

به نام خدا



## برنامه‌سازی ییشوفته

دانشگاه شهید بهشتی · دانشکده مهندسی کامپیوتر

دکتر مجتبی وحیدی اصل

---

آشنایی با شی گرایی

سید محمد حسینی

## فهرست مطالب

1. آشنایی با مفهوم شی گرایی
2. تفاوت کلاس و شی
3. نوشتن چندین کلاس ساده
4. سازنده‌ها
5. نحوه نمایش اشیا و کلاس‌ها در حافظه
6. نحوه ایجاد زباله
7. متدها و متغیرهای نمونه
8. متدها و فیلدهای استاتیک
9. ارسال اشیا به متدها

## مقدمه

تا به اینجا شما با مفاهیم اولیه زبان جاوا آشنا شده‌اید. شما برای حل یک مثال در زبان **جاوا** یا در زبان **سی‌پلاس‌پلاس** متدهایی تعریف و از آنها استفاده می‌کردید اما واقعیت این است که نمیتوان همه‌ی مسائل و نیازهای دنیای واقعی را صرفاً با متدها پیاده‌سازی کرد، به ویژه در پروژه‌های بزرگ این روش باعث افزایش پیچیدگی و کاهش فهم بهتر کد می‌شود. **شی‌گرایی(OOP- Object Oriented Programming)** یکی از مهم‌ترین مفاهیم زبان‌های برنامه‌نویسی مثل **جاوا** می‌باشد. این مفهوم باعث ساختارمند شدن برنامه شده و باعث می‌شود نگهداری کد و تغییر آن ساده‌تر باشد.

## آشنایی با مفهوم شیگرایی

یک شی بیانگر یک موجودیت در دنیای واقعی است که می‌تواند هویت مستقلی داشته . برنامه‌نویسی [شیگرایی](#) به معنی برنامه‌نویسی با استفاده از اشیا می‌باشد.

- یک دانشجو
- یک میز
- یک خودرو
- یک دکمه گرافیکی
- یک وام بانکی (خود بانک)



الان ممکنه سوالی برآتون پیش بیاد که ایجاد کردن این اشیا مثل دنیای خودمون حالت و رفتاری داره؟

- جواب این سوال بله است. هر شی ای که شما داخل یک قطعه برنامه ایجاد میکنید دارای **حالت**, **هویت** و رفتارهای مخصوص به خود است
- **حالت** یک شی شامل مجموعه‌ای از فیلدهای داده‌ای با مقادیر آنها می‌باشد
- **رفتار** یک شی توسط مجموعه متدهای آن تعریف می‌شود

---

یک شی دارای رفتار است

: در شیوه‌های برنامه‌نویسی ساخت‌یافته (غیرشیگرا) داریم

داده‌ها که به صورت انفعالی(غیر فعال) در برنامه استفاده می‌شوند.

توابع، که قادرند بر روی داده‌ها عملیاتی انجام دهند.

با بزرگ‌تر شدن برنامه‌ها، مدیریت کد دشوارتر می‌شود

امکان بروز تداخل بین داده‌ها و توابع وجود دارد

افزودن قابلیت‌های جدید نیازمند تغییرات گسترده است

---

و متدهای مربوطه قرار می‌گیرند. این متدها بر روی داده‌های (Object) در شیوه برنامه‌نویسی شی‌گر برنامه از [اشیا](#) ساخته می‌شود. در درون هر شی داده‌ها در رابطه با برنامه‌نویسی شی‌گرا و اشیا یک سری نکات قابل ملاحظه می‌باشد. [همان شی](#) دستکاری انجام می‌دهند و داده‌های اشیای دیگر را تغییر نخواهند داد:

- است و قادر است کارهایی را انجام دهد (active) یک شیء فعال
  - یک شیء مسئول داده‌های مربوط به خودش است
  - می‌تواند داده‌های خود را برای دیگر اشیاء در معرض نمایش و استفاده قرار دهد
  - مخصوص به خود را دارد و از دیگر اشیاء تمایز است (Identity) هر شیء هویت
  - متفاوت در زمان‌های مختلف باشد (State) یک شیء می‌تواند دارای وضعیت
  - هر شیء توسط متدهای آن مشخص می‌شود (Behavior) رفتار
  - داشته باشند (Interaction) اشیاء می‌توانند با یکدیگر تعامل
  - یک شیء می‌تواند از روی کلاس‌های از پیش تعریف شده ساخته شود
-

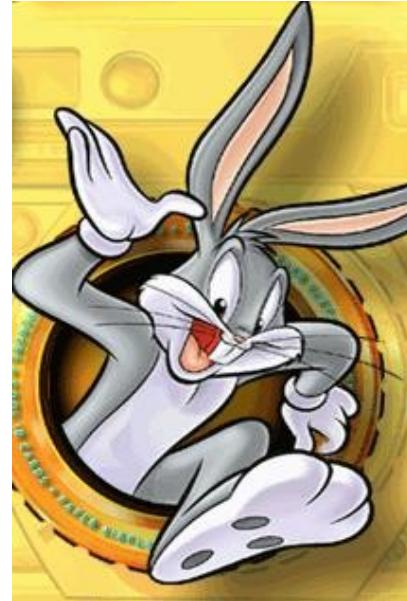
## یک مثال از شی در دنیای واقعی

شما می‌توانید (برای مثال در یک بازی) یک شی خرگوش ایجاد کنید.  
این شی دارای حالت و رفتارهایی است که می‌تواند آنرا از دیگر اشیا متمایز کند.  
این خرگوش می‌تواند داده‌هایی داشته باشد مانند:

- میزان گرسنگی آن را نشان دهد.
- میزان ترسیدن آن را نشان دهد.
- مکان فعلی آن را نشان دهد.
- رنگ آن را مشخص کند.
- سن خرگوش را نشان دهد.

و متدهای زیر را دارا باشد:

- خوردن
- پنهان شدن
- کندن زمین
- دویدن
- خوابیدن



## یک مثال دیگر از اشیا در دنیای واقعی(انسان)

شیئی انسان دارای یکسری **فیلد(ویژگی)هایی** میباشد که میتوانیم از برخی از آنها به صورت زیر یاد کنیم:

- عنوان
- نام
- نام خانوادگی
- تاریخ تولد
- آدرس

همچنین برای یک انسان یکسری **عملیات** مانند عملیات زیر قابل تعریف میباشد:

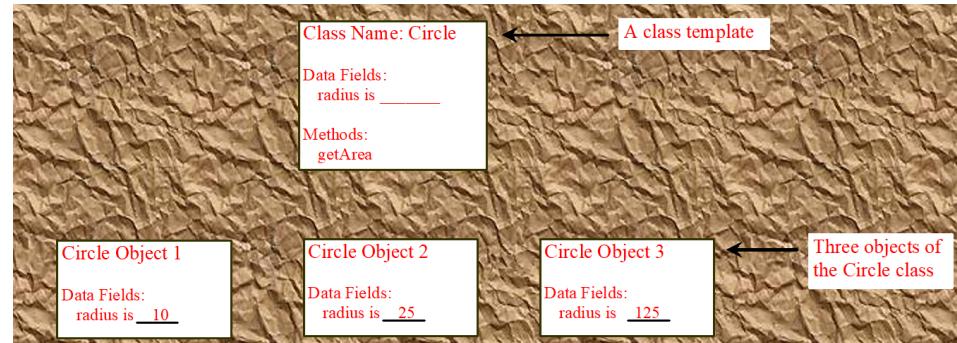
- دانستن این صفات و تغییر دادن آنها

از دانستن این صفات یک مورد دانستن قد یک انسان است. ما میتوانیم **عملیاتی** معرفی کنیم که برای سادهسازی در برنامهنویسی و جهت دانستن قد به برنامه اضافه شده است.

**نکته:** لازم به ذکر است که این عملیات‌ها برای دانستن تمام ویژگی‌های یک کلاس به سادگی در قالب تعریف کردن چند متادقابل انجام است که در بخش‌های جلوتر به صورت مفصل به آنها خواهیم پرداخت.



اشیا



- یک شیئی هم دارای **حالت** و هم رفتار است.
- حالت، تعریف کننده **وضعیت** یک شیئی بوده و رفتار میگوید آن شیئی میتواند **چه کارهایی** انجام بدهد.

## کلاس‌ها

- کلاس‌ها **ساختارهایی** هستند که اشیائی از یک نوع را توصیف می‌کنند.
- این نوع توسط کلاس مشخص می‌شود. کلاس‌ها مانند **قالب‌هایی** هستند که اشیاء از روی آن‌ها ایجاد می‌شوند.
- یک کلاس حاوی **متغیرها** برای توصیف فیلدها و در متدها برای توصیف رفتار اشیاء است.
- علاوه بر این، یک کلاس شکلی خاصی از متدها به نام **سازنده‌ها (constructor)** را فراهم می‌کند که به محض ایجاد یک شیء در آن کلاس فراخوانی می‌شوند.



```
In [29]: !pip install jbang
import jbang
jbang.exec("trust add https://github.com/jupyter-java")
jbang.exec("install-kernel@jupyter-java")
```

```
[notice] A new release of pip is available: 24.2 -> 25.2
[notice] To update, run: python.exe -m pip install --upgrade pip
Requirement already satisfied: jbang in c:\users\mohammad hosseini\appdata\local\programs\python\python312\lib\site-
packages (0.7.0)
```

```
Out[29]: jbang.jbang.CommandResult
```

```
In [30]: class Circle {
    /** The radius of this circle */
    double radius; //data field

    /** Construct a circle object */
    Circle() {
    }

    /** Construct a circle object */
    Circle(double newRadius) {
        radius = newRadius;
    }
```

```

    /** Return the area of this circle */
    double getArea() { //method
        return radius * radius * 3.14159;
    }
}

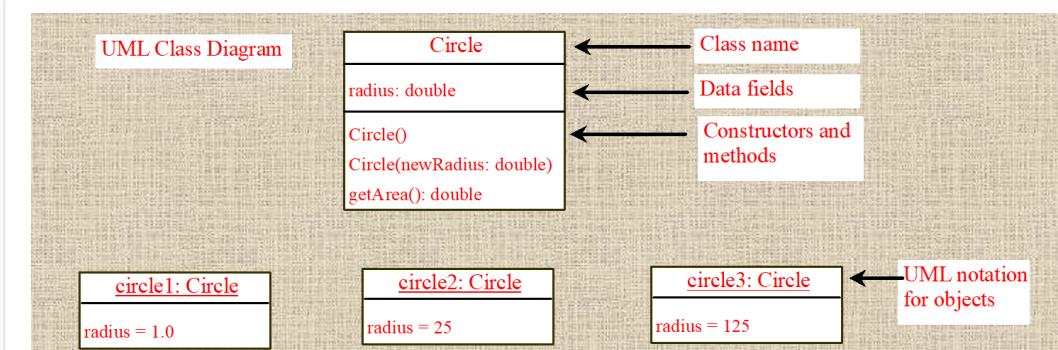
```

Cell In[30], line 1

```
class Circle {
```

SyntaxError: invalid syntax

## کلاس ها



همیشه یک کلاس تعریف کنید و برای کلاس تعریف شده یک **tester** بسازید

### 1 Write your class

```
class Dog {  
    int size;  
    String breed;  
    String name;  
  
    void bark() {  
        System.out.println("Ruff! Ruff!");  
    }  
}
```

instance variables  
a method



### 2 Write a tester (TestDrive) class

just a main method  
(we're gonna put code  
in it in the next step)

```
class DogTestDrive {  
    public static void main (String[] args) {  
        // Dog test code goes here  
    }  
}
```

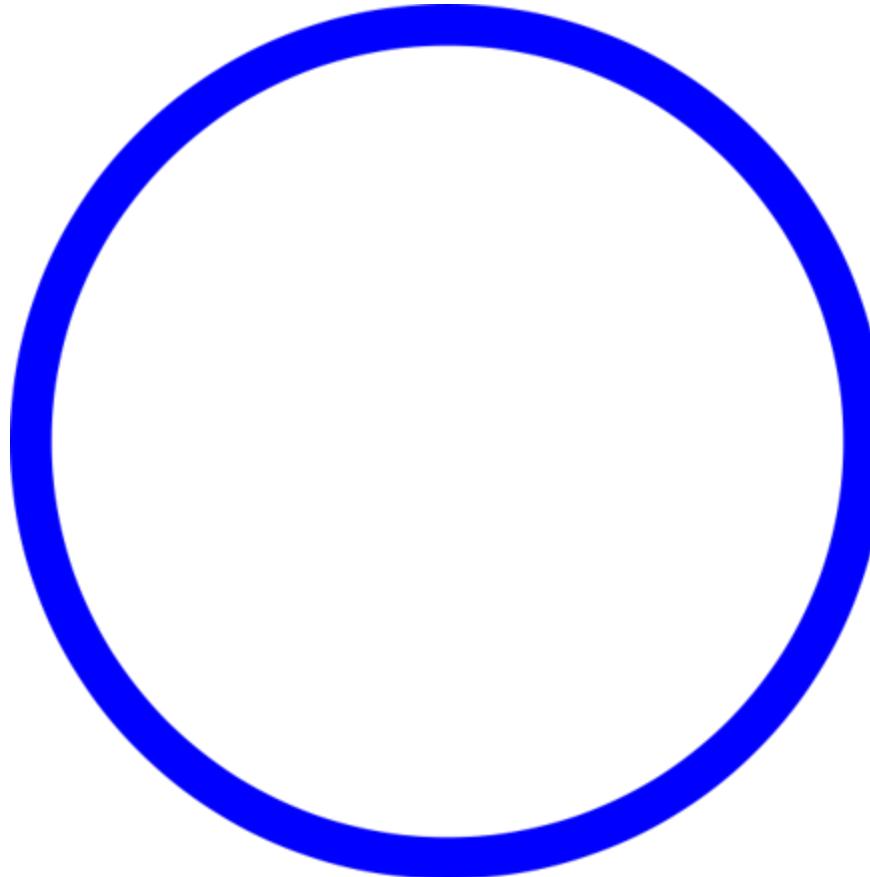
3

In your tester, make an object and access the object's variables and methods

```
class DogTestDrive {  
    public static void main (String[] args) {  
        Dog d = new Dog(); ← make a Dog object  
        d.size = 40; ← use the dot operator(.)  
        d.bark(); ← to set the size of the Dog  
        } ← and to call its bark() method  
    }
```

