دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

دانشکده ریاضی و علوم کامپیوتر

استاد درس: دکتر مهدی قطعی
استاد کارگاه: بهنام یوسفی مهر

بهار ۱۴۰۳

کارگاه کار با کلاس

درس برنامه‌سازی پیشرفته و کارگاه

تا اینجای درس با مباحث مربوط به کلاس‌ها آشنا شدیم و اشیاء گوناگونی از این کلاس‌ها ایجاد کردیم. از ابتدای ترم با کلاس‌هایی مثل String کار و از متدهایشان استفاده کردیم. در ادامه کلاس‌های خودمان را نوشتیم و با اصولی از برنامه‌نویسی شئ‌گرا مثل کپسوله کردن [۱] آشنا شدیم. در این کارگاه با بحث اساسی برنامه‌نویسی شئ‌گرا به طور رسمی‌تر آشنا می‌شویم.

آشنایی با برنامه‌نویسی شئ‌گرا [۲]

برنامه‌نویسی شئ‌گرا یکی از مدل‌ها (یا پارادایم‌های) برنامه‌نویسی است. در درس‌هایی مثل مبانی برنامه‌نویسی اصول اولیه‌ی برنامه‌نویسی مثل توابع گفته می‌شود و شما هم احتمالاً برنامه‌هایی که در آن درس می‌نوشتید بدون ساختار پیچیده و صرفاً دنباله‌ای از دستورات بودند. مدل (یا پارادایمی) که در آن توابعی می‌نویسید و آنگاه آن توابع را فراخوانی می‌کنید تا به ترتیب گام‌های محاسباتی را جلو ببرند، برنامه‌نویسی پروسه‌ای (یا برنامه‌نویسی رویه‌ای) [۳] نامیده می‌شود.
برنامه‌ی شئ‌گرا برنامه‌ای است که از اشیاء تشکیل می‌شود. هر شئ عملکردی دارد که کاربر از آن آگاهی دارد و استفاده می‌کند و هچنین پشت پرده‌ای دارد که مخفی از کاربر است. تا زمانی که این اشیاء عملکردهای مورد نیازمان را در اختیارمان قرار دهند نیازی به دانستن پشت پرده و پیاده‌سازی آنها را نداریم. اما گاهی ممکن است عملکرد خاصی را نیاز داشته باشیم و اینجاست که باید خودمان دست به کار شویم.
همان طور که گفتیم در درسی مانند مبانی برنامه‌نویسی، هدف نوشتن الگوریتم و برنامه‌نویسی رویه‌ای بود. در این مدل برنامه‌نویس قبل از اینکه به ساختمان داده‌ها فکر کند به الگوریتم‌ها فکر می‌کند. در برنامه‌نویسی شئ‌گرا ابتدا به ساختمان داده‌ها و بعد درمورد الگوریتم‌ها و پروسه‌هایی که می‌خواهیم روی آنها اجرا کنیم فکر می‌کنیم. پس منطقی است زمانی سراغ برنامه‌نویسی شئ‌گرا برویم که با ساختارهای پیچیده‌تر سر و کار داریم و باید نظم مشخصی را در پیش بگیریم.
برای مثال یک بازی را در نظر بگیرید که همه‌ی توابع و متغیرهای آن داخل یک برنامه قرار دارد. اگر بخواهیم یکی از شخصیت‌های بازی را تغییر دهیم و یا اسلحه‌ای را به او اضافه کنیم باید در میان تعداد زیادی توابع و متغیر تغییرات خود را اعمال کنیم. اما اگر برنامه را به کلاس‌های گوناگون تقسیم کنیم، تنها کافی است کلاس مربوط به همان شخصیت را تغییر دهیم.

کلاس‌ها

همان طور که می‌دانید کلاس‌ها مشخص می‌کنند که اشیاء چگونه ساخته می‌شوند. به ساختار شئ‌گرای زبان جاوا توجه کنید: هرچه که تا به حال می‌نوشتید داخل یک کلاس بوده. کتابخانه‌ی استاندارد جاوا چندین هزار کلاس برای اهداف مختلفی مثل کار با زمان، تقویم‌ها و برنامه‌نویسی شبکه دارد. با این حال اگر بخواهید برنامه‌های خاص خودتان را بنویسید باید کلاس‌های مخصوص خودتان را تعریف کنید.

