دانشکده مهندسی و علوم کامپیوتر

برنامه نویسی پیشرفته

وحیدی اصل

ارثبری- مقیدسازی- پلی مورفیسم

اعلان یک زیرکلاس

- یک زیرکلاس فیلدهای داده ای و متدها را از ابرکلاس (پدرش) ارث بری می کند. یک فرزند علاوه بر ارث بری فیلدها از پدر می تواند:
  - فیلدهای داده ای جدید داشته باشد.
  - متدهای جدید داشته باشد.
  - متدهای ابرکلاس را بازنویسی (override) کند.

بازنویسی متدهای ابرکلاس در زیرکلاسها

یک زیرکلاس متدهای والدش (ابرکلاس) را ارث بری می کند. برخی مواقع لازم است یک زیرکلاس، پیاده سازی یک متد تعریف شده در ابرکلاسش را تغییر دهد. به این تغییر اصطلاحاً بازنویسی متد (method overriding) گفته می شود.

به عبارت دیگر هرگاه متدی در زیرکلاس بخواهد پیاده سازی خاص خود از متد ارث برده از ابرکلاس را داشته باشد، از بازنویسی استفاده می کند.

مزیت بازنویسی متد در پلی مورفیسم زمان اجرا می باشد.

```
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;
  }
}
```

بازنویسی متدهای ابرکلاس در زیرکلاسها

```
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;
  }
}
```

```
//from class GeometricObject
/** Return a string representation of this object */
  public String toString() {
    return "created on " + dateCreated + "\ncolor: " + color +  " and filled: " + filled;
  }
}
```

- متد باید همنام با متد کلاس والد باشد.
- متد باید پارامتر(های) مشابه کلاس والد را داشته باشد.
- رابطه وراثت برقرار باشد.

اگر از بازنویسی متد استفاده نکنیم

```
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();
}
}
```

Test it Now

```
Output:Vehicle is running
```

مسئله اینجاست: اینکه بخواهیم یک پیاده سازی ویژه از متد run() در زیرکلاس دوچرخه داشته باشیم، باید متد run() را که از کلاس وسیله نقلیه به ارث رسیده است بازنویسی (override) کنیم.

مثال استفاده از بازنویسی متد

- توجه داشته باشید که نام و نوع پارامتر متد بازنویسی شونده باید با متد ابرکلاس یکی باشد.

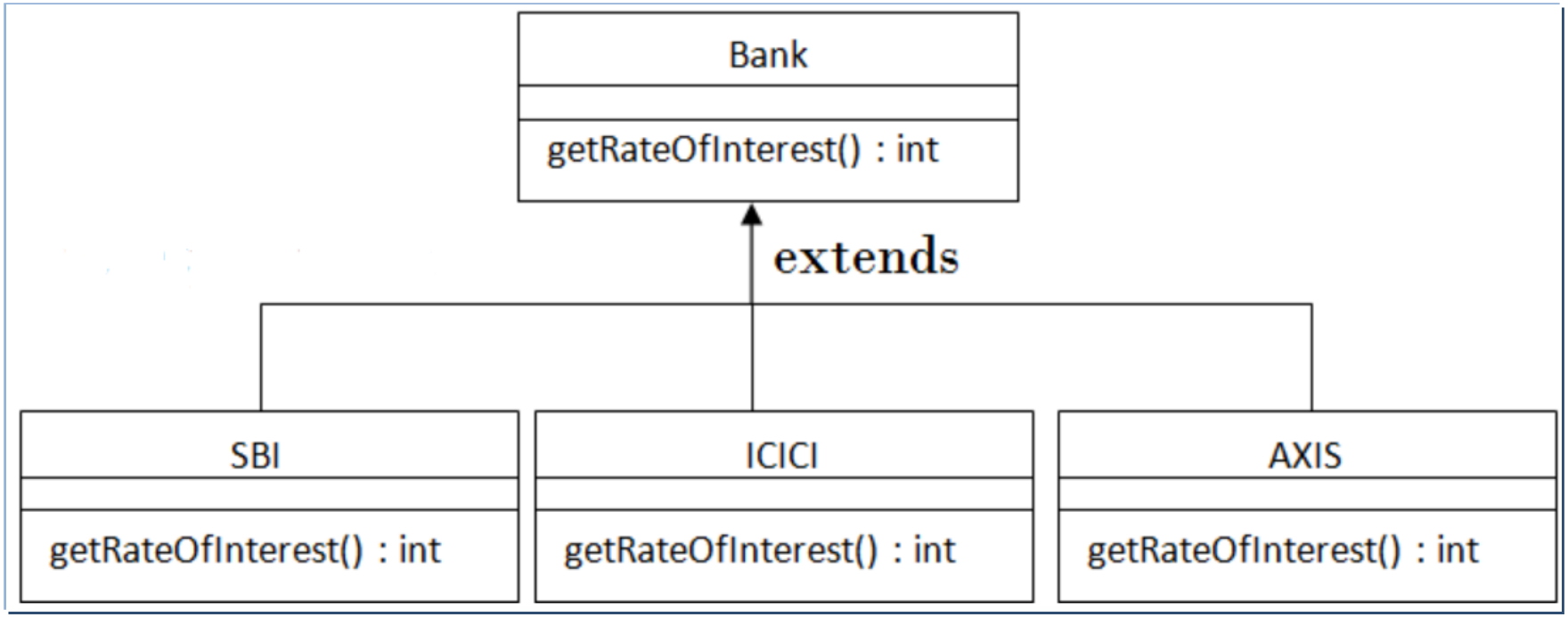
```
class Vehicle{
void run(){System.out.println("Vehicle is running");}
}
class Bike2 extends Vehicle{
void run(){System.out.println("Bike is running safely");}

public static void main(String args[]){
Bike2 obj = new Bike2();
obj.run();
}
```

Test it Now

```
Output:Bike is running safely
```

مثالی از بازنویسی متد–سود بانکی

- سناریوی زیر را در نظر بگیرید: کلاسی به نام Bank داریم که نرخ سود سپرده را برمی گرداند. اما این نرخ سود در هر بانک با بانک دیگر متفاوت است. فرض کنید سه بانک داریم به نامهای SBI، ICIC و AXIS که به ترتیب نرخ سود ۸٪، ۷٪ و ۹٪ برمی گردانند.



