal

^۵ Static Data

^۶ Stack Frame



نوع‌های اولیه و نوع‌های غیر اولیه^۷

همانطور که می‌دانیم جاوا یک زبان کاملاً شیء‌گرا نیست؛ و این موضوع به خاطر وجود نوع‌های اولیه در جاوا است. نوع‌های اولیه، داده‌های اولیه در جاوا می‌باشند که با آنها آشنایی دارید؛ مانند `int`، `char` و `float`. هنگام تعریف یک متغیر از نوع اولیه شیء جدیدی در حافظه هیپ ساخته نخواهد شد و مقدار آن به صورت محلی ذخیره می‌شود؛ اما هنگامی که یک متغیر از نوع غیر اولیه ساخته می‌شود، خانه مربوط به آن در حافظه تنها شامل یک آدرس است که به ابتدای شیء مورد نظر در حافظه هیپ اشاره خواهد کرد. همان‌طور که پیش‌تر هم گفتیم جاوا هنگام استفاده از اشیاء همواره به صورت «pass by value» عمل کرده، که این مسئله هنگام دادن ورودی به یک متد هم صدق می‌کند. ورودی‌هایی که به صورت اولیه باشند، تنها شامل مقدار متغیر می‌شوند و تغییر در ورودی تابع تاثیری بر مقدار اصلی متغیر نخواهد داشت. اما ایجاد تغییر در بدنه متد، بر روی ورودی‌هایی که به صورت شیء هستند، منجر به تغییر در شیء اصلی می‌شود.

Identity vs Equality

همان‌طور که می‌دانید در جاوا برای مشخص کردن برابری دو رشته، از متد `equals` استفاده می‌کنیم و استفاده از عملگر `==` همیشه نتیجه درستی را در بر نخواهد داشت؛ اما هنگام چک کردن برابری دو متغیر از نوع اولیه استفاده از عملگر `==` تقریباً همیشه کارساز خواهد بود (برای اعداد اعشاری به دلیل اینکه مقدار تقریبی نگهداری می‌شود، ممکن است برابری با آنچه انتظار داریم متفاوت باشد).

علت این موضوع مدل حافظه جاوا و تفاوت در طریقه ذخیره سازی اشیاء و متغیرهای اولیه در حافظه است. هنگامی که از عملگر `==` استفاده می‌کنیم، مقداری که در خانه حافظه دو متغیر وجود دارد با هم مقایسه می‌شوند. این مسئله برای متغیرهای اولیه همواره جواب درستی را نتیجه می‌دهد اما همان‌طور که درباره اشیاء گفتیم، خانه مربوط به متغیر آن‌ها در حافظه تنها شامل آدرس آن شیء در حافظه هیپ است؛ بنابراین هنگام استفاده از `==` تنها در صورتی مقدار نهایی، `true` خواهد بود که هر دو متغیر به یک مکان در حافظه اشاره کنند و در واقع هر دو، یک شیء یکسان باشند. برای رفع این مشکل، جاوا استفاده از متد `equals` را پیشنهاد می‌کند. این متد کاری به

^۷ Primitive type & non-primitive type



آدرس اشیاء در حافظه ندارد و تنها با بررسی مقادیر فیلدهای دو شیء نتیجه برابر بودن آنها را مشخص می‌کند. درآینده یاد خواهید گرفت که چگونه متد equals دلخواه خود را برای یک کلاس مشخص کنید. اما اکنون می‌توانید از equals برای رشته‌ها استفاده کنید.

زباله‌روب^۸

همان‌طور که می‌دانید، هنگام ایجاد یک شیء، بخشی از حافظه هیپ به آن اختصاص داده می‌شود. اما اگر به صورت مداوم اقدام به تولید اشیاء جدید کنیم، چه اتفاقی خواهد افتاد؟ در این صورت با توجه به محدود بودن حجم حافظه، پس از مدتی حافظه پر خواهد شد. در جاوا زباله‌روب وظیفه حل این مشکل را دارد.

زباله‌روب تعداد اشاره‌گرهایی را که به یک شیء اشاره می‌کنند می‌شمارد و در صورتی که اشاره‌گری به یک شیء وجود نداشته باشد جاوا متوجه می‌شود که دیگر نیازی به وجود آن شیء نیست و شیء از حافظه پاک خواهد شد. وجود زباله‌روب به این معنی نیست که می‌توانیم بدون نگرانی اقدام به تولید اشیاء جدید کنیم. توجه کنید که زباله‌روب تنها اشیاء بدون اشاره‌گر را پاک می‌کند. پس اگر تعداد زیادی شیء با اشاره‌گر بسازیم، مثلاً یک آری لیست^۹ با تعداد اعضای بسیار بالا، همچنان مشکل کمبود حافظه وجود خواهد داشت. همچنین در صورتی که سرعت ساختن اشیاء جدید توسط برنامه بیشتر از سرعت زباله‌روب باشد، باز هم این مشکل به وجود خواهد آمد.

^۸ Garbage Collector

^۹ Array List



تمرین

تمرین زیر در جهت افزایش مهارت شما در درک مدل حافظه طراحی شده است. کدهای زیر را به دقت بخوانید و در انتها دیاگرام هیپ-استک^{۱۰} برنامه را پس از اجرا شدن آخرین خط متد main رسم کنید (بعد از اجرای خط «System.out.println(area);»).

نکات: می‌توانید بین روش‌های پوینتر مدل^{۱۱} و آدرس مدل^{۱۲}، هر کدام را برای رسم انتخاب کنید. نیازی نیست که داده‌های ایستا را مشخص کنید اما باید آورده^{۱۳} هر شیء را مشخص کنید.

در انتها دیاگرام رسم شده را به استاد درس تحویل داده و عملکرد زباله‌روب در حین اجرای برنامه را به صورت مختصر به استاد درس توضیح دهید.

```
public class Point {  
  
    private final int x;  
    private final int y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX() {  
        return x;  
    }  
  
    public int getY() {  
        return y;  
    }  
}
```

^{۱۰} heap-stack Diagram

^{۱۱} Pointer model

^{۱۲} Address model

^{۱۳} Overhead



```
public class Triangle {  
  
    private final Point p1;  
    private final Point p2;  
    private final Point p3;  
  
    public Triangle(Point p1, Point p2, Point p3) {  
        this.p1 = p1;  
        this.p2 = p2;  
        this.p3 = p3;  
    }  
  
    public double getArea() {  
        return Math.abs(0.5 * (p1.getX() * (p2.getY() - p3.getY())  
            + p2.getX() * (p3.getY() - p1.getY())  
            + p3.getX() * (p1.getY() - p2.getY())));  
    }  
}
```

```
public class Main {  
  
    public static void main(String[] args) {  
        Point p1 = new Point(1, 4);  
        Point p2 = new Point(4, 4);  
  
        Triangle t = new Triangle(p1, p2, new Point(1, 8));  
  
        double area = t.getArea();  
        System.out.println(area);  
    }  
}
```




Wrapper Classes

این کلاس‌ها تنها شامل یک مقدار اولیه^{۱۴} اند، که درواقع می‌توان گفت wrapper class ها روشی برای ذخیره سازی انواع داده اولیه به صورت شی هستند.