[۱] encapsulation
[۲] Object-Oriented Programming (OOP)
[۳] procedural programming

فرض کنید می‌خواهیم یک بازی طراحی کنیم. بازی ما شامل دشمنانی خواهد بود که کلاس مربوط به آن را در کد زیر تعریف می‌کنیم.

```
import java.util.ArrayList;

public class Enemy
{
    int health;
    ArrayList<Weapon> weapons;
}
```

در کارگاه کار با فایل با اصول اولیه‌ی کپسوله‌سازی آشنا شدید. کپسوله‌سازی ترکیب داده‌ها و رفتارها در یک پکیج و مخفی کردن پیاده‌سازی آن از کاربران آن شئ است. تکه داده‌هایی که در اشیاء هستند فیلدهای نمونه [۴] و رویه‌هایی که روی اشیاء عمل می‌کنند متد [۵] نامیده می‌شوند. هر شیئی که در واقع نمونه‌ای[۶] از یک کلاس است، مقادیر خاصی برای فیلدهای نمونه‌اش خواهد داشت. مجموعه‌ی این مقادیر وضعیت [۷] آن شئ نامیده می‌شود. واضح است که هر وقت متدی را روی شئ فراخوانی می‌کنیم ممکن است وضعیت آن تغییر کند.
کلید کپسوله‌سازی این است که متدها هیچ گاه به طور مستقیم فیلدهای نمونه‌ی داخل یک کلاس یا شئ بجز کلاس خودشان را دستکاری نکنند. برنامه‌ها باید فقط از طریق متدهای یک شئ با داده‌های آن شئ ارتباط داشته باشند و بتوانند روی آنها تغییراتی را اعمال کنند. با این کار کلاس مثل جعبه‌ی سیاهی خواهد شد که اثرات خارجی روی آن تأثیر نخواهند داشت و در صورت تغییرات داخلی خود کلاس، تا زمانی که همان متدها برای عوامل خارجی وجود داشته باشند و کار عوامل خارجی را راه بیندازند، پیاده‌سازی خود کلاس برای این عوامل بی‌اهمیت خواهد بود.

تمرین ۱. فیلدهای کلاس *Enemy* را با توجه به اصول کپسوله‌سازی که در کارگاه کار با فایل آموختید تغییر دهید و متدهای مورد نیاز را به آن اضافه کنید.

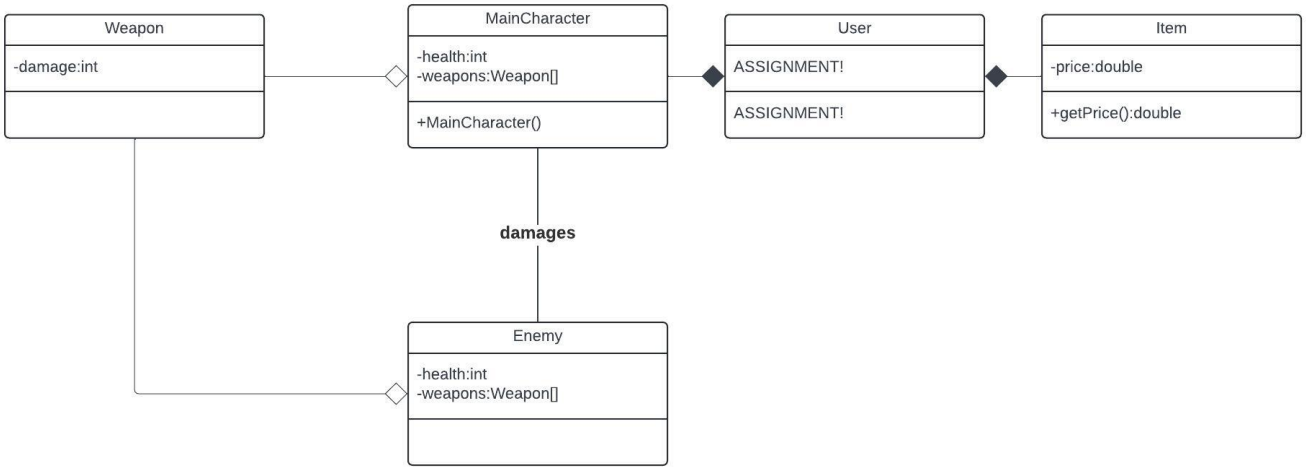
کلاس Weapon و مابقی کلاس‌ها را در قسمت بعدی تعریف می‌کنیم.

