به نام خدا

# برنامه‌سازی پیشرفته

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؛ به‌ویژه در پروژه‌های بزرگ این روش باعث **افزایش پیچیدگی** و **کاهش فهم بهتر کد** می‌شود.

**شیءگرایی (OOP – Object Oriented Programming)** یکی از مهم‌ترین مفاهیم زبان‌های برنامه‌نویسی مانند **جاوا** است. این مفهوم باعث **ساختارمند شدن برنامه** شده و نگهداری و تغییر کد را ساده‌تر می‌کند.

# آشنایی با مفهوم شی‌گرایی

برنامه‌نویسی **شی‌گرا** به معنی برنامه‌نویسی با استفاده از اشیا می‌باشد.

یک شی بیانگر یک موجودیت در دنیای واقعی است که می‌تواند هویت مستقلی داشته باشد.

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

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

- جواب این سوال بله است. هر شی‌ای 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.

در شیوه **برنامه‌نویسی شی‌گر** برنامه از **اشیا** ساخته می‌شود. در درون هر شی **داده‌ها (Object)** و **متدهای مربوطه** قرار می‌گیرند. این متدها بر روی **داده‌های همان شی** دستکاری انجام می‌دهند و داده‌های اشیای دیگر را تغییر نخواهند داد.

در رابطه با **برنامه‌نویسی شی‌گرا** و **اشیا** یک سری نکات قابل ملاحضه می‌باشد:

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

- **نام**
- **نام خانوادگی**
- **تاریخ تولد**

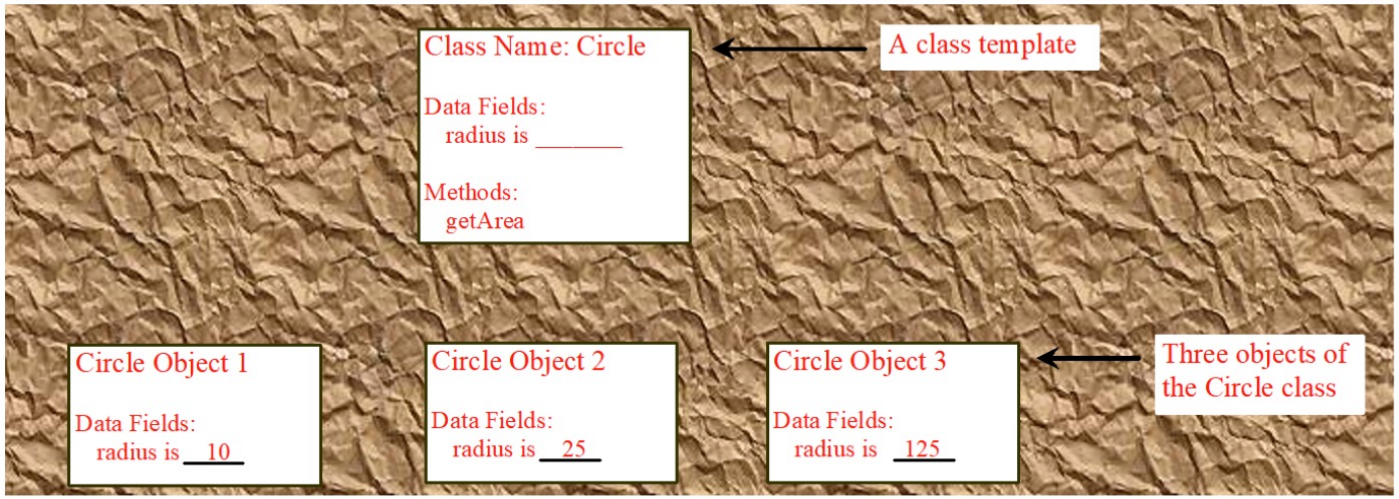
همچنین برای یک انسان یکسری **عملیات** مانند عملیات زیر قابل تعریف می‌باشد:

- **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 شده‌است.

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

# اشیا



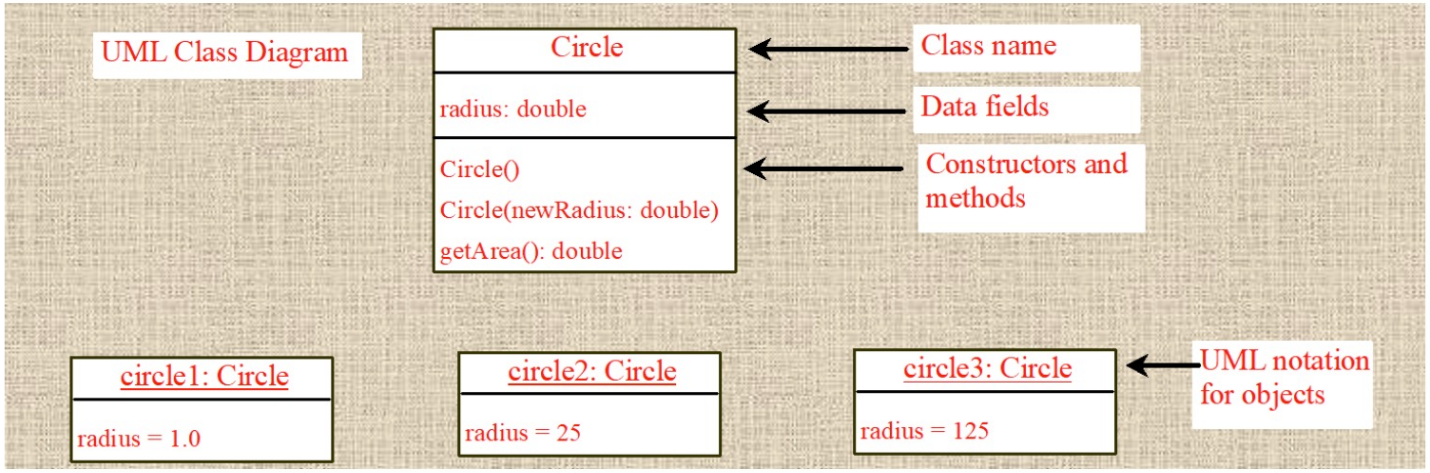

- یک شیئی هم دارای **حالت** و هم رفتار است.
- حالت، تعریف‌کننده **وضعیت** یک شیئی بوده و رفتار میگوید آن شیئی می‌تواند **چه کارهایی** انجام بدهد.

# 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.

```java
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;
    }
}
```

# کلاس‌ها




تصویر بالا یک **نمودار UML کلاس (UML Class Diagram)** برای کلاس **Circle** و چند **شیء (Object)** ساخته‌شده از آن را نشان می‌دهد. این نوع نمودار برای نمایش ساختار یک کلاس، ویژگی‌ها (**Data Fields**) و عملیات یا همان **متدها (Methods)** به کار می‌رود.

# کلاس و اجزای آن

در وسط تصویر یک مستطیل سه‌بخشی با نام **Circle** دیده می‌شود که نشان‌دهنده‌ی **کلاس** است.

- بخش بالایی شامل **Class Name** است: در اینجا نام کلاس **Circle** است.
- بخش میانی شامل **Data Fields (فیلدهای داده‌ای)** است: در تصویر مشخص شده که کلاس **Circle** یک ویژگی یا فیلد به نام **radius** از نوع **double** دارد.
- بخش پایینی شامل **Constructors and Methods (سازنده‌ها و متدها)** است:
  - **Circle()**: سازنده‌ی پیش‌فرض (Constructor)
  - **Circle(newRadius: double)**: سازنده‌ای که می‌تواند شعاع را هنگام ایجاد شیء مشخص کند.
  - **getArea(): double**: متدی برای محاسبه‌ی مساحت دایره.

# اشیاء ساخته‌شده از کلاس

در پایین تصویر سه مستطیل دیده می‌شود که نشان‌دهنده‌ی **اشیاء (Objects)** هستند. هرکدام با **نماد UML برای شیء** نمایش داده شده‌اند:

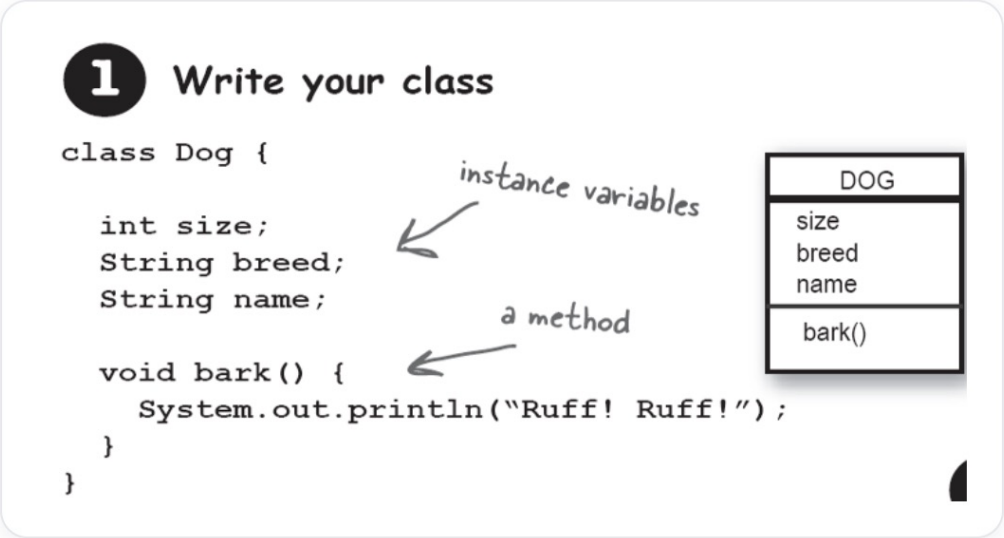
- **circle1: Circle** : دارای مقدار **radius = 1.0**
- **circle2: Circle** : دارای مقدار **radius = 25**
- **circle3: Circle** : دارای مقدار **radius = 125**

این اشیاء نمونه‌هایی از کلاس **Circle** هستند و هرکدام داده‌های مخصوص به خود دارند.

# مفاهیم کلیدی

- **Class (کلاس):** قالب یا نقشه‌ای برای ایجاد اشیاء.
- **Object (شیء):** نمونه‌ای ساخته‌شده از کلاس که داده‌های مخصوص به خود دارد.
- **Data Fields (فیلدهای داده‌ای):** متغیرهایی که ویژگی‌های شیء را ذخیره می‌کنند.
- **Constructors (سازنده‌ها):** توابع ویژه برای ساخت و مقداردهی اولیه به شیء.
- **Methods (متدها):** رفتارها یا عملیات‌هایی که شیء می‌تواند انجام دهد.
- **UML Notation for Objects:** شیوه‌ی استاندارد نمایش اشیاء در نمودار UML.

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



## 2 Write a tester (TestDrive) class

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

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

## 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();
     d.size = 40;
     d.bark();
   }
}
```

make a Dog object

use the dot operator (.) to set the size of the Dog

and to call its bark() method

dot operator

# نوشتن کلاس اصلی

در اولین مرحله یک **کلاس** به نام **Dog** تعریف می‌کنیم. این کلاس شامل **متغیرهای نمونه (Instance Variables)** و یک **متد** است.

متغیرهای نمونه شامل **breed**، **size** و **name** هستند که ویژگی‌های سگ را مشخص می‌کنند.

همچنین در این کلاس یک **متد** به نام **bark()** تعریف شده که وظیفه‌اش چاپ کردن صدای سگ در خروجی است.

این مرحله نشان می‌دهد که چگونه یک کلاس می‌تواند هم داده‌ها (ویژگی‌ها) و هم رفتارها (متدها) را در خود جای دهد.

# ایجاد کلاس تست (TestDrive)

در دومین مرحله یک **کلاس تست** به نام **DogTestDrive** ساخته می‌شود.

این کلاس شامل **متد main** است که نقطه شروع اجرای برنامه محسوب می‌شود. در این مرحله هنوز کدی داخل متد main قرار نمی‌دهیم و فقط ساختار کلی کلاس را آماده می‌کنیم.

هدف از این کلاس تست، آزمایش کردن کلاس Dog و اطمینان از درستی عملکرد آن است.

# ساخت شیء و استفاده از متغیرها و متدها

در این مرحله به سراغ کلاس **DogTestDrive** برمی‌گردیم و در داخل **متد main** یک **شیء** از کلاس Dog می‌سازیم.

برای ساخت شیء از دستور **new** استفاده می‌کنیم و شیء ساخته‌شده را در یک متغیر به نام **d** ذخیره می‌کنیم.

با استفاده از **عملگر نقطه (dot operator)** می‌توانیم به ویژگی‌ها و متدهای شیء دسترسی پیدا کنیم. برای مثال، با دستور **d.size = 40;** اندازه سگ را تنظیم می‌کنیم و با دستور **d.bark();** متد bark() را فراخوانی می‌کنیم تا سگ صدای خود را تولید کند.

# مثالی از تعریف کلاس و ایجاد اشیا

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

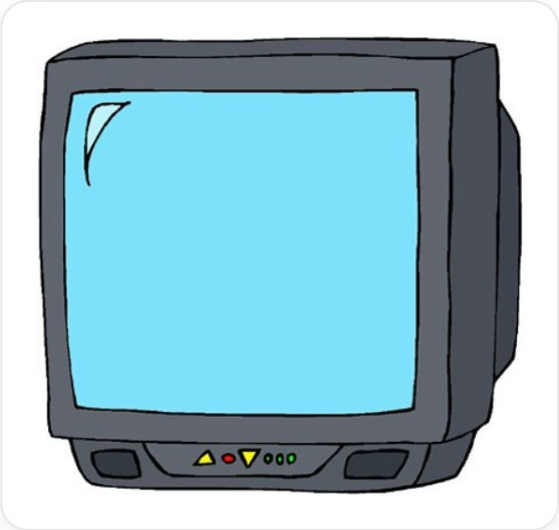
```java
// Define the circle class with two constructors
class Circle1 {
    double radius;

    /** 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;
    }
}
```

```java
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());
    }
}
```

```
The area of the circle of radius 5.0 is 78.53981633974483
The area of the circle of radius 1.0 is 3.141592653589793
The area of the circle of radius 100.0 is 31415.926535897932
```

# 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.