استفاده از wrapper class ها کاربردهای زیادی دارد؛ برای مثال یکی از این کاربردها، استفاده از حالت فراخوانی از مرجع^{۱۵} هنگام ورودی دادن به یک تابع است: اگر بخواهیم در یک تابع ورودی از نوع اولیه را تغییر دهیم، این تغییرات در مقدار اصلی اعمال نخواهند شد؛ اما با استفاده از wrapper class این کار امکان پذیر است. یک کاربرد بسیار مهم دیگر استفاده از مقادیر اولیه در کالکشن‌ها است. پیش‌تر با کالکشن‌ها آشنا شده‌اید. اما باید بدانید کالکشن‌ها تنها امکان ذخیره مقادیر به صورت شی را دارند و شما نمی‌توانید یک کالکشن با مقادیر اولیه داشته باشید. اما با استفاده از wrapper class امکان این کار فراهم شده است.

در جدول زیر می‌توانید لیست wrapper class ها را مشاهده کنید:

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

^{۱۴} primitive type

^{۱۵} call by reference



اتوباکسینگ^{۱۶} و آنباکسینگ^{۱۷}

به فرآیند تبدیل خودکار یک داده اولیه به wrapper class متناظر آن، اتوباکسینگ و به معکوس این فرآیند، آنباکسینگ گفته می‌شود. در واقع هنگام استفاده از یک wrapper class نیازی به استفاده از متدهای مربوطه برای تبدیل نیست و این کار به صورت خودکار انجام می‌پذیرد.

```
3 public class Main {
4
5     public static void main(String[] args) {
6         Integer a = 10; // autoboxing
7         System.out.println(a);
8
9         // This warning means you can use autoboxing
10        Character c = Character.valueOf('D');
11        char primitiveChar = c; // unboxing
12        System.out.println(c);
13        System.out.println(primitiveChar);
14
15
16        Double d = 2.15; // autoboxing
17        // This warning means you can use unboxing
18        double primitiveDouble = d.doubleValue();
19        System.out.println(d);
20        System.out.println(primitiveDouble);
21    }
22
23 }
```

^{۱۶} autoboxing

^{۱۷} unboxing



آموزش کار با اشکال زدا

در ادامه این جلسه قصد داریم شما را با ابزار بسیار مهم به نام اشکال زدا آشنا کنیم. اما مانند هر ابزار دیگری بهتر است قبل از معرفی آن، به معرفی فلسفه وجود آن و دلیل وجود آن بپردازیم:

برخلاف چیزی که اکثر ما درباره شغل برنامه نویسی انتظار داریم، یک برنامه نویس در طول روز بیشتر وقت خود را به دیباگ کردن کدهای نوشته شده توسط خودش یا دیگران اختصاص می دهد. خیلی اوقات، برنامه نویس برای دیباگ کردن نیاز دارد که بداند که در هر قسمت، وضعیت نرم افزار به چه صورتی است؛ به این معنا که مقدار متغیرها، ورودی تابع ها، نتیجه شرطها و ...، دقیقا به چه شکل است.

یک روش بدیهی استفاده از `print statement`ها برای فهمیدن وضعیت نرم افزار در هر نقطه است که می تواند در موارد محدودی اطلاعاتی که برنامه نویس نیاز دارد را به او بدهد. اما به دلیل اینکه پیچیدگی نرم افزارهایی که برنامه نویسان با آن سروکار دارند روز به روز در حال افزایش است، این روش روز به روز ناکارآمدتر به نظر می آید. برای حل این مشکل ابزاری به نام اشکال زدا در اکثر محیطهای توسعه یکپارچه معروف قرار داده شده است که به برنامه نویس کمک می کند که مرحله به مرحله با اجرای برنامه جلو برود و به صورت همزمان تمام پارامترهایی که ممکن است به برنامه نویس در درک کردن وضعیت نرم افزار کمک کند را در اختیار او قرار می دهد.

حال که با اشکال زدا به صورت محدود آشنا شدیم، بهتر است روش فعال کردن آن را در `IntelliJ`، که یکی از قوی ترین اشکال زداها را در بین آی دی ای^{۱۸} های جاوا را دارد، آشنا شویم.

برای اینکه اشکال زدا فعال شود باید یک نقطه توقف^{۱۹} در یکی از خطهای برنامه مشخص کنیم. برای این کار باید سمت راست شماره خط مورد نظر را کلیک کنیم و بعد از آن، یک دایره قرمز رنگ جلوی شماره خط قرار خواهد گرفت؛ مانند شکل زیر:

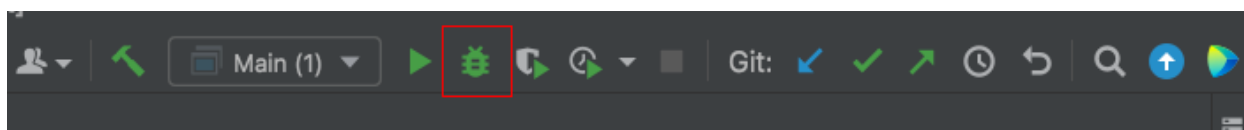
```
32 ● this.socket = new Socket(host: "127.0.0.1", Integer.parseInt(this.gamePort));
```

^{۱۸} IDE

^{۱۹} Breakpoint

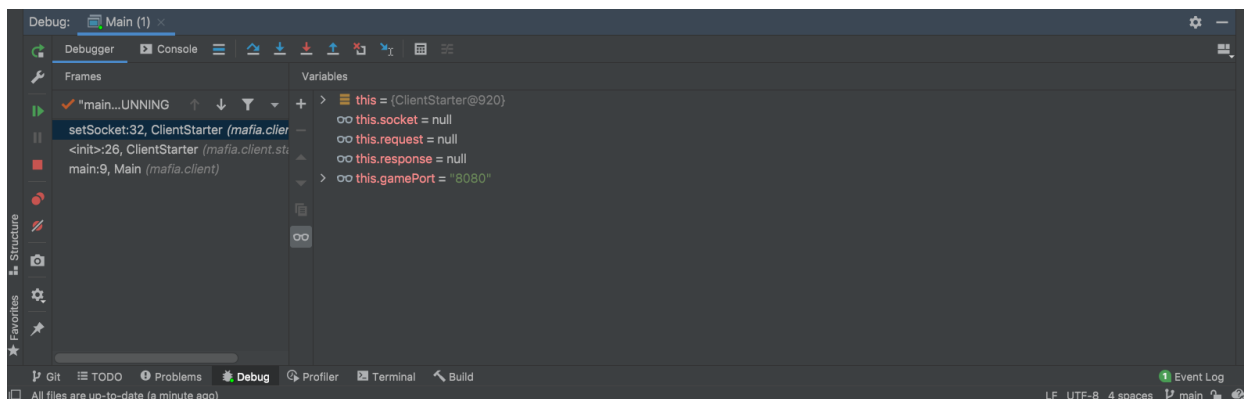


حال، برای اینکه اشکال زدا فعال شود باید نرم افزار را در حالت دیباگ اجرا کنیم. برای این کار کافی است در سمت راست بالای آی دی ای بر روی دکمه مشخص شده در عکس زیر کلیک کنیم:



در این مرحله نرم افزار شروع به اجرا شدن می کند.