UML[۸] و روابط بین کلاس‌ها

UML روشی است که در آن یک ساختار شئ‌گرا و روابطشان را با دیاگرام‌ها نمایش می‌دهیم. در دیاگرام کلاس، یک کلاس با مستطیلی نشان داده می‌شود که نام کلاس در بالای آن قرار دارد. فیلدهای کلاس در خط‌های پایین آن نوشته شده و با یک خط از نام کلاس جدا

[۴] instance fields
[۵] method
[۶] instance
[۷] state
[۸] Unified Modeling Language

می‌شوند. علامت مثبت به معنی عمومی [۹] بودن فیلد و علامت منفی به معنای خصوصی [۱۰] بودن آن است. با این مفاهیم در ادامه‌ی کارگاه آشنا خواهید شد.
متدها به همراه پارامترها و نوع داده‌ی پارامترهایشان هم به طور مشابه در زیر فیلدها نوشته شده و با یک خط از فیلدها جدا می‌شوند.
برای مثال کلاس‌های بازی‌ای که می‌خواهیم طراحی کنیم و روابط بین آنها را به صورت زیر نمایش می‌دهیم.



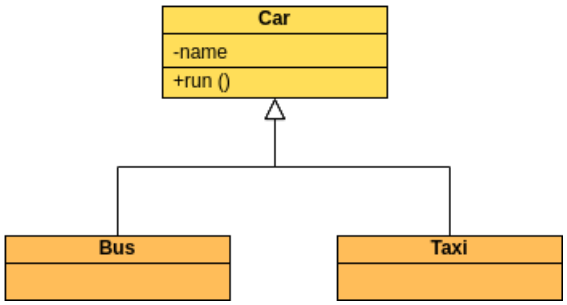
تمرین ۲. کلاس‌های بازی را با توجه با *UML* داده شده تشکیل دهید.

در ادامه با روابط بین کلاس‌ها آشنا می‌شویم.

وراثت (ارث‌بری)[۱۱]

با ارث‌بری در کارگاه‌های بعدی آشنا می‌شوید. در ارث‌بری یک کلاس همه‌ی متدها و فیلدهای کلاس دیگری را به ارث می‌برد و خودش هم می‌تواند متدها و فیلدهایی مضاعف داشته باشد. وراثت در UML به این صورت نشان داده می‌شود:

[۹] public
[۱۰] private
[۱۱] inheritence



در بالا تاکسی و اتوبوس هر دو ماشین هستند و در نتیجه ویژگی‌های ماشین را به ارث می‌برند.

همدستی[۱۲]

همدستی بیانگر نوعی ارتباط بین کلاس‌هاست. مثلاً در بازی ما شخصیت اصلی به دشمن صدمه می‌زند و از این طریق باعث کاهش جان او می‌شود و برعکس.

تراکم[۱۳]

این رابطه نوع خاصی از رابطه‌ی همدستی است که شامل یک کل و یک جزء است. در این رابطه یک کلاس کلی شامل چند نمونه از کلاس دیگر است که هر نمونه می‌تواند مستقل از کلاس کلی وجود داشته باشد؛ یعنی حذف کلاس کلی باعث حذف نمونه‌های آن کلاس نمی‌شود. به رابطه‌ی بین شخصیت اصلی و اسلحه توجه کنید. از بین رفتن شخصیت اصلی باعث از بین رفتن اسلحه‌های او نمی‌شود، بلکه شخصیت‌های دیگر بازی می‌توانند در ادامه این اسلحه‌ها را به دست آورده و استفاده کنند.

ترکیب[۱۴]

این رابطه مانند رابطه‌ی تراکم است، با این تفاوت که نمونه‌های کلاس نمی‌توانند مستقل از کلاس کلی وجود داشته باشند. پس حذف کلاس کلی باعث حذف نمونه‌های کلاس دیگر می‌شود. به رابطه‌ی بین کاربر و شخصیت اصلی یا اجناس[۱۵] توجه کنید. اگر کاربر حساب خود را حذف کند، اجناسی که خریداری کرده به همراه شخصیت اصلی بازی او از بین خواهند رفت.

[۱۲] association
[۱۳] aggregation
[۱۴] composition
[۱۵] items

اشیاء

هر شئ سه ویژگی مهم دارد:

۱. رفتار [۱۶] شئ: چه کارهایی می‌توانید با شئ بکنید یا در واقع چه متدهایی را می‌توانید روی آن فراخوانی کنید؟

۲. وضعیت شئ: چه اتفاقی برای شئ می‌افتد وقتی که این متدها را روی آن فراخوانی می‌کنید؟

۳. هویت [۱۷] شئ: شئ چگونه از بقیه‌ی اشیائی که همان رفتار و وضعیت را دارند متمایز می‌شود؟