```java
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;
    }
```

```java
    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--;
    }
}
```

```java
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);
    }
}
```

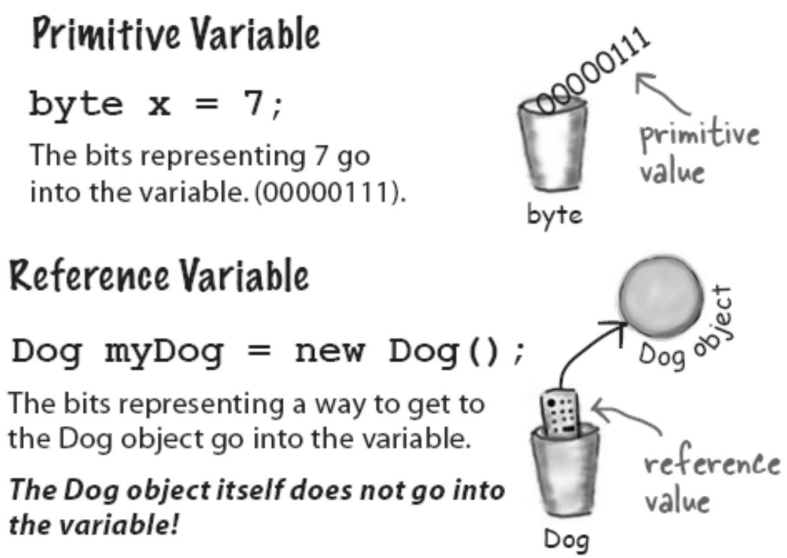
```
tv1's channel is 30 and volume level is 3
tv2's channel is 3 and volume level is 2
```

# نگاهی عمیق‌تر به چگونگی ایجاد اشیا

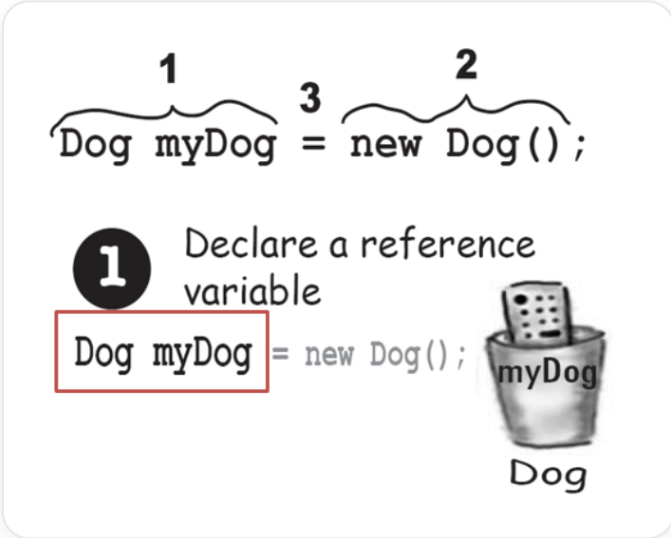
ما با انواع اصلی (Primitive) آشنا شدیم.

اما در جاوا انواع **ارجاعی (refrence types)** هم داریم.
متغیرهای **ارجاعی** به جای نگه‌داشتن 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.

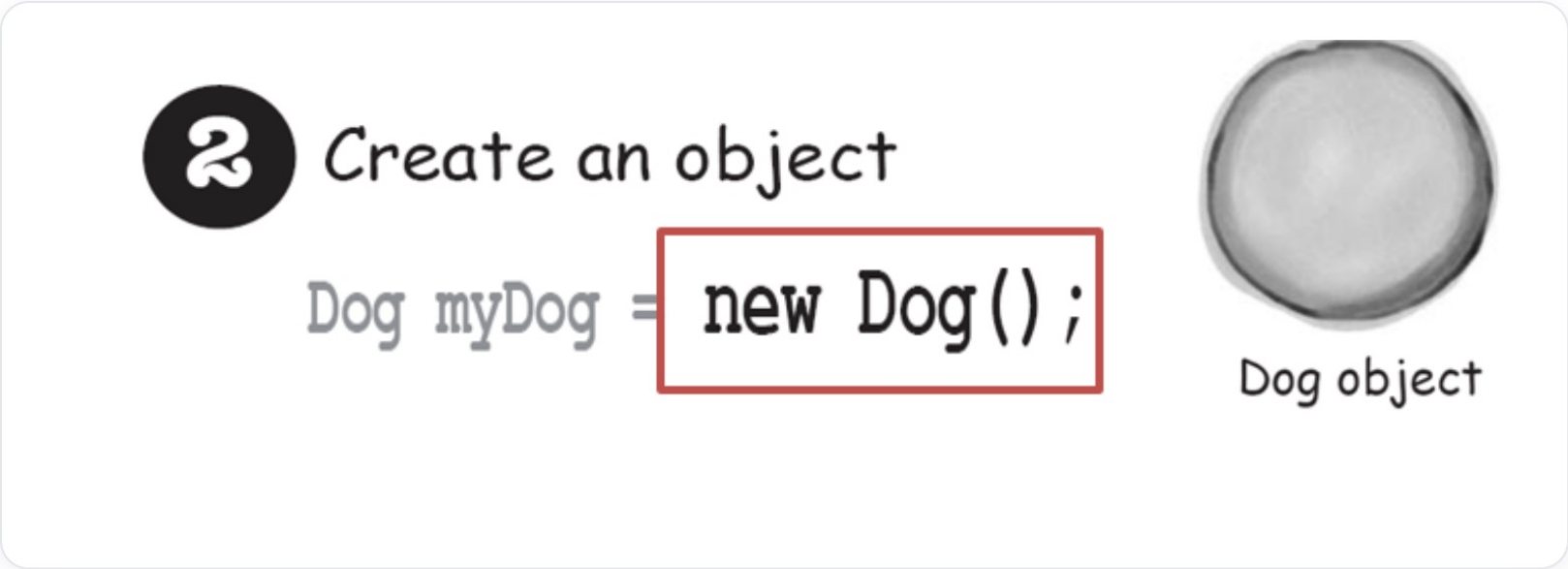
**نکته:** اگر دو متغیر به یک شی اشاره کنند، تغییر یکی روی دیگری هم اثر می‌گذارد.



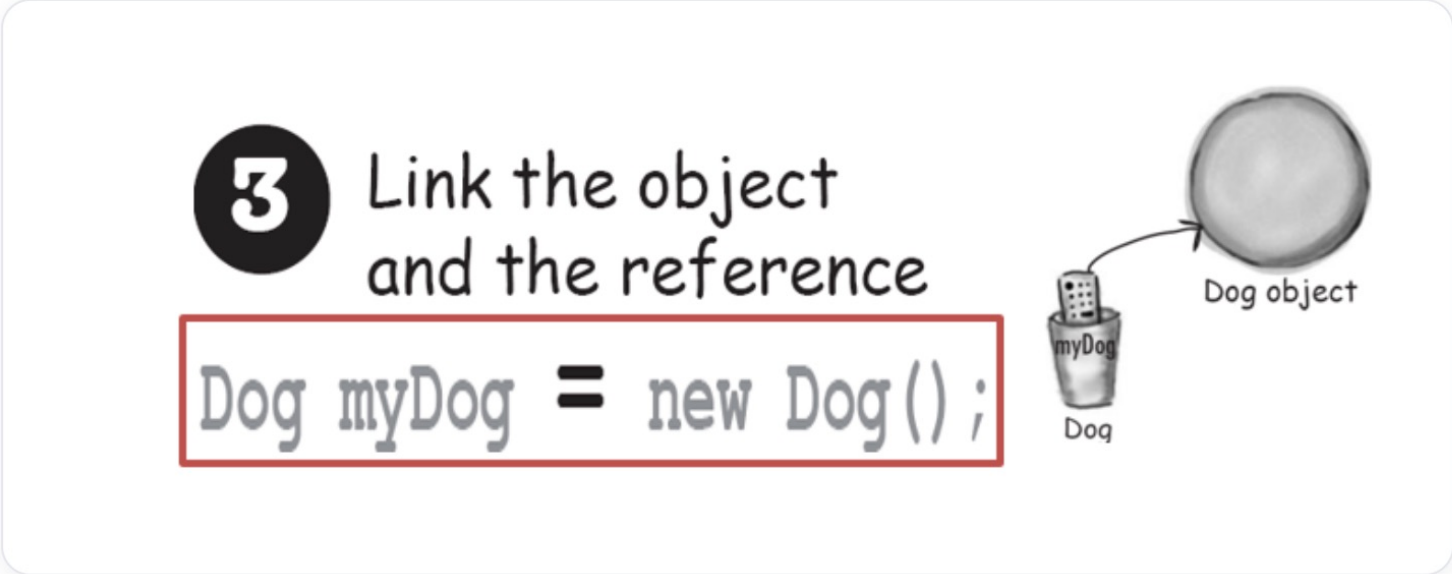
# مراحل ایجاد شی جدید




دستور نشان داده شده در کادرقرمزبه **JVM** می گوید تا فضایی را برای **متغیرارجاعی** اختصاص دهد و نام این متغیر را **myDog** بگذارد. این **متغیرارجاعی** تا ابد از نوع **Dog** خواهد بود. یعنی، ریموت کنترلی که دکمه هایی برای کنترل یک سگ دارد، اما نمی تواند یک **گربه**، یک **دایره** یا یک **ربات** را کنترل کند.

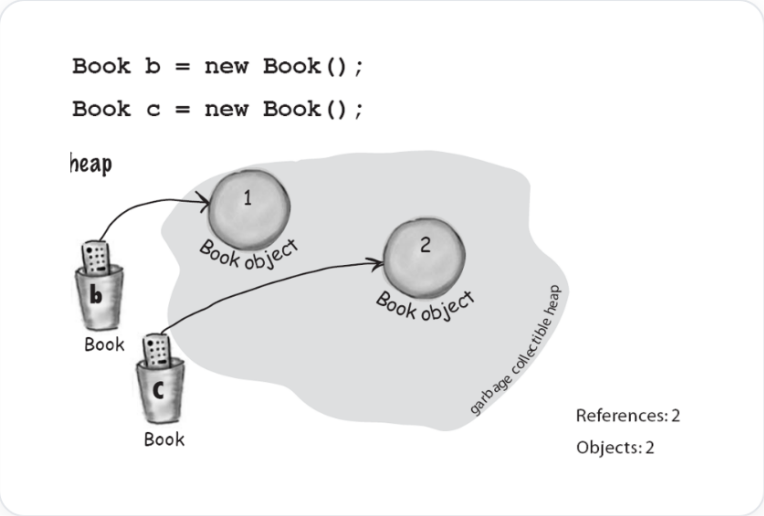



دستور نشان داده شده در کادرقرمزبه **JVM** می گوید تا فضایی را برای شیئ جدید **Dog** در درون **heap** اختصاص دهد.




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

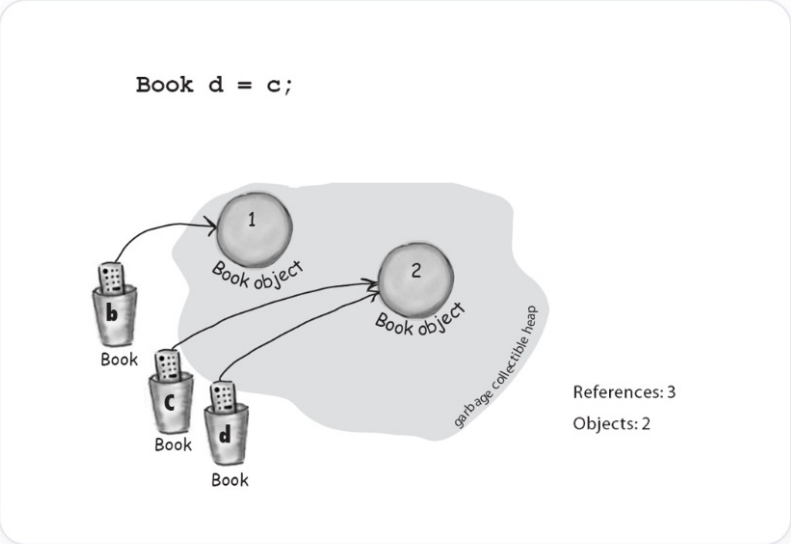
# مثال بیشتر




وقتی در جاوا می‌نویسیم `Book b = new Book();` و `Book c = new Book();` ، دو **متغیر ارجاعی** به نام‌های **b** و **c** ساخته می‌شوند.

هر کدام از این متغیرها با استفاده از دستور `new Book();` یک **شیء جدید** از نوع **Book** را در حافظه **heap** ایجاد می‌کنند. در نتیجه، دو شیء جداگانه در **heap** وجود دارد؛ متغیر **b** به شیء اول و متغیر **c** به شیء دوم اشاره می‌کند.

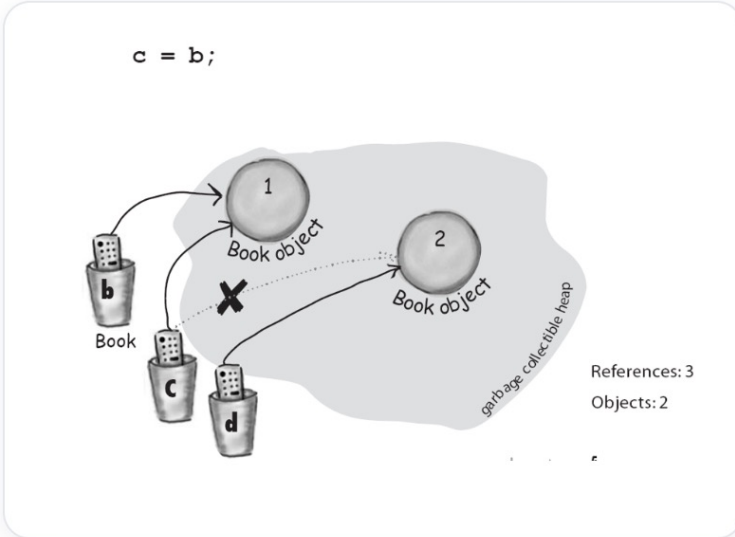
بنابراین، **تعداد ارجاع‌ها (References)** برابر با ۲ و **تعداد اشیاء (Objects)** نیز برابر با ۲ خواهد بود. نکته مهم این است که هر بار که از دستور `new` استفاده می‌کنیم، یک شیء جدید در **heap** ساخته می‌شود، حتی اگر نوع آن یکسان باشد.




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 (Objects)** وجود دارد، اما تعداد **ارجاع‌ها (References)** به ۳ افزایش پیدا می‌کند.




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.

# 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 ساخته‌شده 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 **سربارگذاری 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();
```

مثال

Assign object reference

Create an object

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

دستور `ClassName objectRefVar = new ClassName();` نشان می‌دهد که در **جاوا** وقتی یک **شیء** ساخته می‌شود، همزمان یک **متغیر ارجاعی (Reference Variable)** تعریف و به آن **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 ایجادشده در **heap** دسترسی داشته باشیم و آن را کنترل کنیم.