مثالی از بازنویسی متد–سود بانکی

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

Test it Now

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

نکات بازنویسی متد

- چرا یک متد استاتیک قابل بازنویسی نمی باشد؟
  - زیرا متد استاتیک به یک کلاس مقید شده است نه به اشیای کلاس! اما یک متد نمونه به یک شیئ از کلاس مقید شده است. متدها و فیلدهای استاتیک متعلق به تعریف کلاس در حافظه ایستا هستند که در زمان اجرا قابل تغییر نمی باشند. اما متدهای نمونه در حافظه heap ایجاد می شوند و امکان بازنویسی (تغییر) آنها وجود دارد.

- آیا می توانیم متد main در جاوا را بازنویسی کنیم؟
  - خیر، چون این متد به صورت استاتیک تعریف می شود.

نکته

یک متد نمونه تنها در صورت در دسترس بودن، قابل بازنویسی می باشد.

بنابراین یک متد private نمی تواند بازنویسی شود، چون در خارج از کلاس خود قابل دسترسی نمی باشد.

اگر متد تعریف شده در یک زیرکلاس همنام با متدی در ابرکلاسش باشد که private تعریف شده است، این دو متد کاملاً بی ربط با یکدیگر در نظر گرفته می شوند.

مانند متد نمونه، یک متد استاتیک نیز می تواند ارث بری شود.

با اینحال، یک متد استاتیک نمی تواند بازنویسی شود. اگر متد استاتیک تعریف شده در یک ابرکلاس در یک زیرکلاسش بازتعریف شود، متد تعریف شده در ابرکلاس پنهان (پوشیده) خواهد شد.

Overloading درمقایسه با 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);
  }
}
```

```
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 overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

Overloading درمقایسه با Overriding

- سه تفاوت اصلی میان بازنویسی و سربارگذاری متدها وجود دارد:

بازنویسی متد	سربارگذاری متد
۱) به برنامه نویس امکان می دهد پیاده سازی خاص متد ارث بری شده از کلاس والد را ایجاد کند.	به هدف افزایش خوانایی برنامه ها ایجاد می شود.
همواره در دو کلاسی انجام می شود که رابطه ارث بری میان آنها وجود دارد.	در درون یک کلاس انجام می شود.
پارامترها باید یکسان باشند.	پارامترها باید متفاوت باشند.

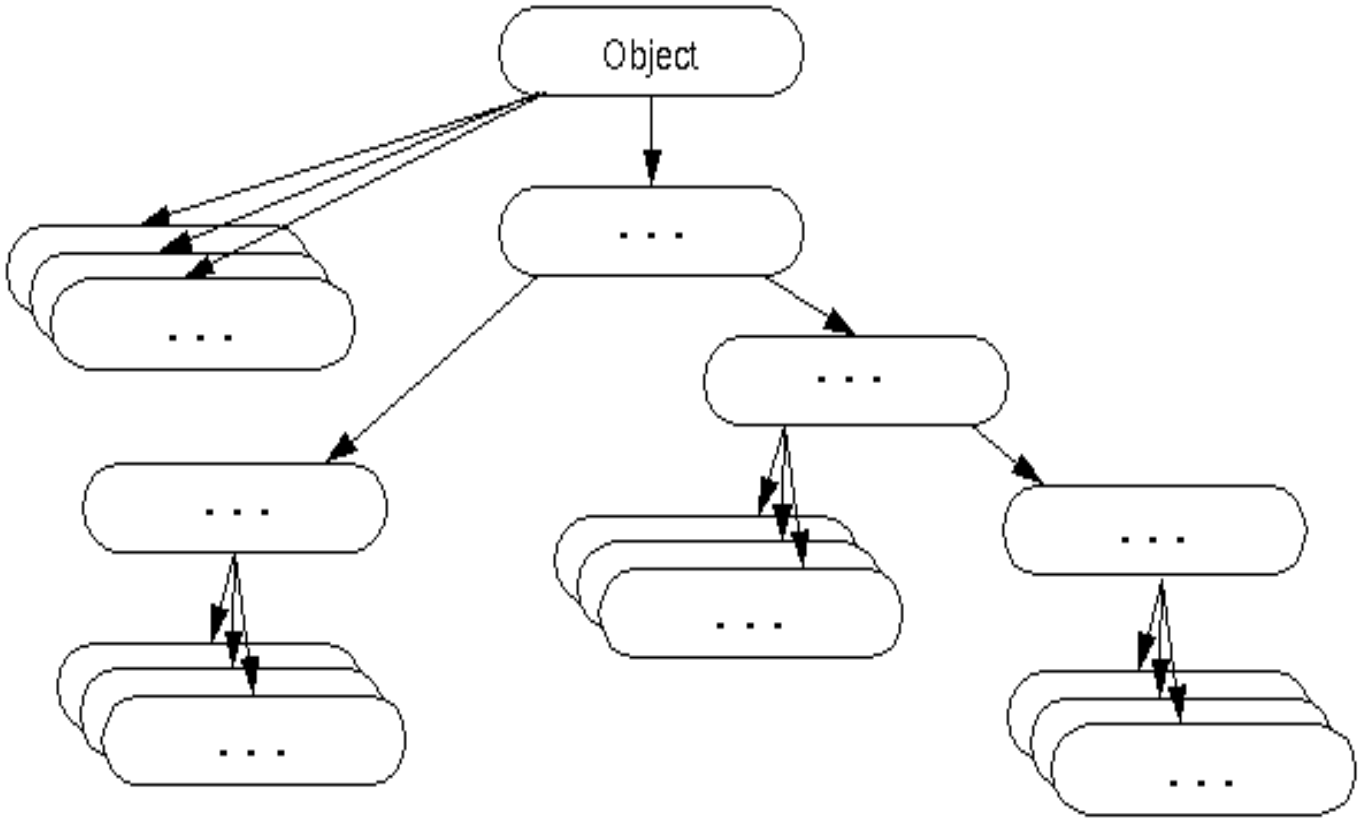
هرکلاس در جاوا به نوعی از نسل کلاس java.lang.Object به حساب می آید. اگر در تعریف یک کلاس، ارث بری اعلام نشود، به طور ضمنی ابرکلاس آن کلاس Object می باشد.

```
public class Circle {
   ...
}
```

Equivalent

```
public class Circle extends Object {
   ...
}
```

کلاس Object پدر همه کلاسهای جاوا است!