رفتار یک شئ را متدهایی که روی آن می‌توانید فراخوانی کنید تعریف می‌کنند. به همین دلیل همه‌ی اشیائی که از یک کلاس ساخته می‌شوند رفتار یکسانی دارند؛ البته با فرض اینکه که کلاس دچار تغییرات نشود.
از طرفی اطلاعاتی که شئ در خود ذخیره می‌کند وضعیت آن را شکل می‌دهند. وضعیت یک شئ ممکن است در طول زمان تغییر کند، ولی این تغییرات باید صرفاً در نتیجه‌ی فراخوانی متدها روی آن باشد؛ در غیر این صورت اصول کپسوله‌سازی شکسته می‌شود.
اما وضعیت شئ به طور کامل آن را توصیف نمی‌کند. ممکن است دو شئ با اطلاعات یکسان ایجاد کنید، ولی این دو همچنان متفاوت از هم باشند. توجه کنید که ماهیت دو شئ که از یک کلاس ساخته شده‌اند همواره متفاوت است، ولی وضعیتشان لزوماً متفاوت نیست و می‌تواند یکی باشد.

استفاده از کلاس‌های از قبل تعریف شده

بدون کلاس‌ها نمی‌توانیم هیچ کاری را در جاوا پیش ببریم. تا به حال از کلاس‌های مختلفی استفاده کرده‌اید و کلاس‌های خودتان را تعریف کرده‌اید. وقتی از بعضی کلاس‌ها استفاده می‌کنیم نیازی به داده‌ها و متغیرها نداریم و فقط از متدهایی که کلاس در اختیار ما قرار می‌دهد بهره می‌بریم.

تمرین ۳. با استفاده از کلاس *Math* بازی‌ای را در کلاسی به نام *GuessMyNumber* بسازید که در آن کاربر باید عددی که به صورت رندوم توسط برنامه در بازه‌ی $(0, 500)$ انتخاب شده را حدس بزند. در هر گام بازی کاربر عددی را از این بازه حدس می‌زند. اگر عدد حدس زده شده همان عدد انتخابی باشد، کاربر برنده می‌شود؛ اما اگر عدد انتخابی از حدس کاربر بیشتر باشد، به کاربر پیامی مبنی بر کم بودن عدد حدس زده شده و در غیر این صورت بیشتر بودن آن نشان داده می‌شود. همچنین در صورتی که تعداد حدس‌ها از ده تا بیشتر شود کاربر می‌بازد.

[۱۶] behavior
[۱۷] identity

یکی از کلاس‌های کاربردی در کتابخانه‌ی استاندارد جاوا Date است. اشیاء این کلاس نمایانگر زمان هستند. به نظر شما چرا برای مفهوم مهمی مثل زمان داده‌ی اولیه[۱۸] مثل int در نظر گرفته نشده؟
برای اینکه بخواهیم شیئی از کلاس Date بسازیم، کانستراکتور آن را فراخوانی می‌کنیم. نام کانستراکتورها همیشه همان نام کلاس است. مثلاً کانستراکتور کلاس Date همان Date است. پس برای ساختن یک شئ Date، کانستراکتور آن را با عملگر new صدا می‌زنیم. خروجی این عملگر یک رفرنس[۱۹] به آن شئ است. یعنی جاوا شئ را در مکانی از حافظه ذخیره می‌کند، ولی مقدار نشانگری که به آن قسمت از حافظه اشاره می‌کند (در زبان جاوا می‌گوییم ارجاع می‌دهد) خروجی داده می‌شود و نه خود شئ.

```
System.out.println(new Date())
```

خط بالا شئ جدیدی می‌سازد که زمان ذخیره شده در آن همان زمانی است که شئ ساخته می‌شود و اطلاعات آن چاپ می‌شود. از طرفی می‌توانید متد toString() را روی آن فراخوانی کنید.

```
String s = new Date().toString();
System.out.println(s);
```

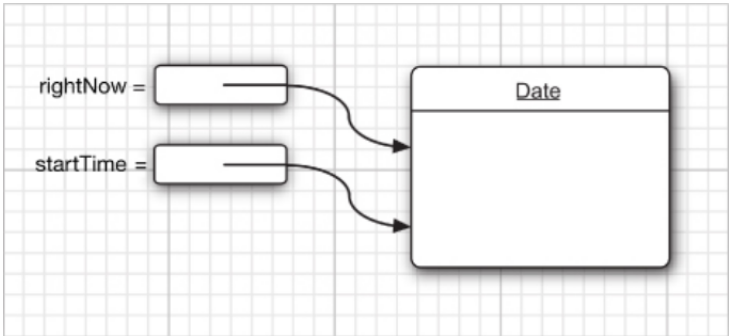
حال اگر در آینده هم نیاز به این شئ داشته باشیم کافی است آن را در یک متغیر به صورت زیر ذخیره کنیم.

```
Date rightNow = new Date();
```

البته توجه کنید که متغیر rightNow خودش یک شئ نیست، بلکه به یک شئ ارجاع می‌دهد. به همین دلیل است که اگر متغیر جدیدی را به صورت زیر تعریف کنیم، آنگاه هر دو متغیر به یک شئ ارجاع خواهند داد.

```
Date startTime; // startTime doesn't refer to any object yet!
startTime = rightNow;
```

این موضوع در شکل زیر به خوبی نشان داده شده است.