## مثالی از تعریف کلاس و ایجاد اشیا

نحوه ایجاد اشیا، دسترسی به داده‌های آنها و استفاده از متدها را با مثال نشان بدهید.



```
In [ ]: public class TestCircle1 {
    /** Main method */
    public static void main(String[] args) {
        // Create a circle with radius 5.0
        Circle1 myCircle = new Circle1(5.0);
        System.out.println("The area of the circle of radius " + myCircle.radius + " is " + myCircle.getArea());

        // Create a circle with radius 1
        Circle1 yourCircle = new Circle1();
        System.out.println("The area of the circle of radius " + yourCircle.radius + " is " + yourCircle.getArea())
```

```

    // Modify circle radius
    yourCircle.radius = 100;
    System.out.println("The area of the circle of radius " + yourCircle.radius + " is " + yourCircle.getArea())
}

// Define the circle class with two constructors
class Circle1 {
    double radius;

    /** Construct a circle with radius 1 */
    Circle1() {
        radius = 1.0;
    }

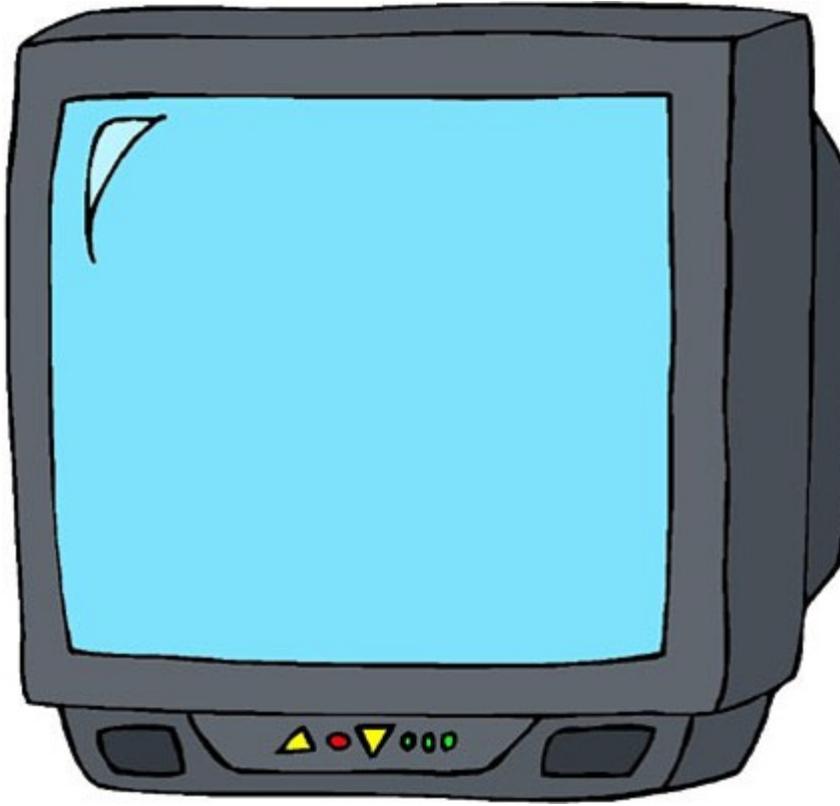
    /** Construct a circle with a specified radius */
    Circle1(double newRadius) {
        radius = newRadius;
    }

    /** Return the area of this circle */
    double getArea() {
        return radius * radius * Math.PI;
    }
}

```

## مثالی از تعریف کلاس و ایجاد اشیا

یک کلاس TV تعریف کنید و نحوه ایجاد اشیا از آن و دسترسی به داده‌ها و متدهای آن را نشان دهید.



```
In [ ]: public class TV {  
    int channel = 1; // Default channel is 1  
    int volumeLevel = 1; // Default volume level is 1  
    boolean on = false; // By default TV is off  
  
    public TV() {  
    }  
  
    public void turnOn() {  
        on = true;  
    }
```

```
public void turnOff() {
    on = false;
}

public void setChannel(int newChannel) {
    if (on && newChannel >= 1 && newChannel <= 120)
        channel = newChannel;
}

public void setVolume(int newVolumeLevel) {
    if (on && newVolumeLevel >= 1 && newVolumeLevel <= 7)
        volumeLevel = newVolumeLevel;
}

public void channelUp() {
    if (on && channel < 120)
        channel++;
}

public void channelDown() {
    if (on && channel > 1)
        channel--;
}

public void volumeUp() {
    if (on && volumeLevel < 7)
        volumeLevel++;
}

public void volumeDown() {
    if (on && volumeLevel > 1)
        volumeLevel--;
}
}
```

```
In [ ]: public class TestTV {
    public static void main(String[] args) {
        TV tv1 = new TV();
        tv1.turnOn();
        tv1.setChannel(30);
```

```
tv1.setVolume(3);

TV tv2 = new TV();
tv2.turnOn();
tv2.channelUp();
tv2.channelUp();
tv2.volumeUp();

System.out.println("tv1's channel is " + tv1.channel
    + " and volume level is " + tv1.volumeLevel);
System.out.println("tv2's channel is " + tv2.channel
    + " and volume level is " + tv2.volumeLevel);
}

}
```

## مثالی از تعریف کلاس و ایجاد اشیا

---

**Dog d = new Dog();  
d.bark();**



think of this

like this



Think of a Dog  
reference variable as  
a Dog remote control.

You use it to get the  
object to do something  
(invoke methods).

# Dog

name

bark()

eat()

chaseCat()

wag();

fetch();



## نگاهی عمیق‌تر به چگونگی ایجاد اشیا

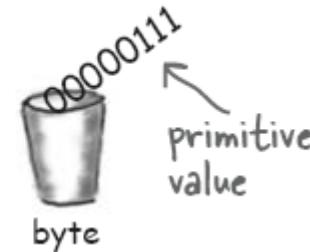
ما با انواع اصلی (Primitive) آشنا شدیم. اما در جاوا انواع **ارجاعی** (**reference types**) هم داریم. متغیرهای **ارجاعی** به جای نکه داشتن خود داده، **آدرس یا ارجاع** شی در حافظه را نگهداری می‌کنند. همه **کلاس‌ها، آرایه‌ها و enum‌ها** در جاوا از نوع ارجاعی هستند. وقتی یک شی با **new** ساخته می‌شود، در **heap** ذخیره می‌شود و متغیر فقط آدرس آن را دارد.

**نکته:** اگر دو متغیر به یک شی اشاره کنند، تغییر یکی روی دیگری هم اثر می‌گذارد.

## Primitive Variable

```
byte x = 7;
```

The bits representing 7 go into the variable. (00000111).

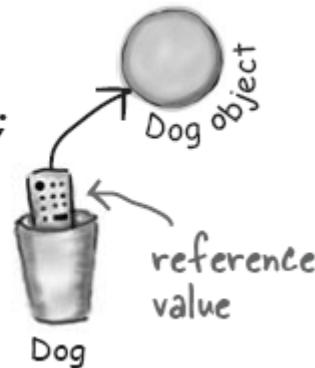


## Reference Variable

```
Dog myDog = new Dog();
```

The bits representing a way to get to the Dog object go into the variable.

*The Dog object itself does not go into the variable!*



مراحل ایجاد شی جدید

**1**                   **2**  
Dog myDog        = new Dog();  
**3**

- ## 1 Declare a reference variable

```
Dog myDog = new Dog();
```



دستور نشان داده شده در کادر قرمزه **JVM** می‌گوید تا فضایی را برای **متغیر جاگعی** اختصاص دهد و نام این متغیر را **myDog** بگذارد. این **متغیر جاگعی** تا ابد از نوع **Dog** خواهد بود. یعنی، ریموت کنترلی که دکمه هایی برای کنترل یک سگ دارد، اما نمی‌تواند یک **گربه**، یک **دایره** یا یک **ریبات** را کنترل کند.

2

## Create an object

Dog myDog = new Dog();



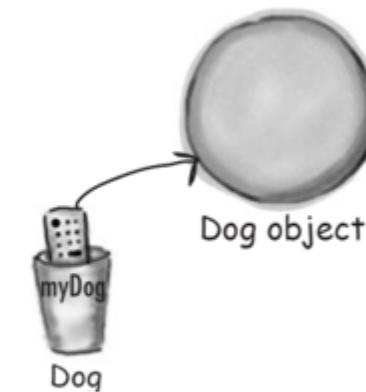
اختصاص دهد **Dog** در درون **myDog** می گوید تا فضایی را برای شیء جدید **JVM heap** دستور نشان داده شده در کادر قرمز به

---

3

## Link the object and the reference

Dog myDog = new Dog();



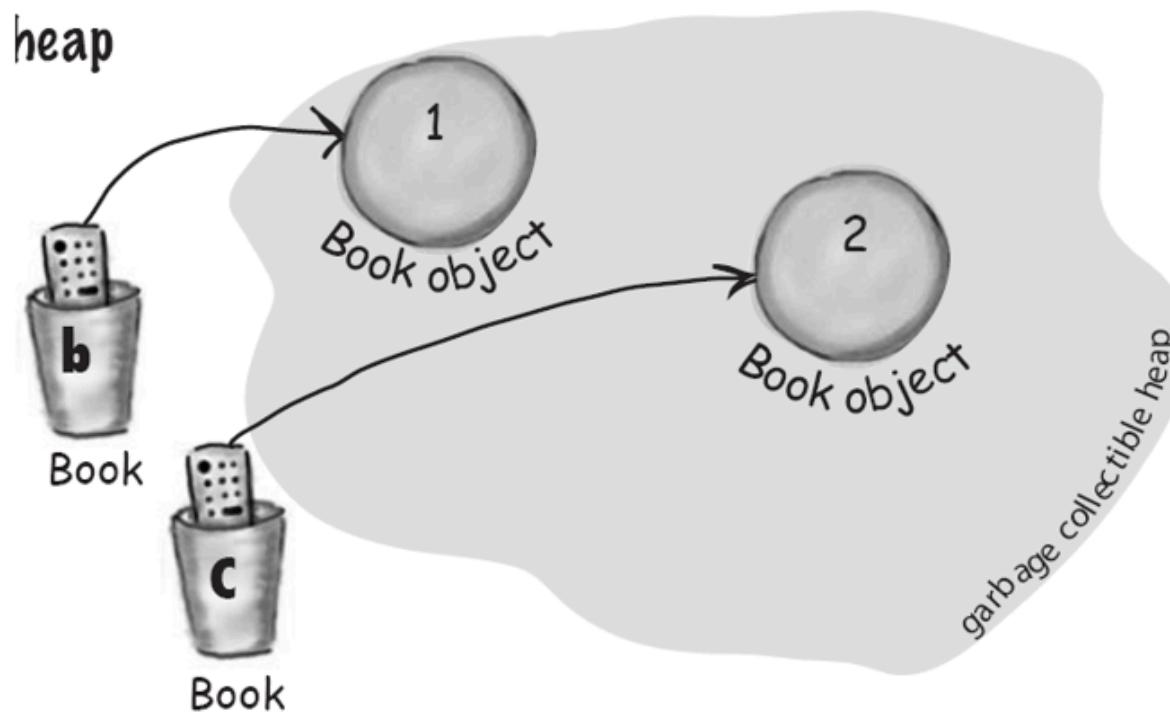
فعال می کند **Dog** مرتبط می کند. به بیان دیگر، ریموت کنترل را برای هدایت شیء جدید از نوع **myDog** شیء جدید را به متغیر ارجاعی

---

## مثال بیشتر

```
Book b = new Book();
```

```
Book c = new Book();
```



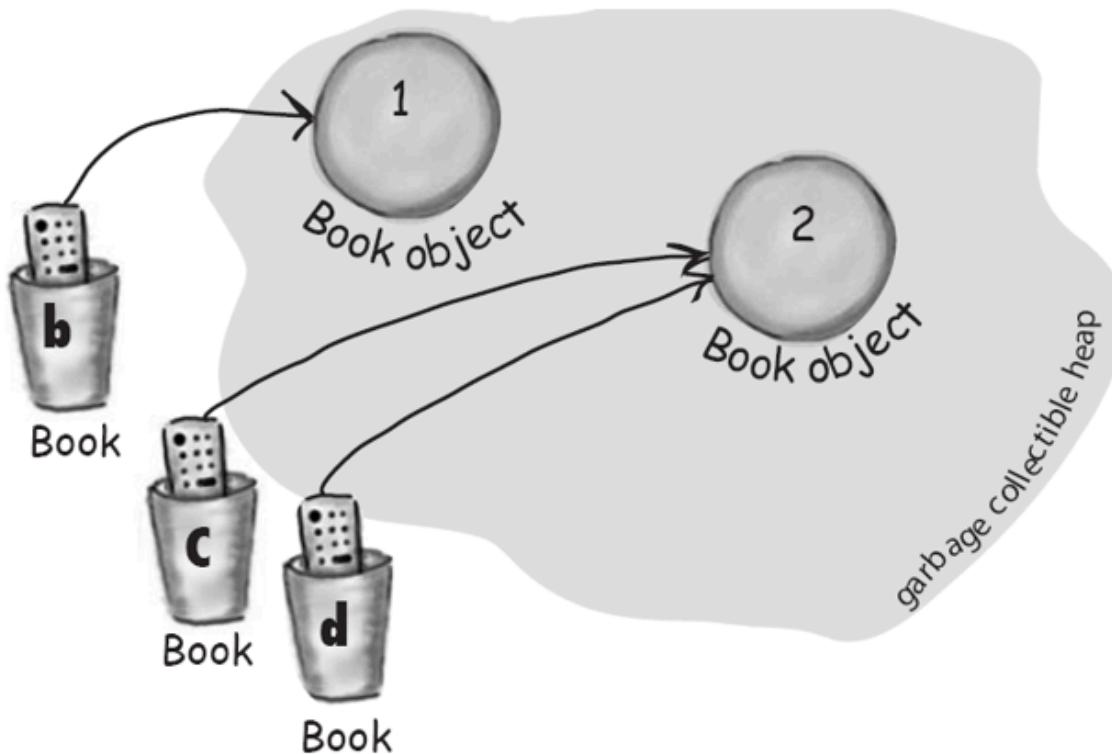
References: 2  
Objects: 2

وقتی در جاوا می‌نویسیم `Book c = new Book();` و `Book b = new Book();`، دو متغیر ارجاعی به نامهای **b** و **c** ساخته می‌شوند و هر کدام با استفاده از دستور `new Book();` یک شیء جدید از نوع `Book` را در حافظه `heap` ایجاد می‌کنند. در نتیجه، دو شیء جداگانه در

وجود دارد؛ متغیر `b` به شیء اول و متغیر `c` به شیء دوم اشاره می‌کند. بنابراین، **تعداد ارجاع‌ها (References)** برابر با `۲` و **تعداد اشیاء (Objects)** (**Objects**) برابر با `۱` نیز برابر با `۲` خواهد بود. نکته مهم این است که هر بار که از دستور `new` استفاده می‌کنیم، یک شیء جدید در `heap` ساخته می‌شود، حتی اگر نوع آن یکسان باشد.