به محض اینکه اجرای کد به نقطه توقف که ما انتخاب کردیم برسد، منویی شبیه به منوی زیر برای ما نشان داده می شود که در واقع همان اشکال زدا است:



در این جلسه سعی می کنیم تا حد مناسبی با توانایی های مختلفی که این اشکال زدا دارد آشنا شویم. توجه داشته باشید که اکثر ویژگی های که در ادامه مطرح می شوند معمولاً در اشکال زداهای دیگر نیز موجود هستند. اما امکان دارد شیوه دقیق نمایش آن ها یا روش کار آن ها مقداری با اشکال زداهایی که در محصولات شرکت JetBrains قرار دارد متفاوت باشد.



معرفی انواع نقطه توقف

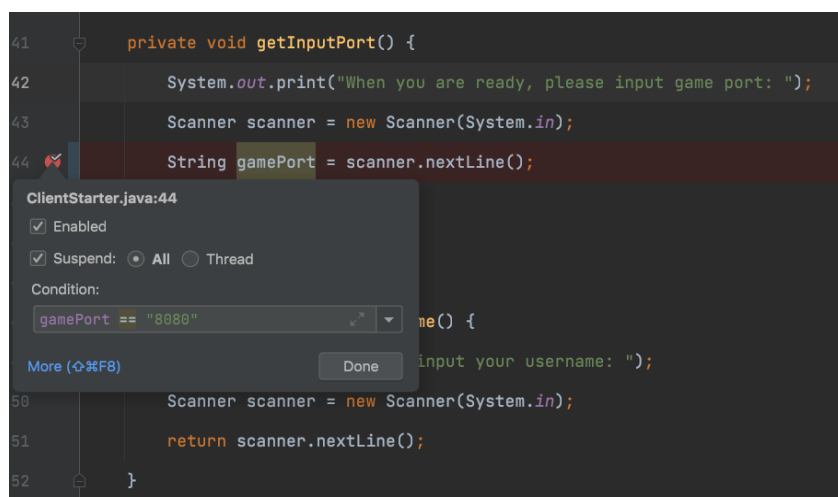
اکثر اوقات نیاز داریم به محض رسیدن اجرا کننده به نقطه توقف، ابزار اشکال‌زدا فعال شود؛ این نوع نقطه توقف به **نقطه توقف نرمال**^{۲۰} معروف است. اما بعضی از اوقات نیاز داریم که نقطه توقف‌ها به صورت پیشرفته‌تری عمل کنند و در شرایط خاصی اشکال‌زدا را فعال کنند که گاهی به این نوع از نقطه توقف‌ها، **نقطه توقف‌های پیشرفته**^{۲۱} نیز گفته می‌شود.

یکی از ویژگی‌های اشکال‌زدا IntelliJ، انواع مختلفی از نقطه توقف‌های پیشرفته آن است که با مهم‌ترین آن‌ها به طور مختصر آشنا می‌شویم.

نقطه توقف شرطی^{۲۲}

این نوع نقطه توقف زمانی اجرا می‌شود که یک شرط مشخص شده در لحظه برخورد اجرا کننده نرم افزار با آن نقطه توقف صادق باشد و اگر این شرایط موجود نباشد، اشکال‌زدا فعال نشده و کد به اجرای معمولی خود ادامه می‌دهد.

نحوه استفاده: برای درست کردن این نقطه توقف، باید بر روی قسمتی که دایره قرمز وجود دارد کلیک راست کنید تا پنجره‌ای مانند زیر برای شما نمایش داده شود و در قسمت Condition شرط خود را، که دقیقاً شبیه به بقیه شرط‌هایی است که در دستورات شرطی قبلاً نوشته‌اید، بنویسید.



^{۲۰} Normal breakpoint

^{۲۱} Advanced breakpoint

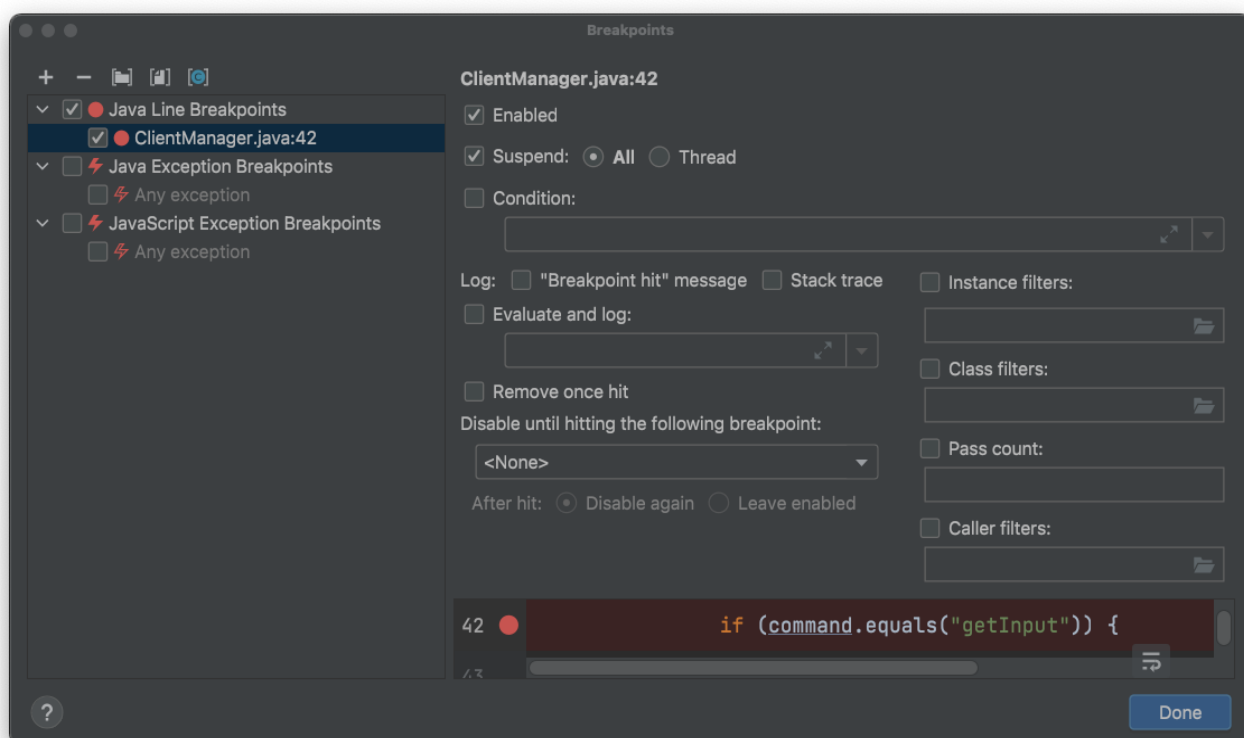
^{۲۲} Conditional breakpoint



نقطه توقف شمارش گذر ۲۳

فرض کنید که در یک حلقه هستیم و نیاز داریم که اگر اجرا کننده برای تعداد مشخصی به یک نقطه توقف رسید، اشکال زدا فعال شود. برای این منظور از نقطه توقف‌های شمارش گذر استفاده می‌کنیم.

