

به نام خدا



# برنامه‌سازی پیشرفته

دانشگاه شهید بهشتی · دانشکده مهندسی و علوم کامپیوتر

دکتر مجتبی وحیدی اصل

پلی مورفیسم

سارا شیری

# فهرست مطالب

1. بازنویسی متدهای ابرکلاس در زیرکلاس

2. مقایسه سربارگذاری متدها و بازنویسی متدها

3. کلاس Object در جاوا

4. پلی مورفیسم

5. مقیدسازی

6. تبدیل اشیا رو به بالا / رو به پایین

7. عملگر instanceof

# اعلان یک زیرکلاس

یک زیرکلاس فیلدهای دادهای و متدها را از ابرکلاس یا همان پدرش ارث بری می‌کند. یک فرزند علاوه بر ارث بری فیلدهای پدر می‌تواند فیلدهای دادهای جدید داشته باشد، متدهای جدید داشته باشد و متدهای ابرکلاس را بازنویسی (override) کند.

## بازنویسی متدهای ابرکلاس در زیرکلاس‌ها

همان‌طور که در اسلاید قبل گفتیم، یک زیرکلاس متدهای پدرش یا همان ابرکلاس را ارث بری می‌کند. بعضی وقت‌ها این زیرکلاس نیاز دارد پیاده‌سازی یک متده تعریف شده در ابرکلاسش را تغییر دهد. به این کار به اصطلاح **بازنویسی متده** یا **method overriding** می‌گوییم. به عبارت دیگر، هرگاه متده در زیرکلاس بخواهد پیاده‌سازی خاص خود از متده شده از ابرکلاس را داشته باشد، از بازنویسی استفاده می‌کند. مزیت بازنویسی متده در پلی مورفیسم زمان اجرا می‌باشد.

### قوانين بازنویسی متده:

متده باید همنام با متده کلاس والد باشد.

متده باید پارامترهای مشابهی با کلاس والد داشته باشد.

رابطه وراثت برقرار باشد.

# اگر از بازنویسی متده استفاده نکنیم

مسئله‌ای که پیش می‌آید: اینکه بخواهیم یک پیاده‌سازی ویژه از متده run() در زیرکلاس دوچرخه داشته باشیم، باید این متده را که از کلاس وسیله نقلیه به ارث رسیده است بازنویسی کنیم.

```
class Vehicle {  
    void run() {  
        System.out.println("Vehicle is running");  
    }  
  
}  
  
class Bike extends Vehicle {  
    public static void main(String args[]) {  
        Bike obj = new Bike();  
        obj.run();  
    }  
}
```

Vehicle is running

# مثال استفاده از بازنویسی متدها

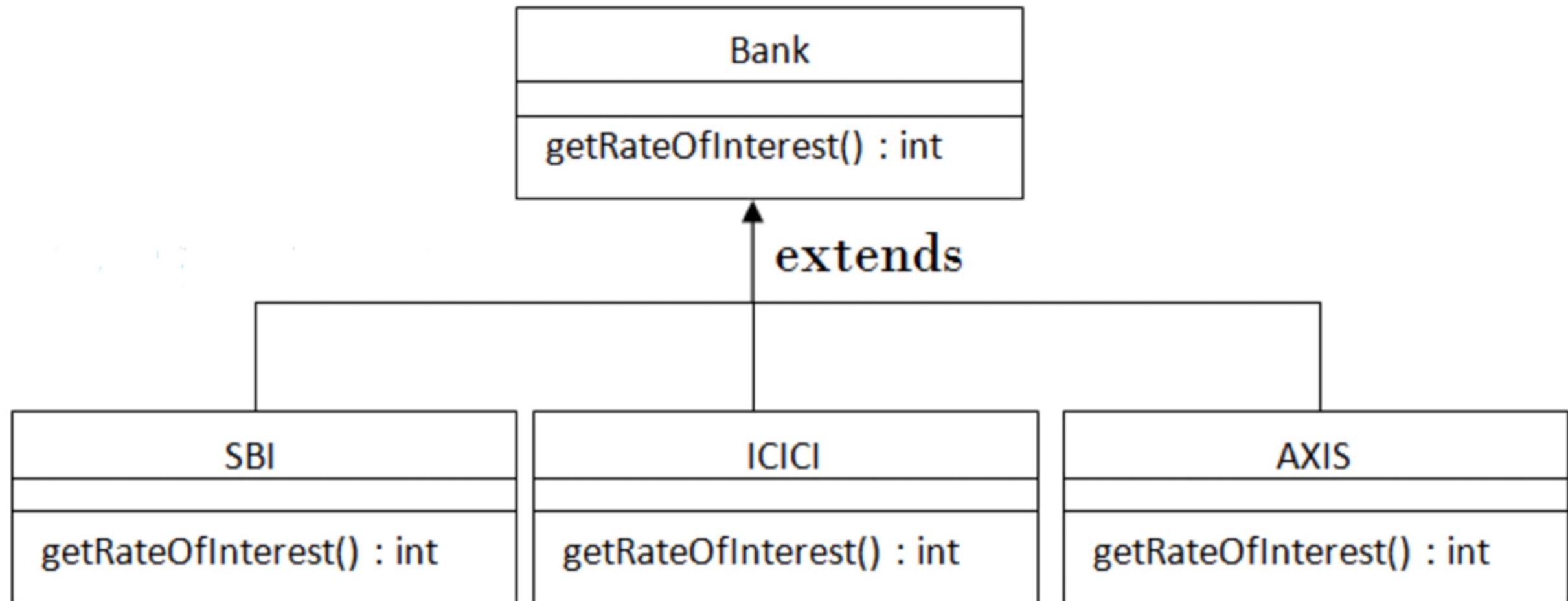
نکته: باید حواسمن باشد که نام و نوع پارامتر متدهای بازنویسی شده باید با متدهای ابرکلاس یکی باشند.

```
class Vehicle {  
    void run() {  
        System.out.println("Vehicle is running");  
    }  
}  
  
class Bike2 extends Vehicle {  
    void run() {  
        System.out.println("Bike is running safely");  
    }  
}  
  
public static void main(String args[]) {  
    Bike2 obj = new Bike2();  
    obj.run();  
}
```