متدهای موجود در کلاس Object

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

متد toString() در کلاس Object

متد toString() رشته ای را بر می گرداند که نمایش دهنده شیئ حاصل از کلاس مربوطه است. پیاده سازی پیش فرض این متد، رشته ای حاوی نام کلاس مربوط به شیئ را برمی گرداند که به دنبال آن علامت @ و شماره شیئی امده است.

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

کد بالا عبارتی نظیر Loan@15037e5 را چاپ می کند.

این پیغام اطلاعات زیادی در اختیار برنامه نویس قرار نمی دهد. در نتیجه در اکثر مواقع متد بازنویسی می شود تا رشته ای حاوی اطلاعات جامع تر در اختیار کاربر قرار دهد.

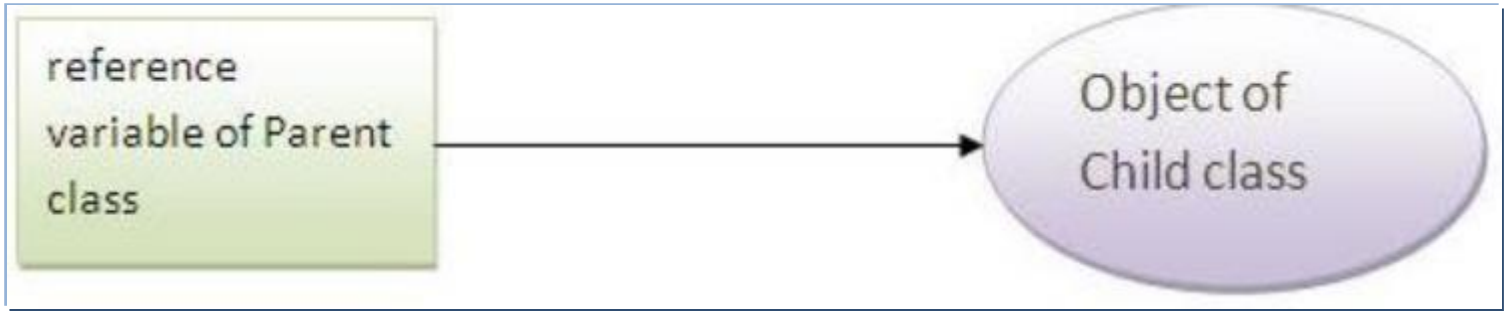
بازنویسی متد toString() در زیرکلاسها

```
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;
  }
}
```

```
//from class GeometricObject
/** Return a string representation of this object */
  public String toString() {
    return "created on " + dateCreated + "\ncolor: " + color +  " and filled: " + filled;
  }
}
```

تبدیل روبه بالا (Upcasting)

- هرگاه متغیر ارجاعی کلاس والد به شیئ کلاس فرزند اشاره کند، می گوییم تبدیل روبه بالا(upcasting) انجام شده است.




```
class A{}
class B extends A{}

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

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

مثالی از پلی مورفیسم زمان اجرا

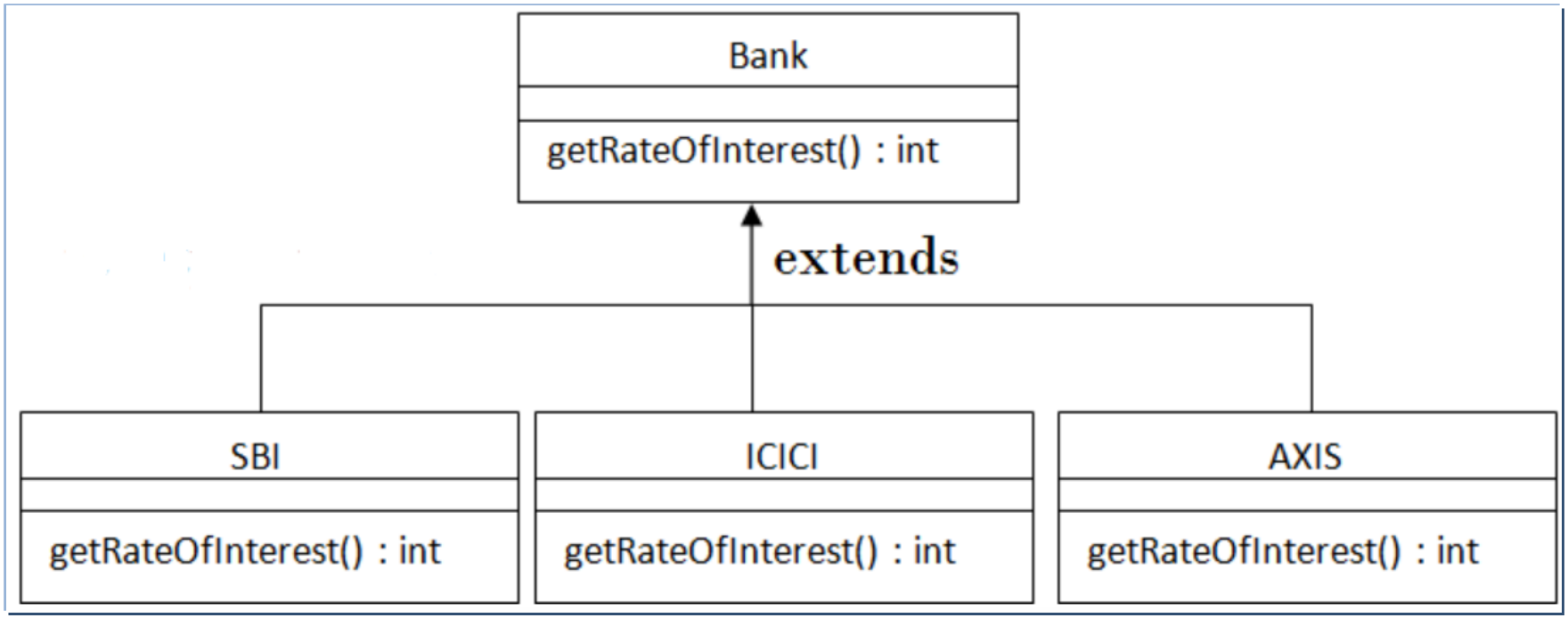
- در این مثال،دو کلاس Bike و Splendor تعریف شده است . کلاس Splender از کلاس Bike ارث بری می کند و متد run() را بازنویسی می کند.
- اینک، متد run() را از طریق متغیر ارجاعی کلاس والد فراخوانی می کنیم.
- چون متغیر ارجاعی به شیئ زیرکلاس اشاره می کند و متد زیرکلاس، متد ابرکلاسش را بازنویسی کرده است، در زمان اجرا متد زیرکلاس فراخوانی می شود.
- چون فراخوانی متد توسط JVM و نه کامپایلر انجام می شود، 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();
  }
}
```

