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[1] پروژه

[1] Debug

فهرست مطالب

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 شئ 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 معمولا 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 شئ 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 کاملا شئ‌گرا 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, float. هنگام تعریف یک متغیر از نوع اولیه شئ 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 می‌باشد که به ابتدای شئ 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 می‌کند و هر متغیر تنها شامل آدرس آن شئ می‌باشد. 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 ورودی‌هایی که به صورت شئ می‌باشند منجر به تغییر در شئ اصلی می‌شوند.

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 == همیشه کارساز خواهد بود.

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 شئ در حافظه هیپ است؛ بنابراین هنگام استفاده از == تنها درصورتی مقدار نهایی، true خواهد بود که هر دو متغیر به یک مکان در حافظه اشاره کنند و در واقع هر دو، یک شئ یکسان باشند. برای رفع این مشکل، جاوا استفاده از متد equals را پیشنهاد می‌کند. این متد کاری به آدرس اشیاء در حافظه ندارد و تنها با بررسی مقادیر فیلدهای دو شئ نتیجه برابر بودن آن‌ها را مشخص می‌کند.

[۷] primitive type & non-primitive type

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.

گاربیج کالکتور[۸]

همان‌طور که می‌دانید، هنگام ایجاد یک شئ، بخشی از حافظه هیپ به آن اختصاص داده می‌شود. اما اگر به صورت مدام 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 گاربیج کالکتور وظیفه حل این مشکل را دارد.

گاربیج کالکتور تعداد اشاره‌گرهایی را که به یک شئ اشاره می‌کنند می‌شمارد و در صورتی‌که اشاره‌گری به یک شئ وجود نداشته باشد جاوا متوجه می‌شود که دیگر نیازی به وجود آن شئ نیست و شئ از حافظه پاک خواهد شد.

وجود گاربیج کالکتور به این معنی نیست که می‌توانیم بدون نگرانی اقدام به تولید اشیاء جدید کنیم. توجه کنید که گاربیج کالکتور تنها اشیاء بدون اشاره‌گر را پاک می‌کند پس اگر تعداد زیادی شئ با اشاره‌گر بسازیم، مثلا یک آری لیست[۹] با تعداد اعضای بسیار بالا، همچنان مشکل کمبود حافظه وجود خواهد داشت.

همچنین در صورتی‌که سرعت ساختن اشیاء جدید توسط برنامه بیشتر از سرعت گاربیج کالکتور 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.

نکات: می‌توانید بین روش‌های پوینتر مدل[۱۱] وآدرس مدل[۱۲]، هرکدام را برای رسم انتخاب کنید. نیازی نیست که داده‌های ایستا را مشخص کنید اما باید اُورهِد[۱۳] هر شئ را مشخص کنید.

در انتها دیاگرام رسم شده را به استاد درس تحویل داده و عملکرد گاربیج کالکتور 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

```java
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())));
    }

}
```

```java
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] می‌باشند. درواقع می‌توان گفت wrapper classها روشی برای ذخیره سازی انواع داده اولیه به صورت شئ هستند.

استفاده از wrapper classها 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 classها 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 شئ را دارند و شما نمی‌توانید یک کالکشن با مقادیر اولیه داشته باشید. اما با استفاده wrapper classها امکان این کار فراهم شده است.

در جدول زیر می‌توانید لیست wrapper classها 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

اتو باکسینگ [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.

```java
public class Main {

    public static void main(String[] args) {
        Integer a = 10; // autoboxing
        System.out.println(a);

        // This warning means you can use autoboxing
        Character c = Character.valueOf('D');
        char primitiveChar = c; // unboxing
        System.out.println(c);
        System.out.println(primitiveChar);


        Double d = 2.15; // autoboxing
        // This warning means you can use unboxing
        double primitiveDouble = d.doubleValue();
        System.out.println(d);
        System.out.println(primitiveDouble);
    }

}
```

[۱۶] autoboxing
[۱۷] unboxing

آموزش کار با دیباگر

در ادامه این جلسه قصد داریم شما را با ابزار بسیار مهم به نام دیباگر 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 می‌دهد. 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 statementها برای فهمیدن وضعیت نرم افزار در هر نقطه است که می‌تواند 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 می‌آید.

برای حل این مشکل ابزاری به نام دیباگر در اکثر محیطهای توسعه یکپارچه معروف قرار داده شده است که به برنامه نویس کمک می‌کند 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.

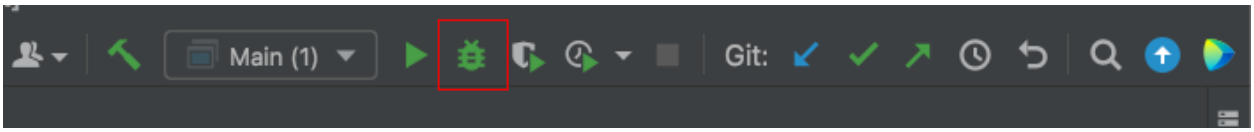
حال که با دیباگر به صورت محدود آشنا شدیم، بهتر است روش فعال کردن آن را در IntelliJ، که یکی از قوی‌ترین دیباگرها را در بین آی‌دی‌ای[۱۸] های جاوا را دارد، آشنا شویم.

برای اینکه دیباگر فعال شود باید یک بریک پوینت[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));
```