Bike is running safely

## مثالی از بازنویسی متدها - سود بانکی

شرح مسئله: کلاسی به نام Bank داریم که نرخ سود سپرده را برمی‌گرداند. اما این نرخ سود در هر بانک با بانک دیگر متفاوت است. فرض کنید سه بانک داریم به نام‌های ICICI، SBI و AXIS که به ترتیب نرخ سود ۸٪، ۷٪ و ۹٪ برمی‌گردانند.



```
class Bank {  
    int getRateOfInterest() {  
        return 0;  
    }  
}  
  
class SBI extends Bank {  
    int getRateOfInterest() {  
        return 8;  
    }  
}  
  
class ICICI extends Bank {  
    int getRateOfInterest() {  
        return 7;  
    }  
}
```

```
class AXIS extends Bank {  
    int getRateOfInterest() {  
        return 9;  
    }  
}  
  
class Test2 {  
    public static void main(String args[]) {  
        SBI s = new SBI();  
        ICICI i = new ICICI();  
        AXIS a = new AXIS();  
  
        System.out.println("SBI Rate of Interest: " + s.getRateOfInterest());  
        System.out.println("ICICI Rate of Interest: " + i.getRateOfInterest());  
        System.out.println("AXIS Rate of Interest: " + a.getRateOfInterest());  
    }  
}
```

```
SBI Rate of Interest: 8  
ICICI Rate of Interest: 7  
AXIS Rate of Interest: 9
```

# نکات بازنویسی متدها

چرا یک متدهای استاتیک قابل بازنویسی نمیباشد؟

چون متدهای استاتیک به یک کلاس محدود شده است نه به اشیای کلاس. اما یک متدهای استاتیک نمیتواند بازنویسی شود. اما متدهای استاتیک در زمان اجرا قابل تغییر نمیباشند. اما متدهای استاتیک در حافظه heap ایجاد میشوند و امکان بازنویسی یا تغییر آنها وجود دارد.

آیا میتوانیم متدهای main در جاوا را بازنویسی کنیم؟

جواب مشخصا خیر است چون متدهای main به صورت استاتیک تعریف میشود.

**نکته:** یک متدهای استاتیک قابل بازنویسی نمیباشد. بنابراین یک متدهای استاتیک private نمیتواند بازنویسی شود، چون در خارج از کلاس خود قابل دسترسی نمیباشد. اگر متدهای استاتیک در یک زیرکلاس همانم با متدهای استاتیک در ابرکلاس باشد که private تعریف شده است، این دو متدهای استاتیک بیاندازند از این نظر گرفته میشوند، مانند متدهای استاتیک private نیز میتوانند ارث بری کنند.

با این حال، یک متدهای استاتیک نمیتواند بازنویسی شود. اگر متدهای استاتیک تعریف شده در یک ابرکلاس بازتعریف شود، متدهای استاتیک در ابرکلاس پنهان (پوشیده) خواهد شد.

# overriding در مقایسه با overloading

سه تفاوت اصلی میان بازنویسی و سربارگذاری متدها وجود دارد:

بازنویسی متده ب برنامه نویس امکان می دهد پیاده سازی خاص متده ارث بری شده از کلاس والد را ایجاد کند و همواره در دو کلاسی انجام می شود که رابطه ارث بری میان آنها وجود دارد و لازم به ذکر است که پارامترها باید یکسان باشد.