Test it Now

```
Output:running safely with 60km.
```

- سناریوی بانک را در نظر بگیرید و با استفاده از پلی مورفیسم متد getRateOfInterest() را فراخوانی کنید.



مثالی دیگر از پلی مورفیسم زمان اجرا–ادامه

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

Test it Now

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

پلی مورفیسم زمان اجرا با فیلد داده ای

- همیشه متدها هستند که بازنویسی می شوند، نه فیلدهای داده ای کلاس.
- بنابراین پلی مورفیسم برروی فیلدهای داده ای قابل دستیابی نمی باشد.
- در مثال زیر، هر دو کلاس دارای فیلد داده ای speedlimit هستند. دسترسی به این فیلد داده ای توسط متغیرارجاعی کلاس والد است که به شیئ زیرکلاس اشاره می کند (ریموت کنترل آن است).
- چون ما به فیلد داده ای دسترسی داریم که بازنویسی نشده است (امکان بازنویسی آن وجود ندارد)، همیشه فقط مقدار فیلد داده ای کلاس والد قابل دسترسی می باشد.

```
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
}
```

Test it Now

پلی مورفیسم زمان اجرا با ارث بری چندسطحی-مثال

```
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();
}
}
```

Test it Now

```
Output: eating
        eating fruits
        drinking Milk
```

مثالی دیگر

از آنجایی که BabyDog متد eat() را بازنویسی نکرده است، متد eat() از کلاس Dog فراخوانی می شود.

.

```
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();
}}
```

Test it Now

- ارتباط دادن یک فراخوانی متد به بدنه متد، اصطلاحاً مقیدسازی (binding) متد به پیاده سازی آن، گفته می شود.
- به طورکلی دو نوع مقیدسازی داریم:
- مقیدسازی ایستا (که به آن مقیدسازی زودهنگام نیزگفته می شود)
- مقیدسازی پویا(که به آن مقیدسازی دیرهنگام نیزگفته می شود)

- ابتدا نوع یک نمونه (instance type) را به طور دقیقتر بررسی می کنیم.

1. متغیرها دارای نوع هستند:

– هر متغیر نوعی دارد که می تواند اصلی (primitive) یا ارجاعی (reference) باشد.

```
int data=30;
```

۲. ارجاعها نیز دارای نوع می باشند.

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

3. اشیا دارای نوع هستند:

– یک شیئی، نمونه ای از یک کلاس مشخص جاوا می باشد، اما در عین حال نمونه ای 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 شیئ در زمان کامپایل (توسط کامپایلر) مشخص شود، می گوییم مقیدسازی ایستا انجام شده است.

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

 public static void main(String args[]){
  Dog d1=new Dog();
  d1.eat();
 }
}
```

- هرگاه نوع شیئی در زمان اجرا مشخص شود، می گوییم مقیدسازی پویا انجام شده است.

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

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

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

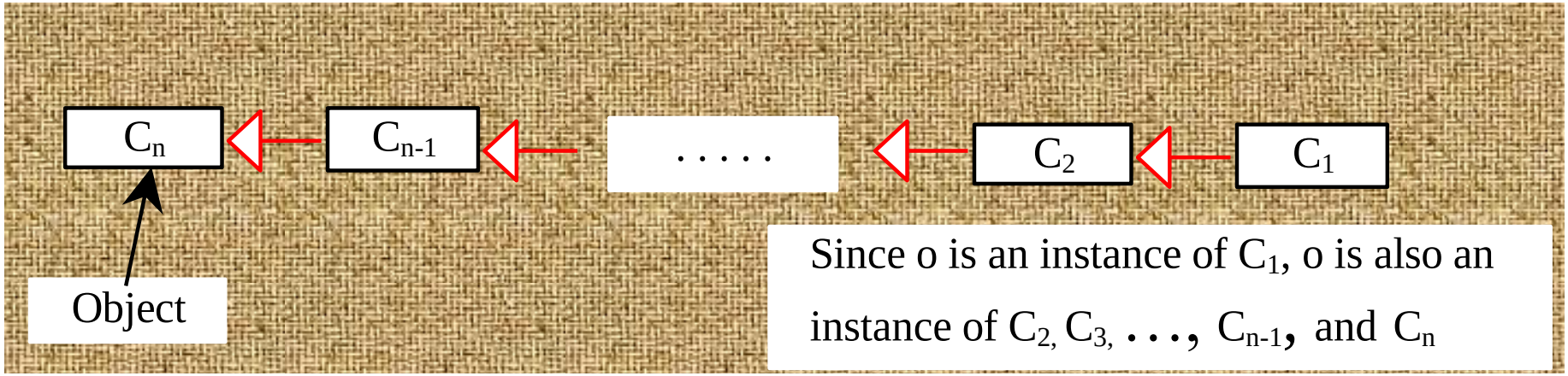
Test it Now