[۱۸] primitive data
[۱۹] reference

یکی از کلاس‌های بهتری که برای کار کردن با تاریخ و زمان وجود دارد، LocalDate است. ابتدا محتویات پکیج java.time را ایمپورت کنید.

```
import java.time.*;
```

حال می‌توانید به جای فراخوانی مستقیم کانستراکتور LocalDate از متد now استفاده کنید.

```
System.out.println(LocalDate.now());
```

همچنین می‌توانید شیئی برای یک تاریخ دلخواه ایجاد کنید. در اینجا شیئی که نماینده‌ی بیست و پنجم مارس سال ۲۰۲۴ است ایجاد می‌کنیم.

```
LocalDate labDate = LocalDate.of(2024, 4, 10);
```

از روی این شئ می‌توانیم سال، ماه و روز را به صورت جداگانه به دست آوریم.

```
int year = labDate.getYear();
int month = labDate.getMonthValue();
int day = labDate.getDayOfMonth();
```

اگر بخواهیم تاریخ هزار روز بعد از کارگاه را به دست آوریم به صورت زیر عمل می‌کنیم.

```
LocalDate aThousandDaysLater = labDate.plusDays(1000);
```

به متدهایی که فراخوانی آنها روی متغیر باعث تغییر متغیر می‌شود جهش‌دهنده [۲۰] و به متدهایی که فقط به اشیاء دسترسی می‌گیرند بدون آنکه باعث تغییرشان شوند دسترسی‌گیرنده [۲۱] می‌گویند.

تمرین ۴. با ذکر دلیل بیان کنید که متد *LocalDate.plusDays* جهش‌دهنده است یا دسترسی‌گیرنده.

تعریف کردن کلاس‌های خودمان

در ادامه مروری بر تعریف کلاس‌ها خواهیم داشت. تا به حال کلاس‌های زیادی را تعریف کردیم، اما بیایید به طور دقیق‌تر آنچه که انجام می‌دادیم را بررسی کنیم. به عنوان مثال کلاس User زیر را که نماینده‌ی کاربران بازی ما خواهند بود در نظر بگیرید.

```
import java.time.LocalDate;
import java.util.ArrayList;

```

[۲۰] mutator
[۲۱] accessor

```java
public class User
{
    private String name;
    private MainCharacter mainCharacter;
    private LocalDate accountCreationDate;
    private LocalDate lastLogin;
    private double fund;
    private ArrayList<Item> items;

    public static void main(String args[])
    {
        User user = new User("Mahsa", new MainCharacter(), 16, 9, 2022);
        user.setLoginMonth(11);
        user.printInformation();
    }

    public User(String name, MainCharacter mainCharacter, int accountCreationDay, int accountCreationMonth, int accountCreationYear)
    {
        lastLogin = accountCreationDate;
    }

    public void setLoginMonth(int month)
    {
        lastLogin = LocalDate.of(lastLogin.getYear(), month, lastLogin.getDayOfMonth());
    }

    public void increaseFund(double value)
    {
        fund += value;
    }
```

```
    public void buyItem(Item item)
    {
        items.add(item);
        fund -= item.getPrice();
    }

    public void printInformation()
    {
        System.out.println("Name: " + name);
        System.out.println("Account created at " + accountCreationDate);
        System.out.println("Last login at " + lastLogin);
    }
}
```

تمرین ۵. *UML* بازی را با توجه به ساختار کلاس *User* که در بالا آمده تکمیل کنید.

تمرین ۶. متدی برای کنترل مقدار دارایی حساب کاربر ایجاد کنید. در صورتی که قرار باشد مقداری بیشتر از دارایی کاربر کم شود باید خطایی مبنی بر کمبود موجودی چاپ شود. متد *buyItem* را با استفاده از متدی که تعریف کرده‌اید بازنویسی کنید.