نحوه استفاده: برای فعالسازی Hit count breakpoint باید مانند مورد قبل عمل کنیم، با این تفاوت که به جای تعریف کردن یک شرط، روی دکمه More کلیک می‌کنیم و پنجره‌ای شبیه به عکس زیر برای نمایش داده می‌شود. سپس در قسمت Pass Count تعداد دفعاتی که باید این خط اجرا شود تا اشکال زدا فعال شود را می‌نویسیم.



توجه داشته باشید که اشکال زدای که در IntelliJ موجود است، انواع نقطه توقف‌های بیشتری دارد که می‌توانید برای آشنایی بیشتر با آن‌ها، به [مستندات رسمی اشکال زدا در وبسایت jetbrains](#) مراجعه کنید.

^{۲۳} pass count breakpoint

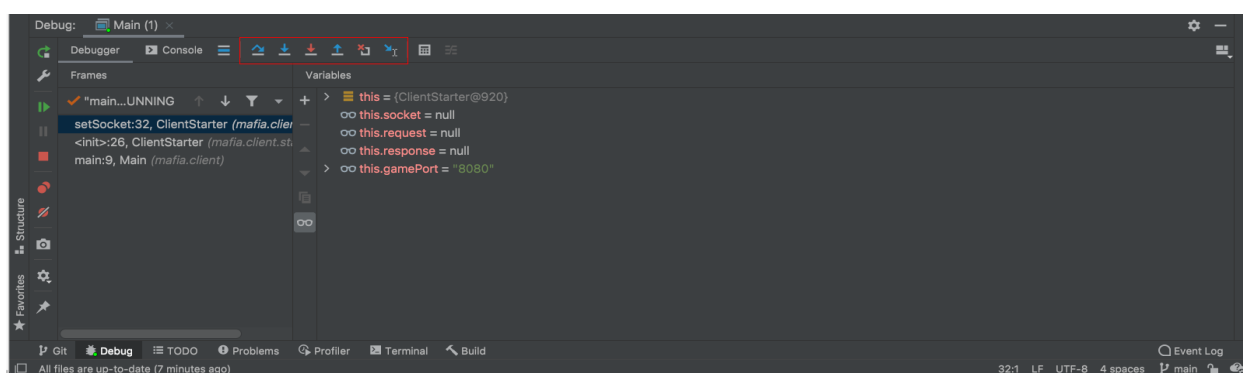


معرفی منوی اصلی اشکال‌زدا

همانطور که ابتدای دستور کار اشاره شد، بعد از رسیدن به یک نقطه توقف، در واقع خود ابزار اشکال‌زدا شروع به کار کرده و به نمایش در می‌آید و تمام اطلاعاتی که ممکن است مفید باشند را به ما نمایش می‌دهد. به دلیل اینکه ابزار اشکال‌زدا کاربردهای زیادی داشته و اطلاعات زیادی را به ما در مورد نرم افزار نمایش می‌دهد، بدیهی است که ممکن است در لحظه اول سردرگم کننده باشد. در ادامه قصد داریم به توضیح بخش‌های مختلف آن پردازیم تا پیچیدگی‌های آن ساده‌تر شود.

معرفی execution toolbar

ابزار اشکال‌زدا می‌تواند کدهای ما را خط به خط اجرا کند و در هر خط وضعیت نرم افزار در آن لحظه^{۲۴} را به صورت دقیق نمایش دهد. برای کنترل دقیق شماره خط کدی که اشاره‌گر به آن اشاره می‌کند (آن خط کدی که اجرای نرم افزار در ابتدای آن متوقف شده است) دکمه‌هایی وجود دارند که در execution toolbar عکس زیر مشخص شده است و به معرفی هر کدام از این دکمه‌ها و کاربرد آن‌ها در دنیای واقعی می‌پردازیم.



^{۲۴} program state



Step over

زمانی که بخواهیم اشاره گر بدون وارد شدن به متد دیگری یک دستور جلو برود، از این دکمه استفاده می‌کنیم. به طوری که اگر اشاره گر در اول فراخوانی یک متد باشد و شما این دکمه را بزنید، دیگر وارد متد نمی‌شوید و اشکال زدا خودش دستورات آن را کامل اجرا کرده و مقدار بازگشتی را محاسبه می‌کند. سپس نتیجه آن را به شما نمایش می‌دهد. به عبارتی، شما نمی‌توانید مراحل اجرای دستورات داخل تابع را به درستی ببینید.

توجه: در صورتی که خط کد مورد نظر، فراخوانی متد نباشد (به طور مثال تعریف کردن یک متغیر)، دکمه step over یا هر دکمه دیگری که در ادامه با آن آشنا می‌شوید، این خط را اجرا می‌کند و به دستور بعدی می‌رود. این دکمه بیشترین کاربرد را در بین دیگر دکمه‌ها دارد. معمولاً، زمانی که اشاره گر روی یک دستور ساده است و یا روی فراخوانی متدی است که از درستی عملکرد آن اطمینان داریم، از این دکمه استفاده می‌کنیم.

Step into

دقیقاً شبیه به step over عمل می‌کند با این تفاوت که اگر در ابتدای یک فراخوانی متد باشیم، اشاره گر را به داخل متد می‌برد و شما می‌توانید بر مراحل اجرای یک متد کاملاً نظارت داشته باشید.

بدیهی است که این دکمه را در حالتی استفاده می‌کنیم که بخواهیم وارد متد شده و مراحل اجرای آن را دقیق مشاهده کنیم.

Force step into

این دکمه بسیار شبیه به step into عمل می‌کند با این تفاوت که اگر متد در یکی از کتابخانه‌های خود جاوا یا داخل پکیجی باشد که به طور پیش فرض آی‌دی‌ای به شما اجازه تغییر دادن کدهای آن را نمی‌دهد^{۲۵}، step into شما را به داخل آن نمی‌برد اما force step into باعث می‌شود که آی‌دی‌ای وارد آن کتابخانه‌ها بشود و شما بتوانید اجرای کدهای این بخش را ببینید.

توجه: بسیاری از کتابخانه‌هایی که در هسته جاوا قرار دارند، برای کاهش حجم و سریع‌تر کردن اجرای کد، فقط نسخه بایت کد^{۲۶} آن‌ها موجود است و jvm وظیفه وصل کردن آن‌ها به کدهای ما را بر عهده می‌گیرد که در این صورت force step into نمی‌تواند وارد تابع شود و از آن می‌گذرد.

این دکمه کاربرد کمتری از دو دکمه دیگر دارد و در موارد خاص که نیاز داریم یکی از توابع زبان یا کتابخانه را دیباگ کنیم از آن استفاده می‌کنیم.