```
Output:dog is eating...
```

در مثال بالا، نوع شیئ نمی تواند در زمان کامپایل توسط کامپایلر مشخص شود، چون یک شیئ (نمونه) از کلاس Dog یک نمونه از کلاس Animal نیز می باشد. درنتیجه کامپایلر نوع دقیق آن را نمی شناسد؛

مقیدسازی پویا

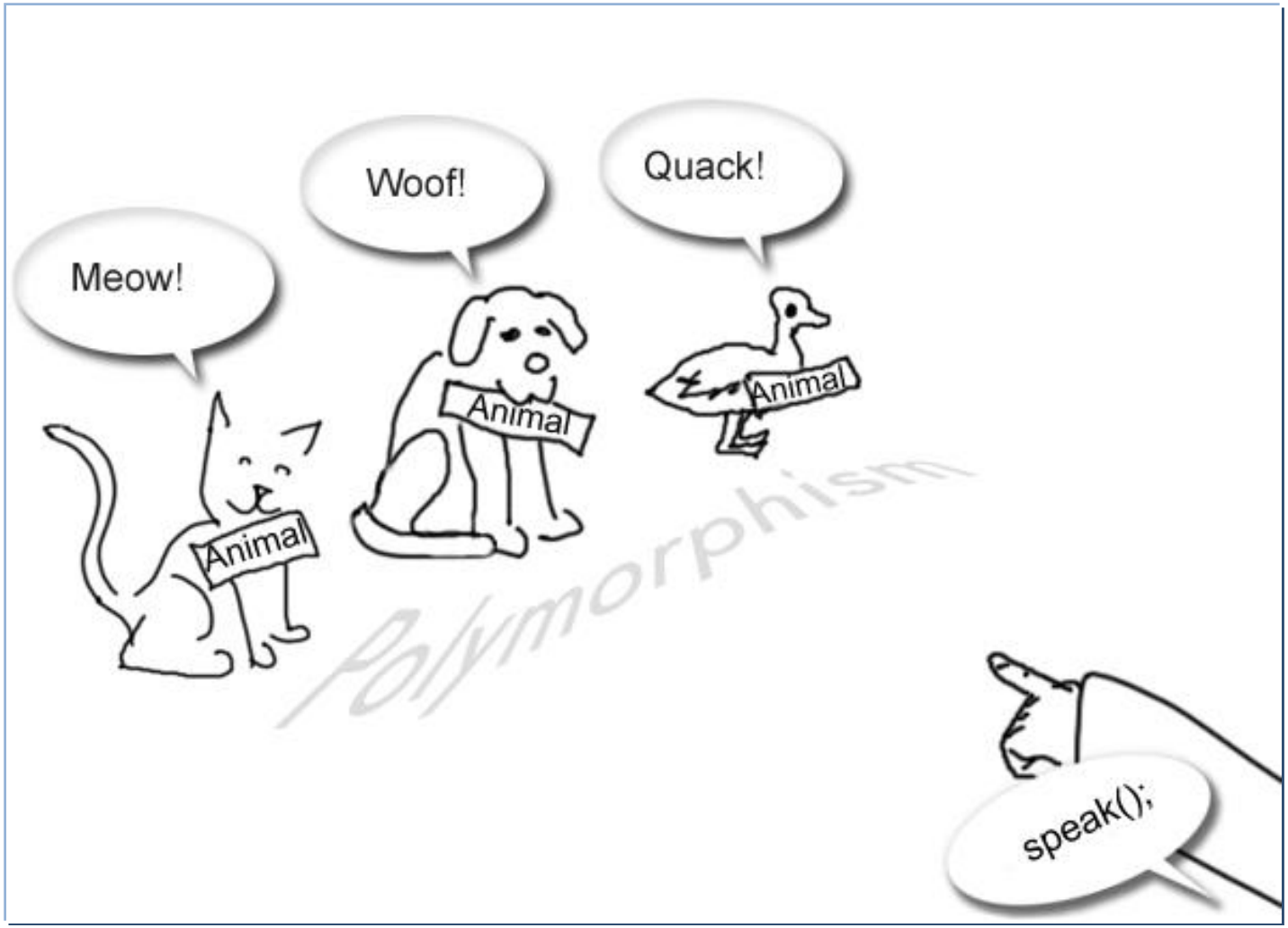
مقیدسازی پویا به صورت زیر عمل می کند: فرض کنید شیئ o یک نمونه از کلاسهای C_1, C_2, C_3, ..., C_{n-1}, C_n باشد به طوری که C_1 یک زیرکلاس C_2 و C_2 یک زیرکلاس C_3 و...و C_{n-1} یک زیرکلاس C_n باشد. یعنی، C_n عمومی ترین کلاس باشد و C_1 اختصاصی ترین کلاس باشد. در جاوا C_n کلاس Object است. اگر o یک متد p رافراخوانی کند، JVM به دنبال پیاده سازی متد p به ترتیب در C_1، C_2،...، C_n می گردد تا زمانی که متد مربوطه را بیابد. به محض یافتن متد، جستجو خاتمه یافته و پیاده سازی شناسایی شده فراخوانی می شود.



تطبیق متد در مقایسه با مقیدسازی متد(Method Matching vs. Binding)

- تطبیق امضای یک متد (signature) و مقیدسازی پیاده سازی یک متد، دو موضوع مختلف هستند.
- کامپایلر تطبیق متد را براساس نوع پارامترها، تعداد و ترتیب پارامترها را در زمان کامپایل انجام می دهد.
- یک متد ممکن است در چندین زیرکلاس بازنویسی شده باشد.
- ماشین مجازی جاوا، به صورت پویا تشخیص می دهد که کدام پیاده سازی متد را در زمان اجرا در نظر بگیرد (مقید سازی کند).

پلی مورفیسم با مثال



تبدیل اشیا (Casting Objects)

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

- این عملگر را می توان برای تبدیل یک شیئ از یک نوع کلاس به شیئ 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 می شود.

تبدیل اشیا (Casting Objects)

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

- دستور زیر شیئ new Student() را از نوع کلاس پدر جدش (Object) تعریف می کند.

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

- دستور بالا یک تبدیل روبه بالا (Upward casting) نامیده می شود و تبدیلی ضمنی (Implicit casting) است.
- این تبدیل در جاوا مجاز است، چون نمونه (شیئی) از کلاس Student به طور خودکار یک نمونه از کلاس Object به حساب می آید.