در اینجا فیلدهای کلاس را تعریف کرده‌ایم و برای اینکه به اصول کپسوله‌سازی پایبند باشیم آنها را خصوصی قرار داده‌ایم. حال نیاز به تعریف کانستراکتور خواهیم داشت. کانستراکتورها زمانی اجرا می‌شوند که اشیائی از کلاس ایجاد می‌کنیم و وضعیت اولیه‌ی اشیاء را تعیین می‌کنند. به تعریف کانستراکتور User توجه کنید. همان طور که می‌بینید کانستراکتور هیچ چیزی برنمی‌گرداند و void هم نیست! همچنین باید دقت کنید که کانستراکتور را نمی‌توانیم خودمان و به صورت دستی فراخوانی کنیم تا دوباره وضعیت یک شئ را به وضعیت اولیه برگرداند. کانستراکتورها در زبان جاوا فقط با عملگر new می‌توانند فراخوانی شوند.

تمرین ۷. کانستراکتور کلاس *User* را تکمیل کنید. در طراحی خود فرض کرده‌ایم اولین لاگین همان زمان ساخت اکانت است. توجه کنید که باید همه‌ی فیلدهای کلاس در کانستراکتور مقدار اولیه‌ای بگیرند. همچنین در هنگام ایجاد هر کابر باید پیامی مثل پیام زیر چاپ شود:

Account created!

Last login at 2023-03-12

توجه کنید که چون متد main را برای کلاس User تعریف کرده‌ایم می‌توانیم آن را مستقیماً با دستور زیر اجرا کنیم.

```
javac User.java
java User
```

تمرین ۸. همه‌ی تاریخ‌های کلاس *User* را از *LocalDate* به *Date* تغییر دهید و متد *setLastLoginMonth* را نیز با استفاده از متد *Date.setMonth* بازنویسی کنید. همچنین در کانستراکتور *LocalDate* را به *Date* تبدیل کنید. حال *User* را دوباره اجرا کنید. چه مشکلی مشاهده می‌کنید؟ علت چیست؟

در کلاس Game چند کاربر به صورت زیر تعریف می‌کنیم.

```
public class Game
{
    public static void main(String[] args)
    {
        User[] users = new User[3];

        users[0] = new User("Ryan", new MainCharacter(), 10, 1, 2024);
        users[1] = new User("Melika", new MainCharacter(), 19, 4, 2022);
        users[2] = new User("Parsa", new MainCharacter(), 12, 3, 2023);

        for (User user : users)
            user.printInformation();
    }
}
```

حال اگر این فایل را کامپایل کنید، کامپایلر به ازای هر کلاس یک فایل با پسوند class. ایجاد می‌کند.

```
javac Game.java
```

در اینجا ما فقط javac را روی Game.java صدا زده‌ایم. در واقع وقتی کامپایلر کلاس‌های دیگر را می‌بیند، به دنبال کلاس آنها (منظور همان فایل‌ها با پسوند class. است) می‌گردد و اگر هرکدام را پیدا نکند، کد جاوایشان را جست‌وجو کرده و کامپایل می‌کند.
بعد از کامپایل کردن با دستور زیر برنامه را اجرا کنید. برای اجرای برنامه باید اسم کلاسی را که متد main در آن است به مفسر بایت‌کد [۲۲] بدهید.

```
java Game
```

آنگاه مفسر شروع به اجرای کد داخل متد main در داخل Game می‌کند. این کد سه شئ User جدید درست کرده و به شما وضعیت هر کدام را نشان می‌دهد.

[۲۲] bytecode interpreter

سطوح دسترسی

اگر به کلاس Game به دقت نگاه کنید متوجه کلمه‌ی public در کنار اسم کلاس می‌شوید که یعنی کلاس عمومی است. هر فایل جاوا فقط می‌تواند یک کلاس عمومی داشته باشد، اما هر چند تا کلاس غیر عمومی مجاز است. سطوح دسترسی مثل خصوصی و عمومی بودن برای متدها و فیلدهای نمونه هم تعریف می‌شود که در ادامه به آنها می‌پردازیم.