ولی سربارگذاری متده ب هدف افزایش خوانایی برنامه ها ایجاد می شود و درون یک کلاس انجام می شود و برخلاف بازنویسی، پارامترها باید متفاوت باشند.

```
//Overriding

public class Test {
    public static void main(String[] args) {
        A a = new A();
        a.p(10);
        a.p(10.0);
    }
}

class B {
    public void p(double i) {
        System.out.println(i * 2);
    }
}

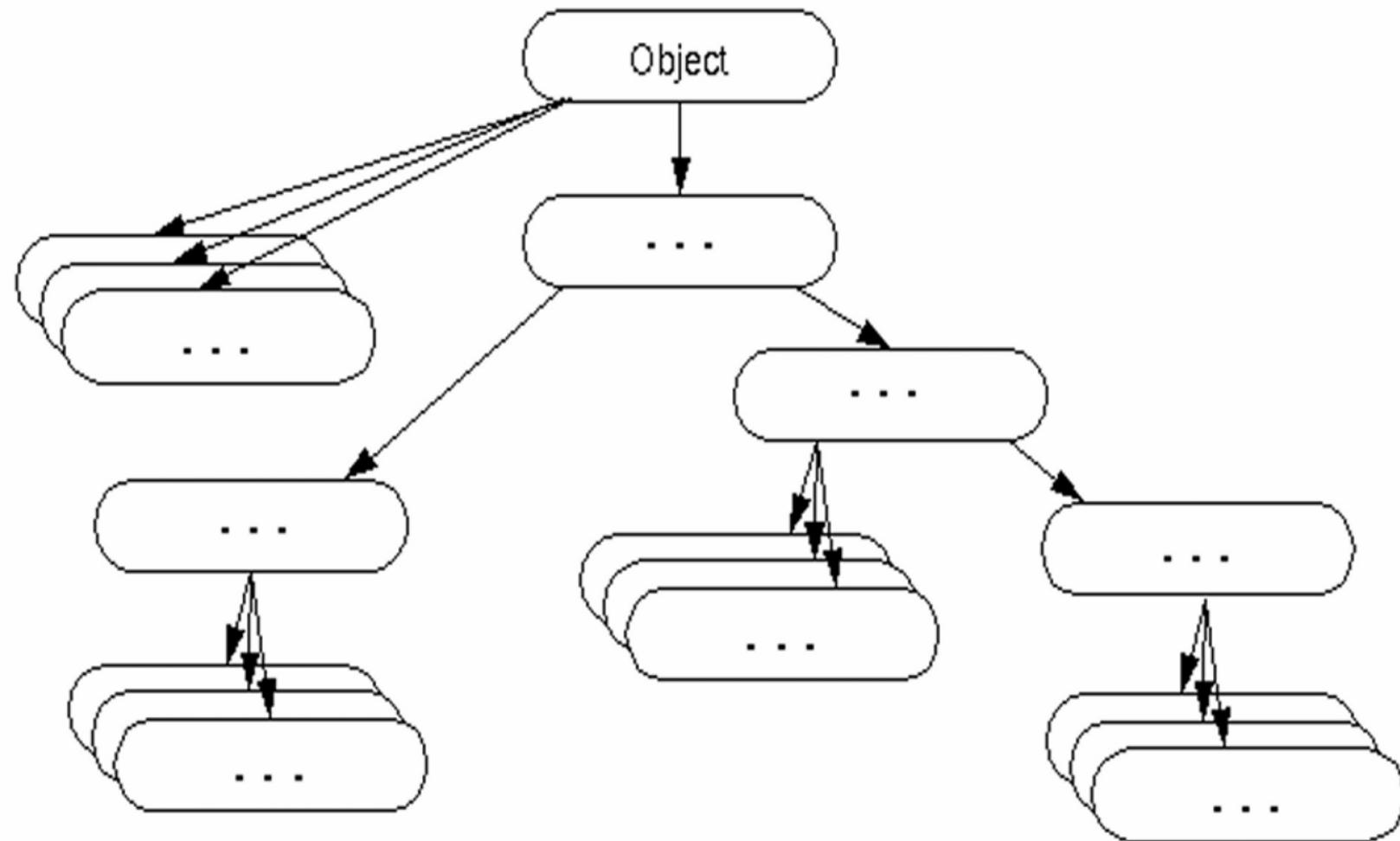
class A extends B {
    //This method overrides the method in B
    public void p(double i) {
        System.out.println(i);
    }
}
```

10.0  
10.0

## کلاس Object و متدهای آن

هر کلاس در جاوا به نوعی از نسل کلاس `java.lang.Object` به حساب می‌آید. اگر در تعریف یک کلاس، ارت بری اعلام نشود، به طور ضمنی ابرکلاس آن کلاس `Object` می‌باشد.

برای مثال `public class Circle extends Object{}` معادل است با `public class Circle{}`



در شکل زیر می‌توانید متدهای موجود در کلاس Object را مشاهده کنید.

Method	Description
<b>public final Class getClass()</b>	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
<b>public int hashCode()</b>	returns the hashcode number for this object.
<b>public boolean equals(Object obj)</b>	compares the given object to this object.
<b>protected Object clone() throws CloneNotSupportedException</b>	creates and returns the exact copy (clone) of this object.
<b>public String toString()</b>	returns the string representation of this object.
<b>public final void notify()</b>	wakes up single thread, waiting on this object's monitor.
<b>public final void notifyAll()</b>	wakes up all the threads, waiting on this object's monitor.
<b>public final void wait(long timeout) throws InterruptedException</b>	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
<b>public final void wait(long timeout,int nanos) throws InterruptedException</b>	causes the current thread to wait for the specified miliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
<b>public final void wait(long timeout,int nanos) throws InterruptedException</b>	causes the current thread to wait for the specified miliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
<b>public final void wait() throws InterruptedException</b>	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
<b>protected void finalize() throws Throwable</b>	is invoked by the garbage collector before object is being garbage collected.

# toString()

این متدهای رشته‌ای را برمی‌گرداند که نمایش‌دهنده شی‌ای حاصل از کلاس مربوطه است. پیاده‌سازی پیش‌فرض این متدهای رشته‌ای حاوی نام کلاس مربوط به شی را برمی‌گرداند که به دنبال آن علامت @ و شماره شی آمده است.

```
Loan loan = new Loan();  
System.out.println(loan.toString());
```

کد بالا عبارتی نظیر Laon@15037e5 را چاپ می‌کند.

این پیام اطلاعات زیادی در اختیار برنامه‌نویس قرار نمی‌دهد. درنتیجه در اکثر موقعیت‌ها بازنویسی می‌شود تا رشته‌ای حاوی اطلاعات جامعتر در اختیار کاربر قرار دهد.

بازنویسی این متدهای در زیرکلاس‌ها به چه صورت است؟

```
public class Circle extends GeometricObject {  
    // Other methods are omitted  
  
    /** Override the toString method defined in GeometricObject */  
    public String toString() {  
        return super.toString() + "\nradius is " + radius;  
    }  
}
```

# تبدیل رو به بالا یا Upcasting

تبدیل رو به بالا یا Upcasting : هرگاه متغیر ارجاعی کلاس والد به شیی کلاس فرزند اشاره کند، می‌گوییم تبدیل رو به بالا انجام شده است.

```
class A{}
```

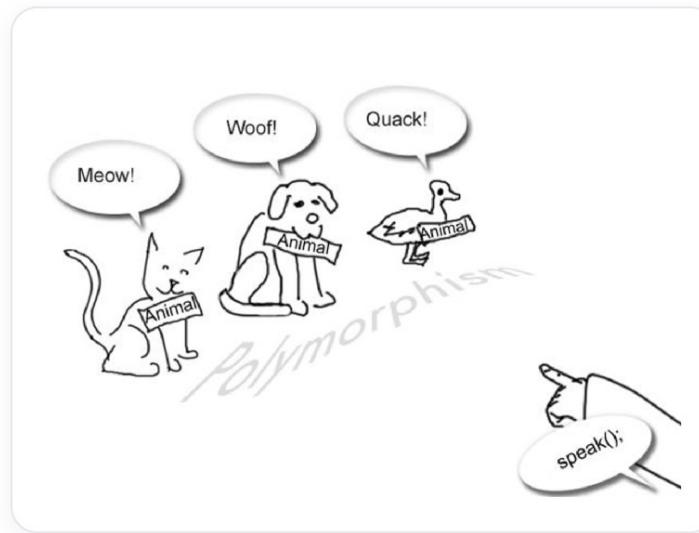
```
class B{}
```

```
A a = new B();
```