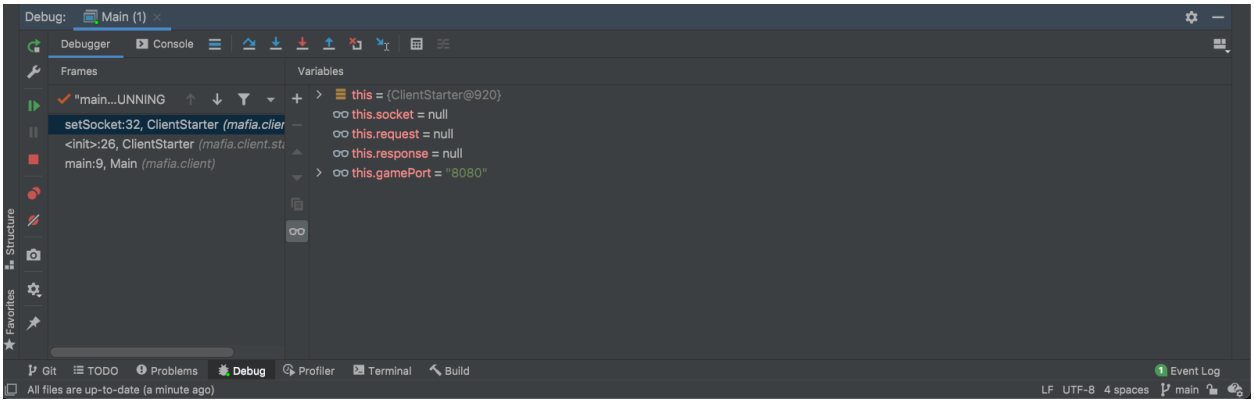
حال، برای اینکه دیباگر فعال شود باید نرم افزار را در حالت دیباگ اجرا کنیم برای این کار کافی است در سمت راست بالای آی دی ای بر روی دکمه مشخص شده در عکس زیر کلیک کنیم:

[۱۸] IDE
[۱۹] breakpoint



در این مرحله نرم‌افزار شروع به اجرا شدن می‌کند.

به محض اینکه اجرای کد به بریک پوینت که ما انتخاب کردیم برسد، منویی شبیه به منوی زیر برای ما نشان داده می‌شود که در واقع همان دیباگر است:



در این جلسه سعی می کنیم تا حد مناسبی با توانایی‌های مختلفی که این دیباگر دارد آشنا شویم.

توجه داشته باشید که اکثر ویژگی‌های که در ادامه مطرح می شوند معمولا در دیباگرهای دیگر نیز موجود هستند. اما امکان دارد شیوه دقیق نمایش آن‌ها یا روش کار آن‌ها مقداری با دیباگرهایی که در محصولات شرکت JetBrains قرار دارد متفاوت باشد.

معرفی انواع بریک پوینت‌ها

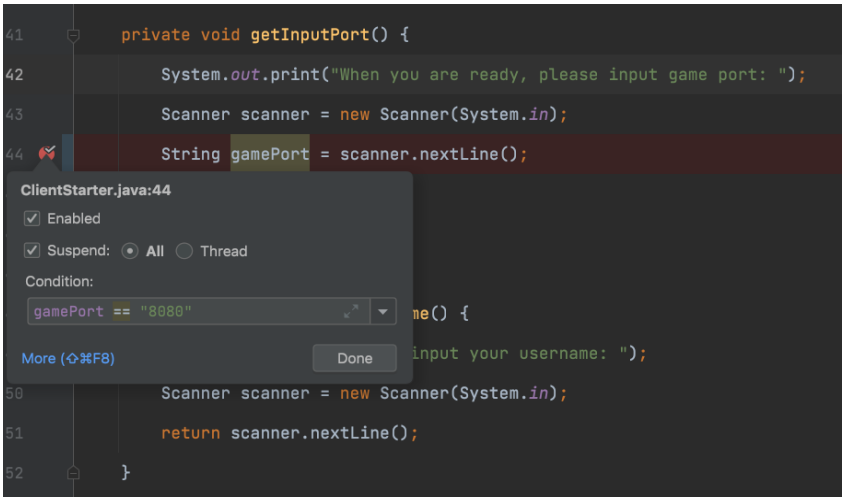
اکثر اوقات نیاز داریم به محض رسیدن اجرا کننده به بریک پوینت، ابزار دیباگر فعال شود؛ این نوع بریک پوینت به **بریک پوینت نرمال**[۲۰] معروف است. اما بعضی از اوقات نیاز داریم که بریک پوینت‌ها به صورت پیشرفته‌تری عمل کنند و در شرایط خاصی دیباگر را فعال کنند که گاها به این نوع از بریک پوینت‌ها بریک **پوینت‌های پیشرفته**[۲۱] نیز گفته می شود.