# دسترسی به اشیا

**ارجاع (دسترسی) به داده‌های درون شیء:**

```
objectRefVar.data
```

مثال: `myCircle.radius`

**فراخوانی متد درون یک شیء:**

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

مثال: `myCircle.getArea()`

در **جاوا**، برای دسترسی به **ویژگی‌ها (Data Fields)** و **متدها (Methods)** از **مرجع شیء (Object Reference)** استفاده می‌کنیم. **ویژگی‌ها** مقادیر را ذخیره می‌کنند و **متدها** رفتار **شیء** را مشخص می‌کنند. به این ترتیب می‌توانیم هم **داده‌های شیء** را بخوانیم و هم **عملیات خاصی** را روی آن انجام دهیم.

# دسترسی به اشیا

وقتی دستور **Circle myCircle = new Circle(5.0);** اجرا می‌شود، بخش سمت چپ یعنی **Circle myCircle** تنها یک **متغیر ارجاعی (Reference Variable)** (مثل یک ریموت کنترل) را اعلان می‌کند که در ابتدا به هیچ **شیء**ی متصل نیست و مقدارش no value است.

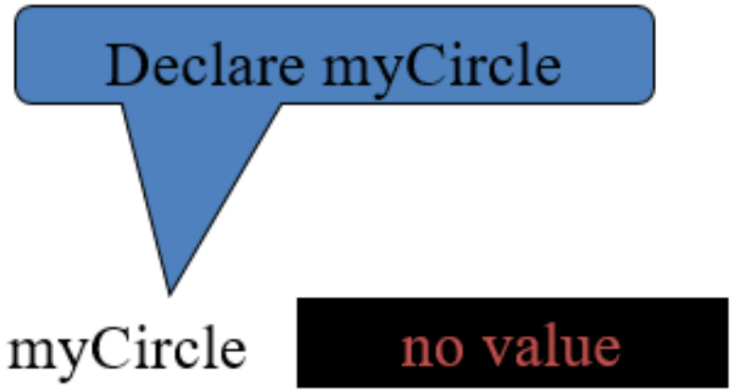
بخش سمت راست یعنی **new Circle(5.0)** یک **شیء جدید** از کلاس **Circle** در حافظه **heap** ایجاد می‌کند که مقدار اولیه‌ی **radius** آن برابر با **۵٫۰** است. سپس عملگر **=** باعث می‌شود مرجع آن **شیء** به **myCircle** اختصاص داده شود و از این لحظه، **myCircle** به آن **شیء** در حافظه اشاره می‌کند.

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** از کلاس **Circle** با مقدار پیش‌فرض (مثلاً **۱٫۰** یا **۰٫۰** بسته به سازنده) می‌سازد و مرجع آن به **yourCircle** داده می‌شود.

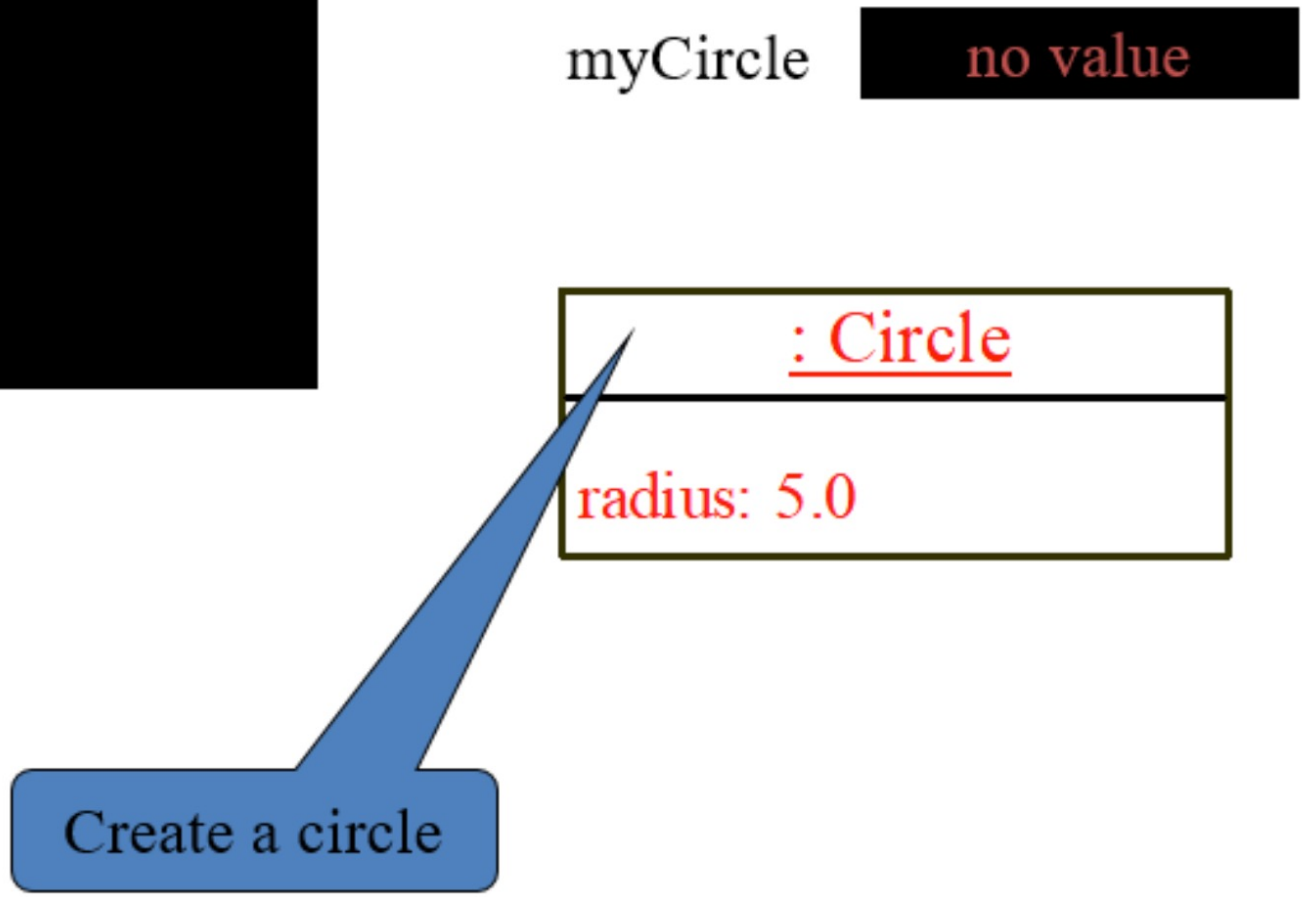
حالا هر **متغیر ارجاعی** به یک **شیء جداگانه** وصل است. در نهایت، دستور **yourCircle.radius = 100;** مقدار **radius** شیئی که **yourCircle** به آن اشاره می‌کند را به **۱۰۰** تغییر می‌دهد، در حالی که **شیء** مربوط به **myCircle** هیچ تغییری نمی‌کند و مقدار **۵٫۰** خودش را حفظ می‌کند.

به طور خلاصه: **اعلان متغیر** فقط یک **ریموت خالی** می‌سازد، **new** یک **شیء جدید** ایجاد می‌کند و عملگر **=** آن‌ها را به هم متصل می‌کند؛ سپس هر تغییری روی **فیلدها (Fields)** فقط روی همان **شیء**ی اثر دارد که **متغیر ارجاعی** به آن اشاره می‌کند.

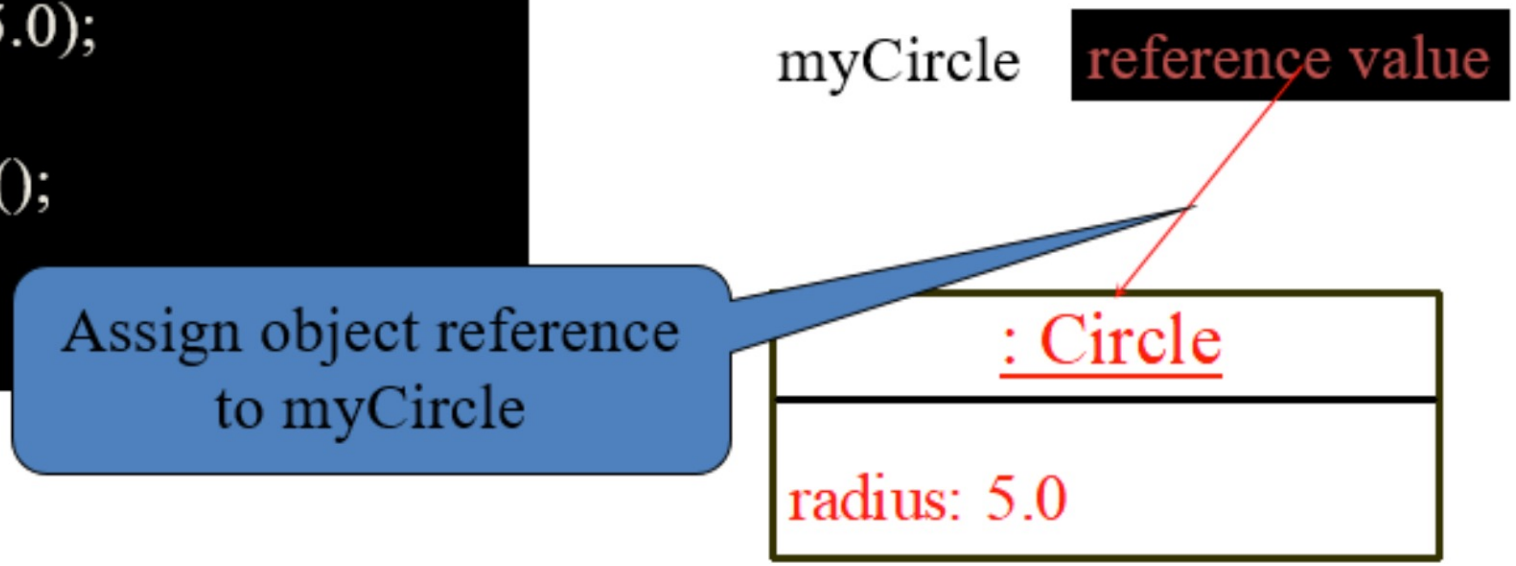
```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



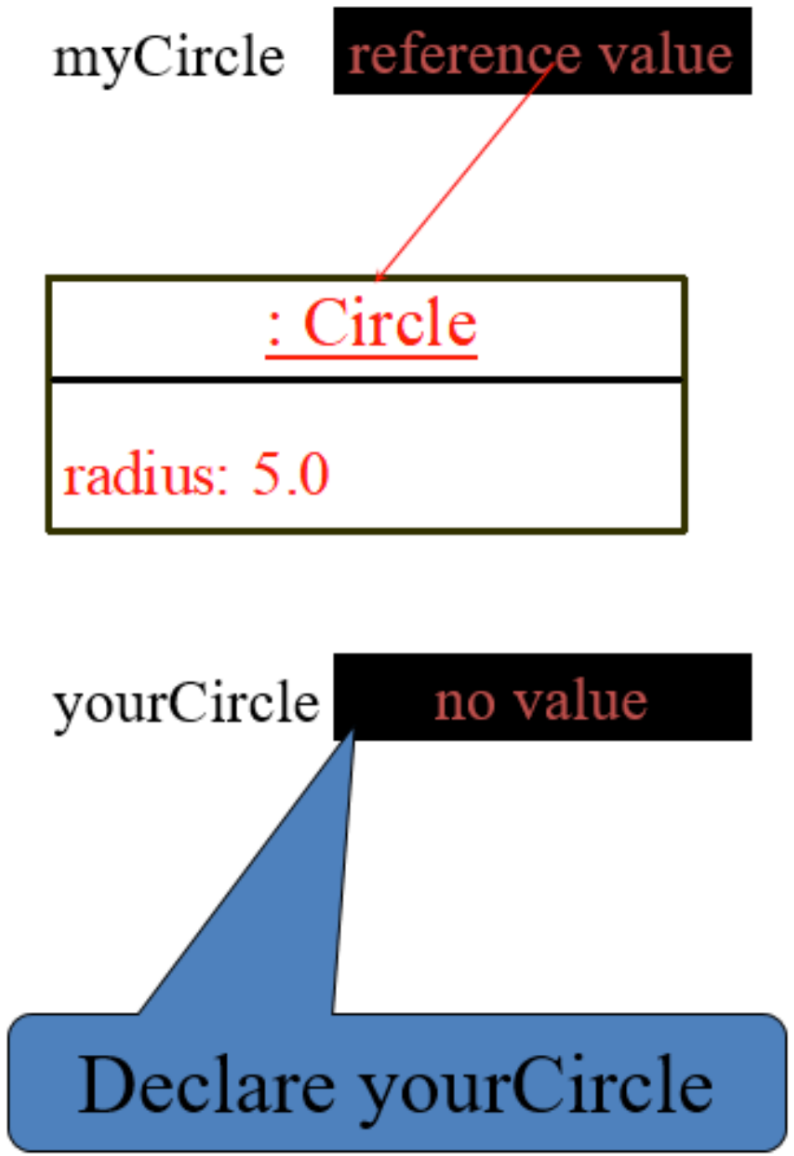
```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