---

# مفهوم پلی مورفیسم

پلی مورفیسم زمان اجرا، فرآیندی است که فراخوانی یک متده بازنویسی شده در زمان اجرا و نه در زمان کامپایل انجام می‌شود. در این فرآیند، یک متده بازنویسی شده از طریق یک متغیر ارجاعی از کلاس والد (ریموت کنترل کلاس والد) فراخوانی می‌شود. تعیین اینکه کدام متده فراخوانی شود، به شی‌ای بستگی دارد که متغیر ارجاعی به عنوان آرگومان به آن اشاره می‌کند.



# مثالی از پلی مورفیسم زمان اجرا

در این مثال دو کلاس Bike و Splender تعریف شده است، کلاس Splender از کلاس Bike ارث بری می‌کند و متد run() را بازنویسی می‌کند. اینک، متد run() را از طریق متغیر ارجاعی کلاس والد فراخوانی می‌کنیم.

چون متغیر ارجاعی به شیی زیرکلاس اشاره می‌کند و متد زیرکلاس، متد ابرکلاس را بازنویسی کرده است، در زمان اجرا متد زیرکلاس فراخوانی می‌شود. چون فراخوانی متد توسط JVM و نه کامپایلر انجام می‌شود، این فراخوانی به پلی مورفیسم زمان اجرا شهرت دارد.

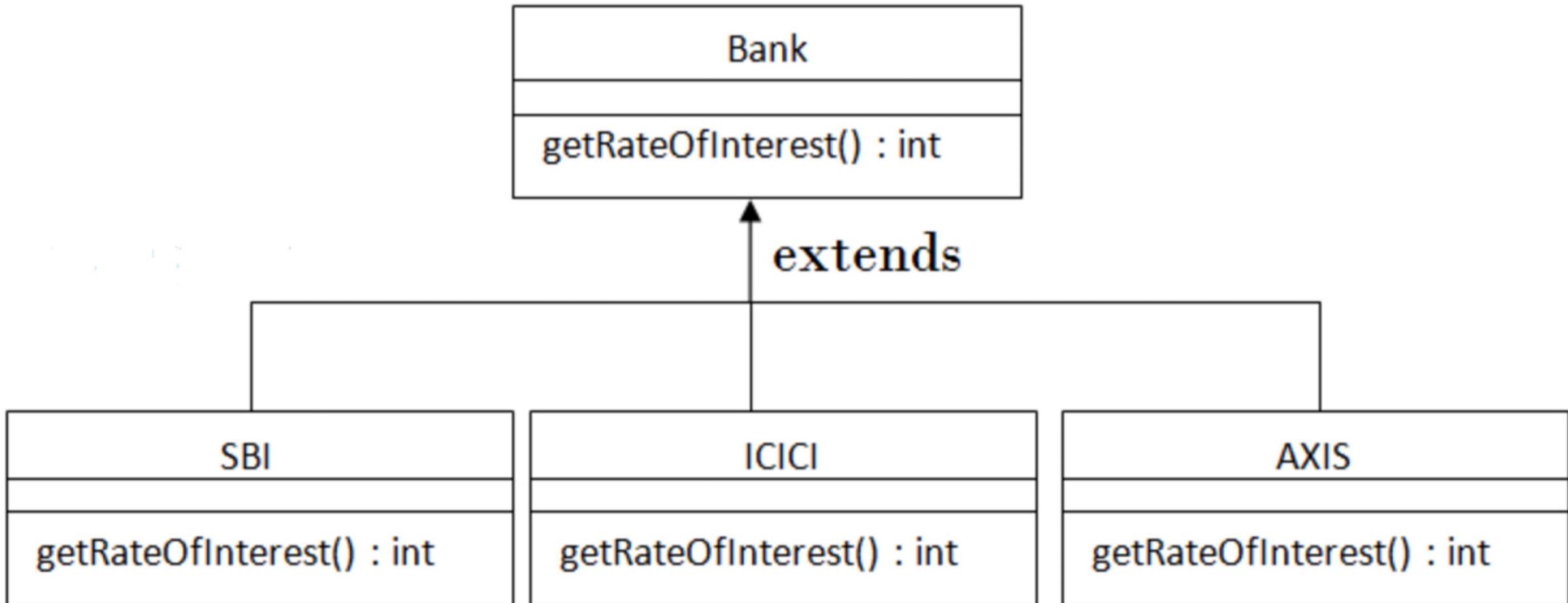
```
class Bike{
    void run(){
        System.out.println("running");
    }
}
class Splender extends Bike{
    void run(){
        System.out.println("running safely with 60km");
    }
}

public static void main(String[] args){
    Bike b = new Splender(); //upcasting
    b.run();
}
```

running safely with 60km

## مثالی دیگر

سناریو بانک را در نظر بگیرید و با استفاده از پلی مورفیسم متدهای `getRateOfInterest()` را فراخوانی کنید.



```
class Bank {  
    int getRateOfInterest() {  
        return 0;  
    }  
  
}  
  
class SBI extends Bank {  
    int getRateOfInterest() {  
        return 8;  
    }  
  
}  
  
class ICICI extends Bank {  
    int getRateOfInterest() {  
        return 7;  
    }  
}
```

```
class AXIS extends Bank {  
    int getRateOfInterest() {  
        return 9;  
    }  
  
}  
  
class Test3 {  
    public static void main(String args[]) {  
        Bank b1 = new SBI();  
        Bank b2 = new ICICI();  
        Bank b3 = new AXIS();  
  
        System.out.println("SBI Rate of Interest: " + b1.getRateOfInterest());  
        System.out.println("ICICI Rate of Interest: " + b2.getRateOfInterest());  
        System.out.println("AXIS Rate of Interest: " + b3.getRateOfInterest());  
    }  
}
```

SBI Rate of Interest: 8  
ICICI Rate of Interest: 7  
AXIS Rate of Interest: 9

# پلی مورفیسم زمان اجرا با فیلد داده‌ای

همیشه متدها هستند که بازنویسی می‌شوند و نه فیلددهای داده‌ای کلاس.