یکی از ویژگی‌های دیباگر IntelliJ، انواع مختلفی از بریک پوینت‌های پیشرفته آن است که با مهم‌ترین آن‌ها به طور مختصر آشنا می‌شویم.

بریک پوینت شرطی[۲۲]

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

نحوه استفاده: برای درست کردن این بریک پوینت، 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 دقیقا شبیه به بقیه شرط‌هایی است که در دستورات شرطی قبلا نوشته‌اید، بنویسید.

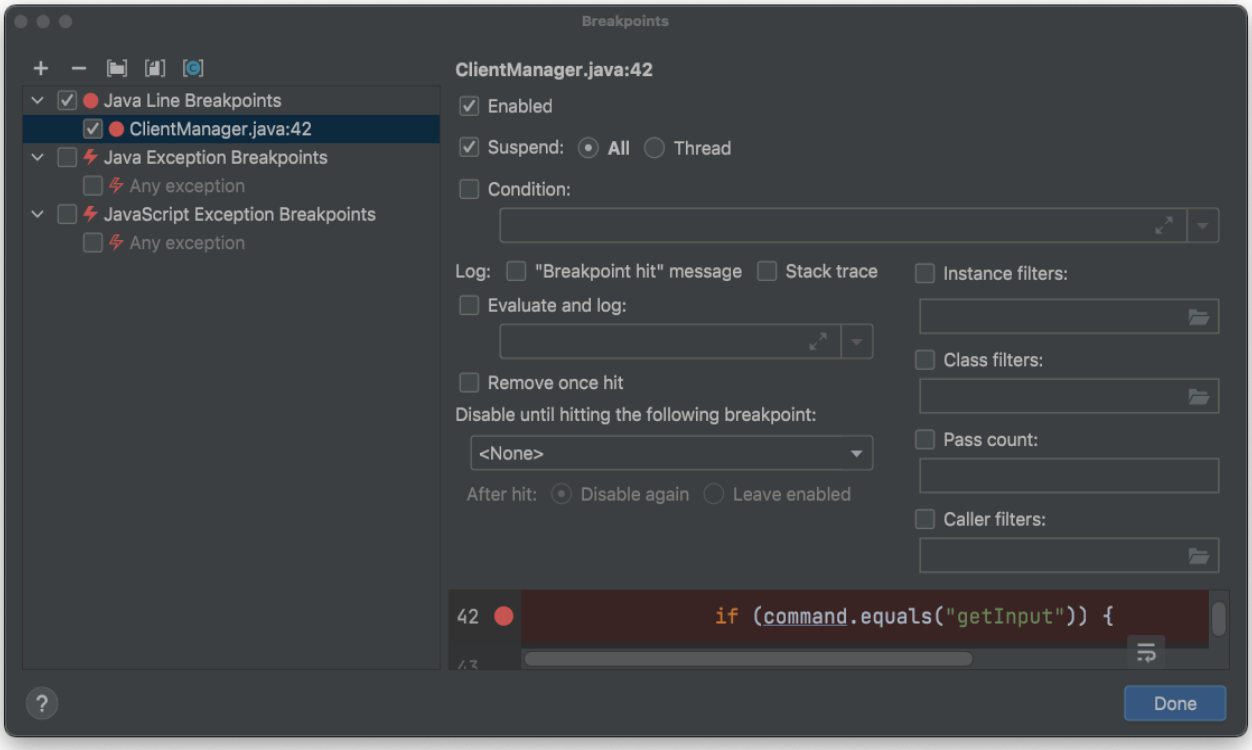


[۲۰] normal breakpoint
[۲۱] advanced breakpoint
[۲۲] conditional breakpoint

بریک پوینت شمارش گذر[۲۳]

فرض کنید که در یک حلقه هستیم و نیاز داریم که اگر اجرا کننده برای تعداد مشخصی به یک بریک پوینت رسید، دیباگر فعال شود. برای این منظور از بریک پوینت‌های 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 دیباگر فعال شود را می‌نویسیم.



توجه داشته باشید که دیباگری که در IntelliJ موجود است، انواع بریک پوینت‌های بیشتری دارد که می‌توانید برای آشنایی بیشتر با آن‌ها، به documentation رسمی debugger در وبسایت jetbrains مراجعه کنید.