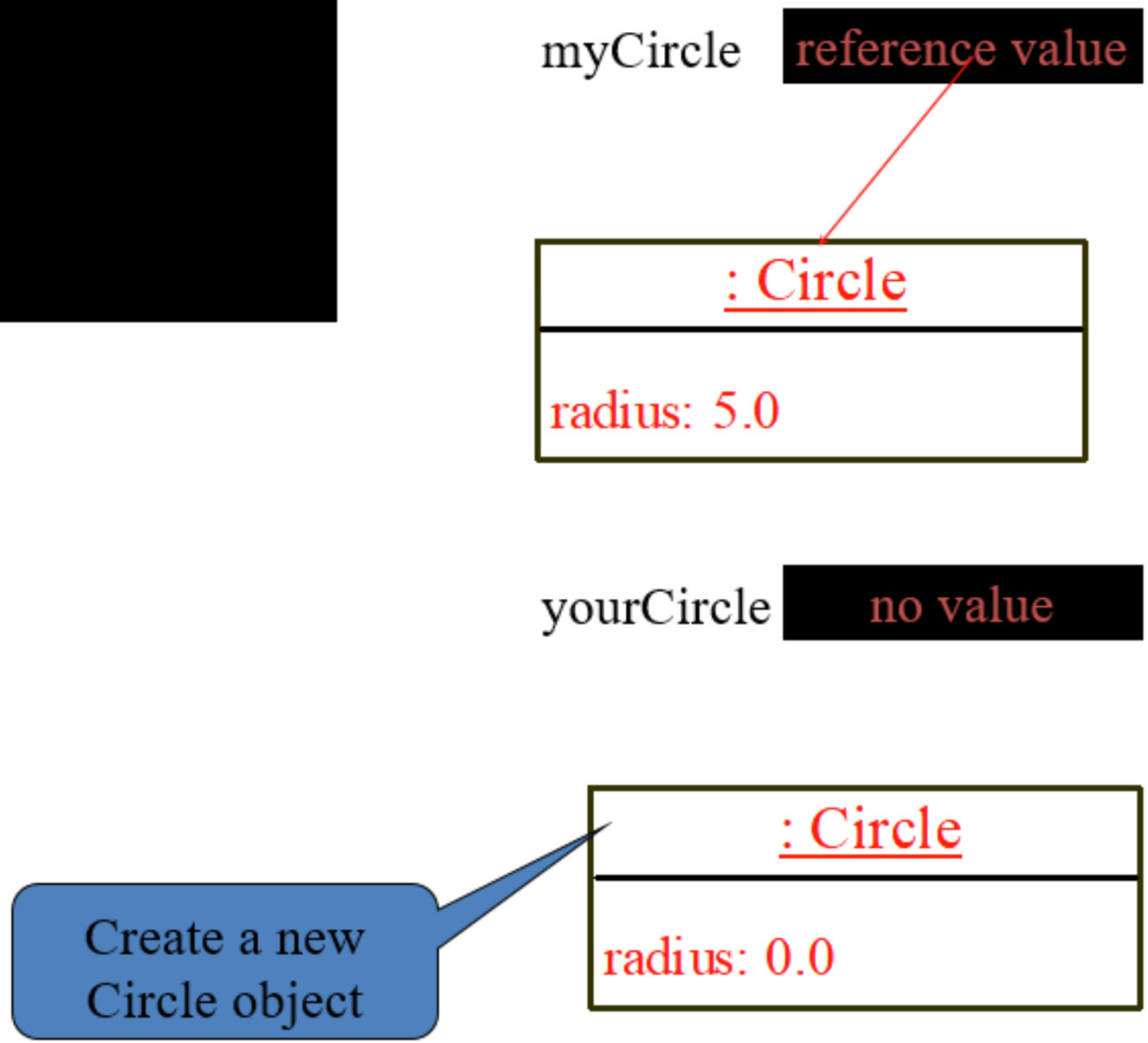
```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



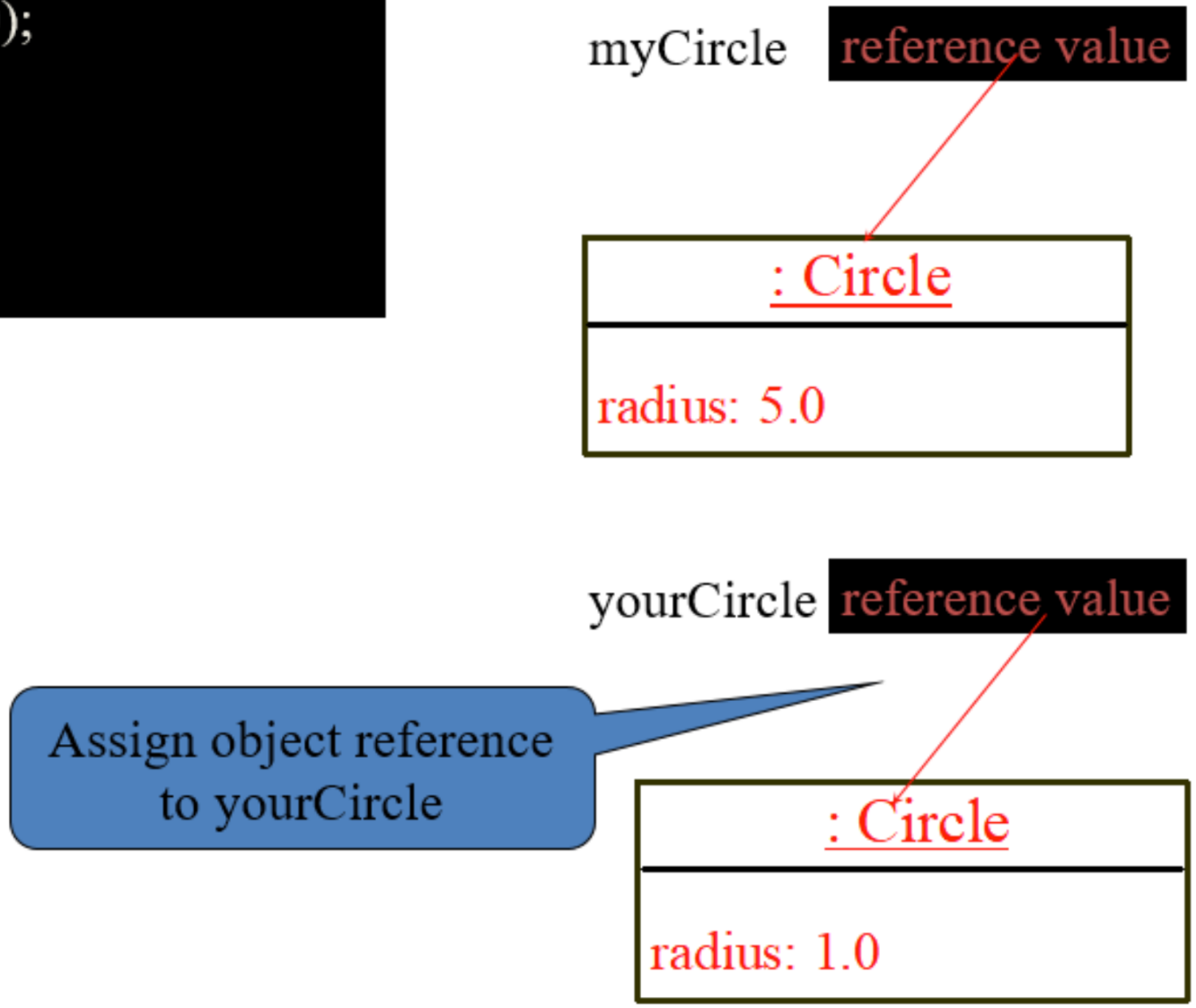
```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



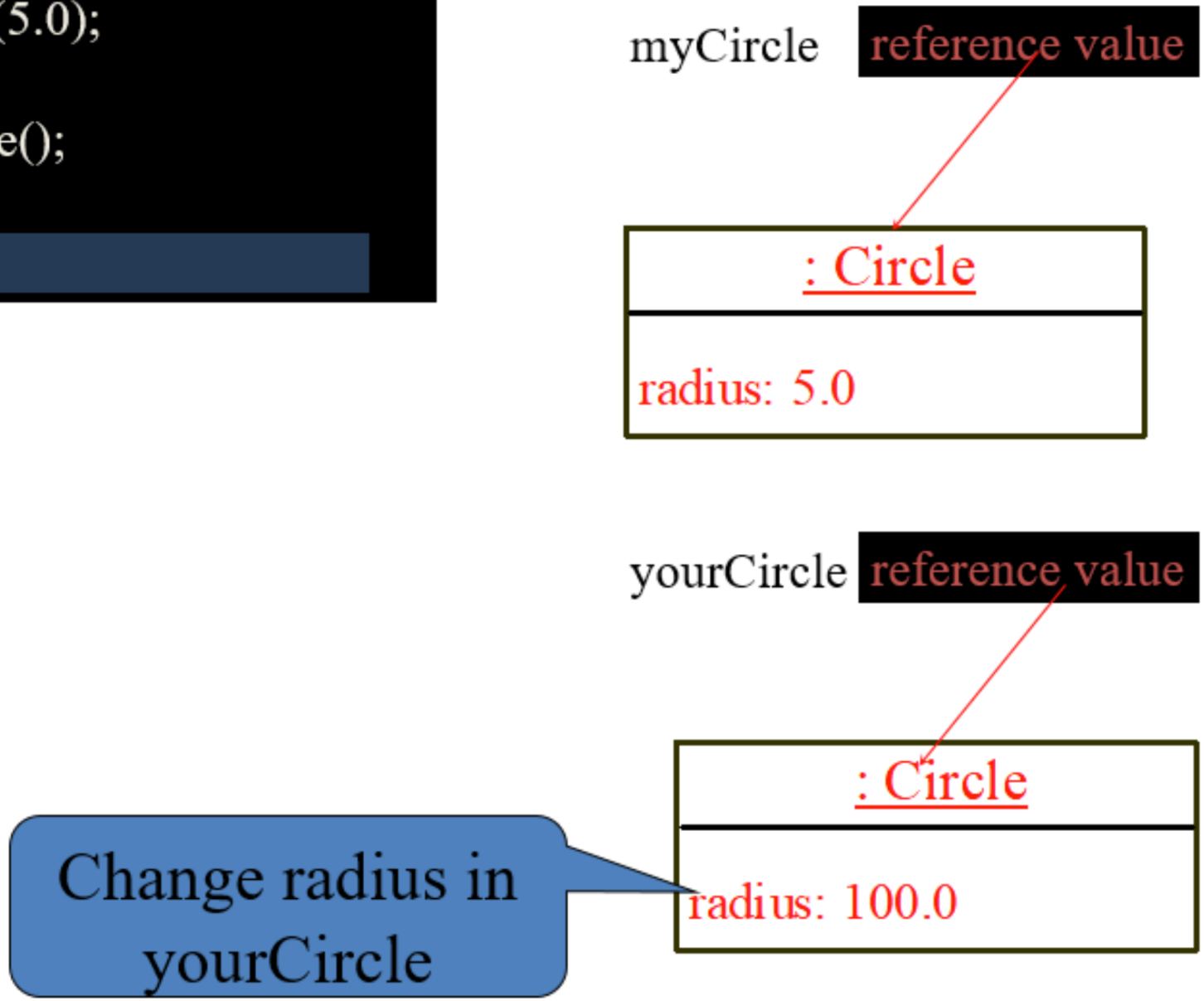
```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



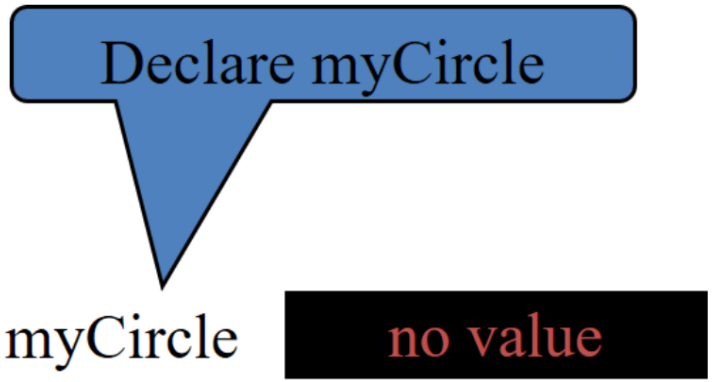
```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