بنابراین پلی مورفیسم بر روی فیلددهای داده‌ای قابل دستیابی نخواهد بود.

در مثال زیر، هر دو کلاس دارای فیلد داده‌ای speedlimit هستند. دسترسی به این فیلد داده‌ای توسط متغیر ارجاعی کلاس والد است که به شی زیرکلاس اشاره می‌کند یا به عبارتی ریموت کنترل آن است.

چون ما به فیلد داده‌ای دسترسی داریم که بازنویسی نشده است و در اصل امکان بازنویسی آن وجود ندارد، همیشه فقط مقدار فیلد داده‌ای کلاس والد قابل دسترسی می‌باشد.

```
class Bike{
    int speedlimit = 90;
}

class Honda3 extends Bike{
    int speedlimit = 150;

    public static void main(String[] args){
        Bike obj = new Honda3();
        System.out.println(obj.speedlimit); //90
    }
}
```

# مثال از پلی مورفیسم زمان اجرا با ارث بری چند سطحی

```
class Animal{
    void eat(){
        System.out.println("eating");
    }
}

class Dog extends Animal{
    void eat(){
        System.out.println("eating fruits");
    }
}
```

```
class BabyDog extends Dog{
    void eat(){
        System.out.println("drinking milk");
    }
}

public static void main(String[] args){
    Animal a1, a2, a3;
    a1 = new Animal();
    a2 = new Dog();
    a3 = new BabyDog();

    a1.eat();
    a2.eat();
    a3.eat();
}
```

eating  
eating fruits  
drinking milk

# مثال دیگر

از آنجایی که eat() متد از کلاس Dog فراخوانی می‌شود.

```
class Animal{
    void eat(){
        System.out.println("animal is eating..");
    }
}

class Dog extends Animal{
    void eat(){
        System.out.println("dog is eating...");
    }
}

class BabyDog1 extends Dog{
    public static void main(String[] args){
        Animal a = new BabyDog1();
        a.eat();
    }
}
```

dog is eating...

# مقید سازی ایستا و مقید سازی پویا

ارتباط دادن یک فراخوانی متده با بدنه متده، اصطلاحاً **مقید سازی (binding)** متده به پیاده سازی آن گفته می شود.

به طور کلی دو نوع مقید سازی داریم: **مقید سازی ایستا** یا به عبارتی مقید سازی زودهنگام و **مقید سازی پویا** که به آن مقید سازی دیرهنگام هم گفته می شود.



قبل از توضیح انواع محدودسازی بهتر است به فهم نوع (type) فیلدها بپردازیم.  
ابتدا نوع یک نمونه (instance type) را به طور دقیق‌تر بررسی می‌کنیم.  
متغیرها دارای نوع هستند : هر متغیر نوعی دارد که می‌تواند **اصلی (primitive)** یا **ارجاعی (reference)** باشد.

```
int data = 30;
```

ارجاع‌ها نیز دارای نوع می‌باشند.

```
class Dog{  
  
public static void main(String[] args){ Dog d1; //Here d1 is a type of Dog}  
}
```

اشیا دارای نوع هستند: یک شی، نمونه‌ای از یک کلاس مشخص جاوا می‌باشد، اما در عین حال نمونه‌ای از کلاس والدش هم هست.

```
class Animal{  
  
class Dog extends Animal{  
  
public static void main(String[] args){ Dog d1 = new Dog() }  
}
```

**مقیدسازی ایستا:** هرگاه نوع یک شی در زمان کامپایل (توسط کامپایلر) مشخص شود، می‌گوییم **مقیدسازی ایستا** انجام شده است.

**مقیدسازی پویا:** هرگاه نوع شی در زمان اجرا مشخص شود، می‌گوییم **مقیدسازی پویا** انجام شده است.

مقیدسازی پویا به این شکل عمل می‌کند: فرض کنید شی  $o$  یک نمونه از کلاس‌های  $C_1, C_2, \dots, C_{n-1}, C_n$  باشد، به طوری که  $C_1$  یک زیرکلاس  $C_2$  و  $C_2$  یک زیرکلاس  $C_3$  و ... و  $C_{n-1}$  یک زیرکلاس  $C_n$  باشد. یعنی  $C_n$  عمومی‌ترین کلاس باشد و  $C_1$  اختصاصی‌ترین کلاس باشد. در جاوا  $C_n$  همان کلاس `Object` است. اگر  $o$  یک متاد  $p$  را فراخوانی کند، JVM به دنبال پیاده‌سازی متاد  $p$  به ترتیب در  $C_1, C_2, \dots, C_n$  می‌گردد تا زمانی که متاد مربوطه را بیابد. به محض یافتن متاد، جستجو خاتمه یافته و پیاده‌سازی شناسایی شده فراخوانی می‌شود.