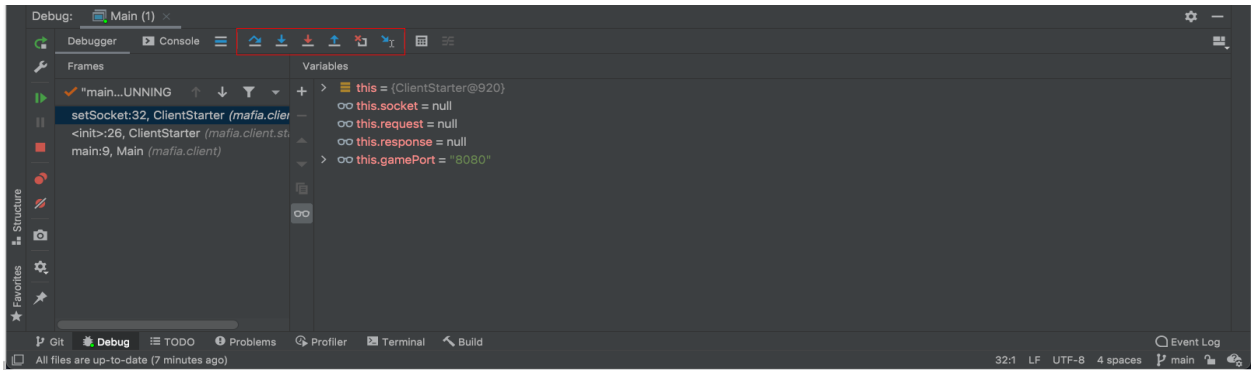
[۲۳] pass count breakpoint

معرفی منوی اصلی دیباگر

همانطور که ابتدای دستور کار اشاره شد، بعد از رسیدن به یک بریک پوینت، در واقع خود ابزار دیباگر 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 دیباگر کاربردهای زیادی داشته و اطلاعات زیادی را به ما در مورد نرم افزار نمایش می دهد، بدیهی است که ممکن است در لحظه اول سردرگم کننده باشد. در ادامه قصد داریم به توضیح بخش‌های مختلف آن بپردازیم تا پیچیدگی‌های آن ساده‌تر شود.

معرفی execution toolbar

ابزار دیباگر 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 درابتدای 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 اشاره‌گر بدون وارد شدن به متد دیگری یک دستور جلو برود، از این دکمه استفاده می‌کنیم. به طوری که اگر اشاره‌گر در اول فراخوانی یک متد باشد و شما این دکمه را بزنید، دیگر وارد متد نمی‌شوید و دیباگر خودش به داخل متد رفته، دستورات آن را کامل اجرا می‌کند و مقدار بازگشتی را محاسبه کرده و 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 باشد) دکمه step over یا هر دکمه دیگری که در ادامه با آن آشنا می‌شوید این خط را بدون هیچ مشکلی اجرا می‌کنند و به دستور بعدی می‌روند.

این دکمه بیشترین کاربرد را در بین دیگر دکمه‌ها دارد. معمولا، زمانی که اشاره‌گر 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

دقیقا شبیه به step over عمل می‌کند با این تفاوت که اگر در ابتدای یک فراخوانی متد باشیم، اشاره‌گر را به داخل متد می‌برد و شما می‌توانید بر مراحل اجرای یک متد کاملا 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 نمی برد 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 نمی تواند وارد تابع شود و از آن می‌گذرد.

[۲۵] read-only mode
[۲۶] bytecode

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.

Step out

برخی اوقات هنگامی که داخل یک متد هستم، به این نتیجه می‌رسیم که متد هیچ مشکلی ندارد و دیگر نیازی به ادامه اجرای 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 دیباگر 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 دیباگر دقت کنید:

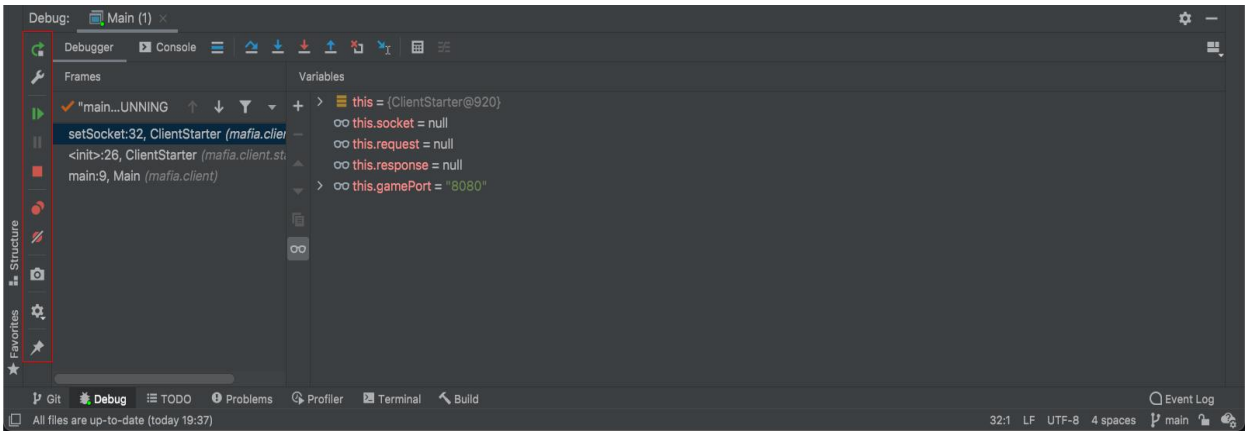
۱) در زبان‌های مبتنی بر کامپایلر[۲۷]، در صورتی که دیباگر را اجرا کنید و کدهایی که نوشته‌اید را تغییر دهید، دیباگر آن تغییرات را نادیده گرفته و از آن‌ها رد می‌شود. زیرا زمانی که شما کد خود را در حالت دیباگ[۲۸] اجرا می‌کنید ابتدا باید این کد کامپایل شده و بعد اجرا شود. در نتیجه اگر کد خود را تغییر دهید،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