^{۲۵} read-only mode

^{۲۶} bytecode



Step out

برخی اوقات هنگامی که داخل یک متد هستیم، به این نتیجه می‌رسیم که متد هیچ مشکلی ندارد و دیگر نیازی به ادامه اشکال‌زدایی آن نیست. این دکمه اجرای متد را تا آخر ادامه می‌دهد و سپس اشاره‌گر را به جایی که تابع از آنجا صدا زده شده می‌برد و اگر مقداری را بازگشت شده باشد، آن را نیز در متغیری ذخیره کرده و ادامه اجرای نرم افزار را جلو می‌برد.

بیشتر کاربرد این دکمه در حالتی است که یا به اشتباه وارد بدنه یک متد شده‌اید یا به صورت کلی از کارکرد درست آن مطمئن شده‌اید و قصد اتلاف وقت و جلو رفتن اجرا را تا اتمام اجرای متد ندارید.

Run to cursor

هنگامی که اشکال‌زدا در حال اجرا است، شما می‌توانید قسمت‌های دیگر کد را ببینید. در صورتی که بخواهید به جایی که اشاره‌گر قرار دارد منتقل شوید، این قسمت دکمه را می‌زنیم و جای دقیق اشاره‌گر منتقل می‌شویم.

به دو نکته زیر در مورد اشکال‌زدا دقت کنید:

(۱) در زبان‌های مبتنی بر کامپایلر^{۲۷}، در صورتی که اشکال‌زدا را اجرا کنید و کدهایی که نوشته‌اید را تغییر دهید، اشکال‌زدا آن تغییرات را نادیده گرفته و از آن‌ها رد می‌شود. زیرا زمانی که شما کد خود را در حالت دیباگ^{۲۸} اجرا می‌کنید، ابتدا باید این کد کامپایل شده و بعد اجرا شود. در نتیجه اگر کد خود را تغییر دهید، این تغییرات در بایت کد که در حافظه تحت کنترل IntelliJ است، منعکس نخواهد شد و اشکال‌زدا نمی‌تواند متوجه تغییرات شما شود. در این صورت باید دوباره نرم افزار را در حالت دیباگ اجرا کنید.

(۲) در اکثر اشکال‌زدهایی که تا به حال ساخته شده است، نمی‌توان به عقب برگشت و انجام یک دستور را ملغی کرد. برای بازگشت به عقب، نیاز دارید دوباره نرم افزار را در حالت دیباگ اجرا کنید.

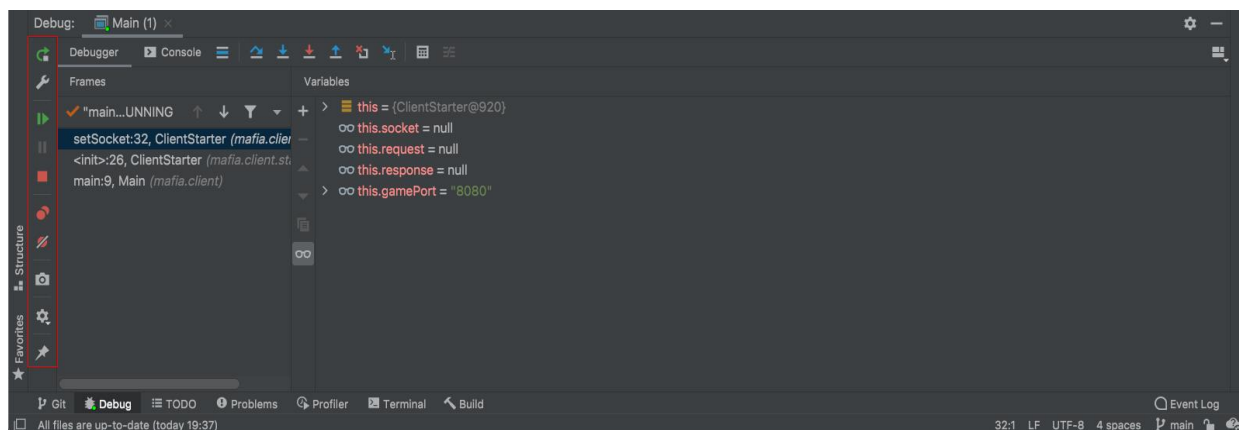
^{۲۷} compiler-based

^{۲۸} debug mode



معرفی status toolbar

بعد از آشنا شدن با execution toolbar، به قسمت status toolbar می‌رویم که در عکس زیر مشخص شده است. این قسمت مربوط به کنترل وضعیت اشکال‌زدا است که به معرفی دکمه‌های مختلف آن می‌پردازیم.



Rerun

این دکمه برای زمانی است که بخواهیم دوباره نرم افزار را از اول در حالت دیباگ اجرا کنیم. در واقع این دکمه میانبری است برای سریع‌تر کردن فرایند بستن اشکال‌زدا و دوباره اجرا کردن نرم افزار در حالت دیباگ.

Edit configuration

به صورت کلی IntelliJ و دیگر محصولات JetBrains، برای اجرای نرم افزار به صورت عادی و یا در حالت دیباگ، این اجازه را به ما می‌دهد که برخی از تنظیمات مربوط به اجرا کننده را بتوانیم کنترل کنیم.^{۲۹} با زدن این دکمه وارد run configuration های مربوط به اشکال‌زدا می‌شوید.

Resume program

با زدن این دکمه، اجرای نرم افزار به صورت معمول تا وقتی که به نقطه توقف بعدی برسیم یا در صورتی که نقطه توقف دیگری موجود نباشد، تا اتمام اجرای نرم افزار ادامه پیدا می‌کند.

^{۲۹} run configuration



Stop

با زدن این دکمه، اجرای نرم افزار در حالت دیباگ در همان نقطه متوقف می شود و اشکال زدا بسته خواهد شد.

Show breakpoint

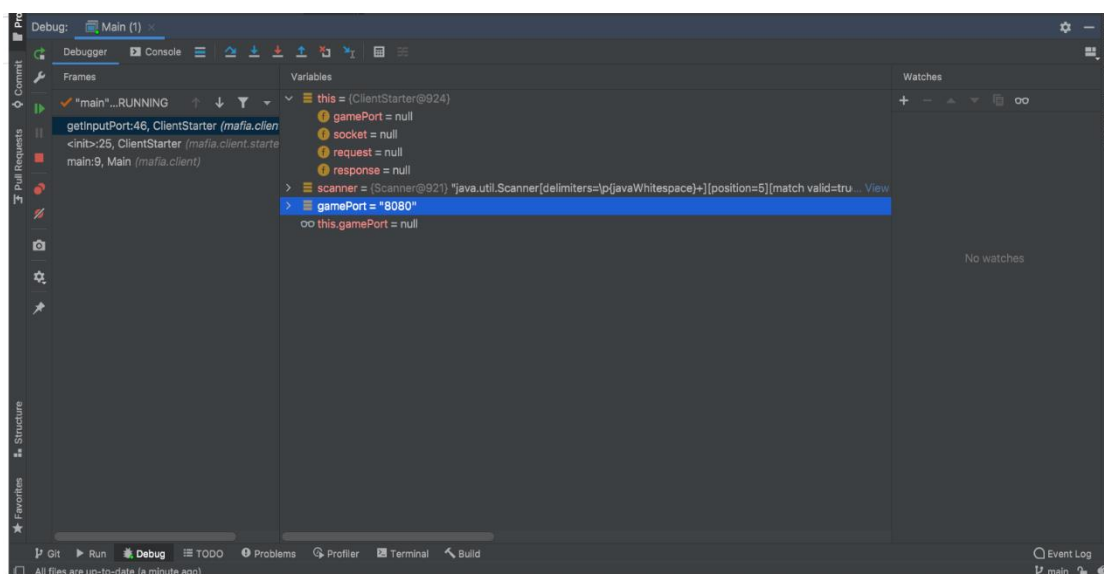
در IntelliJ و به صورت کلی تمام محصولات JetBrains، می توان لیستی از تمام نقطه توقف هایی که نرم افزار دارد تهیه کرد و تنظیمات آن ها را تغییر داد. اگر این دکمه را بزنید به آن لیست منتقل می شوید.