/\*مقدیدسازی ایستا\*/

```
class Dog{  
    private void eat() {  
        System.out.println("dog is eating...");  
    }  
  
    public static void main(String[] args){  
        Dog d1 = new Dog();  
        d1.eat();  
    }  
}
```

dog is eating...

```
/*مقیدسازی بوسیل*/\n\n\nclass Animal{\n    void eat() {\n        System.out.println("animal is eating...")\n    }\n}\n\n\nclass Dog extends Animal{\n    private void eat() {\n        System.out.println("dog is eating...");\n    }\n\n    public static void main(String[] args){\n        Animal a = new Dog();\n        a.eat();\n    }\n}
```

dog is eating...

در مثال بالا نوع شی نمی‌تواند در زمان کامپایل توسط کامپایلر مشخص شود، چون یک شی (نمونه) از کلاس Dog یک نمونه از کلاس Animal نیز می‌باشد.  
در نتیجه کامپایلر نوع دقیق آن را نمی‌شناسد.

# تطبیق متد در مقایسه با مقيدسازی متد یا به عبارتی Binding

تطبیق امضای یک متد (signature) و مقيدسازی پياده‌سازی یک متد، دو موضوع مختلف هستند.

کامپایلر تطبیق متد را براساس نوع پارامترها، تعداد و ترتیب پارامترها را در زمان کامپایل انجام می‌دهد.

یک متد ممکن است در چندین زیرکلاس بازنویسی شده باشد.

ماشین مجازی جاوا، به صورت پویا تشخیص می‌دهد که کدام پياده‌سازی متد را در نظر بگیرد (مقيدسازی کند).

---

# تبديل اشیا (Casting Objects)

در جلسات اول گفتیم که عملگر casting برای تبدیل متغیرهای یک نوع اصلی به نوع اصلی دیگر استفاده می‌شود.

این عملگر را می‌توان برای تبدیل یک شی از یک نوع کلاس به شی‌ای از کلاسی دیگر از طریق سلسله مراتب وراثتی مورد استفاده قرار داد.

متد زیر را در نظر بگیرید.

```
public static void m(Object x){ System.out.println(x.toString()); }
```

در این متد یک متغیر ارجاعی از کلاس Object به عنوان پارامتر به متد ارسال می‌شود.

دستور زیر شی () new Student() را از نوع کلاس پدر جدش (Object) تعریف می‌کند.

```
Object o = new Student();
```

دستور بالا یک تبدیل رو به بالا Upward casting نامیده می‌شود و تبدیلی ضمنی (Implicit casting) است. این تبدیل در جاوا مجاز است چون نمونه از کلاس Student به طور خودکار یک نمونه از کلاس Object به حساب می‌آید.

## سوالی مهم پیش می‌آید: چرا تبدیل لازم است؟

فرض کنید می‌خواهید ارجاع به شی `o` از کلاس `Object` را به متغیری از نوع `Student` با استفاده از دستور زیر انتساب دهید.

با این دستور، یک خطای کامپایلری رخ خواهد داد. سوال اینجاست: چرا دستور `Object o =new Student()` اجرا می‌شود اما `o` کامپایل نمی‌شود؟

دلیل این است که شی `Student` همیشه یک نمونه از کلاس `Object` به حساب می‌آید، اما یک شی از `Object` لزوماً یک نمونه از `Student` نیست. حتی اگر از نظر شما `o` واقعاً شی‌ای از `Student` باشد، کامپایلر آنقدر باهوش نیست که این مسئله را درک کند.

برای اینکه به کامپایلر بگوییم `o` یک شی از `Student` است، از تبدیل آشکار (`explicit`) استفاده می‌کنیم. قاعده مورد استفاده، مشابه قاعده تبدیل نوع اصلی می‌باشد (استفاده از دو پرانتز برای نوع شی‌ای که می‌خواهیم تبدیل را بر روی آن انجام دهیم).

```
Student b = (Student)o; //Explicit casting
```

---

# تبديل از کلاس والد به کلاس فرزند (Downcasting)

تبديل آشكار يا همان Explicit در هنگام تبديل ريموت كنترل يك شي از جنس کلاس والد به کلاس فرزندش استفاده مي شود. اين تبديل هميسه موفقيتآمييز نمي باشد!

```
Apple x = (Apple)fruit;
```

```
Orange x = (Orange)fruit;
```

## عملگر instanceof

از اين عملگر به اين هدف استفاده مي کنيم تا تست کنيم آيا يك شي نمونه اي از کلاس موردنظر مي باشد يا خير؟

```
Object myObject = new Circle();
...//Some lines of code
/* Perform casting if myObject is an instance of Circle */
if(myObject instanceof Circle){
    System.out.println("The circle diameter is " + ((Circle)myObject).getDiameter());
}
...
```

## مثال‌هایی از عملگر : instanceof

یک شی از کلاس فرزند، یک شی از کلاس والد نیز هست. برای مثال، اگر Dog از Animal ارث بری کند، شی‌ای از Dog هم توسط کلاس Dog و هم توسط Animal می‌تواند مورد ارجاع قرار بگیرد.

```
class Animal { }

class Dog1 extends Animal {
    public static void main(String[] args) {
        Dog1 d = new Dog1();
        System.out.println(d instanceof Animal);
    }
}
```

true

اگر عملگر instanceof را با متغیر ارجاعی استفاده کنیم که دارای مقدار null است، false برخواهد گرداند.