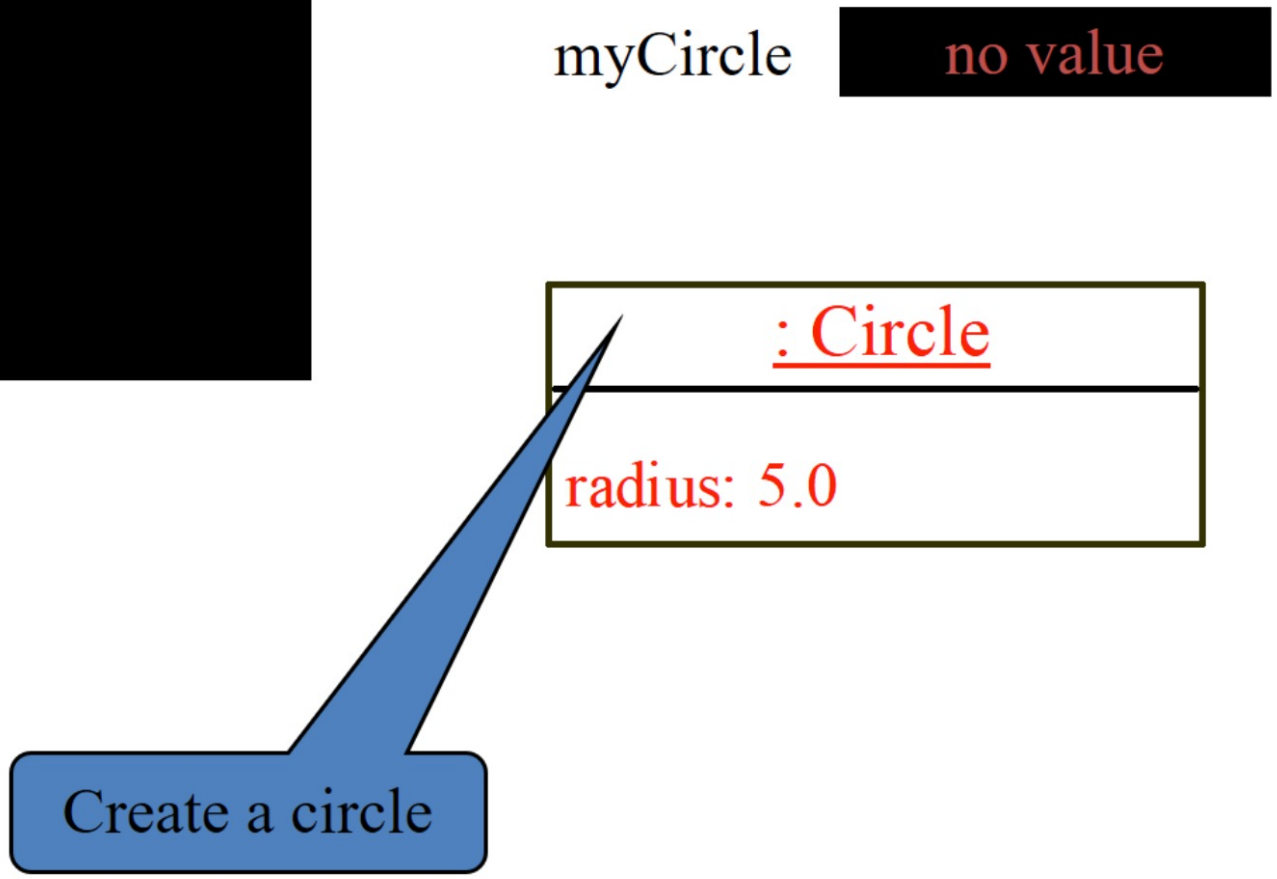
```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```

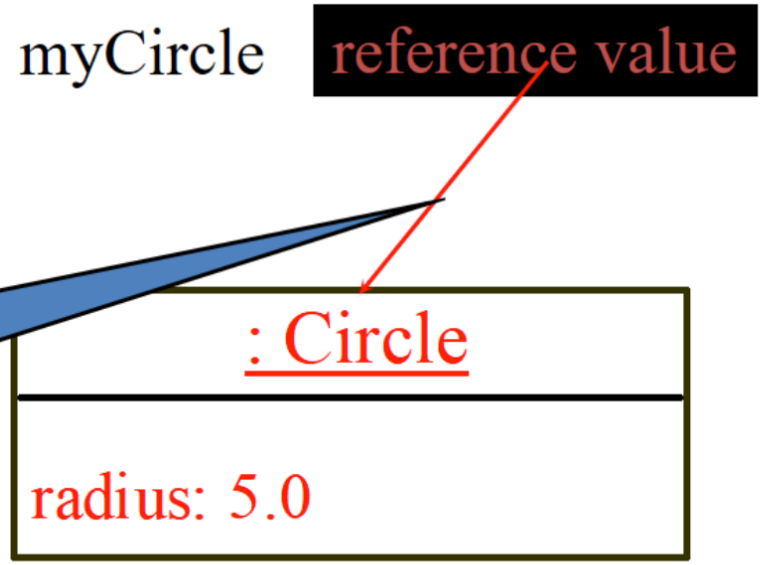


```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```

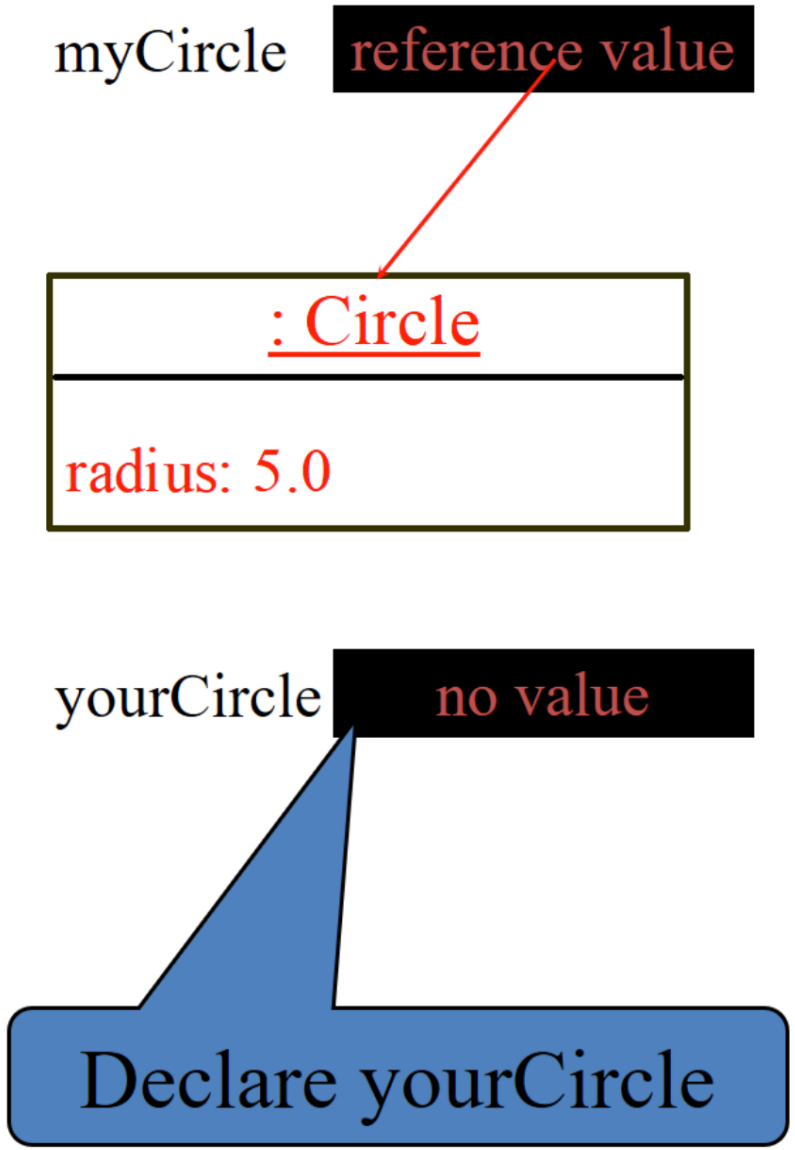


```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```

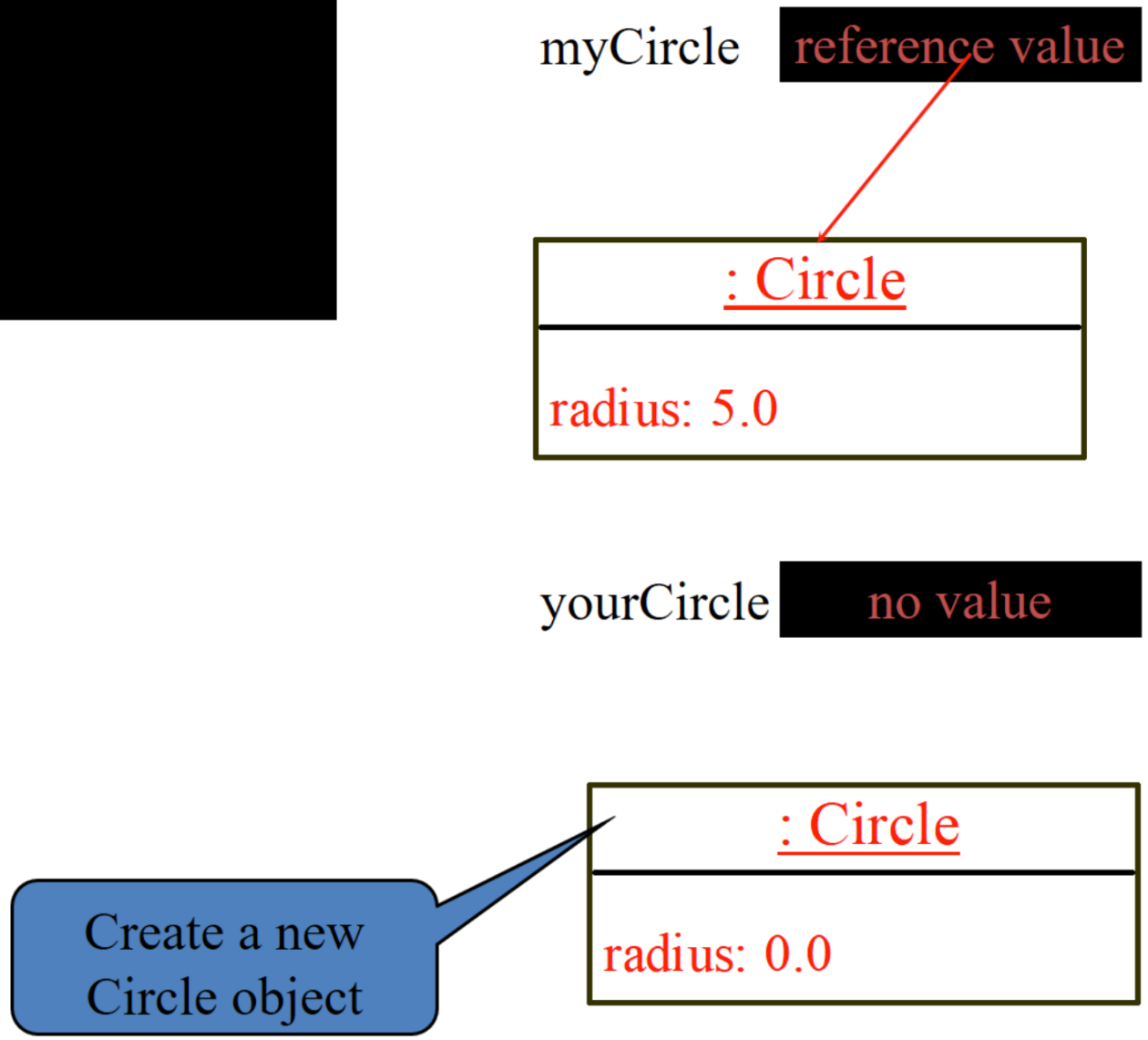
Assign object reference to myCircle



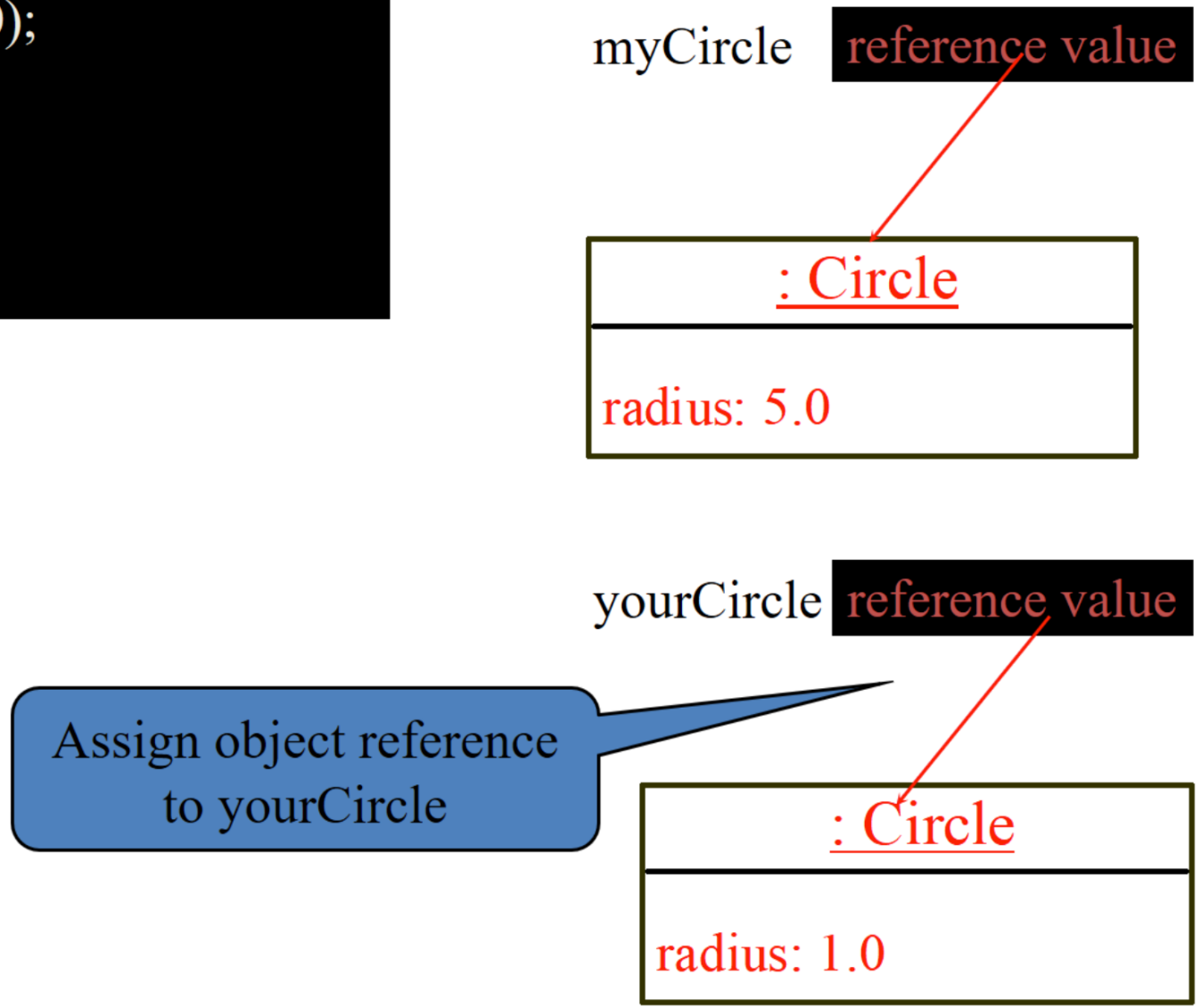
```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