۲) در تمام دیباگرهای مختلفی 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 دیباگر 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 دیباگر 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 configurationهای مربوط به دیباگر می‌شوید.

Resume program

با زدن این دکمه، اجرای نرم افزار به صورت معمول تا وقتی که به بریک پوینت بعدی برسیم یا در صورتی که بریک پوینت 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 می‌شود و دیباگر بسته خواهد شد.

Show breakpoint

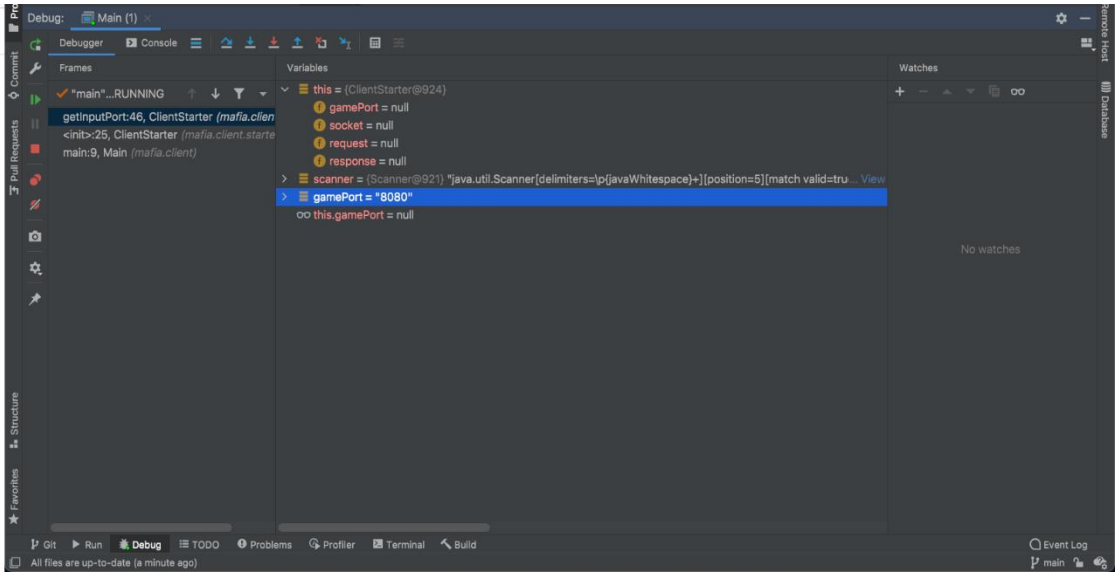
در IntelliJ و به صورت کلی تمام محصولات JetBrains، می‌توان لیستی از تمام بریک پوینت‌هایی که نرم‌افزار دارد تهیه کرد و تنظیمات آن‌ها را تغییر داد. اگر این دکمه را بزنید به آن لیست منتقل می‌شوید.

Mute breakpoints

هنگامی که قصد دارید که دیگر بریک پوینت‌ها عمل نکنند، از این دکمه استفاده کنید.

معرفی قسمت متغیرها [۳۰]

یکی از مهم‌ترین وظیفه‌های دیباگر 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 می‌دهد و در صورتی که متغیر یک شئ باشد، می‌توانیم با کلیک بر روی آیکون سمت چپ مقدار فیلد[۳۲] های آن شئ را نیز ببینیم.