مثال زیر را ببینید:

```
class Dog2 {  
    public static void main(String[] args) {  
        Dog2 d = null;  
        System.out.println(d instanceof Dog2);  
    }  
}
```

false

## تبدیل رو به پایین (Downcasting) با عملگر instanceof

وقتی یک متغیر ارجاعی از کلاس فرزند شی‌ای از کلاس والد را کنترل کند، می‌گوییم تبدیل رو به پایین انجام شده است. اگر این تبدیل مستقیماً انجام شود، کامپایلر خطأ خواهد گرفت.

```
Dog d = new Animal(); //Compilation error
```

اگر مستقیماً با typecasting انجام شود (گذاشتن کلاس فرزند در پرانتز)، دچار خطای مان اجرا خواهد شد!

```
Dog d = (Dog)new Animal();
```

```
//Compiles successfully but ClassCastException is thrown at runtime
```

اما به کمک عملگر instanceof و با بررسی نوع شی، تبدیل رو به پایین انجام خواهد شد.

```
class Animal{ }

class Dog3 extends Animal {
    static void method(Animal a) {
        if(a instanceof Dog3) {
            Dog3 d = (Dog3)a; //downcasting
            System.out.println("ok downcasting performed");
        }
    }

    public static void main(String[] args) {
        Animal a = new Dog3();
        Dog3.method(a);
    }
}
```

ok downcasting performed

# تبدیل رو به پایین استفاده نکنیم instanceof از عملگر (Downcasting)

اگر نخواهیم از عملگر instanceof برای تبدیل رو به پایین استفاده کنیم، به شکل زیر عمل می‌کنیم:

```
class Animal { }
class Dog4 extends Animal {
    static void method(Animal a) {
        Dog4 = (Dog4)a; //downcasting
        System.out.println("ok downcasting performed");
    }
}

public static void main(String[] args) {
    Animal a = new Dog4();
    Dog4.method(a);
}
```

در این مثال چون a را از کلاس والد تعریف کردیم که به شی کلاس فرزند اشاره می‌کند، شی ایجاد شده، در واقع نمونه‌ای از کلاس Dog می‌باشد. در نتیجه در داخل method a متغیر a بدون اشکال به متغیر Dog تبدیل می‌شود.

اما اگر دستور را به صورت زیر بنویسیم با خطای زمان اجرا رو به رو می‌شویم:

```
;()Animal a = new Animal  
;Dog4.method(a)
```

Now ClassException but not in case of instanceof operator//

---

```
interface printable{  
  
class A implements printable{  
    public void a() {  
        System.out.println("a method");  
    }  
}  
  
class B implements printable{  
    public void b() {  
        System.out.println("b method");  
    }  
}
```

```
class Call{  
    void invoke(printable p){//upcasting  
        if(p instanceof A){  
            A a = (A)p; //downcasting  
            a.a();  
        }  
        if(p instanceof B){  
            B b = (B)p; //downcasting  
            b.b();  
        }  
    }  
}
```

```
class Test4{  
    public static void main(String[] args){  
        printable p = new B();  
        Call c = new Call();  
        c.invoke(p);  
    }  
}
```

b method

# مثال پلی مورفیسم و casting

برنامه‌ای بنویسید که دو شی هندسی ایجاد کند: یک دایره، یک مستطیل

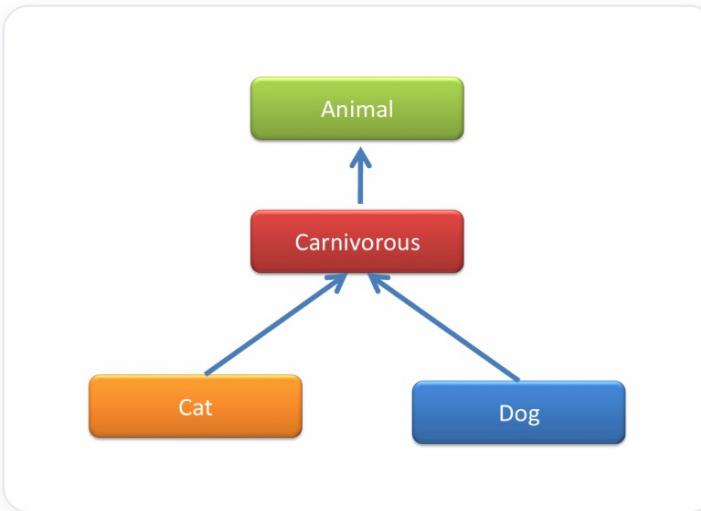
کافیست کلاسی جدید تعریف کنید و متدهای main را درون آن بنویسید. سپس از کلاس والد Object دو متغیر ارجاعی (object2, object1)، یکی به دایره و دیگری به مستطیل تعریف کنید.

سپس متدهای displayGeometricObject را برای دو شی جدید فراخوانی کنید.

پس از اتمام متدهای main را این گونه تعریف کنید: این متدهای استاتیک، پارامتری از نوع Object داشته باشد. اگر پارامتر ارسالی دایره بود، یک تبدیل رو به پایین انجام شده و متدهای مساحت و قطر دایره فراخوانی شوند. اگر پارامتر ارسالی مستطیل بود، متدهای مساحت آن فراخوانی شود.

```
public class CastingDemo {  
  
    /** Main method */  
    public static void main(String[] args) {  
        // Declare and initialize two objects  
        Object object1 = new Circle4(1);  
        Object object2 = new Rectangle1(1, 1);  
  
        // Display circle and rectangle  
        displayObject(object1);  
        displayObject(object2);  
    }  
  
    /** A method for displaying an object */  
    public static void displayObject(Object object) {  
        if (object instanceof Circle4) {  
            System.out.println("The circle area is " + ((Circle4)object).getArea());  
            System.out.println("The circle diameter is " +((Circle4)object).getDiameter());  
        }  
        else if (object instanceof Rectangle1) {  
            System.out.println("The rectangle area is " +(Rectangle1)object).getArea());  
        }  
    }  
}
```

# تبديل رو به بالا / رو به پايين - مثال جانوران



به اين نكته توجه کنيد که: با تبديل اشيا شما هويت واقعی شی را تغيير نمی دهيد. بلکه به شی مربوطه برجسب متفاوتی می زنيد.