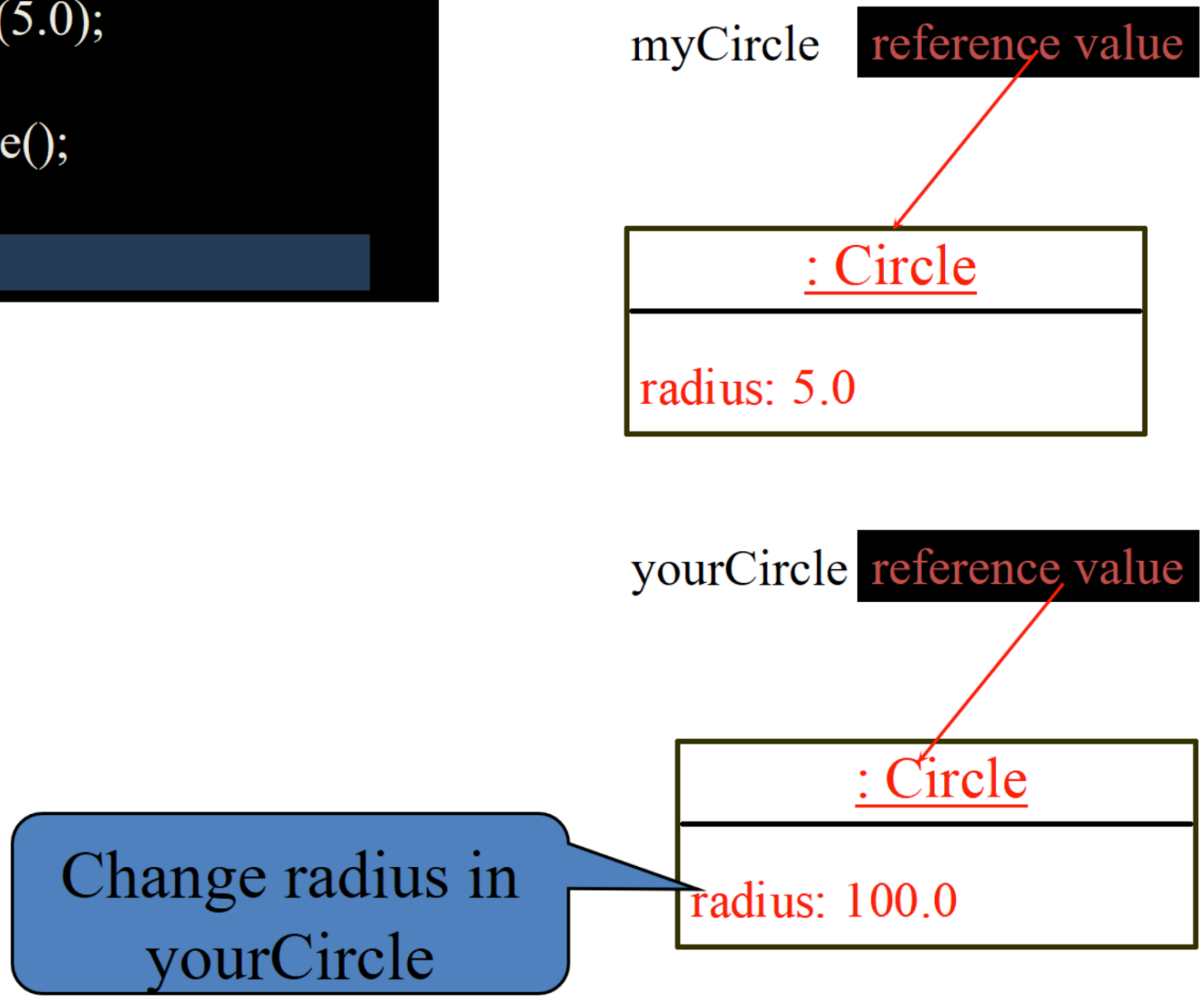
```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



# 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.

**مقادیر پیش‌فرض در جاوا:**
- برای فیلد داده‌ای از نوع **ارجاعی**: مقدار **null**
- برای نوع **عددی**: مقدار **0**
- برای نوع **بولین**: مقدار **false**
- برای نوع **کاراکتری**: مقدار **'\u0000'**

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

```java
class Student {
    String name;
    int age;
    boolean isScienceMajor;
    String gender;
}
```

```java
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);
    }
}
```

```
name? null
age? 0
isScienceMajor? false
gender? null
```

# مثال

در این مثال، **متغیرهای x** و **y** به عنوان **متغیر محلی (Local Variables)** درون **متد main** تعریف شده‌اند.

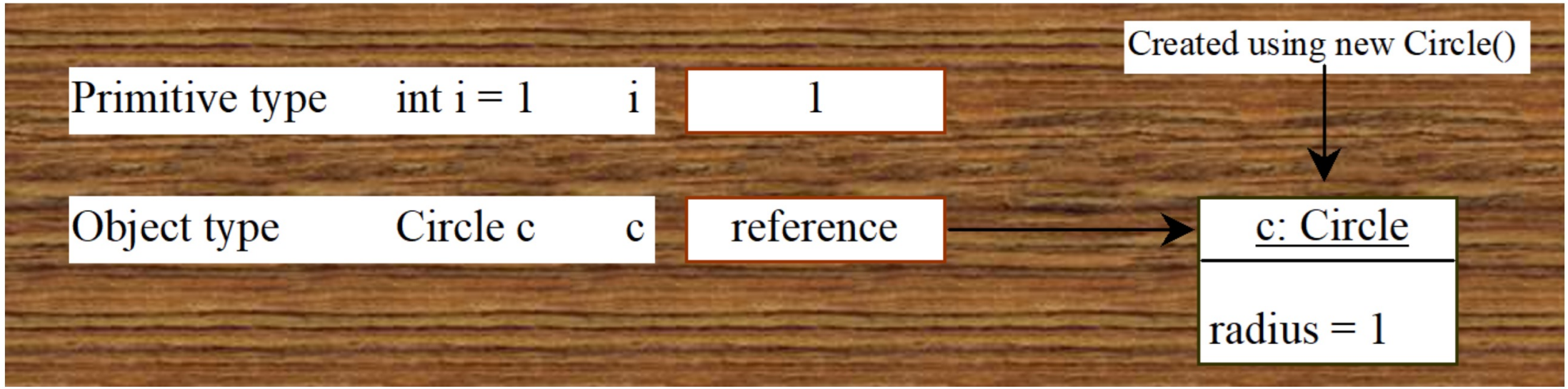
**جاوا** برای **متغیرهای محلی** هیچ **مقدار پیش‌فرضی** در نظر نمی‌گیرد. به همین دلیل، اگر بخواهیم قبل از **مقداردهی اولیه** از آن‌ها استفاده کنیم، برنامه با **خطای زمان کامپایل (Compile-Time Error)** مواجه خواهد شد.

به طور خلاصه: **متغیرهای محلی باید قبل از استفاده حتماً مقداردهی اولیه شوند.**

```java
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);
    }
}
```

```
⊗ 0.0s

|           System.out.println("x is " + x);
variable x might not have been initialized

|           System.out.println("y is " + y);
variable y might not have been initialized
```

# تفاوت‌های میان انواع داده‌ای اصلی (اولیه) و انواع ارجاعی




در زبان **جاوا** دو نوع داده وجود دارد: **انواع داده‌ای اصلی (Primitive Types)** و **انواع ارجاعی (Object Types)**.

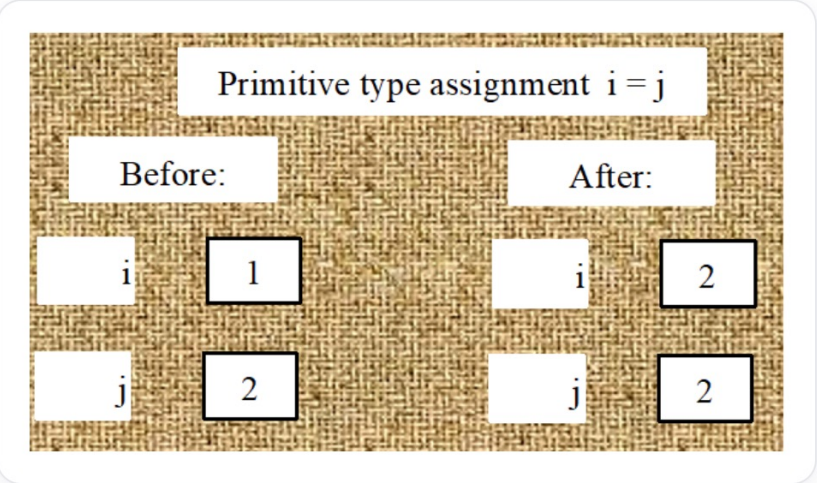
**انواع داده‌ای اصلی** مانند `int i = 1;` مستقیماً **مقدار** را در خود ذخیره می‌کنند. به این معنا که متغیر **i** مستقیماً عدد **1** را نگه می‌دارد.

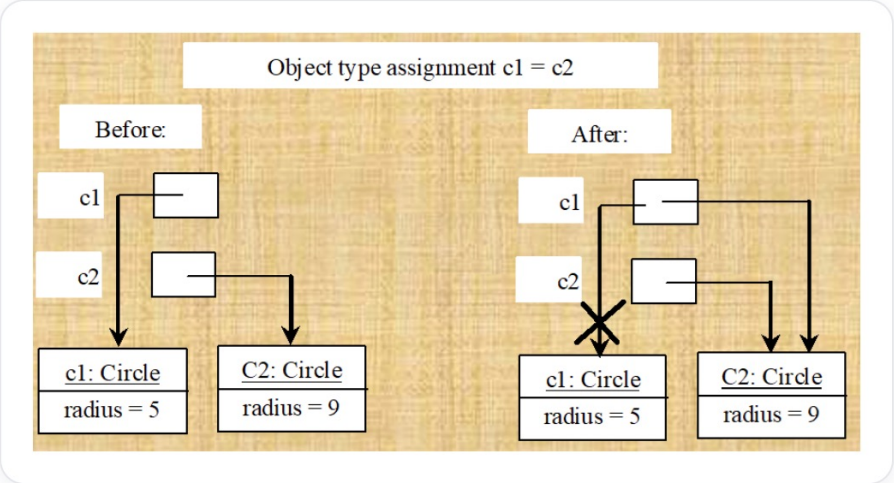
اما در **انواع ارجاعی** مانند `Circle c;` ، متغیر **c** فقط یک **ارجاع (Reference)** به **شیء** ذخیره می‌کند.

این **شیء** با استفاده از دستور `new Circle()` ساخته می‌شود و متغیر **c** به **آدرس** آن شیء در **حافظه** اشاره خواهد کرد.

برای مثال، اگر یک **دایره** با شعاع **1** ساخته شود، مقدار شعاع درون **شیء** ذخیره می‌شود اما متغیر **c** فقط به آن **شیء** در حافظه اشاره می‌کند.

# تفاوت‌های میان انواع داده‌ای اصلی (اولیه) و انواع ارجاعی




این عکس‌ها تفاوت رفتار **انتساب در انواع اصلی (Primitive)** و **انواع ارجاعی (Object)** را نشان می‌دهند.

در بخش چپ، با انتساب `i = j` مقدار عددی متغیر **j** مستقیماً در **i** کپی می‌شود؛ بنابراین بعد از عمل، **i** مقدار جدید (مثلاً **2**) 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** قبلیِ **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.

# جمع‌آوری زباله (Garbage Collection)

همان‌طور که در **توضیحات قبلی** و **عکس‌های قبلی** دیدیم، پس از دستور انتساب `c1 = c2` ، متغیر **c1** به همان **شیء**ی اشاره می‌کند که توسط **c2** مورد ارجاع قرار گرفته است.

در این حالت، **شیء**ی که قبلاً توسط **c1** مورد ارجاع قرار می‌گرفت و دیگر هیچ **متغیر ارجاعی** 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.

**Garbage 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 **متغیر ارجاعی** به یک **شیء** اشاره نکند، آن شیء **غیرقابل‌دسترسی** محسوب می‌شود و **Garbage Collector** آن را آزاد می‌کند. این فرآیند باعث جلوگیری از **Memory Leak** (نشت حافظه) شده و استفاده بهینه‌تری از منابع سیستم به همراه دارد.

البته باید توجه داشت که اجرای **Garbage Collector** بر اساس زمان‌بندی خاص **JVM** است و الزاماً در لحظه‌ای که ما انتظار 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 دقت داشت که تعداد زیادی **شیء** غیرضروری ایجاد نشود تا بار اضافی بر روی **Garbage Collector** تحمیل نگردد.

**نکته:** اگر مطمئن هستید یک **شیء** دیگر مورد نیاز نیست، می‌توانید **متغیر ارجاعی** کنترل‌کنندهٔ آن را به مقدار **null** تنظیم کنید؛ مانند:

```
c1 = null;
```

با این کار، وقتی هیچ مرجع دیگری به آن **شیء** وجود نداشته 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، متد toString از شیء تاریخ فراخوانی می‌شود و تاریخ و زمان فعلی سیستم به‌صورت 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 modifer

| java.util.Date | |
|---|---|
| +Date() | Constructs a Date object for the current time. |
| +Date(elapseTime: long) | Constructs a Date object for a given time in milliseconds elapsed since January 1, 1970, GMT. |
| +toString(): String | Returns a string representing the date and time. |
| +getTime(): long | Returns the number of milliseconds since January 1, 1970, GMT. |
| +setTime(elapseTime: long): void | 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 سپری‌شده** از January 1, 1970 (Epoch time) ایجاد کرد.

برای نمایش تاریخ و زمان در قالب **رشته** نیز می‌توان از متد toString() استفاده کرد.

## سازنده‌ها (Constructors):

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

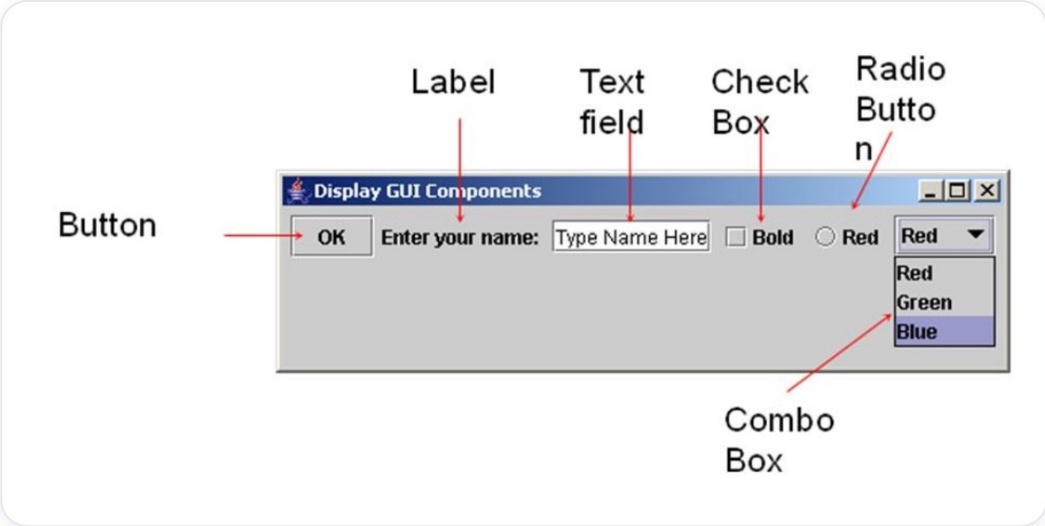
## متدهای مهم کلاس Date:

- toString(): String
  **تاریخ** و **زمان** را به صورت یک **رشته قابل‌خواندن** (مثل Wed Feb 15 09:40:19 IRST 2017) برمی‌گرداند.
- getTime(): long
  تعداد **میلی‌ثانیه‌های سپری‌شده** 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.

# نمایش مولفه‌های گرافیکی (GUI)

هنگامی که می‌خواهید برنامه‌ای جهت ایجاد **واسط‌های گرافیکی کاربر (GUI)** بنویسید، می‌توانید از کلاس‌های جاوا نظیر **JFrame**، **JButton**، **JRadioButton**، **JComboBox** و **JList** برای تولید فریم‌ها، دکمه‌ها، دکمه‌های رادیویی، جعبه‌های کمبو، لیست‌ها و غیره استفاده کنید.

در اینجا به کمک کلاس **JFrame** دو پنجره ساده ایجاد می‌کنیم.



```java
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 مولفه‌های 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 پنجرهٔ گرافیکی را در جاوا نشان می‌دهد. در هر مرحله، یک **متغیر ارجاعی** (reference) مانند frame1 یا frame2 به یک شیء از نوع **JFrame** اشاره می‌کند و با فراخوانی متدهای آن، ویژگی‌های پنجره مانند **عنوان**، **اندازه** و **نمایان بودن** تنظیم می‌شود.

### ۱) اعلان، ایجاد و انتساب در یک دستور

با دستورِ زیر شیء ساخته می‌شود و مرجع آن در متغیر frame1 قرار می‌گیرد:

```
JFrame frame1 = new JFrame();
```

در این لحظه شیء **JFrame** ایجاد شده ولی هنوز ویژگی‌هایش مقداردهی نشده است (مثلاً عنوان تهی و اندازه پیش‌فرض است). frame1 صرفاً یک **ارجاع** به این شیء در حافظه است.

### ۲) تنظیم عنوان پنجره
با فراخوانی:

```
frame1.setTitle("Window 1");
```

ویژگی **title** در شیء مرتبط با frame1 مقدار «Window 1» می‌گیرد.

### ۳) تنظیم اندازهٔ پنجره
با فراخوانی:

```
frame1.setSize(200, 150);
```

عرض و ارتفاع پنجره به ترتیب روی **۲۰۰** و **۱۵۰** پیکسل تنظیم می‌شود (ویژگی‌های **width** و **height** شیء تغییر می‌کنند).

### ۴) قابل‌نمایش کردن پنجره
با فراخوانی:

```
frame1.setVisible(true);
```

ویژگی **visible** برابر **true** می‌شود و پنجرهٔ frame1 روی صفحه نمایش داده می‌شود.

### ۵) ساخت پنجرهٔ دوم
دوباره همان فرایند را برای یک مرجع جدید انجام می‌دهیم:

```
JFrame frame2 = new JFrame();
```

حالا frame2 به شیء جدیدی از نوع **JFrame** اشاره می‌کند که مستقل از پنجرهٔ اول است.

### ۶) مقداردهی ویژگی‌های پنجرهٔ دوم

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

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

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

### نکات تکمیلی مهم

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

```
frame1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame2.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

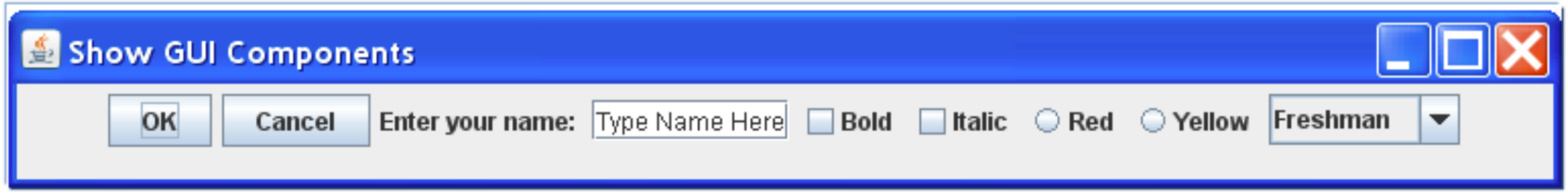
– در صورت نیاز می‌توانید مکان ظاهر شدن پنجره‌ها را نیز تعیین کنید؛ مثلاً:

```
frame1.setLocation(200, 100);
frame2.setLocation(410, 100);
```

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

# اضافه نمودن مولفه های گرافیکی به پنجره

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



```java
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");
```

```java
        // 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)

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

این متغیرها روی حافظه‌ی **heap** ساخته می‌شوند و تا وقتی شیء وجود دارد (و توسط **Garbage Collector** پاک نشده) باقی می‌مانند. دسترسی به آن‌ها از طریق شیء انجام می‌شود، مثلاً: `obj.field`

این متغیرها می‌توانند سطح دسترسی‌های مختلف داشته باشند (مثل **public / private**) و حتی مقدار اولیه هم داشته باشند. تفاوت اصلی آن‌ها با متغیرهای **static** این است که متغیرهای نمونه برای هر شیء یک نسخه جداگانه می‌سازند، در حالی که متغیر **static** فقط یک نسخه مشترک برای کل کلاس دارد.

---

## 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 کلیدواژه **this** به اعضای همان شیء اشاره کنند.

برای استفاده از آن‌ها باید متد را روی یک شیء صدا بزنیم، مثلاً: `obj.doSomething()` اگر شیئی وجود نداشته باشد، این متدها را نمی‌توان از زمینه **static** فراخوانی کرد.

متدهای نمونه کاربردهای زیادی دارند، مثلاً: **encapsulation** (مثل **getter/setter**)، بازنویسی متدها (**override**) در کلاس‌های فرزند، و پیاده‌سازی **polymorphism** در برنامه‌نویسی شیءگرا.

# متغیرها، ثابت‌ها و متدهای ایستا

وقتی چیزی را در **جاوا** با کلیدواژه **static** تعریف می‌کنیم، یعنی آن عضو به خود **class** تعلق دارد، نه به شیءهایی که از آن ساخته می‌شوند.

بنابراین، یک متغیر یا متد **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 جداگانه**ای 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** فراخوانی کرد، 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 متد از کلیدواژه‌ی **static** استفاده کنیم.

# مثالی از Static Variables, Constants and Methods

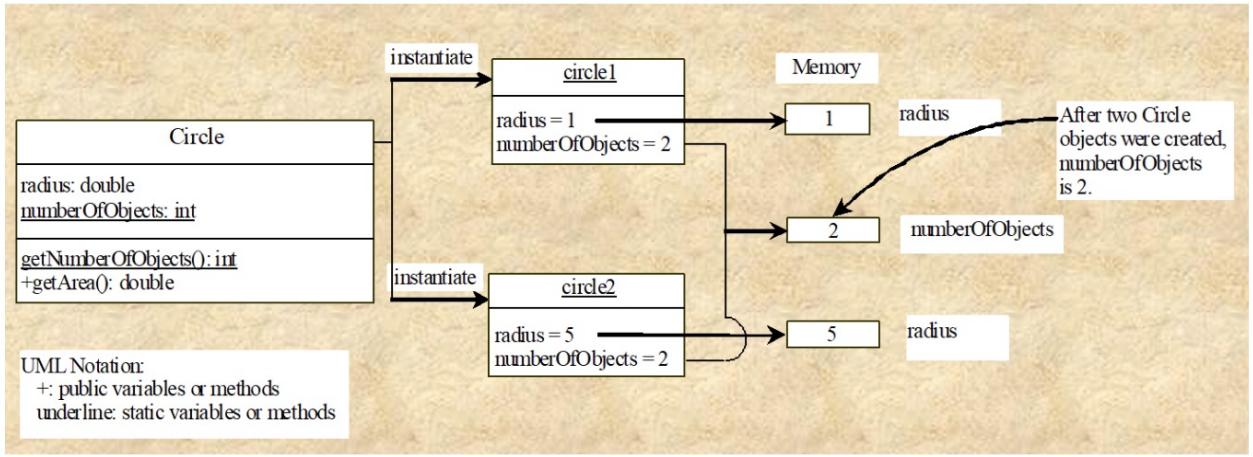
- برنامه‌ای بنویسید و در آن نقش **متغیرهای نمونه** و **متغیرهای کلاسی** را به همراه نحوه استفاده از آن‌ها نشان دهید.
- این مثال، به یک متغیر کلاسی **numberOfObjects** یک واحد اضافه می‌کند تا تعداد اشیای ساخته‌شده از کلاس **Circle** را حساب کند.

```java
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;
  }
}
```

```java
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.numberOfObjects);

    // 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.numberOfObjects + ")");

    // 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.numberOfObjects + ")");
    System.out.println("c2: radius (" + c2.radius + ") and number of Circle objects (" + c2.numberOfObjects + ")");
  }
}
```

```
Before creating objects
The number of Circle objects is 0

After creating c1
c1: radius (1.0) and number of Circle objects (1)

After creating c2 and modifying c1
c1: radius (9.0) and number of Circle objects (2)
c2: radius (5.0) and number of Circle objects (2)
```

# نحوه نمایش متغیرهای نمونه و ایستا در حافظه




این شکل نحوه‌ی تفاوت بین **متغیرهای نمونه (instance)** و **متغیرهای ایستا (static)** را در حافظه نشان می‌دهد.

در سمت چپ کلاسی به نام **Circle** داریم که شامل موارد زیر است:

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

وقتی یک شیء جدید می‌سازیم (**instantiate**)، مثلاً **circle1**، شعاع آن روی **1** قرار می‌گیرد و مقدار **numberOfObjects** یک واحد زیاد می‌شود. با ساخت شیء دوم (**circle2**)، شعاع آن **5** می‌شود و **numberOfObjects** دوباره یک واحد افزایش یافته و به **2** می‌رسد.

نکتهٔ کلیدی: **radius** یک ویژگیِ اختصاصیِ هر شیء است (برای هر شیء مقدار جدا دارد)، اما **numberOfObjects** یک ویژگیِ مشترکِ همهٔ اشیای کلاس است و فقط **یک نسخه** از آن در حافظه نگه‌داری می‌شود. به همین دلیل پس از ساخت دو شیء، مقدار آن **2** می‌شود.

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

# سطح دسترسی به فیلدها و متدها

در **برنامه‌نویسی شیءگرا**، هر **متغیر** یا **متد** 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 **متد** از هر جای برنامه و توسط هر **کلاس** دیگری قابل مشاهده و استفاده است.
- **private**: یعنی **متغیر** یا **متد** فقط در همان **کلاس**ی که تعریف شده قابل مشاهده است و بیرون از آن کلاس به‌طور مستقیم قابل دسترسی نیست.

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

# یک نکته مهم!!

```java
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

```java
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.

در این تصویر مفهوم سطح دسترسی **private** به‌خوبی نشان داده شده است. وقتی یک متغیر یا متد در جاوا **private** تعریف می‌شود، یعنی فقط از درون همان کلاس قابل دسترسی است و هیچ کلاس دیگری امکان دسترسی مستقیم به آن را ندارد.

در بخش سمت چپ، کلاسی به نام **Foo** داریم که یک متغیر **x** و یک متد **convert** را به صورت **private** تعریف کرده است. چون دسترسی به این اعضا درون همان کلاس انجام می‌شود، کد معتبر است و بدون خطا اجرا خواهد شد. بنابراین (a) صحیح است.

در بخش سمت راست، کلاسی دیگر به نام **Test** سعی کرده به متغیر **x** و متد **convert** از کلاس **Foo** دسترسی پیدا کند. این کار اشتباه است زیرا هر دو عضو **private** تعریف شده‌اند و فقط درون کلاس **Foo** قابل استفاده هستند. به همین دلیل (b) نادرست است و کد به خطا منجر می‌شود.

نتیجه‌گیری مهم این است که اعضای **private** برای محافظت از داده‌ها استفاده می‌شوند و باید تنها از طریق متدهای **getter** و **setter** یا دیگر متدهای عمومی (**public**) قابل دسترسی باشند. این ویژگی همان اصل **Encapsulation** در برنامه‌نویسی شیءگراست.

# چرا باید فیلدهای داده‌ای private باشند؟

در زبان **Java** بهتر است **فیلدهای داده‌ای** یک کلاس به صورت **private** تعریف شوند. این کار باعث **محافظت** از داده‌ها می‌شود و همچنین **نگهداری** از کلاس را آسان‌تر می‌کند.