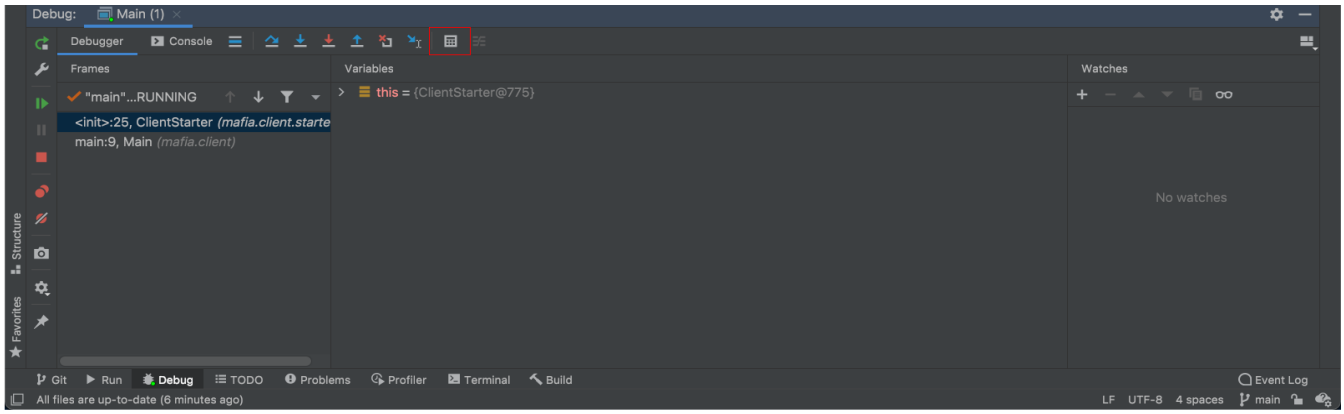
[۳۰] variable
[۳۱] scope
[۳۲] field

معرفی watcherها

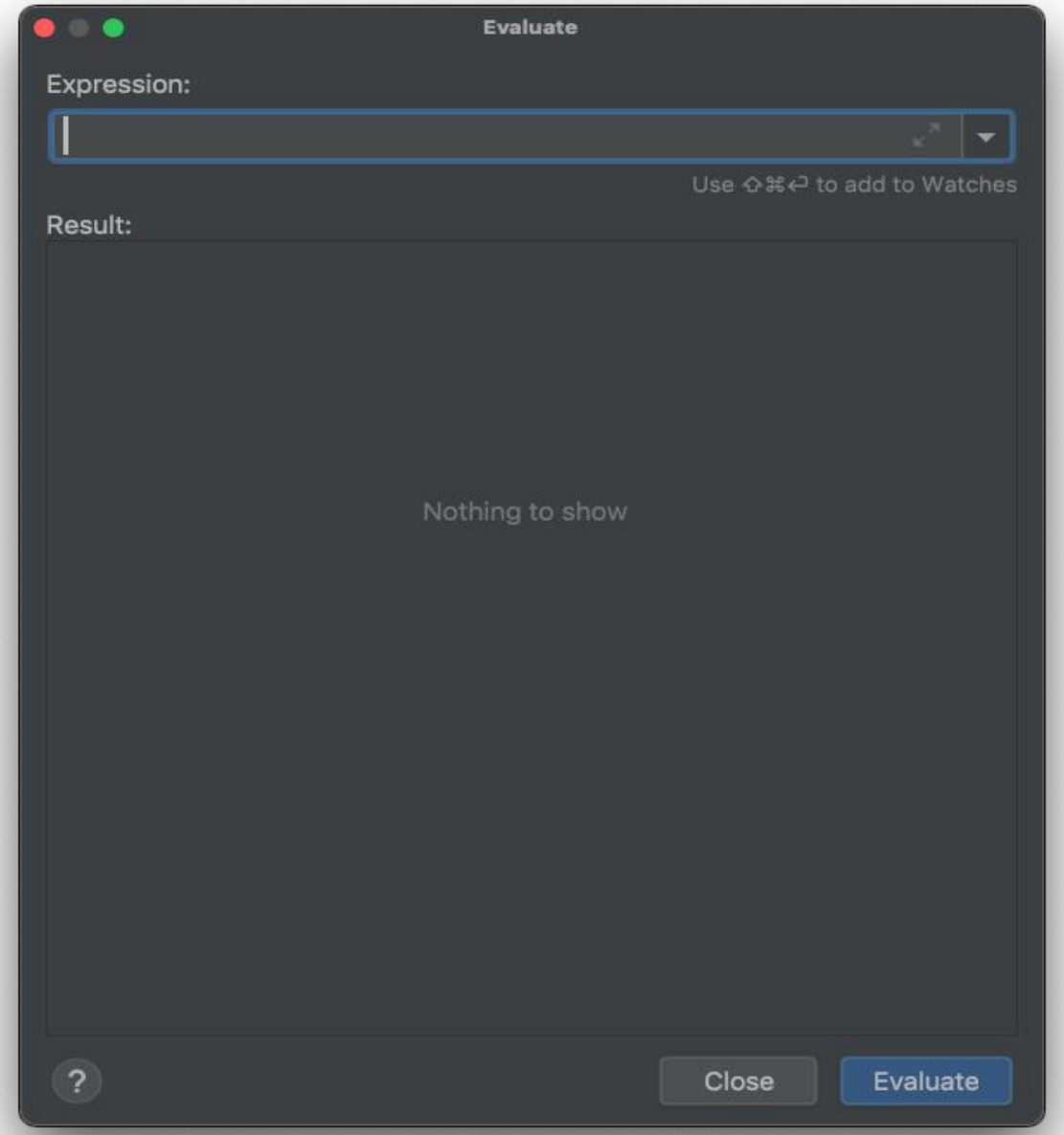
برخی اوقات، بررسی متغیرهایی که در یک اسکوپ موجود هستند، سخت‌تر می‌شود. در این حالت می‌توانیم به صورت drag and drop یا با کلیک راست کردن بر روی متغیر و انتخاب کردن گزینه add to watches آن را به قسمت watches اضافه کنیم و تغییرات آن را در هر مرحله راحت‌تر متوجه شویم.