Mute breakpoints

هنگامی که قصد دارید که دیگر نقطه توقف ها عمل نکنند، از این دکمه استفاده کنید.

معرفی قسمت متغیرها^{۳۰}

یکی از مهم ترین وظیفه های اشکال زدا نشان دادن مقدار متغیرهای برنامه، به صورت زنده در هر خط و داخل هر تابع است. این قسمت مقدار متغیرها را که در اسکوپ^{۳۱} فعلی موجود هستند، نشان می دهد و در صورتی که متغیر یک شی باشد، می توانیم با کلیک بر روی آیکون سمت چپ مقدار فیلد^{۳۲} های آن شی را نیز ببینیم.



^{۳۰} variable

^{۳۱} scope

^{۳۲} field

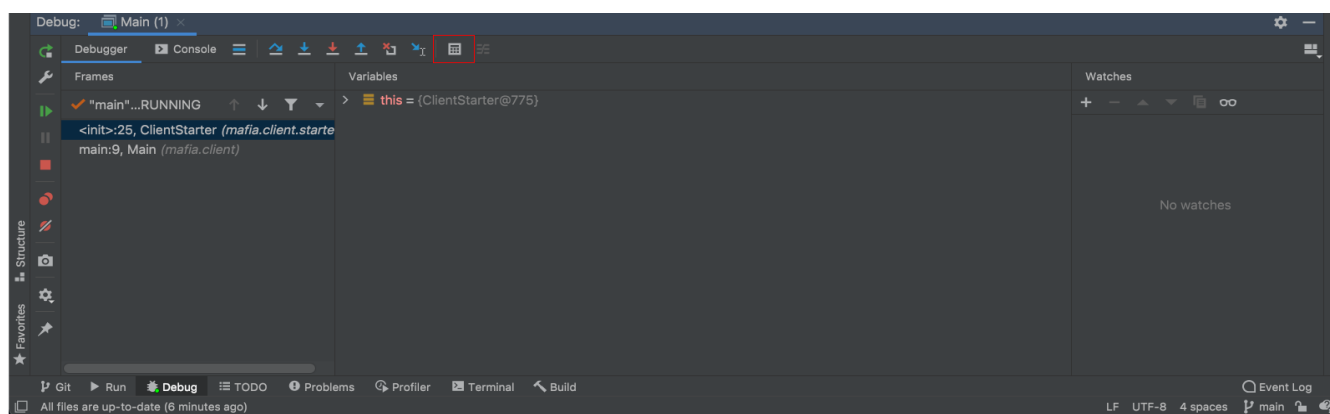


معرفی watchها

برخی اوقات، بررسی متغیرهایی که در یک اسکوپ موجود هستند، به دلیل زیاد شدن تعداد آنها، سخت‌تر می‌شود. در این حالت می‌توانیم به صورت drag and drop یا با کلیک راست کردن بر روی متغیر و انتخاب کردن گزینه add to watches آن را به قسمت watches اضافه کنیم و تغییرات آن را در هر مرحله راحت‌تر متوجه شویم.

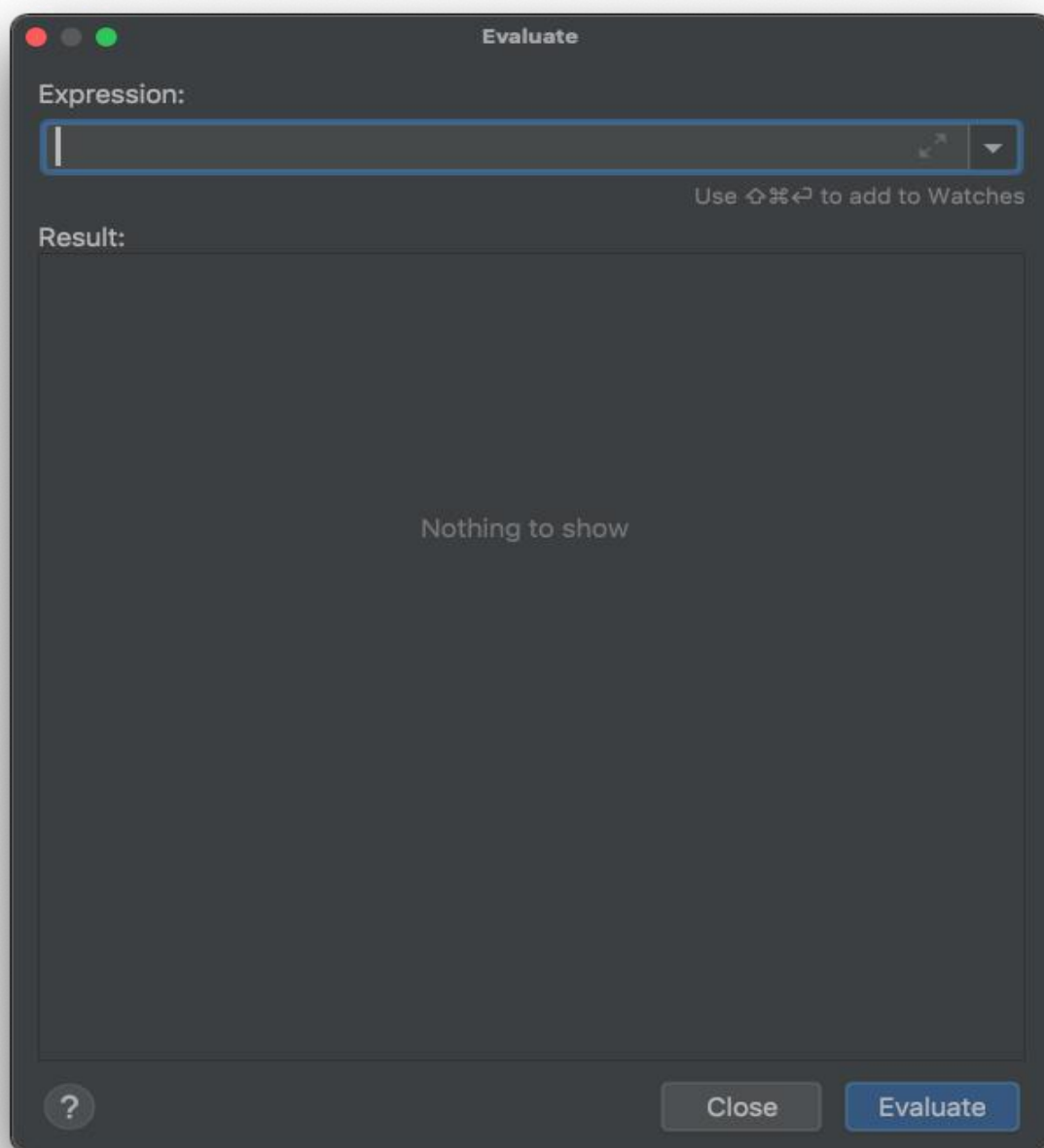
معرفی ویژه قسمت evaluate expression

یکی از قدرتمندترین قسمت‌های اشکال‌زدا شرکت JetBrains، قسمتی به نام evaluate expression است. دکمه‌ای که در عکس زیر مشخص شده است (در قسمت بالا و در انتهای execution toolbar) این ویژگی را فعال می‌کند.