---

`Book d = c;`

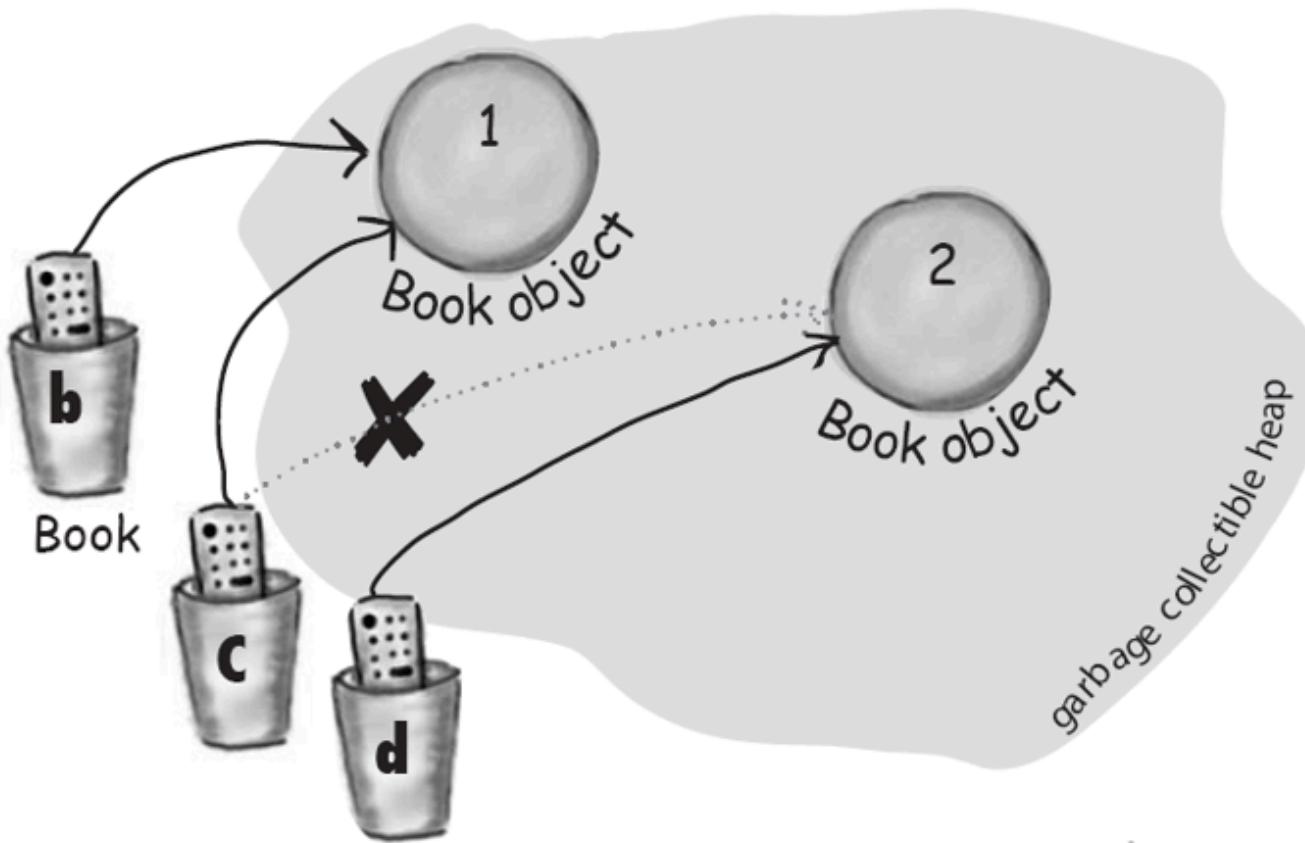


References: 3

Objects: 2

وقتی در جاوا می‌نویسیم `Book d = c;` متغیر `d` ایجاد می‌شود و به همان شیئی اشاره می‌کند که متغیر `c` به آن وصل است. در این حالت متغیر `b` همچنان به شیء شماره ۱ اشاره دارد، در حالی که متغیرهای `c` و `d` هر دو به شیء شماره ۲ متصل هستند. بنابراین در حافظه `heap` همچنان فقط ۲ شیء وجود دارد، اما تعداد ارجاع‌ها (References) به ۳ افزایش پیدا می‌کند.

`c = b;`



وقتی در جاوا می‌نویسیم `c = b;`، متغیر **c** به همان شیئی اشاره می‌کند که متغیر **b** به آن متصل است، یعنی شیء شماره ۱. در نتیجه ارتباط قبلی متغیر **c** با شیء شماره ۲ از بین می‌رود. بنابراین، متغیرهای **b** و **c** هر دو به شیء شماره ۱ اشاره می‌کنند و متغیر **d** همچنان به شیء شماره ۲ متصل است. در این وضعیت در حافظه **heap** همچنان فقط ۲ شیء (**Objects**) وجود دارد، اما تعداد ارجاع‌ها (**References**) برابر با ۳ است.

In [ ]:

```
public class TV {  
    int channel = 1; // Default channel is 1  
    int volumeLevel = 1; // Default volume level is 1  
    boolean on = false; // By default TV is off  
  
    // This is our constructor  
    public TV() {  
    }  
  
    public void turnOn() {  
        on = true;  
    }  
  
    public void turnOff() {  
        on = false;  
    }  
  
    public void setChannel(int newChannel) {  
        if (on && newChannel >= 1 && newChannel <= 120)  
            channel = newChannel;  
    }  
  
    public void setVolume(int newVolumeLevel) {  
        if (on && newVolumeLevel >= 1 && newVolumeLevel <= 7)  
            volumeLevel = newVolumeLevel;  
    }  
  
    public void channelUp() {  
        if (on && channel < 120)  
            channel++;  
    }  
  
    public void channelDown() {  
        if (on && channel > 1)  
            channel--;  
    }  
  
    public void volumeUp() {  
        if (on && volumeLevel < 7)
```

```
        volumeLevel++;
    }

    public void volumeDown() {
        if (on && volumeLevel > 1)
            volumeLevel--;
    }
}
```

## سازنده‌ها

سازنده‌ها نوع خاصی از متدها هستند که برای ایجاد اشیاء فراخوانی می‌شوند.

---

```
Circle() {  
}
```

```
Circle(double newRadius) {  
    radius = newRadius;  
}
```

---

یک **سازنده** بدون پارامتر **no-arg constructor** نامیده می‌شود. **سازنده‌ها** باید همنام با **کلاس** خود باشند و هیچ مقدار **برگشتی** حتی **void** هم ندارند. آن‌ها هنگام ایجاد شیء با استفاده از عملگر **new** فراخوانی می‌شوند و نقش مهمی در مقداردهی اولیه به اشیای ساخته شده از آن کلاس دارند. به عبارت دیگر، **سازنده‌ها** کمک می‌کنند تا یک شیء از همان ابتدا وضعیت و داده‌های مشخصی داشته باشد. نکته مهم دیگر این است که اگر شما هیچ سازنده‌ای تعریف نکنید، **کامپایلر جاوا** به طور خودکار یک **سازنده پیش‌فرض** ایجاد می‌کند. همچنین می‌توانید چندین سازنده با پارامترهای متفاوت تعریف کنید تا امکان **سر برگذاری**

سازنده‌ها (Constructor Overloading) را داشته باشید. برای مثال: `new Circle();` یک شیء از کلاس Circle با مقادیر پیش‌فرض ایجاد می‌کند، در حالی که `new Circle(5.0);` یک شیء از همان کلاس می‌سازد اما مقدار اولیه شعاع آن را برابر با ۵ قرار می‌دهد.

---

## ارجاع به یک متغیر ارجاعی!

برای ارجاع به یک شیء، آن را به یک متغیر از نوع ارجاعی (reference) منتسب کنید.

برای اعلان یک متغیر از نوع ارجاعی، از قاعده‌ی نحوی زیر استفاده کنید:

```
ClassName objectRefVar;
```

به عنوان مثال:

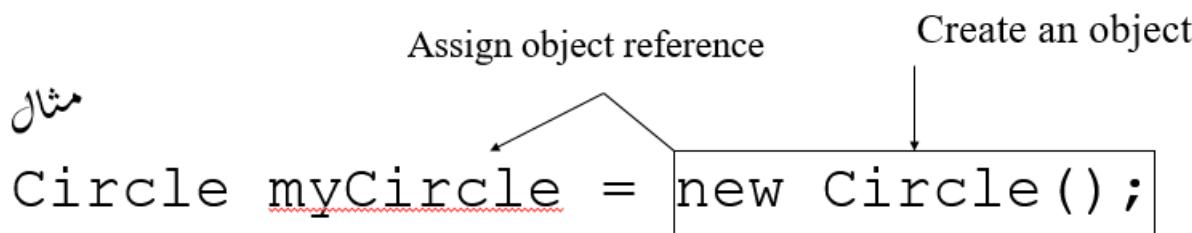
```
Circle myCircle;
```

در اینجا متغیر `myCircle` از نوع ارجاعی تعریف شده و می‌تواند به یک شیء از کلاس Circle ارجاع دهد.

---

## ارجاع به یک متغیر ارجاعی!

```
ClassName objectRefVar = new ClassName();
```



دستور `ClassName objectRefVar = new ClassName();` نشان می‌دهد که در جاوا وقتی یک شیء ساخته می‌شود، هم‌زمان یک متغیر ارجاعی تعریف و به شیء جدید متصل می‌گردد. برای مثال، در دستور `Circle myCircle = new Circle();` بخش `new Circle()` یک شیء جدید ایجاد می‌کند و بخش `= myCircle` آن را به متغیر ارجاعی متصل می‌سازد. این ساختار باعث می‌شود که بتوانیم به شیء تازه ایجاد شده در heap دسترسی داشته باشیم و آن را کنترل کنیم.

## دسترسی به اشیا

ارجاع (دسترسی) به داده‌های درون شیء:

`objectRefVar.data`

فرآخوانی متد درون یک شیء: مثال:

`myCircle.radius`

```
objectRefVar.methodName(arguments)
```

در جاوا، برای دسترسی به ویژگی‌ها و متدهای یک شیء از [مراجع شیء](#) استفاده می‌کنیم. ویژگی‌ها (data fields) مقادیر رامثال: `(myCircle.getArea())` ذخیره می‌کنند و متدها (methods) رفتار شیء را مشخص می‌کنند. به این ترتیب می‌توانیم هم داده‌های شیء را بخوانیم و هم عملیات خاصی را روی آن اجام دهیم.

---

## دسترسی به اشیا

وقتی دستور **Circle myCircle = new Circle(5.0);** اجرا می‌شود، بخش سمت چپ یعنی **Circle myCircle** تنها یک متغیر ارجاعی (مثل یک ریموت کنترل) را اعلان می‌کند که در ابتدا به هیچ شیئی متصل نیست و مقدارش **no value** است. بخش سمت راست یعنی **new Circle(5.0)** یک شیء جدید از کلاس **Circle** در حافظه **heap** ایجاد می‌کند که مقدار اولیه‌ی **radius** آن برابر با  $5.0$  است. سپس عملگر **=** باعث می‌شود مرجع آن شیء به **myCircle** اختصاص داده شود و از این لحظه، **myCircle** به آن شیء در حافظه اشاره می‌کند. به همین ترتیب، در خط **Circle yourCircle = new Circle();** یک متغیر ارجاعی دیگر به نام **yourCircle** ایجاد می‌شود که در ابتدا **no value** دارد، اما بخش راست یعنی **new Circle()** یک شیء جدید **Circle** با مقدار پیش‌فرض (مثلاً  $1.0$  یا  $0.0$  بسته به سازنده) می‌سازد و مرجع آن به **yourCircle** داده می‌شود. حالا هر متغیر ارجاعی به شیء جداگانه‌ای وصل است. در نهایت، دستور **yourCircle.radius = 100;** مقدار **radius** شیئی که به آن اشاره می‌کند را به  $100$  تغییر می‌دهد، در حالی که شیء مربوط به **myCircle** هیچ تغییری نمی‌کند و مقدار  $0.0$  خودش را حفظ می‌کند. به طور خلاصه: **اعلان متغیر فقط یک ریموت خالی می‌سازد، new یک شیء جدید می‌سازد، و عملگر = تصویر این توضیحات آن‌ها را به هم متصل می‌کند؛ سپس هر تغییری روی فیلد‌ها فقط روی همان شیئی اثر دارد که متغیر به آن اشاره می‌کند.** را می‌توانید در تصاویر زیر مشاهده کنید:

---

Declare myCircle

myCircle      no value

```
Circle myCircle = new Circle(5.0);
```

```
Circle yourCircle = new Circle();
```

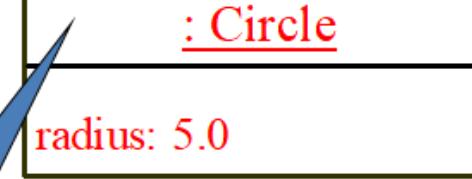
```
yourCircle.radius = 100;
```

```
Circle myCircle = new Circle(5.0);
```

myCircle      no value

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```



Create a circle

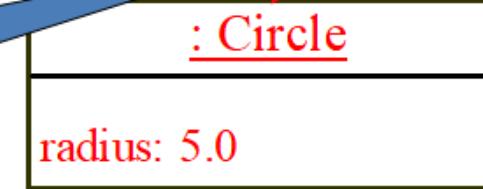
```
Circle myCircle = new Circle(5.0);
```

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

Assign object reference  
to myCircle

myCircle **reference value**

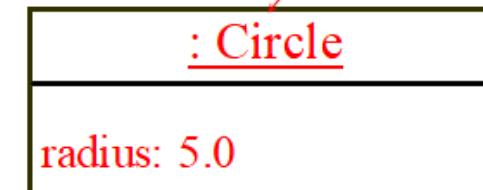


```
Circle myCircle = new Circle(5.0);
```