با جلوگیری از **تغییر مستقیم داده‌ها** توسط کدهای بیرونی، امکان **کنترل دسترسی** از طریق متدهای **getter** و **setter** فراهم می‌شود. این موضوع به پیاده‌سازی اصل **Encapsulation** در **برنامه‌نویسی شی‌گرا** کمک کرده و باعث افزایش **امنیت** و کاهش احتمال بروز خطا می‌گردد.

همچنین می‌توان قبل از تغییر مقدار فیلدها، فرآیند **اعتبارسنجی** را انجام داد تا از ورود داده‌های نامعتبر به سیستم جلوگیری شود.

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;

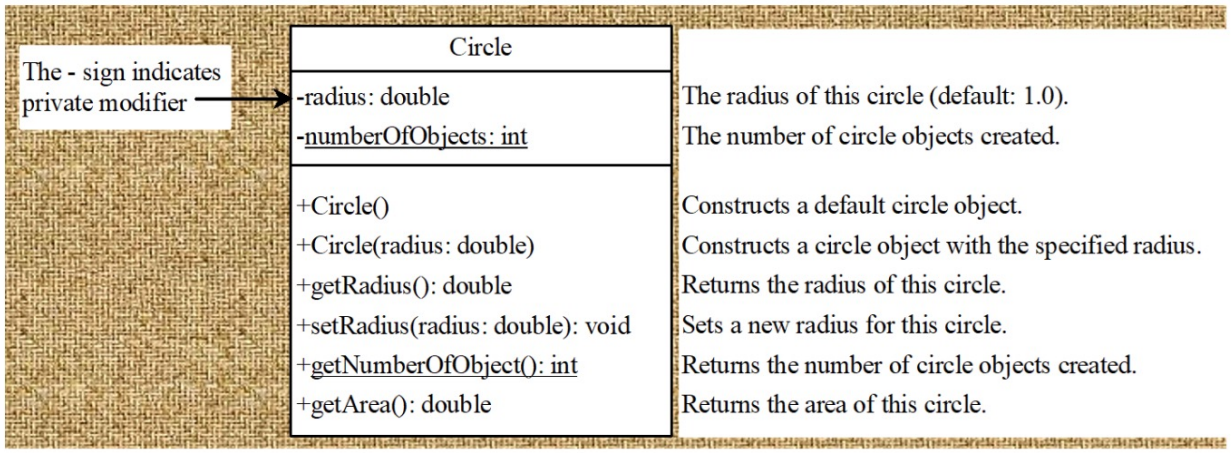
```
class C1 {
  ...
}
```

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

package p2;

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

# مثالی از کپسوله‌بندی فیلد داده‌ای




در زبان **Java** یکی از اصول مهم **برنامه‌نویسی شیءگرا**، **کپسول‌بندی (Encapsulation)** است. کپسول‌بندی یعنی این‌که **فیلدهای داده‌ای (Data Fields)** یا همان **متغیرهای کلاس**، به صورت **private** تعریف شوند تا مستقیماً از بیرون کلاس قابل دسترسی نباشند.

در عوض، برای **دسترسی** یا **تغییر دادن مقدار** آن‌ها از متدهای **getter** و **setter** استفاده می‌کنیم. به این ترتیب، داده‌ها درون کلاس **ایمن‌تر** هستند و برنامه‌نویس می‌تواند کنترل کند که چگونه مقادیر **خوانده** یا **تغییر داده** شوند.

مثلاً در مثال **Circle**، فیلد **radius** به صورت **private** تعریف شده است. برای خواندن مقدار آن از متد getRadius() و برای تغییر مقدار آن از متد setRadius(double radius) استفاده می‌شود. این روش هم باعث **افزایش امنیت داده** می‌شود و هم کد را **منظم‌تر** و **قابل‌نگهداری‌تر** می‌کند.

```java
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;
        numberOfObjects++;
    }

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

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

    /** Return numberOfObjects */
    public static int getNumberOfObjects() {
        return numberOfObjects;
    }

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

```java
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());
    }
}
```

```
The area of the circle of radius 5.0 is 78.53981633974483
The area of the circle of radius 5.5 is 95.03317777109125
```

# ارسال اشیا به متدها

در **Java** دو شیوهٔ اصلی برای **ارسال پارامتر** وجود دارد.

در حالت **ارسال با مقدار (Pass by Value)** برای **انواع اولیه (primitive)** مانند **int** یا **double**، یک **کپی از مقدار** به متد داده می‌شود. هر تغییری که داخل متد روی پارامتر انجام شود، روی متغیر اصلی اثر نمی‌گذارد.

در حالت **ارسال با مقدارِ ارجاع (Reference Value)** برای **اشیا و آرایه‌ها**، در واقع **کپیِ ارجاع** (آدرس شیء) به متد داده می‌شود. به همین دلیل اگر **ویژگی‌های شیء** داخل متد تغییر کنند، این تغییرات روی **شیء اصلی** نیز اعمال خواهد شد، چون هر دو به یک محل در حافظه اشاره می‌کنند.

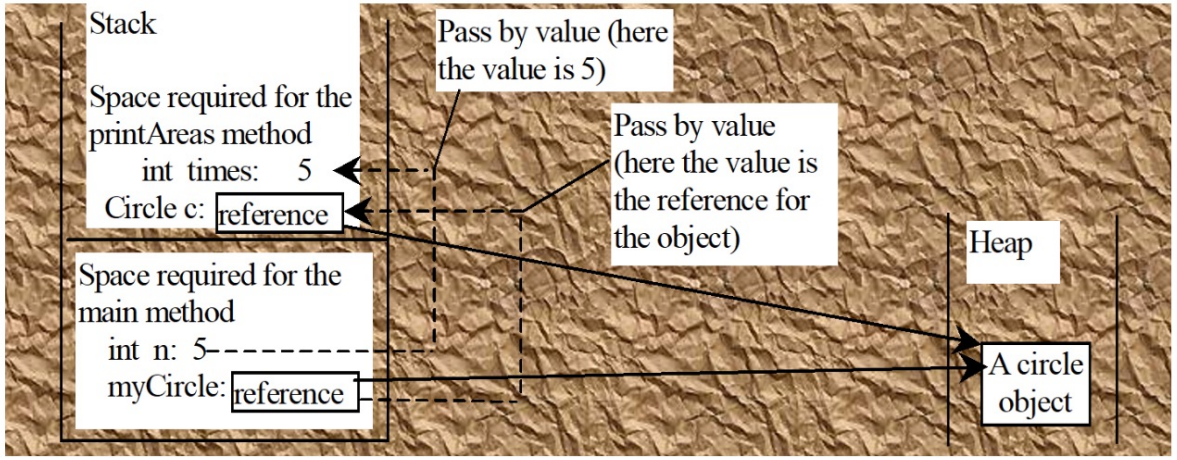
با این حال، اگر داخل متد پارامتر ارجاعی دوباره به یک **شیء جدید** انتساب داده شود، این تغییر فقط در همان متد باقی می‌ماند و **مرجع بیرونی** همچنان به شیء قبلی اشاره می‌کند.

این رفتار باعث می‌شود که تغییر مستقیم روی **حالت شیء** از متد قابل مشاهده باشد، اما تغییر **مرجع** آن تنها در محدودهٔ متد اثر کند.

```java
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--;
        }
    }
}
```

```
Radius          Area
1.0             3.141592653589793
2.0             12.566370614359172
3.0             28.274333882308138
4.0             50.26548245743669
5.0             78.53981633974483

Radius is 6.0
n is 5
```

# ارسال اشیا به متدها




در زبان **Java** تمام مقادیر به صورت **Pass by Value** به متدها ارسال می‌شوند. این موضوع ممکن است در ابتدا کمی گیج‌کننده به نظر برسد، مخصوصاً زمانی که با **اشیا** سر و کار داریم. تصویر بالا دقیقا همین مفهوم را نشان می‌دهد.

در بخش **Stack** فضای لازم برای اجرای متدها و متغیرهای محلی آن‌ها نگهداری می‌شود. برای مثال، در متد اصلی (**main method**) یک متغیر **int n** با مقدار ۵ و یک متغیر به نام **myCircle** تعریف شده است. متغیر **myCircle** در حقیقت یک **reference** (آدرس در حافظه) به یک شی از نوع **Circle** در **Heap** است.

وقتی متد **printAreas** فراخوانی می‌شود، دو مقدار به آن پاس داده می‌شوند:

- مقدار متغیر **n** که یک عدد صحیح (۵) است. این مقدار مستقیماً به صورت **pass by value** به متد ارسال می‌شود و در **Stack** فضای جدیدی برای آن ایجاد می‌گردد.
- مقدار متغیر **myCircle** که در واقع همان **reference** (آدرس شی در heap) است. این آدرس نیز به صورت **pass by value** کپی شده و در متد **printAreas** در متغیری به نام **c** قرار می‌گیرد.

نکته مهم این است که در **Java** حتی وقتی یک شیء به متد ارسال می‌شود، چیزی که پاس داده می‌شود خود شیء نیست، بلکه **reference** آن است. از آنجایی که این **reference** نیز به صورت **by value** پاس داده می‌شود، در واقع دو متغیر (در متد اصلی و در متد فراخوانی شده) هر دو به یک شی در **Heap** اشاره می‌کنند. بنابراین تغییر دادن داده‌های داخل شی (مثلاً تغییر شعاع دایره) باعث تغییر در همان شی واحد در حافظه خواهد شد.

به همین دلیل اگر داخل متد **printAreas** مقداری از ویژگی‌های شی دایره تغییر کند، پس از بازگشت به متد اصلی نیز این تغییرات قابل مشاهده خواهند بود، چون هر دو متغیر به یک **Circle object** در **Heap** اشاره می‌کنند. اما اگر خود **reference** به شی جدیدی اشاره داده شود، این تغییر تنها در متد جاری باقی می‌ماند و تأثیری روی متغیر متد اصلی ندارد.

در یک جمع‌بندی کلی:

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

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

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

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

نکات مهم:

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

نکات تکمیلی که باید بدانید:

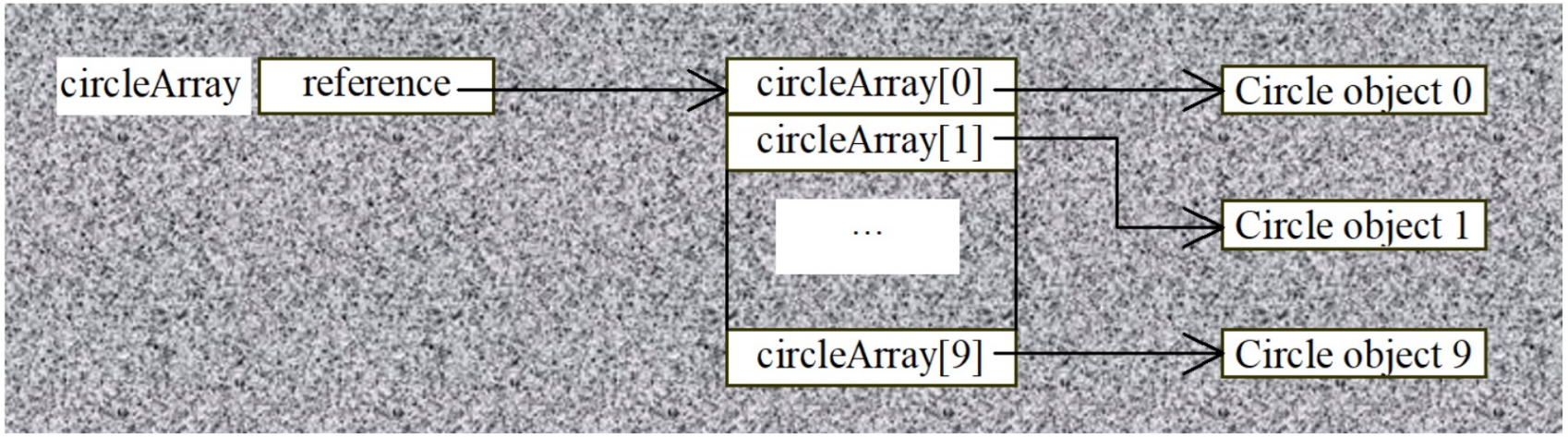
- برای مقداردهی اولیه عناصر آرایه می‌توان از یک **حلقه for** استفاده کرد. مثلاً:

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

این کار باعث می‌شود تمام خانه‌های آرایه به اشیای جدیدی از نوع Circle ارجاع دهند.

- آرایه‌ها در Java اندازه ثابت دارند. یعنی وقتی آرایه‌ای مثل **new Circle[10]** ساخته می‌شود، تعداد خانه‌های آن دیگر قابل تغییر نیست.
- در سطح مفهومی، می‌توان گفت که در اینجا دو سطح ارجاع داریم:
  - یک ارجاع برای کل آرایه (**circleArray**).
  - چند ارجاع داخل آرایه که هر کدام می‌توانند به یک شی Circle در حافظه Heap اشاره کنند.

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


circleArray
reference
circleArray[0]
circleArray[1]
...
circleArray[9]
Circle object 0
Circle object 1
Circle object 9

```java
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;
    }
```

```java
    /** 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 خطایی می‌دهد؟ (فقط نام خطا را بنویسید)

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

۲) درست/نادرست: جملهٔ زیر را بررسی کنید و دلیل کوتاه بنویسید: «با اجرای دستور **new Circle[10]** ده شیء از نوع **Circle** ساخته می‌شود.»

۳) جای خالی را پر کنید: پس از اجرای کد زیر، مقدار پیش‌فرض هر خانهٔ آرایهٔ **circles** چیست؟

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

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

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

۵) تکمیل کد: با توجه به اصول گفته‌شده، حلقهٔ for را طوری کامل کن که همهٔ خانه‌ها با **Circle**های جدید مقداردهی شوند.

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

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

```
Circle[] a = new Circle[4];
a[0] = new Circle();
a[2] = new Circle();
```

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

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

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

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

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

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

## پایان

در صورت هرگونه سوال یا پیشنهاد میتونید با من در ارتباط باشید
:)

gmail: mr.mohamad.hoseini05@gmail.com
telegram: @MHosseiniR