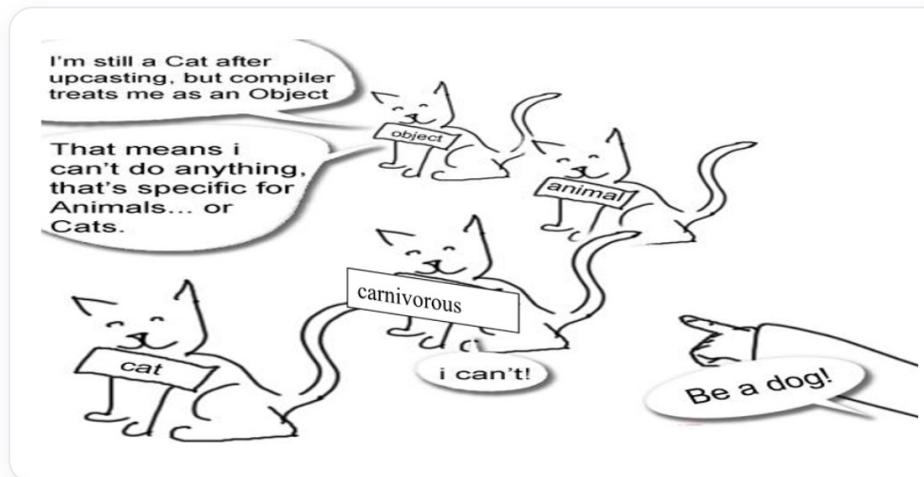
برای مثال، اگر شما یک Cat ایجاد کرده و بر روی آن تبديل رو به بالا انجام دهيد، این تبديل سبب نمی شود که شی اوليه دیگر گربه (Cat) نباشد!

این شی همچنان گربه است. اما با او مانند هر جانور (Animal) دیگری رفتار خواهد شد و خصوصيات (فیلدهای) ویژه گربه پنهان می شود تا زمانی که دوباره به گربه تبديل رو به پايين شود.

همانطور که در کد مربوط به اين مثال مشاهده می کنيد، Cat پس از تبديل همچنان Cat باقی مانده و به جای آن یک شی گوشتخوار (Carnivorous) ایجاد نشده است. فقط برجسب گوشتخوار به آن زده شده که کاملا مجاز است، چون یک گربه، یک گوشتخوار هم می باشد!

```
Cat c = new Cat();  
System.out.println(c);  
Carnivorous m = c; // upcasting  
System.out.println(m);  
/* This printed:  
Cat@a90653  
Cat@a90653 */
```

توجه داشته باشید اگرچه دو شی گوشتخوار هستند، گربه نمی‌تواند به سگ تبدیل شود. شکل زیر این مطلب را نشان می‌دهد:



## خودکار انجام شدن تبدیل رو به بالا:

برای تبدیل رو به بالا نیاز نیست برنامهنویس به صورت دستی این کار را انجام دهد. برای مثال

Carnivorous m = new Cat(); معادل است با Carnivorous m = (Carnivorous)new Cat();

اما تبدیل رو به پایین همیشه باید به صورت دستی انجام شود:

```
Cat c1 = new Cat();
```

```
Animal a = c1; //automatic upcasting to Animal
```

```
Cat c2 = (Cat) a; //manual downcasting back to a Cat
```

---

# تفاوت تبدیل رو به بالا و تبدیل رو به پایین

چرا تبدیل رو به بالا به صورت خودکار انجام می‌شود اما تبدیل رو به پایین باید به طور دستی انجام شود؟

چون تبدیل رو به بالا هرگز با خطا رو به رو نخواهد شد. اما اگر مجموعه‌ای از جانوران مختلف داشته باشیم و بخواهیم با تبدیل رو به پایین همه آنها را به گربه تبدیل کنیم، در زمان اجرا این احتمال وجود دارد که برخی از این جانوران واقعاً گربه نباشند (مثلًا اردک باشند) و در نتیجه فرآیند تبدیل رو به پایین با پیام ClassCastException با شکست مواجه خواهد شد.

به همین دلیل است که از عملگر instanceof استفاده می‌کنیم تا تست کند آیا شی مربوط، نمونه‌ای از کلاس گربه می‌باشد یا خیر.

مثال زیر را در نظر بگیرید :

```
Cat c1 = new Cat();  
  
Animal a = c1; //upcasting to Animal  
  
if(a instanceof Cat){ // testing if the Animal is a Cat  
  
System.out.println("It's a Cat!  
  
Now i can safely downcast it to a Cat, without a fear of failure");  
  
Cat c2 = (Cat)a;}
```

مشخص است که تبدیل همواره در هر دو جهت قابل انجام نمی باشد. اگر با دستور "new Carnivorous()" یک شی گوشتخوار ایجاد کرده باشید شی مورد نظر نمی تواند مستقیما به یک سگ یا یک گربه تبدیل شود، چون در واقع هیچ یک از آنها نمی باشد!

برای مثال تبدیل زیر نادرست است.

```
Carnivorous m = new Carnivorous();
```

```
Cat c = (Cat)m;
```

این کد با پیام زمان اجرای `java.lang.ClassCastException` را به زور می‌خواهیم گوشت‌خواری را که لزوماً گربه نیست، به گربه تبدیل کنیم.

در حالت کلی برای اینکه بدانیم چه موقع از تبدیل مناسب استفاده کنیم، این سوال را از خود بپرسیم: آیا گربه یک گوشت‌خوار است؟ بله، هست. پس تبدیل گربه به گوشت‌خوار قابل انجام است!

دوباره بپرسیم: آیا یک گوشت‌خوار یک گربه است؟ پاسخ، خیر است. پس تبدیل گوشت‌خوار به گربه نمی‌تواند انجام بشود.

---

# 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 را مرور کن و سعی کن بدون نگاه کردن به متن درس، به اون ها پاسخ بدی.

- مهم‌ترین کاربرد عملگر instanceof توی چیه؟
- موقع بازنویسی متدهای ابرکلاس های اون باید حواسمن به چه نکاتی باشه؟
- تفاوت بین مقیدسازی ایستا و پویا در چه چیزیه؟

## پایان

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

gmail: sarashiri0906@gmail.com