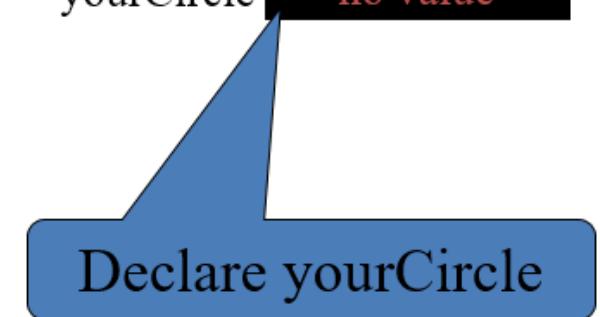
```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

myCircle **reference value**



yourCircle **no value**

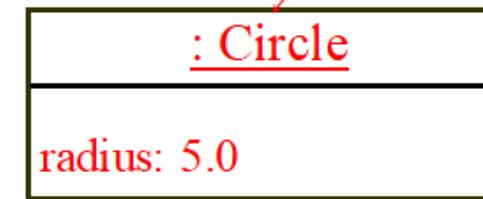


```
Circle myCircle = new Circle(5.0);
```

```
Circle yourCircle = new Circle();
```

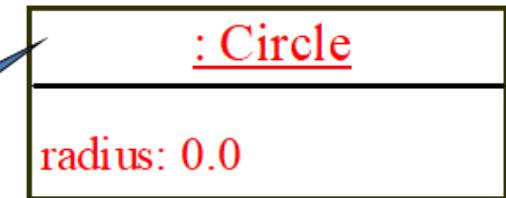
```
yourCircle.radius = 100;
```

myCircle    reference value



yourCircle    no value

Create a new  
Circle object



```
Circle myCircle = new Circle(5.0);
```

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

myCircle **reference value**

: Circle

radius: 5.0

yourCircle **reference value**

: Circle

radius: 1.0

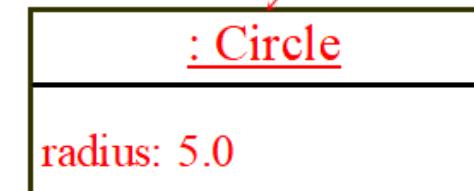
Assign object reference  
to yourCircle

```
Circle myCircle = new Circle(5.0);
```

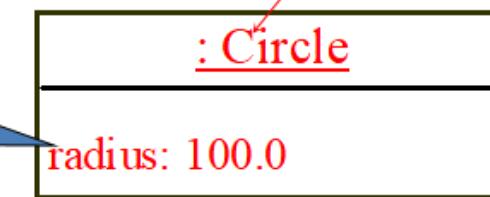
```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

myCircle reference value



yourCircle reference value



Change radius in  
yourCircle

Declare myCircle

myCircle

no value

```
Circle myCircle = new Circle(5.0);
```

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

---

```
Circle myCircle = new Circle(5.0);
```

myCircle      no value

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

: Circle

radius: 5.0

Create a circle

```
Circle myCircle = new Circle(5.0);
```

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

Assign object reference  
to myCircle

myCircle    reference value

: Circle

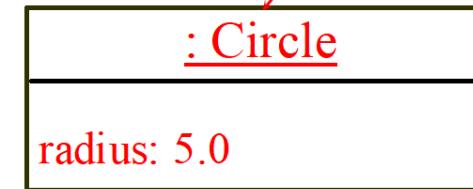
radius: 5.0

```
Circle myCircle = new Circle(5.0);
```

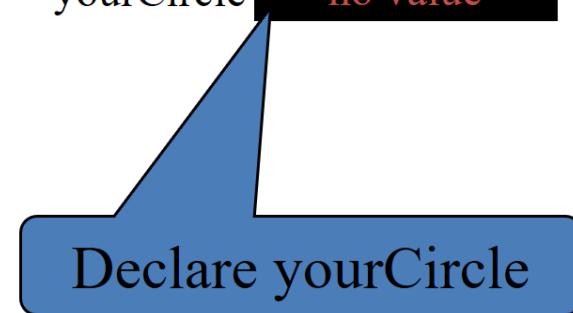
```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

myCircle **reference value**



yourCircle **no value**

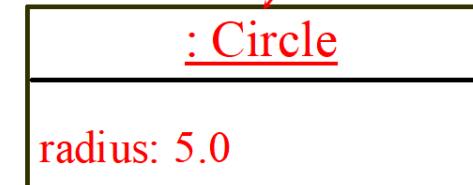


```
Circle myCircle = new Circle(5.0);
```

```
Circle yourCircle = new Circle();
```

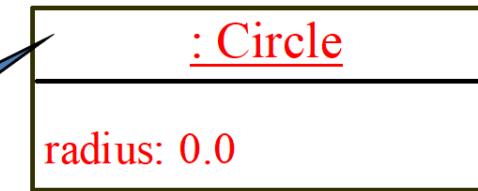
```
yourCircle.radius = 100;
```

myCircle **reference value**



yourCircle **no value**

Create a new  
Circle object

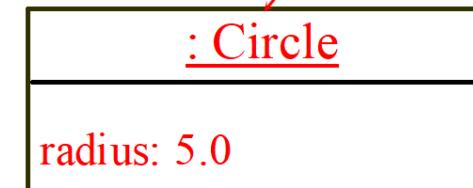


```
Circle myCircle = new Circle(5.0);
```

```
Circle yourCircle = new Circle();
```

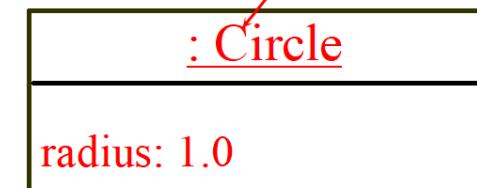
```
yourCircle.radius = 100;
```

myCircle **reference value**



yourCircle **reference value**

Assign object reference  
to yourCircle

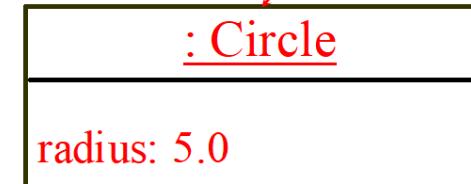


```
Circle myCircle = new Circle(5.0);
```

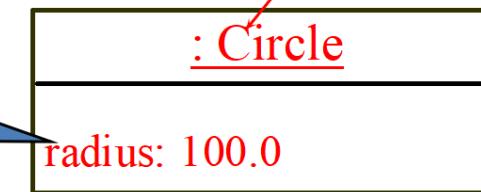
```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

myCircle **reference value**



yourCircle **reference value**



Change radius in  
yourCircle

مقدار null

اگر یک متغیر از نوع ارجاعی (**reference type**) تعریف شود اما به هیچ شیئی ارجاع داده نشود، مقدار آن به طور پیشفرض **null** خواهد بود. برای مثال در قطعه کد زیر:

```
Circle myCircle;
```

متغیر **myCircle** فقط اعلان شده است و هنوز به هیچ شیئی متصل نیست؛ بنابراین مقدار آن **null** است، یعنی هیچ آدرس/مرجعی به یک شیء واقعی در حافظه ندارد.

### مقادیر پیشفرض برای فیلدهای داده‌ای

اگر یک متغیر از نوع ارجاعی (**reference type**) تعریف شود اما به هیچ شیئی ارجاع داده نشود، مقدار آن به طور پیشفرض **null** خواهد بود. برای مثال در قطعه کد زیر:

```
Circle myCircle;
```

متغیر **myCircle** فقط اعلان شده است و هنوز به هیچ شیئی متصل نیست؛ بنابراین مقدار آن **null** است، یعنی هیچ آدرس/مرجعی به یک شیء واقعی در حافظه ندارد.

- برای فیلد داده‌ای از نوع **ارجاعی**: مقدار **null**
- برای نوع **عددی**: مقدار **0**
- برای نوع **بولین**: مقدار **false**
- برای نوع **کاراکتری**: مقدار '**u00000\**'

با این حال، جاوا برای یک متغیر محلی درون یک متغیر پیشفرض در نظر نمی‌گیرد و اگر قبل از استفاده مقداردهی نشود، باعث خطا خواهد شد.

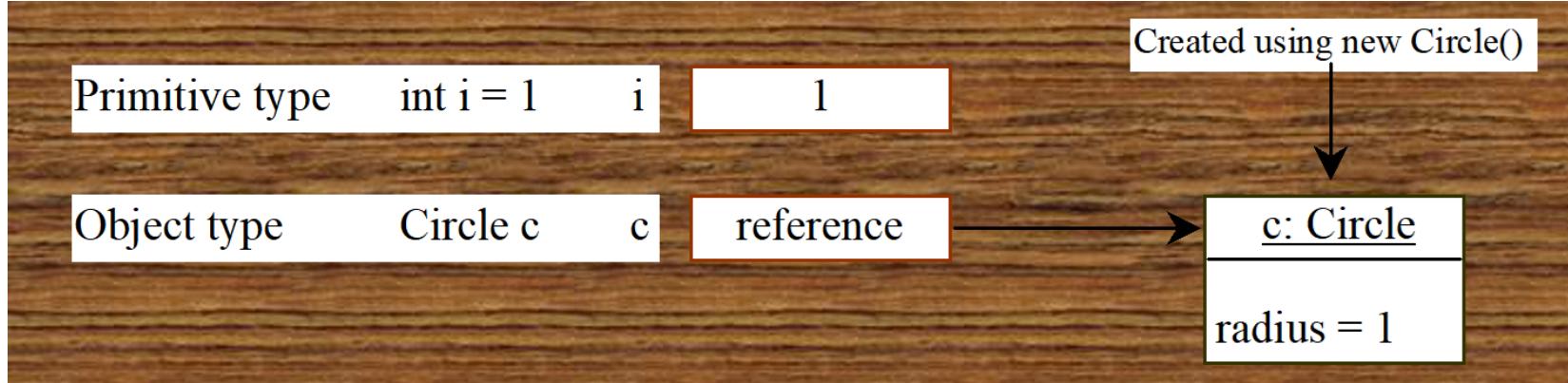
```
In [ ]: public class Test {  
    public static void main(String[] args) {  
        Student student = new Student();  
        System.out.println("name? " + student.name);  
        System.out.println("age? " + student.age);  
        System.out.println("isScienceMajor? " + student.isScienceMajor);  
        System.out.println("gender? " + student.gender);  
    }  
}
```

## مثال

هیچ مقدار پیش‌فرضی در (local variables) تعریف شده‌اند. جاوا برای متغیرهای محلی **main** به عنوان متغیرهای محلی درون متدهای **y** و **x** در این مثال، متغیرهای نظر نمی‌گیرد. به همین دلیل، اگر بخواهیم قبل از مقداردهی اولیه از آن‌ها استفاده کنیم، برنامه با خطای زمان کامپایل مواجه خواهد شد. به طور خلاصه: **متغیرهای محلی باید قبل از استفاده حتماً مقداردهی اولیه شوند**.

```
In [ ]: public class Test {  
    public static void main(String[] args) {  
        int x;      // x has no default value  
        String y;   // y has no default value  
        System.out.println("x is " + x);  
        System.out.println("y is " + y);  
    }  
}
```

تفاوت‌های میان انواع داده‌ای اصلی (اولیه) و انواع ارجاعی

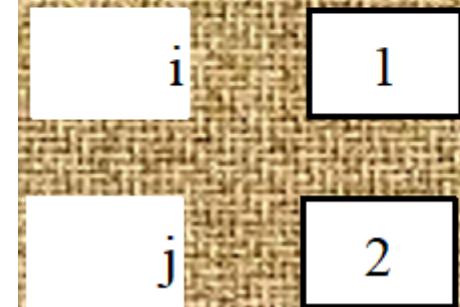


انواع داده‌ای اصلی مانند `int i = 1;` مستقیماً (Primitive Types) و انواع ارجاعی (Object Types). در زبان جاوا دو نوع داده وجود دارد: انواع داده‌ای اصلی مقدار را در خود ذخیره می‌کنند. به این معنا که متغیر `i` مستقیماً عدد `1` را نگه می‌دارد. اما در انواع ارجاعی مانند `Circle c`, متغیر `c` فقط یک ارجاع به شیء ذخیره می‌کند. این شیء با استفاده از دستور `new Circle()` ساخته می‌شود و متغیر `c` به آدرس آن شیء در حافظه اشاره خواهد کرد. (Reference) برای مثال، اگر دایره‌ای با شعاع `1` ساخته شود، مقدار درون شیء ذخیره شده اما متغیر `c` فقط به آن اشاره می‌کند.