سطح دسترسی متدها و فیلدهای نمونه

به متدهایی که تا به حال در هرکدام از کلاس‌ها تعریف کردید توجه کنید. در تعریف همگی اینها کلمه‌ی public به چشم می‌خورد. کلیدواژه‌ی public یعنی هر متدی در هر کلاسی می‌تواند متد را فراخوانی کند. گاهی اوقات که نیاز به متدهای کمکی برای محاسباتی داریم که یک متد عمومی به آن نیاز دارد، می‌توانیم متدهای خصوصی تعریف کنیم. این متدها به درد کاربران خارجی نمی‌خورند و در نتیجه نیازی نیست که عمومی تعریف شوند. متدهای خصوصی را می‌توانیم در صورتی که دیگر نیازی به آنها نبود حذف کنیم و به دنبال جایگزین برایشان باشیم، ولی از آنجا که کاربران خارجی از متدهای عمومی استفاده می‌کنند نمی‌توانیم به راحتی آنها را حذف کنیم.
درمورد فیلدهای نمونه هم به طور مشابه همین موضوع برقرار است. اگر خصوصی تعریف شوند متدها و کلاس‌های بیرونی به آنها اجازه‌ی دسترسی و تغییر نخواهند داشت و اگر عمومی تعریف شوند، کلاس‌ها و متدهای بیرونی می‌توانند به آنها دسترسی داشته و آنها را تغییر دهند.

سطح دسترسی مبتنی بر کلاس

می‌دانید که یک متد می‌تواند به داده‌های خصوص شیئی که روی آن فراخوانی می‌شود دسترسی داشته باشد، ولی آیا می‌دانستید که یک متد می‌تواند به داده‌های خصوصی همه‌ی اشیاء کلاسش دسترسی بگیرد؟!
به عنوان مثال متد زیر را در نظر بگیرید که دو کاربر را با هم مقایسه می‌کند.

```
1    public boolean equals(User other)
2    {
3        return name.equals(other.name);
4    }
```

حال اگر متد را روی کاربری مثل user1 فراخوانی کنیم و user2 را به صورت پارامتر به آن بدهیم، آنگاه متد به اسم user2 که یک داده‌ی خصوصی است هم دسترسی خواهد داشت.

کلیدواژه‌ی final

final کلیدواژه‌ای است که می‌تواند در تعریف فیلدهای نمونه به کار برود. مثلاً می‌توانیم نام کاربر را final تعریف کنیم.

```
1    private final String name;
```

چنین فیلدهایی باید در زمان ایجاد شئ مقداردهی شوند و در نتیجه باید در کانستراکتور آنها را مقداردهی کنید. بعد از آن دیگر هیچ تغییری در آنها قابل قبول نیست. پس متد setter هم نمی‌توان برای آنها تعریف کرد.
این کلیدواژه زمانی مفید است که نوع داده‌ی فیلد، اولیه یا یک کلاس تغییرناپذیر[۲۳] باشد؛ کلاسی تغییرناپذیر است که هیچ کدام از متدهایش نتوانند اشیاء آن را تغییر دهند؛ مثل String.

null

هر داده‌ی مرجع (رفرنس) را که مقداردهی نکنیم به صورت پیش فرض مقدار null می‌گیرد. به عنوان مثال در کانستراکتور User خط مربوط به مقداردهی اولیه‌ی name را حذف کنید. حال Game را دوباره کامپایل کرده و اجرا کنید. دقت کنید که حتماً باید برنامه را قبل از اجرا یک بار دیگر کامپایل کنید. پس از اجرا باید به جای اسم هر کاربر null چاپ شده باشد. اگر بخواهیم عبارت بامعنی‌تری چاپ شود می‌توانیم کد زیر را در کانستراکتور User به کار ببریم.

```
1        if (name == null)
2            this.name = "Unknown";
3        else
4            this.name = name;
```

روش فشرده‌تر استفاده از کلاس java.util.Objects است.

```
1        this.name = Objects.requireNonNullElse(name, "Unknown");
```

[۲۳] immutable

پارامترهای ضمنی[۲۴] و صریح[۲۵]

متدها روی اشیاء عمل می‌کنند و به فیلدهای نمونه‌شان دسترسی می‌گیرند. مثلاً متد increaseFund از کلاس User را در نظر بگیرید. این متد در زمان فراخوانی فیلد نمونه‌ی fund را تغییر دهد. مثلاً فراخوانی آن روی یک کاربر فرضی مثل user را در نظر بگیرید.

```
user.increaseFund(100);
```

اثر کد بالا به این صورت است که مقدار user.fund به اندازه‌ی ۱۰۰ افزایش می‌یابد. در واقع به صورت ضمنی کد زیر اجرا می‌شود.

```
user.fund += 100;
```

متد increaseFund دو پارامتر دارد. اولین پارامتر آن که پارامتر ضمنی[۲۶] نامیده می‌شود، شیئی از نوع User است که نمی‌توانیم صریحاً آن را در تعریف متد مشاهده کنیم. دومین پارامتر که عدد داخل پرانتز بعد از نام متد است یک پارامتر صریح نامیده می‌شود که می‌توانیم صریحاً آن را همراه با نوع داده‌اش در تعریف متد ببینیم.
در هر متدی کلیدواژه‌ی this به پارامتر ضمنی متد ارجاع می‌دهد. پس اگر بخواهید می‌توانید متد increaseFund را به صورت زیر بنویسید.

```
    public void increaseFund(double value)
    {
        this.fund += value;
    }
```

عده‌ای از برنامه‌نویس‌ها این شیوه را می‌پسندند چراکه تفاوت بین فیلدهای نمونه و متغیرهای محلی را به وضوح مشخص می‌کند.