```
m(o);
```

چرا تبدیل لازم است؟

- فرض کنید می خواهید ارجاع به شیئ o از کلاس Object را به متغیری از نوع Student با استفاده از دستور زیر انتساب دهید.

```
Student b = o;
```

- با این دستور، یک خطای کامپایلری رخ خواهد داد. سوال اینجاست: چرا دستور Object o = new Student() اجرا می شود اما Student b = o کامپایل نمی شود؟
  - دلیل این است که شیئ Student همیشه یک نمونه از کلاس Object به حساب می آید، اما یک شیئ از Object لزوماً یک نمونه از Student نیست.
  - حتی اگر از نظر شما o واقعاً شیئی از Student باشد، کامپایلر آنقدر باهوش نیست که این مسئله را درک کند.
- برای اینکه به کامپایلر بگوییم o یک شیئ از Student است، از تبدیل اشکار (explicit) استفاده می کنیم.
- قاعده مورد استفاده، مشابه قاعده تبدیل نوع اصلی می باشد (استفاده از دو پرانتز برای نوع شیئی که می خواهیم تبدیل را برروی ان انجام دهیم.
- مثال:

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

تبدیل از کلاس والد به کلاس فرزند(Downcasting)

- تبدیل آشکار (Explicit) در هنگام تبدیل ریموت کنترل یک شیئ از جنس کلاس والد به کلاس فرزندش استفاده می شود.
- این تبدیل همیشه موفقیت آمیز نمی باشد!

```
Apple x = (Apple)fruit;

Orange x = (Orange)fruit;
```

عملگر instanceof

از عملگر instanceof به این هدف استفاده می کنیم تا تست کنیم آیا یک شیئ، نمونه ای از کلاس موردنظر می باشد یا خیر؟

```
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());
  ...
}
```

مثال–instanceof عملگر

```
class Simple1{
 public static void main(String args[]){
 Simple1 s=new Simple1();
 System.out.println(s instanceof Simple);//true
 }
}
```

Test it Now

```
Output:true
```

عملگر instanceof–مثالی دیگر

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

```
class Animal{}
class Dog1 extends Animal{//Dog inherits Animal

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

Test it Now

```
Output:true
```

instanceof با متغیر ارجاعی که دارای مقدار null است.

- اگر عملگر instanceof را با متغیرارجاعی 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
 }
}
```

Test it Now

```
Output:false
```

تبدیل روبه پایین (downcasting) با عملگر instanceof

- وقتی یک متغیر ارجاعی از کلاس فرزند شیئی از کلاس والد را کنترل کند، می گوییم تبدیل روبه پایین انجام شده است. اگر این تبدیل مستقیماً انجام شود کامپایلر خطا خواهد گرفت.

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

- اگر مستقیما با typecasting انجام شود(گذاشتن کلاس فرزند در پرانتز)، خطای زمان اجرای خواهد داد!

```
Dog d=(Dog)new Animal();
//Compiles successfully but ClassCastException is thrown at runtime
```

تبدیل روبه پایین (downcasting) با عملگر instanceof –ادامه

- اما به کمک عملگر instanceof و با بررسی نوع شیئ، 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);
  }

}
```

Test it Now

```
Output:ok downcasting performed
```

تبدیل روبه پایین (downcasting) اگر از عملگر instanceof استفاده نکنیم

- اگر نخواهیم از عملگر instanceof برای تبدیل روبه پایین استفاده نکنیم، به شکل زیر عمل می کنیم:

```
class Animal { }
class Dog4 extends Animal {
  static void method(Animal a) {
      Dog4 d=(Dog4)a;//downcasting
      System.out.println("ok downcasting performed");
  }
   public static void main (String [] args) {
    Animal a=new Dog4();
    Dog4.method(a);
  }
}
```

Test it Now

```
Output:ok downcasting performed
```

- در این مثال چون a را از کلاس والد تعریف کردیم که به شیئ کلاس فرزند اشاره می کند، شیئ ایجاد شده، در واقع نمونه ای از کلاس Dog می باشد. در نتیجه درداخل method متغیر a بدون اشکال به متغیر Dog تبدیل می شود.