معرفی ویژه قسمت evaluate expression

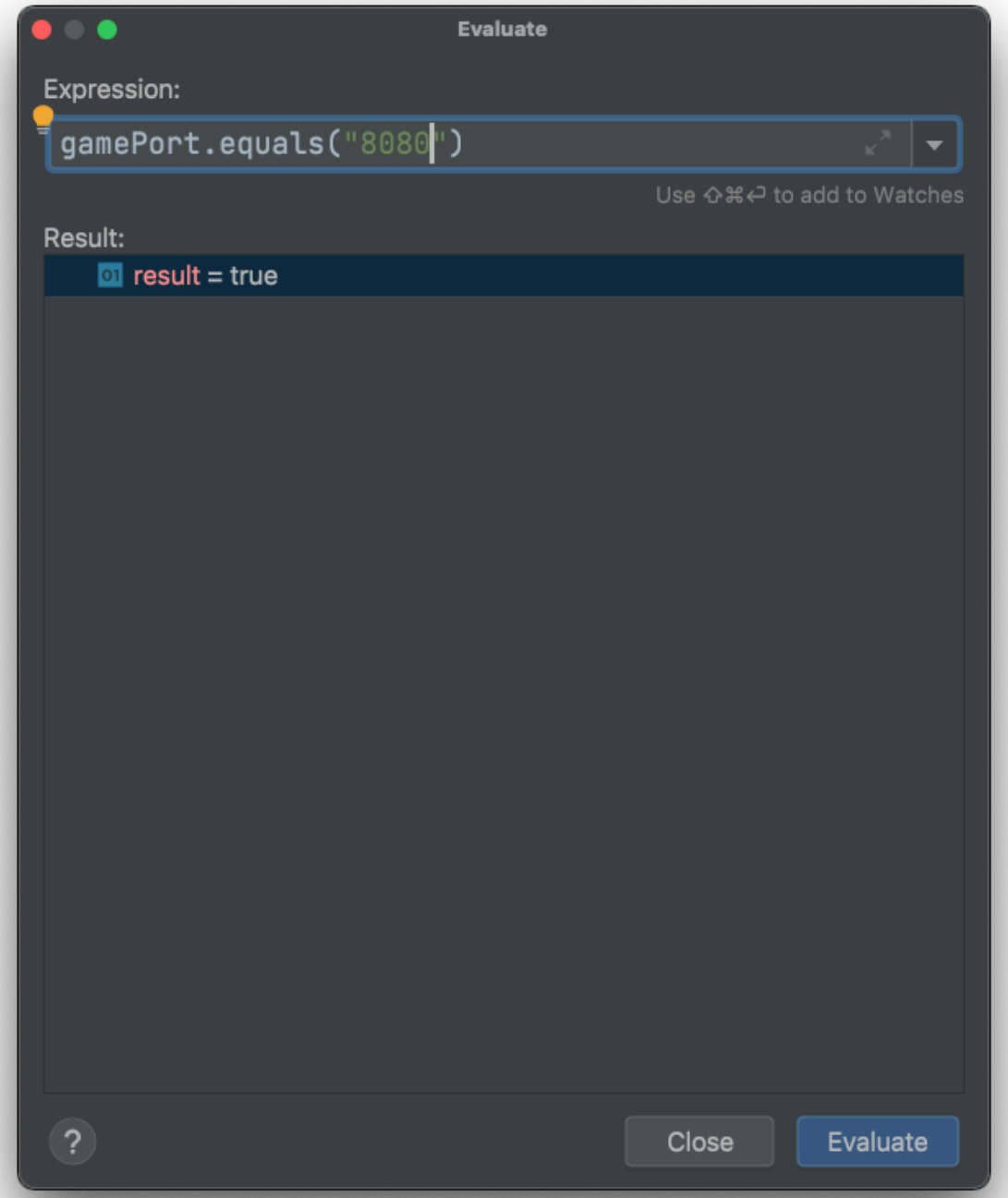
یکی از قدرتمندترین قسمت‌های دیباگر شرکت JetBrains، قسمتی به نام evaluate expression است. دکمه‌ای که در عکس زیر مشخص شده است (در قسمت بالا و در انتهای execution toolbar) این ویژگی را فعال می‌کند.