کلیدواژه‌ی **static** در جاوا

یکی از کلیدواژه‌هایی که آن را به خصوص در تعریف متد main دیده‌اید static است. در اینجا به این کلیدواژه بیشتر می‌پردازیم.
چنانچه در تعریف یک فیلد از این کلیدواژه استفاده کنیم، آنگاه آن فیلد متعلق به کلاس خواهد بود و به ازای هر شئ کلاس به طور جداگانه تعریف نمی‌شود. به عنوان مثال فرض کنید می‌خواهیم به هر کاربر یک شناسه یا id بدهیم. دو فیلد جدید به صورت زیر تعریف می‌کنیم.

```
    private static int nextId = 1;
    private int id;
```

حال هر کاربر یک شناسه‌ی مخصوص به خود را دارد، ولی فیلد nextId بین همه‌ی آنها مشترک است.

[۲۴] implicit
[۲۵] explicit
[۲۶] پارامتر ضمنی target یا receiver هم نامیده می‌شود.

تمرین ۹. با استفاده از دو فیلد *nextId* و *id* کلاس کاربر را به گونه‌ای تغییر دهید که در زمان ساخت هر کاربر، شناسه‌ی کاربر جدید یکی از شناسه‌ی کاربری که قبل از آن ایجاد شده بیشتر باشد.

شاید فکر کنید که اگر یک فیلد را static تعریف کنید آنگاه همیشه ثابت خواهد بود، اما در زبان جاوا چنین نیست! در واقع فیلدهای ثابت را همان طور که دیدید با کلیدواژه‌ی final تعریف می‌کنیم. اما ترکیب این دو کلیدواژه چه معنی‌ای خواهد داشت؟
در واقع ترکیب این دو کلیدواژه در زبان جاوا معمول‌تر است و باعث می‌شود که یک متغیر هم ثابت بوده و هم متعلق به کلاس باشد و نه هر شئ جداگانه. به عنوان مثال عدد پی در کلاس Math به صورت زیر تعریف شده است.

```
    private static final double PI = 3.14159265358979323846;
```

در حال حاضر می‌توانید با Math.PI به عدد پی دسترسی داشته باشید، ولی اگر کلیدواژه‌ی static نبود، آنگاه نیاز داشتید شیئی از این کلاس ایجاد کنید تا به عدد پی دسترسی داشته باشید و هر شیئی هم عدد پی مخصوص خودش را داشت!
علاوه بر فیلدهای static، متدهای static هم داریم. اگر متدی با این کلیدواژه تعریف شود آنگاه روی شئ عمل نمی‌کند. پس می‌توانید آنها را متدهای بدون پارامتر ضمنی this در نظر بگیرید. چنین متدی به فیلدهای static دسترسی دارد ولی نمی‌تواند به فیلدهای غیر static دسترسی داشته باشد، چراکه این فیلدها به ازای هر شئ به طور جداگانه تعریف می‌شوند.

تمرین ۱۰. متد خصوصی و *static* به نام *advanceId* را در کلاس کاربر تعریف کنید به گونه‌ای که *nextId* را در هنگام فراخوانی افزایش داده و خروجی آن را طوری تعیین کنید که در کانستراکتور *User* با استفاده از این متد شناسه‌ی هر کاربر جدید را طبق همان منطق قبلی تعیین کنیم. آنگاه با تغییر در متد *printInformation* و اجرای *Game*، صحت برنامه‌ی خود را تایید کنید.

تمرین ۱۱. چرا متد *main* را *static* تعریف می‌کنیم؟

تمرین ۱۲. بازی‌ای که در تمرین ۳ ساختیم سخت است. می‌خواهیم آن را به گونه‌ای تغییر دهیم که عدد انتخاب شده توسط برنامه در بازه‌ی کوچکتری باشد و همچنین محدودیت حدس‌های کاربر هم بیشتر باشد. فیلدهای مناسبی برای این اعداد با توجه به کلیدواژه‌هایی که آموختید در کلاس *GuessMyNumber* تعریف کنید.