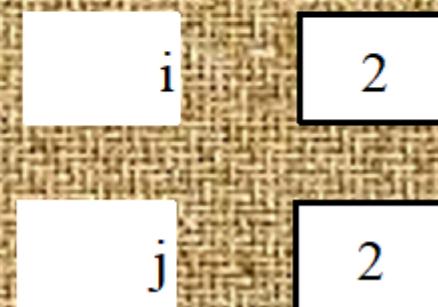
تفاوت‌های میان انواع داده‌ای اصلی (اولیه) و انواع ارجاعی

Primitive type assignment  $i = j$

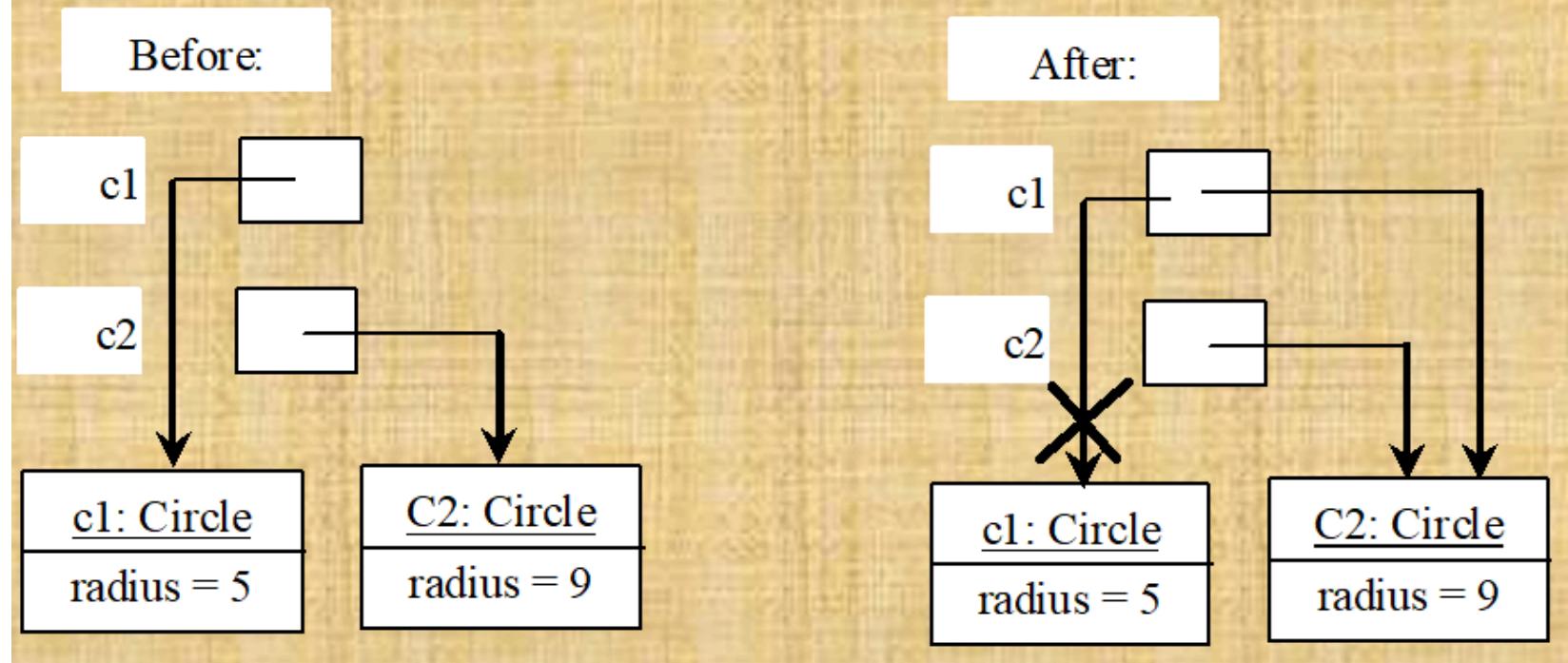
Before:



After:



## Object type assignment $c1 = c2$



این عکس‌ها تفاوت رفتار انتساب در انواع اصلی (Primitive) و انواع ارجاعی (Object) را نشان می‌دهند. در بخش چپ، با انتساب  $j = i$  مقدار عددی متغیر  $j$  مستقیماً در  $i$  کپی می‌شود؛ بنابراین بعد از عمل،  $i$  مقدار جدید (مثلاً ۲) را دارد و  $j$  مقدار قبلی خودش را حفظ می‌کند. اما در بخش راست، با انتساب  $c1 = c2$  تنها مرجع شیء کپی می‌شود، نه خود شیء؛ در نتیجه هر دو متغیر `c1` و `c2` به یک شیء مشترک اشاره می‌کنند (در تصویر شیئی با `radius = 9`). شیء قبلی `c1` (مثلاً با `radius = 5`) اگر مرجعی به آن وجود نداشته باشد، توسط Garbage Collector حذف خواهد شد. به این ترتیب، در انواع اصلی «کپی مقدار» انجام می‌شود، اما در انواع ارجاعی «کپی مرجع» و اشتراک در یک شیء اتفاق می‌افتد.

## جمع آوری زباله (garbage collection)

همان‌طور که در **توضیحات قبلی** و **عکس‌های قبلی** دیدیم، پس از دستور انتساب  $c2 = c1$ ، متغیر **c1** به همان شیئی اشاره می‌کند که توسط **c2** مورد ارجاع قرار گرفته است. در این حالت، شیئی که قبلاً توسط **c1** مورد ارجاع قرار می‌گرفت و دیگر هیچ متغیری به آن اشاره ندارد، بلااستفاده باقی می‌ماند. به چنین شیئی **اصطلاحاً زباله** گفته می‌شود. این زباله‌ها در زبان **جاوا** به صورت خودکار توسط **JVM** مدیریت و جمع‌آوری (**Garbage Collection**) می‌شوند. **Collection** در جاوا یکی از مهم‌ترین ویژگی‌ها است که باعث می‌شود مدیریت حافظه ساده‌تر شود. برخلاف زبان‌هایی مثل C++ یا C که برنامه‌نویس باید حافظه را به صورت دستی آزاد کند، در **جاوا** این کار به طور خودکار توسط **JVM** انجام می‌شود. زمانی که هیچ متغیری به یک شیء اشاره نکند، آن شیء **غیرقابل دسترسی** محسوب می‌شود و **Garbage Collector** آن را آزاد می‌کند. این فرآیند باعث جلوگیری از **Memory Leak** (نشت حافظه) شده و استفاده البته باید توجه داشت که اجرای **Garbage Collector** بر اساس زمان‌بندی خاص **JVM** است و در لحظه‌ای که ما انتظارهاین‌تری از منابع سیستم می‌گردد. داریم اجرا نمی‌شود. این موضوع می‌تواند در برخی موارد باعث کاهش سرعت برنامه برای مدت کوتاهی شود. به همین دلیل، در طراحی برنامه‌های بزرگ باید **نکته**: اگر مطمئnid یک شیء دیگر مورد نیازدقت داشت که تعداد زیادی شیء غیرضروری ایجاد نشود تا باز اضافی بر روی **Garbage Collector** تحمیل نگردد. با این کار، وقتی هیچ مرجع دیگری به آن شیء وجود نداشته‌نیست، می‌توانید **متغیر ارجاعی** کنترل‌کننده آن را به مقدار **null** تنظیم کنید؛ مانند: `C1 = null;` باشد. **JVM** در فرآیند **Garbage Collection** آن را به‌طور خودکار آزاد می‌کند.

## نمونه‌ای از استفاده از کلاس **Date** در **جاوا**

در این مثال، با استفاده از کلاس **java.util.Date** یک شیء جدید از تاریخ ساخته می‌شود. دستور زیر یک نمونه از **Date** را ایجاد کرده و آن را در متغیر **date** ذخیره می‌کند:

```
java.util.Date date = new java.util.Date();
System.out.println(date.toString());
```

با اجرای این کد، متده `toString` از شیء `Tarix` فراخوانی می‌شود و تاریخ و زمان فعلی سیستم به صورت یک رشته نمایش داده خواهد شد. خروجی رشته‌ای مشابه زیر خواهد بود:

```
Wed Feb 15 09:40:19 IRST 2017
```

همان‌طور که مشاهده می‌کنید، این رشته شامل **روز هفته**، **ماه**، **روز ماه**، **ساعت**، **منطقه زمانی** و **سال** است. به این ترتیب، کلاس `Date` در جاوا امکان کار با تاریخ و زمان فعلی سیستم را به ساده‌ترین شکل فراهم می‌کند.

## ادامه کلاس تقویم

The + sign indicates public modifier	<b>java.util.Date</b>	
	<code>+Date()</code>	Constructs a Date object for the current time.
	<code>+Date(elapseTime: long)</code>	Constructs a Date object for a given time in milliseconds elapsed since January 1, 1970, GMT.
	<code>+toString(): String</code>	Returns a string representing the date and time.
	<code>+getTime(): long</code>	Returns the number of milliseconds since January 1, 1970, GMT.
	<code>+setTime(elapseTime: long): void</code>	Sets a new elapse time in the object.

---

برای نمایش تاریخ و زمان در جاوا از کلاس `java.util.Date` استفاده می‌شود. این کلاس قابلیت ایجاد یک شیء جدید از تاریخ و زمان فعلی را دارد و همچنین می‌توان تاریخ مشخصی را بر اساس تعداد میلی‌ثانیه‌های سپری شده از (Epoch time) January 1, 1970 ایجاد کرد. برای نمایش تاریخ و زمان در قالب رشته نیز می‌توان از متد `toString()` کمک گرفت.

### سازنده‌ها :`(Constructors)`

- `Date()` یک شیء `Date` بر اساس زمان فعلی سیستم می‌سازد.
- `Date(Date(elapseTime: long))` یک شیء `Date` بر اساس تعداد میلی‌ثانیه‌های گذشته از January 1, 1970 می‌سازد.

### متدهای مهم کلاس `Date`

- `toString(): String` تاریخ و زمان را به صورت یک رشته قابل خواندن (مثل Feb 15 09:40:19 IRST 2017) برمی‌گرداند.
- `getTime(): long` تعداد میلی‌ثانیه‌های گذشته از January 1, 1970 تا زمان فعلی شیء را برمی‌گرداند.
- `setTime(elapseTime: long)` یک زمان جدید (بر حسب میلی‌ثانیه از 1970/01/01) را به شیء اختصاص می‌دهد.

نکته:

علامت `+` در دیاگرام نشان‌دهنده `public` بودن سازنده‌ها و متدها است. این یعنی می‌توان آن‌ها را از هر جایی در برنامه فراخوانی کرد.

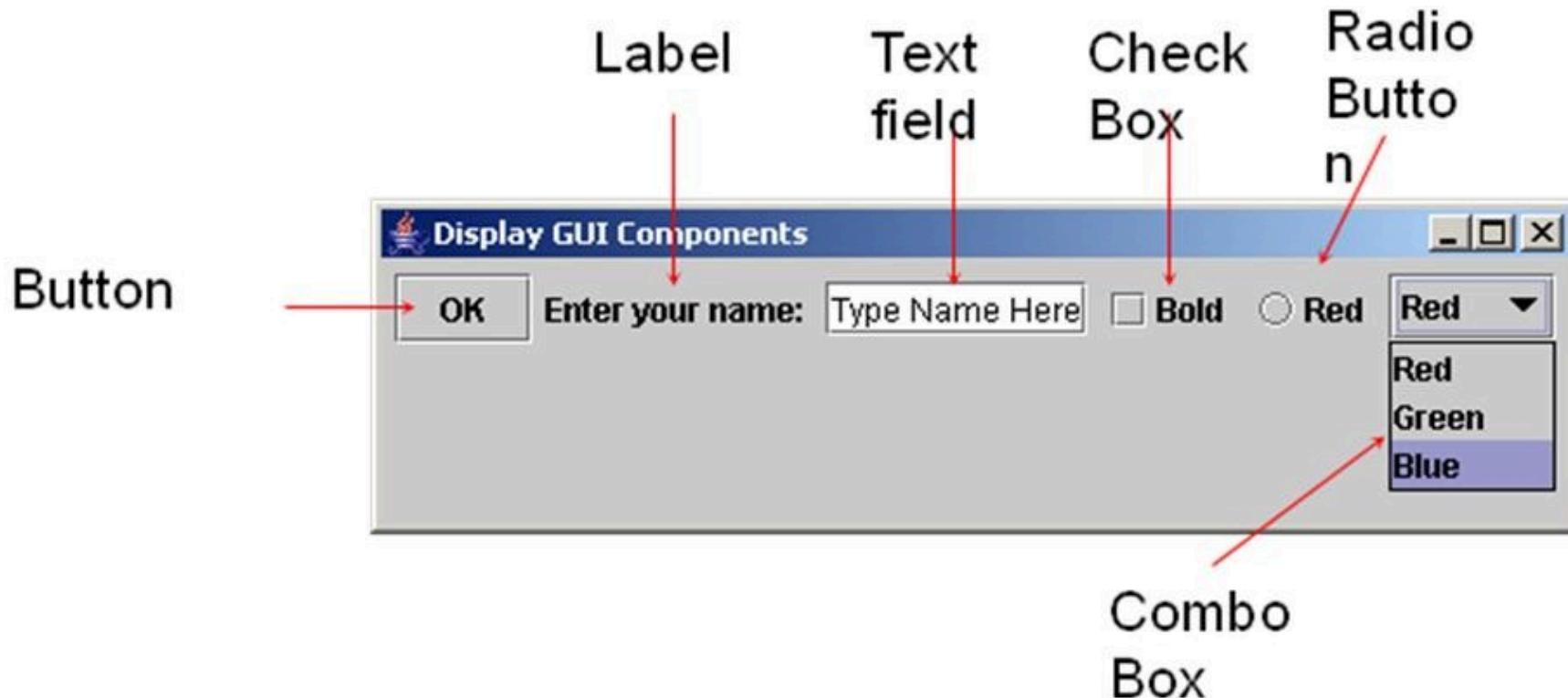
### توضیحات تکمیلی:

کلاس `Date` ابزار پایه‌ای برای کار با زمان است، اما در نسخه‌های جدید جاوا، کلاس‌های پیشرفته‌تری مثل `LocalDateTime`, `LocalDate`, `LocalTime` و `Duration` معرفی شده‌اند که امکانات بیشتر و دقیق‌تر دارند. با این حال، همچنان در بسیاری از کدها و پروژه‌های قدیمی، متدها و سازنده‌های کلاس `Date` پرکاربرد هستند.

---

### نمایش مولفه‌های گرافیکی (GUI)

هنگامی که میخواهید برنامه‌ای جهت ایجاد واسطه‌های گرافیکی کاربر (GUI) بنویسید، میتوانید از کلاس‌های جاوا نظریر **JRadioButton**, **JButton**, **JFrame** برویسید، در اینجا به کمک کلاس **JFrame** برای تولید فریم‌ها، دکمه‌ها، دکمه‌های رادیویی، جعبه‌های کمبو، لیست‌ها و غیره استفاده کنید. پنجره ساده ایجاد می‌کنیم.



```
In [ ]: import javax.swing.JFrame;

public class TestFrame {
    public static void main(String[] args) {
        JFrame frame1 = new JFrame();
        frame1.setTitle("Window 1");
        frame1.setSize(200, 150);
        frame1.setLocation(200, 100);
```

```
frame1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame1.setVisible(true);

JFrame frame2 = new JFrame();
frame2.setTitle("Window 2");
frame2.setSize(200, 150);
frame2.setLocation(410, 100);
frame2.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame2.setVisible(true);
}

}
```

## نمایش مولفه های گرافیکی (GUI)

```
JFrame frame1 = new JFrame();
frame1.setTitle("Window 1");
frame1.setSize(200, 150);
frame1.setVisible(true); JFrame
frame2 = new JFrame();
frame2.setTitle("Window 2");
frame2.setSize(200, 150);
frame2.setVisible(true);
```

---

این مجموعه تصاویر روند ایجاد دو پنجره گرافیکی را در جاوا نشان می‌دهد. در هر مرحله، یک **متغیر ارجاعی** (reference) مانند frame1 یا frame2 به یک شء

۱) اعلان، ایجاد و انتساب در یک از نوع **JFrame** اشاره می‌کند و با فراخوانی متدهای آن، ویژگی‌های پنجره مانند **عنوان**، **اندازه** و **نمایان بودن** تنظیم می‌شود.