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

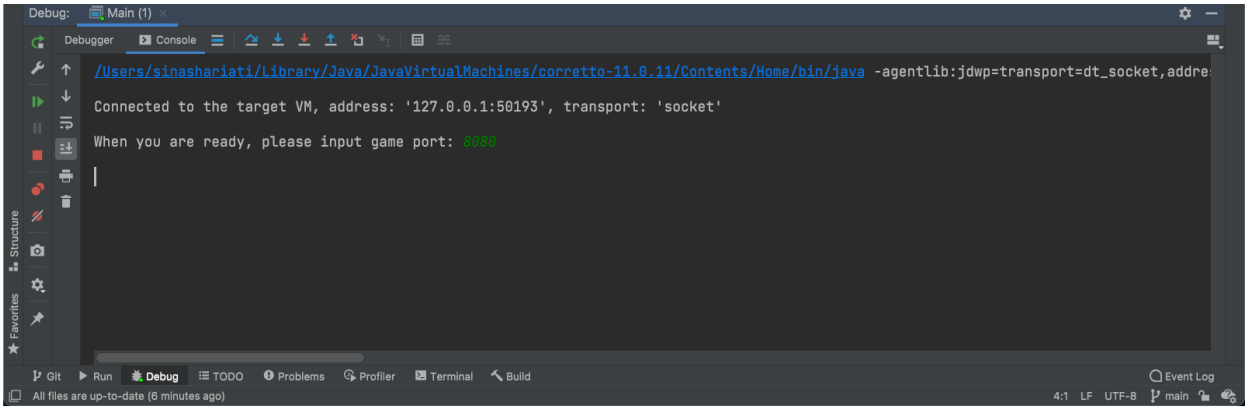


که در داخل قسمت expression می‌توانید هر کدی را که می‌خواهید، وارد کنید و جواب آن را دقیقا مطابق با شرایط برنامه در خط فعلی دریافت کنید:



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 دیباگر 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)

[۳۳] console

تمرین استفاده از دیباگر

برای اینکه استفاده از دیباگر را بهتر یاد بگیرید و بتوانید آن را در پروژه‌ای 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 دقیقا 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

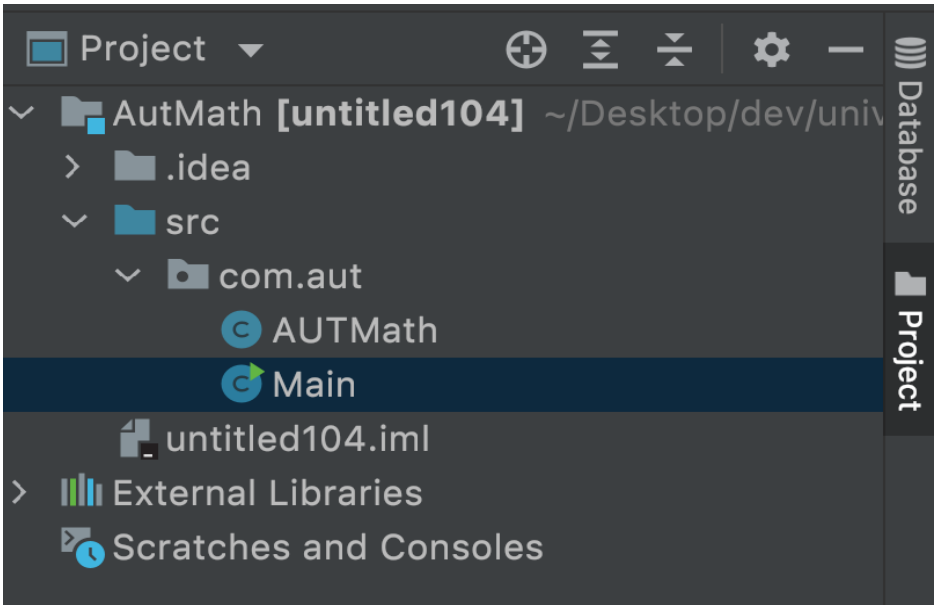
یکی از سخت‌ترین مهارت‌هایی که در دیباگ کردن به آن نیاز دارید توانایی آشنا شدن با پایگاه کد[۳۶] است؛ در غیر این صورت، روند دیباگ کردن برای شما بسیار سخت یا غیر ممکن خواهد شد.

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 می کند 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 اورهایی 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.