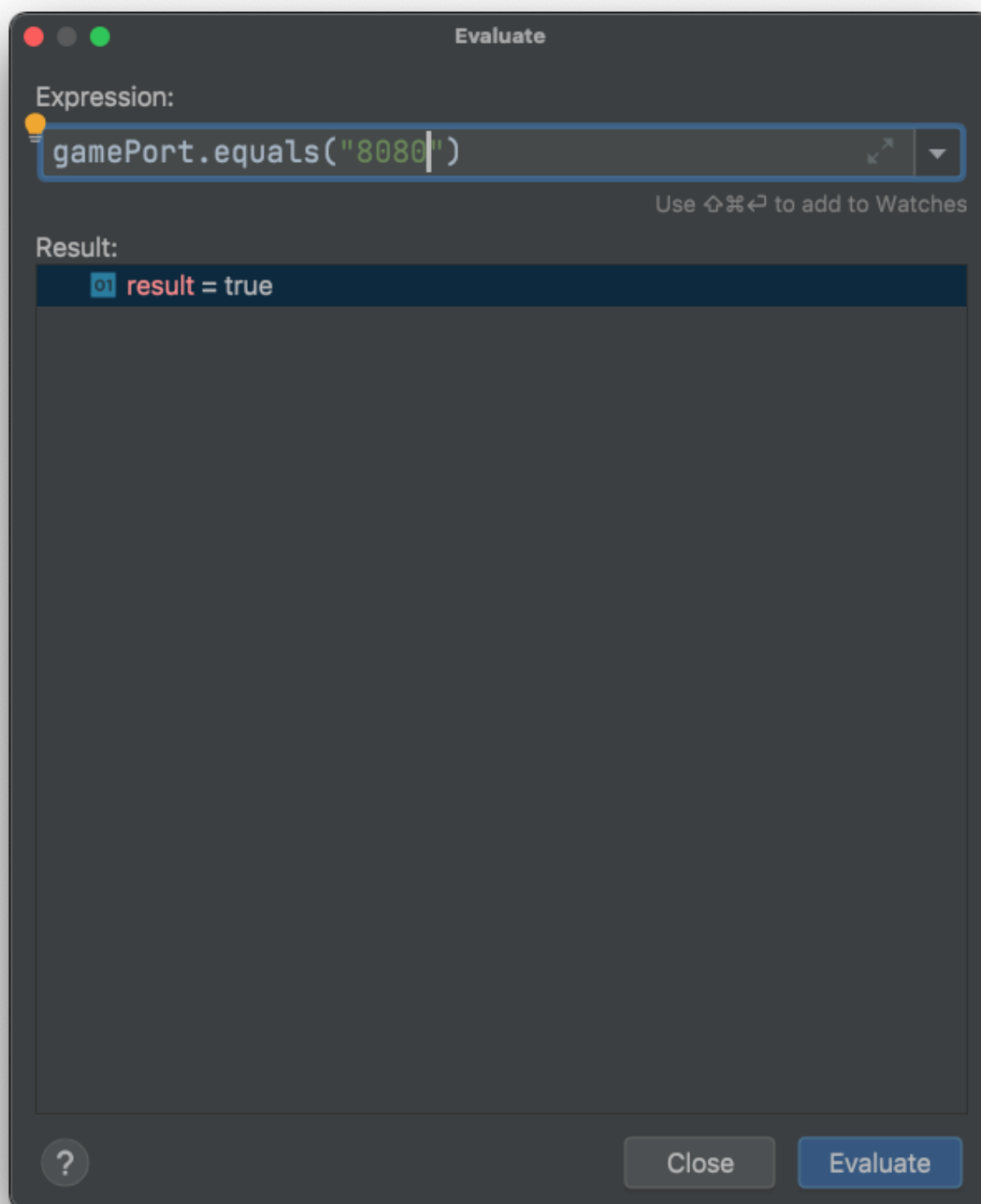


با کلیک کردن بر روی این دکمه، پنجره‌ای شبیه به عکس زیر باز می‌شود:





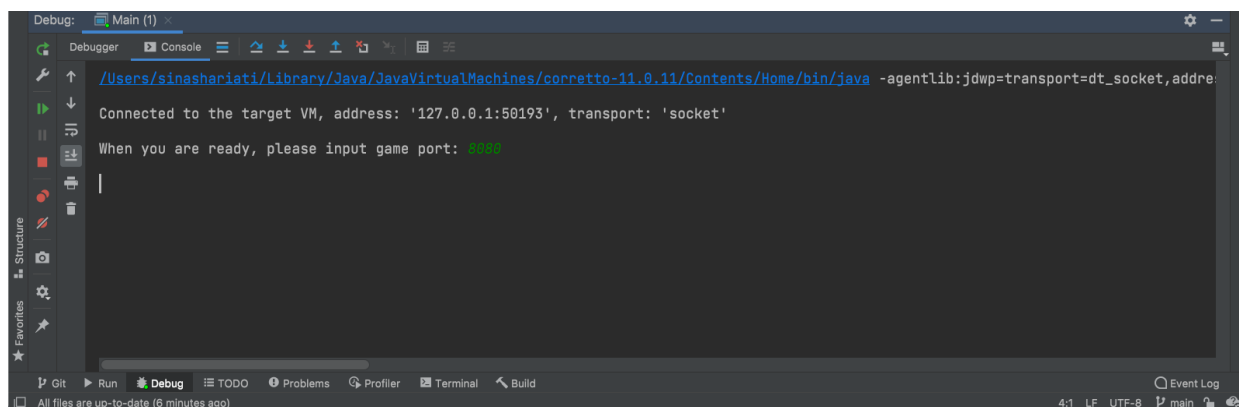
که در داخل قسمت expression می‌توانید هر کدی را که می‌خواهید، وارد کنید و جواب آن را دقیقاً مطابق با شرایط برنامه در خط فعلی دریافت کنید:





معرفی قسمت کنسول ۳۳

هر خروجی که به صورت `system.out.print` دارید، در کنسول چاپ می‌شود و در صورتی که با استفاده از `scanner` قصد داشته باشید که از کاربر ورودی بگیرید، آن را باید در این قسمت وارد کنید.