تبدیل روبه پایین (downcasting)اگر از عملگر instanceof استفاده نکنیم

- اگر نخواهیم از عملگر instanceof برای تبدیل روبه پایین استفاده نکنیم، به شکل زیر عمل می کنیم:

```
class Animal { }
class Dog4 extends Animal {
  static void method(Animal a) {
      Dog4 d=(Dog4)a;//downcasting
      System.out.println("ok downcasting performed");
  }
   public static void main (String [] args) {
    Animal a=new Dog4();
    Dog4.method(a);
  }
}
```

Test it Now

```
Output:ok downcasting performed
```

- اما اگر دستور را به صورت زیر می نوشتیم با خطای زمان اجرا رو به رو می شدیم:

```
Animal a=new Animal();
Dog.method(a);
//Now ClassCastException but not in case of instanceof operator
```

کاربرد مهم عملگر instanceof

```
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();
}
 }
 }//end of Call class

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

Test it Now

```
Output: b method
```

مثال پلی مورفیسم و casting

برنامه ای بنویسید که دو شیئی هندسی ایجاد کند: یک دایره، یک مستطیل

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

سپس متد displayGeometricObject را برای دو شیئی جدید فراخوانی کنید.

پس از اتمام main متد displayGeometricObject را این گونه تعریف کنید: این متد 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، متد مساحت آن فراخوانی شود.

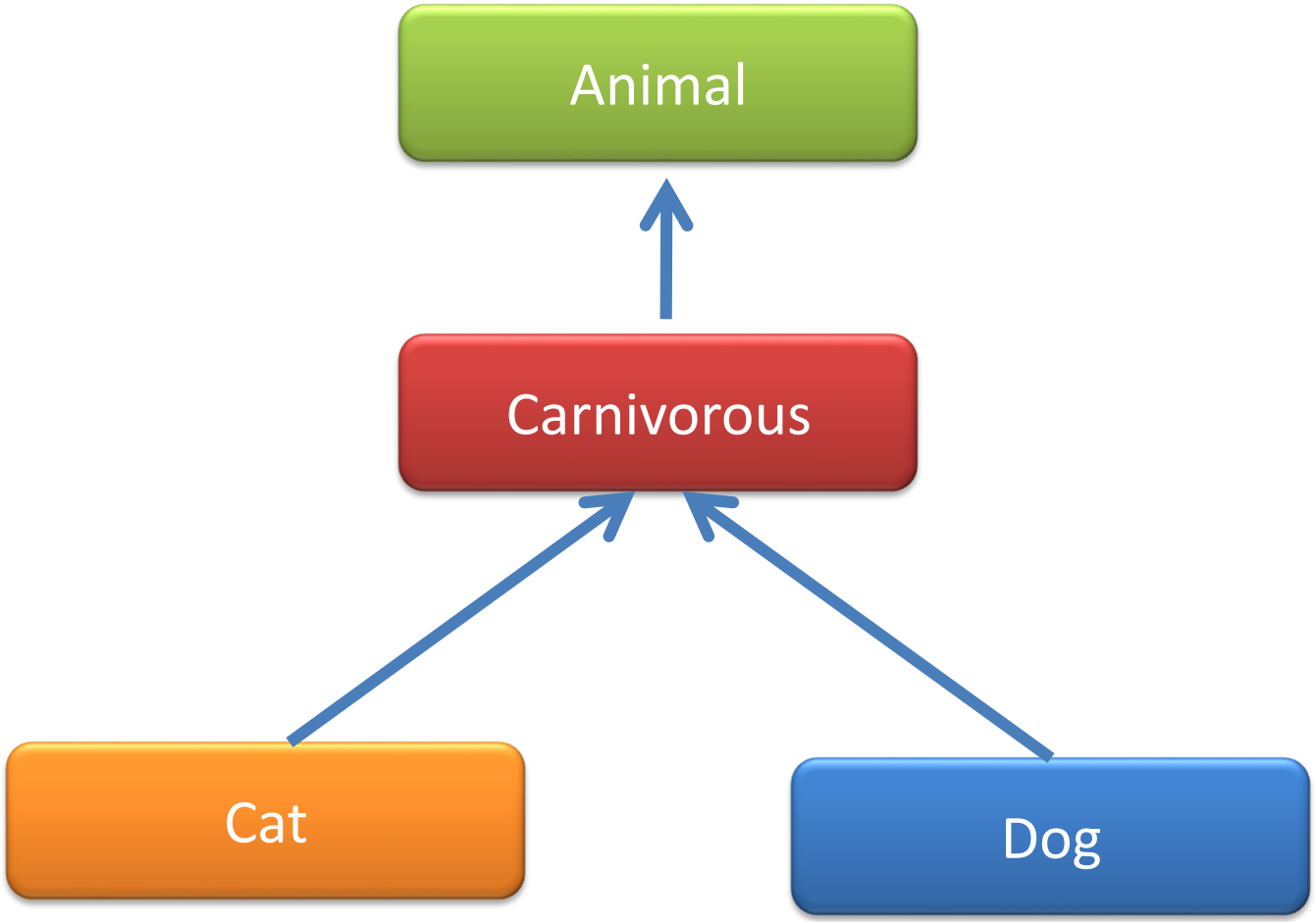
TestPolymorphismCasting

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

مثالی دیگر از تبدیل رو به پایین

```
public class CastingDemo {

/**
 * @param args
 */
public static void main(String[] args) {
    AOne obj = new Bone();
    ((Bone) obj).method2();
}
}

class AOne {
public void method1() {
    System.out.println("this is superclass");
}
}