#### دستور

در این لحظه شء **JFrame** ایجاد شده ولی; `JFrame frame1 = new JFrame();` با دستور زیر شء ساخته می‌شود و مرجع آن در متغیر frame1 قرار می‌گیرد:

۲) **تنظیم عنوان** هنوز ویژگی‌هایش مقداردهی نشده است (مثلاً عنوان تهی و اندازه پیش‌فرض است). frame1 صرفاً یک **ارجاع** به این شء در حافظه است.

#### پنجره

۳) تنظیم اندازه پنجره ویژگی **title** در شیء مرتبط با frame1 مقدار «Window 1» می‌گیرد. عرض و ارتفاع پنجره به ترتیب روی ۲۰۰ و ۱۵۰ پیکسل تنظیم می‌شود (ویژگی‌های **width** و **height** شیء **frame1.setSize(200, 150);** فاصله بین پنجره با عنوان است).

۴) قابل نمایش کردن پنجره می‌کند.

۵) ساخت پنجره دوم ویژگی **visible** برابر **true** می‌شود و پنجره frame1 روی صفحه نمایش داده می‌شود. حالا frame2 به شیء جدیدی از نوع **JFrame** اشاره می‌کند که frame2 = new JFrame();

۶) مقداردهی ویژگی‌های پنجره دوم مستقل از پنجره اول است.

عنوان، اندازه و نمایش آن مشابه پنجره اول تنظیم می‌شود:

```
frame2.setTitle("Window 2");  
frame2.setSize(200, 150);
```

نکات پس از این دستورات هر دو پنجره با عنوانین «Window 1» و «Window 2» و اندازه یکسان روی صفحه قابل مشاهده‌اند.

تکمیلی مهم

- توصیه می‌شود برای بستن صحیح برنامه هنگام بستن پنجره از دستور زیر استفاده کنید:

```
frame1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

- در صورت نیاز می‌توانید مکان ظاهر شدن پنجره‌ها را نیز تعیین کنید؛ مثلاً:

```
frame1.setLocation(200, 100);
```

جمع‌بندی: در این روند می‌بینید که یک متغیر ارجاعی مانند frame1 یا frame2 صرفاً **مراجع** شیء است و با متدهای **setVisible** و **setLocation** تغییر می‌کند تا پنجره با عنوان، اندازه و حالت نمایش دلخواه روی صفحه ظاهر شود.

---

```
JFrame frame1 = new JFrame();
frame1.setTitle("Window 1");
frame1.setSize(200, 150);
frame1.setVisible(true); JFrame
frame2 = new JFrame();
frame2.setTitle("Window 2");
frame2.setSize(200, 150);
frame2.setVisible(true);
```

frame1

reference

: JFrame  
title:  
width:  
height:  
visible:

Declare, create,  
and assign in one  
statement

```
JFrame frame1 = new JFrame();
frame1.setTitle("Window 1");
frame1.setSize(200, 150);
frame1.setVisible(true); JFrame
frame2 = new JFrame();
frame2.setTitle("Window 2");
frame2.setSize(200, 150);
frame2.setVisible(true);
```

frame1

reference

: JFrame  
title: "Window 1"  
width:  
height:  
visible:

Set title property

```
JFrame frame1 = new JFrame();
frame1.setTitle("Window 1");
frame1.setSize(200, 150);
frame1.setVisible(true);
JFrame frame2 = new JFrame();
frame2.setTitle("Window 2");
frame2.setSize(200, 150);
frame2.setVisible(true);
```

frame1

reference

: JFrame  
title: "Window 1"  
width: 200  
height: 150  
visible:

Set size property

```
JFrame frame1 = new JFrame();
frame1.setTitle("Window 1");
frame1.setSize(200, 150);
frame1.setVisible(true);

JFrame frame2 = new JFrame();
frame2.setTitle("Window 2");
frame2.setSize(200, 150);
frame2.setVisible(true);
```

frame1

reference

: JFrame  
title: "Window 1"  
width: 200  
height: 150  
visible: true

Set visible  
property

```
JFrame frame1 = new JFrame();
frame1.setTitle("Window 1");
frame1.setSize(200, 150);
frame1.setVisible(true);

JFrame frame2 = new JFrame();
frame2.setTitle("Window 2");
frame2.setSize(200, 150);
frame2.setVisible(true);
```

frame1 reference  
: JFrame  
title: "Window 1"  
width: 200  
height: 150  
visible: true

frame2 reference  
: JFrame  
title:  
width:  
height:  
visible:

Declare, create,  
and assign in one  
statement

```
JFrame frame1 = new JFrame();
frame1.setTitle("Window 1");
frame1.setSize(200, 150);
frame1.setVisible(true);
JFrame frame2 = new JFrame();
frame2.setTitle("Window 2");
frame2.setSize(200, 150);
frame2.setVisible(true);
```

frame1 reference

: JFrame  
title: "Window 1"  
width: 200  
height: 150  
visible: true

frame2 reference

: JFrame  
title: "Window 2"  
width:  
height:  
visible:

Set title property

```
JFrame frame1 = new JFrame();
frame1.setTitle("Window 1");
frame1.setSize(200, 150);
frame1.setVisible(true);
JFrame frame2 = new JFrame();
frame2.setTitle("Window 2");
frame2.setSize(200, 150);
frame2.setVisible(true);
```

frame1 | reference

: JFrame  
title: "Window 1"  
width: 200  
height: 150  
visible: true

frame2 | reference

: JFrame  
title: "Window 2"  
width: 200  
height: 150  
visible:  
Set size property

```
JFrame frame1 = new JFrame();
frame1.setTitle("Window 1");
frame1.setSize(200, 150);
frame1.setVisible(true);
JFrame frame2 = new JFrame();
frame2.setTitle("Window 2");
frame2.setSize(200, 150);
frame2.setVisible(true);
```

frame1 reference

: JFrame  
title: "Window 1"  
width: 200  
height: 150  
visible: true

frame2 reference

: JFrame  
title: "Window 2"  
width: 200  
height: 150  
visible: true

Set visible  
property

اضافه نمودن مولفه های گرافیکی به پنجره

- این مولفه های گرافیکی با استفاده از کلاس های کتابخانه جاوا به سادگی ایجاد می شوند.
- در قطعه کد بعدی نحوه ایجاد این مولفه ها نشان داده شده است.



```
In [ ]: import javax.swing.*;

public class GUIComponents{
    public static void main(String[] args){
        // Create a button with text OK
        JButton btnOK = new JButton("OK");

        // Create a button with text Cancel
        JButton btnCancel = new JButton("Cancel");

        // Create a label with text "Enter your name: "
        JLabel lblName = new JLabel("Enter your Name: ");

        // Create a text field with text "Type Name Here"
        JTextField txtName = new JTextField("Type Name Here");

        // Create a check box with text Bold
        JCheckBox chkBold = new JCheckBox("Bold");

        // Create a check box with text Italic
        JCheckBox chkItalic = new JCheckBox("Italic");

        // Create a radio button with text red
        JRadioButton rdbRed = new JRadioButton("Red");

        // Create a radio button with text yellow
        JRadioButton rdbYellow = new JRadioButton("Yellow");
    }
}
```

```

// Create a combo box with several choices
JComboBox jcmbColor = new JComboBox(new String[]{"Freshman", "Sophomore", "Junior", "Senior"});

// Create a panel to group components
 JPanel panel = new JPanel();
 panel.add(btnOK);           // Add the OK button to the panel
 panel.add(btnCancel);       // Add the Cancel button to the panel
 panel.add(lblName);         // Add the label to the panel
 panel.add(txtName);         // Add the text field to the panel
 panel.add(chkBold);         // Add the check box to the panel
 panel.add(chkItalic);        // Add the check box to the panel
 panel.add(rdbRed);          // Add the radio button to the panel
 panel.add(rdbYellow);        // Add the radio button to the panel
 panel.add(jcmbColor);        // Add the combo box to the panel

JFrame frame = new JFrame(); // Create a frame
frame.add(panel);           // Add the panel to the frame
frame.setTitle("Show GUI Components");
frame.setSize(450, 100);
frame.setLocation(200, 100);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
}
}

```

## متغیرها و متدهای نمونه‌ای

### متغیرهای نمونه (Instance variables)

بسازیم، این متغیرها برای آن شیء ایجاد (**object**) تعریف می‌شوند، اما داخل متدها قرار ندارند. هر وقت از آن کلاس یک شیء **class** متغیرهای نمونه داخل یک ساخته می‌شوند و **heap** این متغیرها روی حافظه‌ی «شیء state» می‌شوند و مقدار مخصوص به خودشان را دارند. به همین دلیل به آن‌ها می‌گوییم «حالت یا

این `obj.field` باقی می‌مانند. دسترسی به آن‌ها از طریق شیء انجام می‌شود، مثلً (پاک نشده **Garbage Collector** و توسط) تا وقتی شیء وجود دارد و حتی مقدار اولیه هم داشته باشند. تفاوت اصلی آن‌ها با متغیرهای **public / private** (مثل) متغیرها می‌توانند سطح دسترسی‌های مختلف داشته باشند. فقط یک نسخه مشترک برای کل کلاس دارد **static** این است که متغیرهای نمونه برای هر شیء یک نسخه جداگانه می‌سازند، در حالی که متغیر **static**

## متدهای نمونه (Instance methods)

متدهای نمونه عملیاتی هستند که روی یک شیء مشخص اجرا می‌شوند. این متدها به متغیرهای نمونه همان شیء دسترسی مستقیم دارند و می‌توانند با برای استفاده از آن‌ها باید متده را روی یک شیء صدابزنیم، مثلً `obj.doSomething()` اگر شیئی وجود به اعضای همان شیء اشاره کنند **this** کلیدواژه: متدهای نمونه کاربردهای زیادی دارند، مثلً فراخوانی کرد **static** نداشته باشد، این متده را نمی‌توان از زمینه در برنامه‌نویسی شیءگرا **polymorphism** در کلاس‌های فرزند، و پیاده‌سازی **override** بازنویسی متدها

## متغیرها ثابت‌ها و متدهای ایستا

وقتی چیزی را در جاوا **static** تعریف می‌کنیم، یعنی آن عضو به خود **class** تعلق دارد، نه به شیء‌هایی که از آن ساخته می‌شوند. پس متغیر یا متده **static** فقط یک نسخه در کل برنامه دارد و همه‌ی شیء‌های آن کلاس از همان نسخه مشترک استفاده می‌کنند. برای مثال، اگر یک متغیر معمولی تعریف کنیم، هر شیء کپی جداگانه‌ای از آن را دارد؛ اما اگر همان متغیر را **static** کنیم، همه‌ی شیء‌ها به همان مقدار واحد دسترسی خواهند داشت. همچنین متدهای **static** نیازی به ساخت شیء ندارند و می‌توان آن‌ها را مستقیماً با نام کلاس صدابزنیم، در حالی که متدهای معمولی باید روی یک شیء اجرا شوند. اگر یک متغیر **static** را همراه با **final** تعریف کنیم، مقدارش ثابت و تغییرناپذیر می‌شود و در تمام شیء‌ها مشترک خواهد بود. برای مشخص کردن **static** بودن کافی است در تعریف متغیر یا متده از کلیدواژه **static** استفاده کنیم.

## مثالی از Static Variables, Constants and Methods

- برنامه‌ای بنویسید و در آن نقش **متغیرهای نمونه** و **متغیرهای کلاسی** را به همراه نحوه استفاده از آن‌ها نشان دهید.
- را حساب کند **Circle** یک واحد اضافه می‌کند تا تعداد اشیای ساخته شده از کلاس **numberOfObjects** این مثال، به یک متغیر کلاسی.

```
In [ ]: public class Circle2 {  
    /** The radius of the circle */  
    double radius;  
  
    /** The number of the objects created */  
    static int numberOfObjects = 0;  
  
    /** Construct a circle with radius 1 */  
    Circle2() {  
        radius = 1.0;  
        numberOfObjects++;  
    }  
  
    /** Construct a circle with a specified radius */  
    Circle2(double newRadius) {  
        radius = newRadius;  
        numberOfObjects++;  
    }  
  
    /** Return numberOfObjects */  
    static int getNumberOfObjects() {  
        return numberOfObjects;  
    }  
  
    /** Return the area of this circle */  
    double getArea() {
```

```
        return radius * radius * Math.PI;
    }
}
```

```
In [ ]: public class TestCircle2 {
    /** Main method */
    public static void main(String[] args) {
        System.out.println("Before creating objects");
        System.out.println("The number of Circle objects is " + Circle2.number0fObjects);

        // Create c1
        Circle2 c1 = new Circle2();

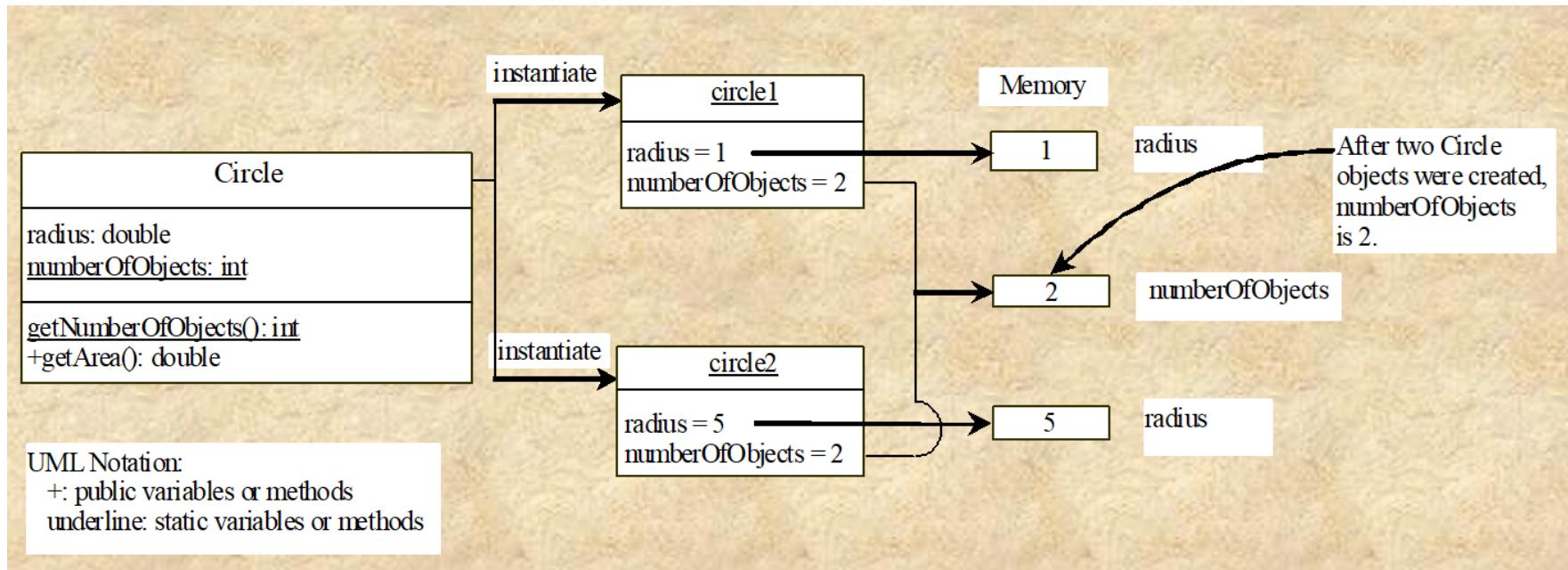
        // Display c1 BEFORE c2 is created
        System.out.println("\nAfter creating c1");
        System.out.println("c1: radius (" + c1.radius + ") and number of Circle objects (" + c1.number0fObjects + ")");