توجه داشته باشید که اشکال‌زدا ابزاری بسیار کاربردی است و ویژگی‌های بسیار زیادی دارد و تسلط بر آن می‌تواند بر روی کیفیت پروژه‌هایی که در آینده نزدیک با آن‌ها سروکار دارید بسیار موثر باشد؛ بنابراین برای آشنایی بیشتر شما، دو ویدئو از خود شرکت JetBrains قرار می‌دهیم:

[IntelliJ IDEA. Debugger Essentials \(۲۰۲۱\)](#)

[IntelliJ IDEA. Debugger Advanced \(۲۰۲۱\)](#)



تمرین استفاده از اشکال زدا

برای اینکه استفاده از اشکال زدا را بهتر یاد بگیرید و بتوانید آن را در پروژه‌های بسیار ساده‌تر از پروژه‌های واقعی امتحان کنید، سعی کرده‌ایم برای شما پروژه‌ای ساده طراحی کنیم که باگ‌های مناسبی داشته باشد تا بستر مناسبی برای تمرین شما فراهم کند.

آماده سازی پروژه

ابتدا قصد داریم پروژه را بر روی کامپیوتر خود دانلود کرده و آن را آماده کنیم. در مرحله اول به لینک مخزن در GitHub بروید و بعد پروژه را فورک^{۳۴} کنید؛ این عمل باعث می‌شود که دقیقاً یک کپی از این پروژه را در پروفایل خود داشته باشید. در مرحله بعد پروژه را کلون^{۳۵} کنید و سپس آن را با استفاده از IntelliJ باز کنید.

توضیح پروژه

یکی از سخت‌ترین مهارت‌هایی که در دیباگ کردن به آن نیاز دارید توانایی آشنا شدن با پایگاه کد^{۳۶} است؛ در غیر این صورت، روند دیباگ کردن برای شما بسیار سخت یا غیر ممکن خواهد شد. از طرفی آشنا شدن با پروژه‌های واقعی که بسیار بزرگ‌تر از نمونه بیان شده در این تمرین ساده هستند، سخت‌تر است و زمان و انرژی بیشتری از شما می‌گیرد و برای حل این مشکل باید با صبر بیشتری به بررسی دقیق کدها و محل دقیق ایجاد باگ‌ها بپردازید. در این پروژه قصد داریم یک نمونه بسیار ساده‌تر از کتابخانه [Math](#) که در java/core موجود است را بسازیم.

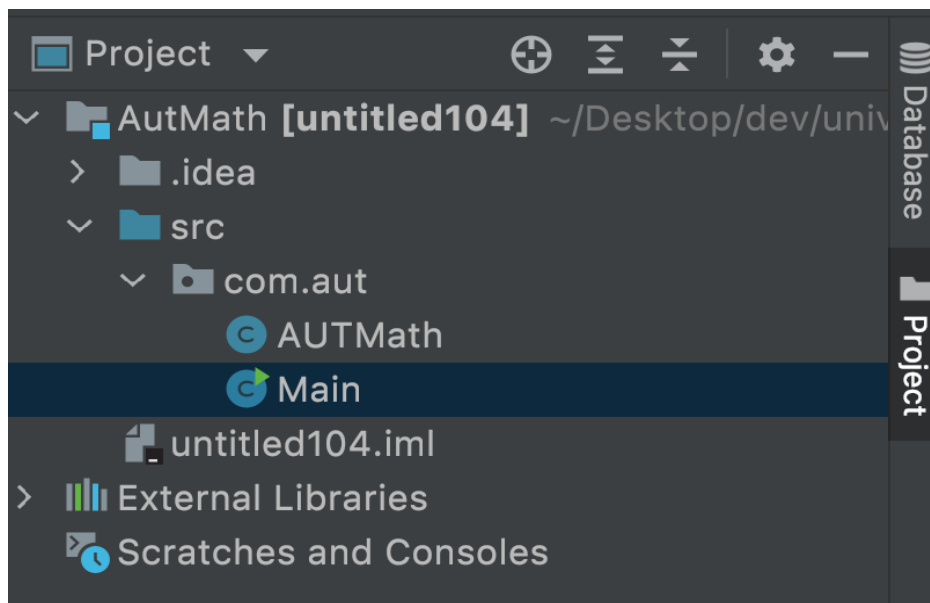
^{۳۴} fork

^{۳۵} clone

^{۳۶} codebase



ساختار پروژه به صورت زیر است:



در این پروژه، یک کلاس به نام **Main** (نقطه شروع برنامه) و یک کلاس دیگر به نام **AUTMath** موجود است. در کلاس **Main** دو تابع داریم با نام‌های **Main** و **assertResult** که وظیفه هر کدام را به اختصار توضیح می‌دهیم:

- **Main**: نقطه ورودی نرم افزار است که در بدنه خود توابع مختلف **AUTMath** را صدا می‌زند و نتیجه آن را به **assertResult** پاس می‌دهد تا آن را بررسی کند.

- **assertResult**: وظیفه این تابع این است که نتیجه هر تابع و مقداری که به صورت منطقی انتظار داریم، خروجی آن تابع به ما بدهد را گرفته و آن‌ها را مقایسه می‌کند.

اگر مقدار بدست آمده با مقدار مورد انتظار برابر نباشد یک ارور چاپ می‌کند و با استفاده از دستور

```
System.exit(1);
```

اجرای نرم افزار را متوقف می‌کند. اگر مقدار مورد انتظار با مقدار بدست آمده برابر باشد روند کار را ادامه می‌دهد.

در قسمت بعدی به توضیح **AUTMath** می‌پردازیم که به نوعی قسمت اصلی نرم افزار است:



این کلاس چند تابع مختلف برای شبیه سازی اعمال ریاضی مختلف دارد که بعضی از آن‌ها مشکلاتی دارند (با استفاده از ارورهایی که `assertResult` در کنسول چاپ می‌کند می‌توانید نقطه شروعی برای بررسی کدها پیدا کنید).

کلاس `AUTMath` شامل توابع زیر می‌باشد که برای هر کدام جاواداک مناسب نوشته شده است که با استفاده آن و اسم تابع، هدف هر تابع را می‌توانید درک کنید:

```
public static int sum(int num1, int num2)

public static int subtract(int num1, int num2)

public static int multiply(int num1, int num2)

public static int divide(int num1, int num2)

public static int factorial(int number)

public static int pow(int base, int power)
```

توجه داشته باشید که دیباگ کردن یک مهارت و در عین حال یک هنر است که هر برنامه‌نویس شیوه خاص خودش را برای انجام آن داشته و هر فرد نیاز به مدت زمان متفاوتی برای تسلط به این مهارت دارد.

به صورت کلی پیشنهاد می‌کنیم در صورتی که در پروژه‌هایی که با آن‌ها سروکار دارید، نمی‌توانید (حتی بعد از تخصیص زمان مناسب به آن) مشکل نرم افزار را حل کنید، کمی از کامپیوتر فاصله بگیرید و مدت زمانی را به کار دیگری غیر از برنامه‌نویسی و استراحت کردن بپردازید و بعد از پیدا کردن تمرکز دوباره شروع به دیباگ کردن کنید.