 class Bone extends AOne {

public void method2() {
    System.out.println("this is subclass");
}
}
```

تبدیل رو به بالا/روبه پایین–مثال جانوران



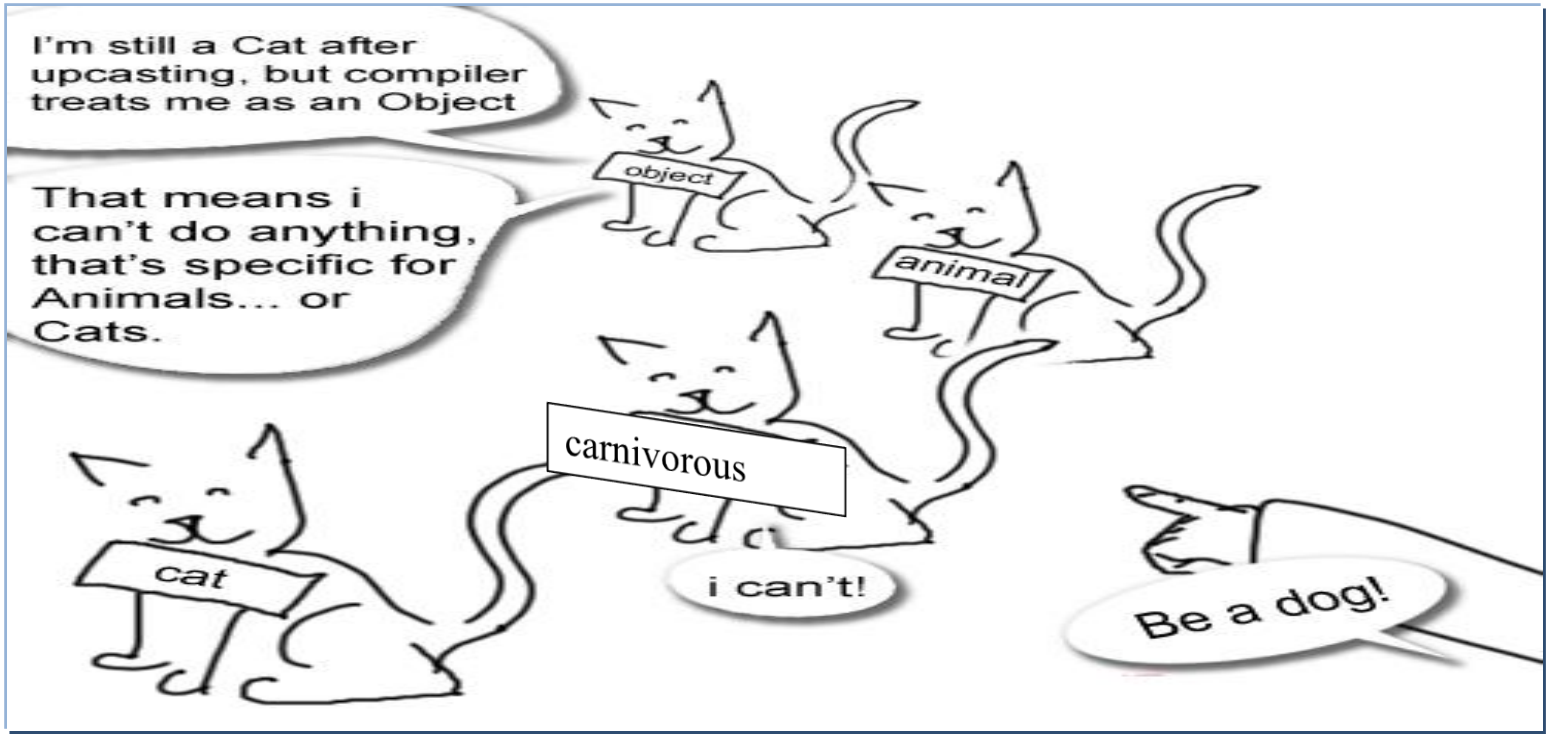
- به این نکته توجه کنید که: با تبدیل اشیا شما هویت واقعی شیئ را تغییر نمی دهید. بلکه به شیئ مربوطه برچسب متفاوتی می زنید.
- برای مثال، اگر شما یک Cat ایجاد کرده و برروی آن تبدیل روبه بالا انجام دهید، این تبدیل سبب نمی شود که شیئی اولیه دیگر گربه (Cat) نباشد!
- این شیئ همچنان گربه است. اما با او مانند هر جانور (Animal) دیگری رفتار خواهد شد و خصوصیات (فیلدهای) ویژه گربه پنهان می شود تا زمانی که دوباره به گربه تبدیل روبه پایین شود.
- در اسلاید بعدی کد مربوط به شیئ را پیش و پس از تبدیل مشاهده کنید.

تبدیل در مثال جانوران

همانطور که مشاهده می کنید، Cat پس از تبدیل همچنان Cat باقیمانده و به جای ان یک شیئ گوشتخوار (Carnivorous) ایجاد نشده است. فقط برچسب گوشتخوار به آن زده شده که کاملاً 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 هر دو شیئ گوشتخوار هستند، گربه نمی تواند به سگ تبدیل شود. شکل زیر این مطلب را نشان می دهد:



خودکار انجام شدن تبدیل روبه بالا

- برای تبدیل روبه بالا نیاز نیست برنامه نویس به طور دستی اینکار را انجام دهد. برای مثال

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

- معادل است با:

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

- اما تبدیل روبه پایین همیشه باید به طور دستی انجام شود:

```
Cat c1 = new Cat();
 Animal a = c1; //automatic upcasting to Animal
 Cat c2 = (Cat) a; //manual downcasting back to a Cat
```

- چرا تبدیل رو به بالا به صورت خودکار انجام می شود اما تبدیل رو به پایین باید به طور دستی انجام شود؟
- چون تبدیل روبه بالا هرگز با خطا روبه رو نخواهد شد.
- اما اگر مجموعه ای 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(مثلا اردک باشند) و در نتیجه فرآیند تبدیل رو به پایین با پیغام ClassCastException با شکست مواجه خواهد شد.

تفاوت تبدیل روبه بالا و تبدیل روبه بالا

- به همین دلیل است که از عملگر "instanceof" استفاده می کنیم تا تست کند آیا شیئ مربوطه نمونه ای از کلاس گربه می باشد 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 شیئ گوشتخوار ایجاد کرده باشید شیئ مورد نظر نمی تواند مستقیما به یک سگ یا یک گربه تبدیل شود، چون در واقع هیچ یک از آنها نمی باشد!

برای مثال:

- Carnivorous m = new Carnivorous();
- Cat c = (Cat)m;

- تبدیلی نادرست است.

تفاوت تبدیل روبه بالا و تبدیل روبه بالا

- کد اسلایده قبل با پیغام زمان اجرای "java.lang.ClassCastException رو به رو می شود چون به زور می خواهیم 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 نمی تواند انجام شود.