        // Create c2
        Circle2 c2 = new Circle2(5);

        // Modify c1
        c1.radius = 9;

        // Display c1 and c2 AFTER c2 was created
        System.out.println("\nAfter creating c2 and modifying c1");
        System.out.println("c1: radius (" + c1.radius + ") and number of Circle objects (" + c1.number0fObjects + ")");
        System.out.println("c2: radius (" + c2.radius + ") and number of Circle objects (" + c2.number0fObjects + ")");
    }
}
```

نحوه نمایش متغیرهای نمونه و ایستا در حافظه



داریم که **Circle** در سمت چپ کلاسی به نام **Circle** را در حافظه نشان می‌دهد (**instance**) و متغیرهای ایستا (**static**) و متغیرهای نمونه (**instance**) این شکل نحوه تفاوت بین **متغیرهای نمونه** و **متغیرهای ایستا** (static) را در حافظه نشان می‌دهد: شامل موارد زیر است:

- **radius**: یک متغیر نمونه برای هر شیء (دایره) به صورت جداگانه.
- **numberOfObjects**: یک متغیر **static** که به کل کلاس تعلق دارد و بین همه اشیا مشترک است.
- **getNumberOfObjects()** و **getArea()**.

یک واحد زیاد می‌شود. با ساخت **numberOfObjects** شیء آن روی **1** قرار می‌گیرد و مقدار **circle1**, **M** مثلاً، وقتی یک شیء جدید می‌سازیم یک ویژگی اختصاصی هر **radius** نکته کلیدی. دوباره یک واحد افزایش یافته و به **2** می‌رسد **circle2**, **M** می‌شود و **2** شیء دوم یک ویژگی مشترک همه اشیای کلاس است و فقط **یک نسخه** از آن در حافظه نگه داری **numberOfObjects** شیء است (برای هر شیء مقدار جدا دارد)، اما می‌شود. به همین دلیل پس از ساخت دو شیء، مقدار آن **2** می‌شود.

- **هر شیء (state) متغیر نمونه**: داده مخصوص هر شیء.
- داده مشترک بین تمام اشیای یک کلاس **static** متغیر.

## سطح دسترسی به فیلدها و متدها

در برنامه‌نویسی شئ‌گرا، هر متغیر یا متده می‌تواند سطح دسترسی مشخصی داشته باشد تا تعیین شود چه بخش‌هایی از برنامه به آن دسترسی دارند. دو سطح دسترسی پرکاربرد عبارتند از:

- **public:** یعنی متغیر یا متده از هر جای برنامه و توسط هر کلاس دیگری قابل مشاهده و استفاده است.

- **private:** یعنی متغیر یا متده فقط در همان کلاسی که تعریف شده قابل مشاهده است و بیرون از آن کلاس به‌طور مستقیم قابل دسترسی نیست.

استفاده می‌کنیم که به آن‌ها **get** و **set** دسترسی داشته باشیم یا آن‌ها را تغییر بدھیم، از متدهای **private** برای اینکه بتوانیم به داده‌های گفته می‌شود. این متدها امکان کنترل شده‌ای برای خواندن و تغییر مقادیر فراهم می‌کنند.

---

یک نکته مهم!!

```

public class Foo {
    private boolean x;

    public static void main(String[] args) {
        Foo foo = new Foo();
        System.out.println(foo.x);
        System.out.println(foo.convert());
    }

    private int convert(boolean b) {
        return x ? 1 : -1;
    }
}

```

(a) This is OK because object foo is used inside the Foo class

```

public class Test {
    public static void main(String[] args) {
        Foo foo = new Foo();
        System.out.println(foo.x);
        System.out.println(foo.convert(foo.x));
    }
}

```

(b) This is wrong because x and convert are private in Foo.

تعريف می‌شود، یعنی فقط از درون **private** به خوبی نشان داده شده است. وقتی یک متغیر یا متد در جاوا **private** در این تصویر مفهوم سطح دسترسی و یک **x** داریم که یک متغیر **Foo** در بخش سمت چپ، کلاسی به نام **.همان** کلاس قابل دسترسی است و هیچ کلاس دیگری امکان دسترسی مستقیم به آن را ندارد تعریف کرده است. چون دسترسی به این اعضا درون **همان** کلاس انجام می‌شود، کد معتبر است و بدون خطأ اجرا خواهد **convert** را به صورت **private** متد دسترسی پیدا کند. این کار **Foo** از کلاس **Test** و متد **x** سعی کرده به متغیر **convert** در بخش سمت راست، کلاس دیگر به نام **.صحیح** است (a) شد. بنابراین نادرست است و کد به خطأ منجر (b) قابل استفاده هستند. به همین دلیل **Foo** تعریف شده‌اند و فقط درون کلاس **private** اشتباه است زیرا هر دو عضو **private** و **convert** برای محافظت از داده‌ها استفاده می‌شوند و باید تنها از طریق متد‌های **private** نتیجه‌گیری مهم این است که اعضای می‌شود یا دیگر **getter** و **setter** برای **Encapsulation** قابل دسترسی باشند. این ویژگی همان اصل (**public**) متد‌های عمومی در برنامه‌نویسی شیء‌گراست.

چرا باید فیلد‌های داده‌ای **private** باشند؟

تعریف شوند. این کار باعث **محافظت** از داده‌ها می‌شود و همچنین **نگهداری** از کلاس را **private** بهتر است فیلدهای داده‌ای یک کلاس به صورت **Java** در زبان فراهم **getter** و **setter** آسان‌تر می‌کند. علاوه بر این، با جلوگیری از **تغییر مستقیم داده‌ها** توسط کدهای بیرونی، امکان **کنترل دسترسی** از طریق متدهای در برنامه‌نویسی شی‌گرا کمک کرده و باعث افزایش **امنیت** و کاهش احتمال بروز خطا می‌گردد. **Encapsulation** می‌شود. این موضوع به پیاده‌سازی اصل همچنین می‌توان قبل از تغییر مقدار فیلدها، فرآیند **اعتبارسنجی** را انجام داد تا داده‌های نامعتبر وارد سیستم نشوند.

---

```
package p1;
```

```
public class C1 {  
    public int x;  
    int y;  
    private int z;  
  
    public void m1() {  
    }  
    void m2() {  
    }  
    private void m3() {  
    }  
}
```

```
public class C2 {  
    void aMethod() {  
        C1 o = new C1();  
        can access o.x;  
        can access o.y;  
        cannot access o.z;  
  
        can invoke o.m1();  
        can invoke o.m2();  
        cannot invoke o.m3();  
    }  
}
```

```
package p2;
```

```
public class C3 {  
    void aMethod() {  
        C1 o = new C1();  
        can access o.x;  
        cannot access o.y;  
        cannot access o.z;  
  
        can invoke o.m1();  
        cannot invoke o.m2();  
        cannot invoke o.m3();  
    }  
}
```

```
package p1;
```

```
class C1 {  
    ...  
}
```

```
public class C2 {  
    can access C1  
}
```

```
package p2;
```

```
public class C3 {  
    cannot access C1;  
    can access C2;  
}
```

مثالی از کپسوله‌بندی فیلد داده‌ای

The - sign indicates private modifier →

Circle	
<u>-radius: double</u>	The radius of this circle (default: 1.0).
<u>-numberOfObjects: int</u>	The number of circle objects created.
+Circle()	Constructs a default circle object.
+Circle(radius: double)	Constructs a circle object with the specified radius.
+getRadius(): double	Returns the radius of this circle.
+setRadius(radius: double): void	Sets a new radius for this circle.
+getNumberOfObject(): int	Returns the number of circle objects created.
+getArea(): double	Returns the area of this circle.

یا (**Data Fields**) است. کپسولبندی یعنی این که **فیلد های داده ای (Encapsulation)** یکی از اصول مهم برنامه نویسی شیء گرا، کپسولبندی Java در زبان تعریف شوند تا مستقیماً از بیرون کلاس قابل دسترسی نباشند. در عوض برای دسترسی یا تغییر دادن مقدار آنها از **private** همان متغیرهای کلاس، به صورت استفاده می‌کنیم. به این ترتیب داده‌ها درون کلاس ایمن‌تر هستند و برنامه‌نویس می‌تواند کنترل کند که چگونه مقدارها خوانده یا **getter** و **setter** متدهای تعریف شده و برای خواندن آن از متدهای **getRadius()** و برای تغییرش از متدهای **setRadius(double radius)** فیلد **radius** را تغییر داده شوند. مثلاً در مثال استفاده می‌شود. این کار هم امنیت داده را بالا می‌برد و هم کد را منظم‌تر و قابل نگهداری‌تر می‌کند.

```
In [ ]: public class Circle3 {
    /** The radius of the circle */
    private double radius = 1;

    /** The number of the objects created */
    private static int numberOfObjects = 0;

    /** Construct a circle with radius 1 */
    public Circle3() {
        numberOfObjects++;
    }
}
```

```
/** Construct a circle with a specified radius */
public Circle3(double newRadius) {
    radius = newRadius;
    number0f0bjects++;
}

/** Return radius */
public double getRadius() {
    return radius;
}

/** Set a new radius */
public void setRadius(double newRadius) {
    radius = (newRadius >= 0) ? newRadius : 0;
}

/** Return number0f0bjects */
public static int getNumber0f0bjects() {
    return number0f0bjects;
}

/** Return the area of this circle */
public double getArea() {
    return radius * radius * Math.PI;
}
}
```

```
In [ ]: public class TestCircle3 {
    /** Main method */
    public static void main(String[] args) {
        // Create a Circle with radius 5.0
        Circle3 myCircle = new Circle3(5.0);

        System.out.println("The area of the circle of radius "
            + myCircle.getRadius() + " is " + myCircle.getArea());

        // Increase myCircle's radius by 10%
        myCircle.setRadius(myCircle.getRadius() * 1.1);
        System.out.println("The area of the circle of radius "
```

```
+ myCircle.getRadius() + " is " + myCircle.getArea());  
}  
}
```

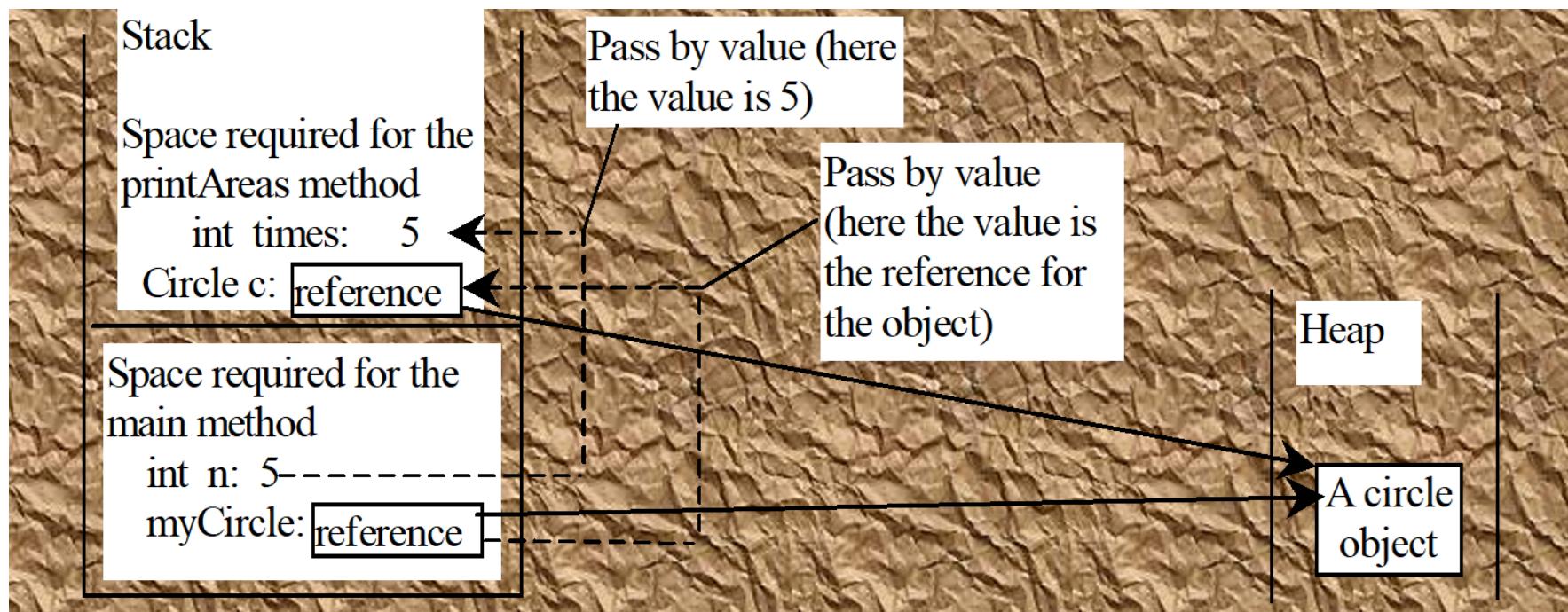
## ارسال اشیا به متدها

یک کپی، int یا double یا مانند int double از مقدار (primitive) برای انواع اولیه (Pass by Value) دو شیوه اصلی برای ارسال پارامتر وجود دارد. در حالت ارسال با مقدار Java در (Reference) از مقدار به متده می‌شود و هر تغییری که داخل متده روی پارامتر انجام شود، روی متغير اصلی اثر نمی‌گذارد. در حالت ارسال با مقدار ارجاع برای اشیا و آرایه‌ها، در واقع کپی ارجاع (آدرس شیء) به متده می‌شود. به همین دلیل اگر ویژگی‌های شیء داخل متده تغییر کنند، این تغییرات روی Value شیء اصلی نیز اعمال خواهد شد، چون هر دو به یک محل در حافظه اشاره می‌کنند. با این حال اگر داخل متده پارامتر ارجاعی دوباره به یک شیء جدید انتساب داده شود، این تغییر فقط در همان متده باقی می‌ماند و مرجع بیرونی همچنان به شیء قبلی اشاره می‌کند. این رفتار باعث می‌شود که تغییر مستقیم روی حالت شیء از متده قابل مشاهده باشد، اما تغییر مرجع آن تنها در محدوده متده اثر کند.

```
In [ ]: public class TestPassObject {  
    /** Main method */  
    public static void main(String[] args) {  
        // Create a Circle object with radius 1  
        Circle3 myCircle = new Circle3(1);  
  
        // Print areas for radius 1, 2, 3, 4, and 5.  
        int n = 5;  
        printAreas(myCircle, n);  
  
        // See myCircle.radius and times  
        System.out.println("\n" + "Radius is " + myCircle.getRadius());  
        System.out.println("n is " + n);  
    }  
  
    /** Print a table of areas for radius */  
    public static void printAreas(Circle3 c, int times) {
```

```
System.out.println("Radius \t\t" + "Area");
while (times >= 1) {
    System.out.println(c.getRadius() + "\t\t" + c.getArea());
    c.setRadius(c.getRadius() + 1);
    times--;
}
}
```

## ارسال اشیا به متدها



به متدها ارسال می‌شوند. این موضوع ممکن است در ابتدای کمی گیج‌کننده به نظر برسد، مخصوصاً **Pass by Value** تمام مقادیر به صورت **Java** در زبان فضای لازم برای اجرای متدها و متغیرهای محلی آن‌ها **Stack** در بخش زمانی که با **اشیا** سر و کار داریم. تصویر بالا دقیقاً همین مفهوم را نشان می‌دهد **myCircle** تعریف شده است. متغیر **myCircle** با مقدار ۰ و یک متغیر به نام **int n** یک متغیر (**main method**) نگهداری می‌شود. برای مثال، در متدهایی که **printAreas** در حقیقت یک فراخوانی می‌شود، دو مقدار به آن پاس داده **Circle** در **Heap** وقتی متدهای **printAreas** و **Circle** به یک شی از نوع (آدرس در حافظه) **reference** در حقیقت یک می‌شوند:

- فضای جدیدی برای آن **Stack** به متند ارسال می‌شود و در **pass by value** که یک عدد صحیح (۰) است. این مقدار مستقیماً به صورت **n** مقدار متغیر ایجاد می‌گردد.
  - کپی شده و در متند **pass by value** است. این آدرس نیز به صورت (آدرس شی در) **heap** که در واقع همان **myCircle** مقدار متغیر **printAreas** می‌گردد **c** در متغیری به نام **printAreas**.

آن است. از آنجایی **reference** حتی وقتی یک شیء به متدهای ارسال می‌شود، چیزی که پاس داده می‌شود خود شیء نیست، بلکه **Java** نکته مهم این است که در اشاره **Heap** پاس داده می‌شود، در واقع دو متغیر (در متدهای اصلی و در متدهای فراخوانی شده) هر دو به یک شی در **by value** نیز به صورت **reference** که این به همین دلیل اگر داخل متدهای داده شده این تغییر را در همان شی واحد در حافظه خواهد شد مقداری از ویژگی‌های شی دایره تغییر کند، پس از بازگشت به متدهای اصلی نیز این تغییرات قابل مشاهده خواهند بود، چون هر دو متغیر به یک **printAreas** به شی جدیدی اشاره داده شود، این تغییر تنها در متدهای باقی می‌ماند و تأثیری روی متغیر **reference** اشاره می‌کنند. اما اگر خود **Circle object** در **Heap** در یک جمع‌بندی کلی متمدد اصلی ندارد:

- انجام می‌شود و تغییر در آن‌ها داخل متدها بر متغیر اصلی ندارد **pass by value** به متدها، همیشه با (**int, double** مانند) ارسال مقادیر اولیه است.
  - به همین خاطر متدهای **heap** (آدرس شی در حافظه) انجام می‌شود، اما چیزی که ارسال می‌شود **pass by reference** ارسال اشیا نیز با می‌تواند روی همان شی اصلی تغییرات اعمال کند.

این تفاوت اساسی باعث می‌شود که دانشجویان تازه‌کار بهتر درک کنند چرا گاهی تغییرات درون متد روی شی اصلی اعمال می‌شود و چرا در مورد انواع داده‌های ساده چنین چیزی رخ نمی‌دهد.

آرایه‌ای از اشیا

تعریف می‌کند که ظرفیت ۱۰ خانه دارد **Circle** می‌توان آرایه‌ای از اشیا ایجاد کرد. دستور زیر یک آرایه از نوع **Java** در زبان:

```
Circle[] circleArray = new Circle[10];
```

ایجاد **Circle** به اشیای (references) ساخته نمی‌شودارجاع‌ها این دستور یک آرایه ایجاد می‌کند، اما توجه داشته باشید که در این لحظه ۱۰ شی از نکات مهم واقعی به آن اختصاص داده شود **Circle** دارد تا زمانی که یک شی **null** می‌گردد. به بیان دیگر، هر خانه از این آرایه در ابتداء مقدار

- اشاره کند **Heap آرایه‌ای از اشیا در واقع آرایه‌ای از متغیرهای ارجاعی** است. یعنی هر عنصر آرایه می‌تواند به یک شی خاص در حافظه ارجاع داده باشد. در غیر **Circle** نوشته می‌شود، ابتدا باید خانه شماره ۱ به یک شی واقعی از نوع **circleArray[1].getArea()** وقتی عبارتی مثل خواهد داد **NullPointerException** این صورت اجرای این دستور خطای
- اشاره می‌کند که شامل ۱۰ خانه (برای ۱۰ **Heap** نگهداری شده و به فضایی در **Stack** یک ارجاع به کل آرایه است، یعنی متغیری که در **circleArray** خود ارجاع) است.
- باقی **null** است. تا زمانی که شی جدیدی به آن تخصیص داده نشود، مقدار آن **Circle** یک ارجاع به یک شی از نوع **circleArray[1]** هر عنصر مانند می‌ماند.

نکات تکمیلی که باید بدانید:

- استفاده کرد. مثلاً **for** برای مقداردهی اولیه عناصر آرایه می‌توان از یک **حلقه**:

```
for (int i = 0; i < circleArray.length; i++) {  
    circleArray[i] = new Circle();  
}
```

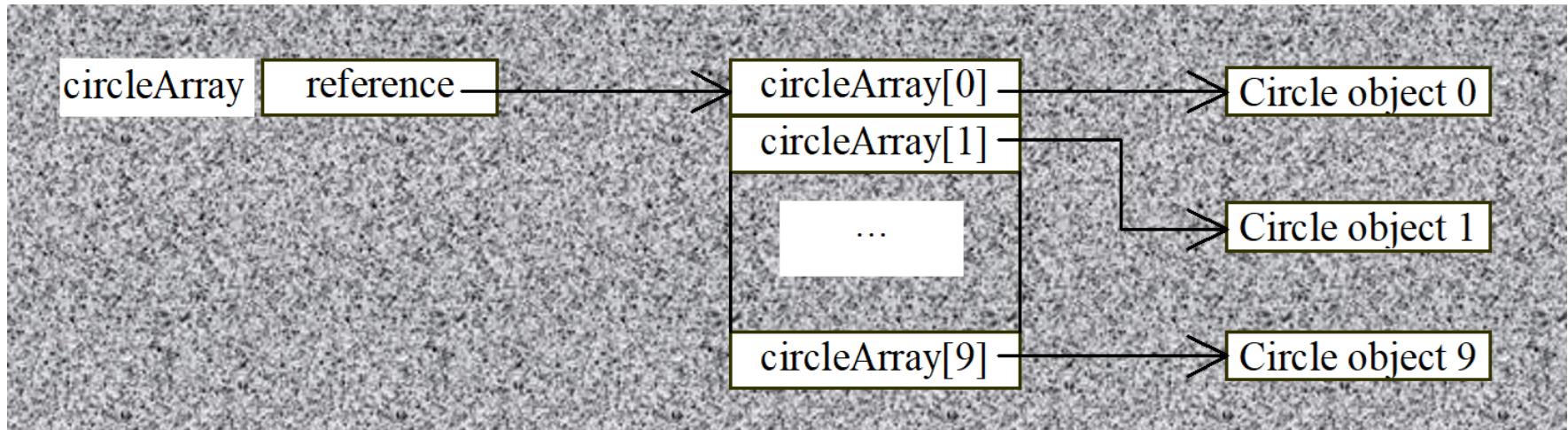
ارجاع دهنده **Circle** این کار باعث می‌شود تمام خانه‌های آرایه به اشیای جدیدی از نوع

ساخته می‌شود، تعداد خانه‌های آن دیگر قابل تغییر نیست **new Circle[10]** اندازه ثابت دارند. یعنی وقتی آرایه‌ای مثل **Java** آرایه‌ها در

در سطح مفهومی، می‌توان گفت که در اینجا دو سطح ارجاع داریم

- یک ارجاع برای کل آرایه (**circleArray**).
- اشاره کنند **Heap** در حافظه **Circle** چند ارجاع داخل آرایه که هر کدام می‌توانند به یک شی

بسیار پرکاربرد هستند، اما باید همواره به این نکته توجه داشت که صرف ایجاد آرایه، باعث ایجاد اشیای داخل آن نمی‌شود، بلکه Java در نتیجه، آرایه‌های اشیا در تنها ظرفی برای نگهداری ارجاع‌ها ساخته می‌شود.



```
In [ ]: public class TotalArea {
    /** Main method */
    public static void main(String[] args) {
        // Declare circleArray
        Circle3[] circleArray;

        // Create circleArray
        circleArray = createCircleArray();

        // Print circleArray and total areas of the circles
        printCircleArray(circleArray);
    }

    /** Create an array of Circle objects */
    public static Circle3[] createCircleArray() {
        Circle3[] circleArray = new Circle3[5];

        for (int i = 0; i < circleArray.length; i++) {
            circleArray[i] = new Circle3(Math.random() * 100);
        }
    }
}
```

```
        }
        // Return Circle array
        return circleArray;
    }

/** Print an array of circles and their total area */
public static void printCircleArray(Circle3[] circleArray) {
    System.out.printf("%-30s%-15s\n", "Radius", "Area");
    for (int i = 0; i < circleArray.length; i++) {
        System.out.printf("%-30f%-15f\n",
            circleArray[i].getRadius(),
            circleArray[i].getArea());
    }

    System.out.println("-----");

    // Compute and display the result
    System.out.printf("%-30s%-15f\n", "The total areas of circles is",
        sum(circleArray));
}

/** Add circle areas */
public static double sum(Circle3[] circleArray) {
    // Initialize sum
    double sum = 0;

    // Add areas to sum
    for (int i = 0; i < circleArray.length; i++)
        sum += circleArray[i].getArea();

    return sum;
}
}
```

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.

۱) خروجی منطقی: با توجه به توضیحات، اجرای دستور زیر بدون مقداردهی عناصر آرایه چه خطای می‌دهد؟ ( فقط نام خط را بنویسید)

```
Circle[] arr = new Circle[3];  
double a = arr[0].getArea(); // ?
```

» ساخته می‌شود **new Circle[10]** ده شیء از نوع **Circle** درست/نادرست: جمله زیر را بررسی کنید و دلیل کوتاه بنویسید: «با اجرای دستور (۱) چیست؟ **circles** جای خالی را پر کنید: پس از اجرای کد زیر، مقدار پیش‌فرض هر خانه آرایه (۳)

```
Circle[] circles = new Circle[5];
```

۴) چند انتخابی: کدام گزینه بهترین توصیف از «دو سطح ارجاع» در متن است؟

- و هیچ ارجاعی به خود آرایه وجود ندارد Heap یک ارجاع به هر شیء در (A).
- اشاره می‌کنند Heap یک ارجاع برای کل آرایه + چند ارجاع داخل آرایه که هر کدام به یک شیء در (B).
- ایجاد نمی‌شود Heap و هیچ چیز در Stack فقط متغیرهای محلی در (C).
- هستند (by value) آرایه و عناصر آن هر دو مقداردهی مقداری (D).

۵) های جدید مقداردهی شوند **Circle** را طوری کامل کن که همه خانه‌ها با for تکمیل کد: با توجه به اصول گفته شده، حلقة

```
Circle[] circleArray = new Circle[10];  
for (int i = 0; i < circleArray.length; i++) {  
    // TODO: assign a new Circle to each slot  
}
```

۶) را فراخوانی می‌کند؟ **Circle** پیش‌بینی نتیجه: کد زیر چند بار سازنده کلاس

```
Circle[] a = new Circle[4];  
a[0] = new Circle();
```

```
a[2] = new Circle();
```

۷) را «متغیر ارجاعی» می‌نامیم؟ در یک یا دو جمله توضیح بده **circleArray[1]** مفهومی: چرا عبارت

۸) قابل تغییر است **new Circle[10]** درست/نادرست: اندازه آرایه در جاوا پس از ساختن با دستور

۹) اشکالیابی: مشکل این کد چیست و چگونه آن را رفع می‌کنی؟

```
Circle[] arr = new Circle[2];  
System.out.println(arr[1].getArea());
```

۱۰) تعریف شده است **Circle** در کلاس (**getArea()** کدنویسی کوتاه: تابعی بنویس که مساحت همه دایره‌های یک آرایه را جمع بزند. فرض کن متدها

```
// write a static method sumAreas(Circle[] arr) that returns a double
```

## پایان

در صورت هرگونه سوال یا پیشنهاد میتوانید با من در

ارتباط باشید:

mr.mohamad.hoseini05@gmail.com telegram:

@MHosseiniR