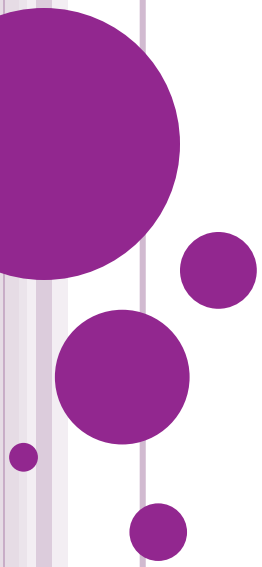
انجمن جاوا کاپ تقدیم می‌کند

دوره برنامه‌نویسی جاوا

چندریختی
Polymorphism



صادق علی‌اکبری

حقوق مؤلف

- کلیه حقوق این اثر متعلق به انجمن جاواکاپ است
- بازنشر یا تدریس آن‌چه توسط جاواکاپ و به صورت عمومی منتشر شده است، با ذکر مرجع (جاواکاپ) بلامانع است
- اگر این اثر توسط جاواکاپ به صورت عمومی منتشر نشده است و به صورت اختصاصی در اختیار شما یا شرکت شما قرار گرفته، بازنشر آن مجاز نیست
- تغییر محتوای این اثر بدون اطلاع و تأیید انجمن جاواکاپ مجاز نیست

سرفصل مطالب

- چندریختی (Polymorphism)
- کاربرد چندریختی
- کلاس‌ها و متدهای انتزاعی (Abstract)
- متدها و کلاس‌های final
- انقیاد پویا (Dynamic Binding)
- اطلاعات نوع داده شیء در زمان اجرا

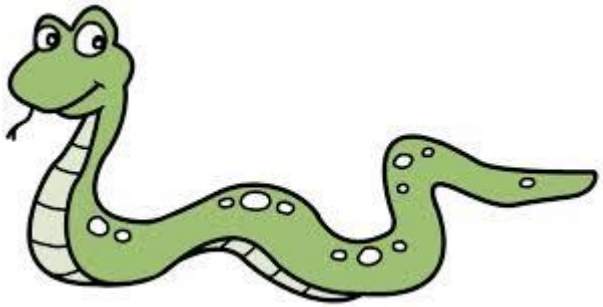
مفهوم چندریختی (Polymorphism)

مثال: برنامه شبیه‌سازی حیوانات

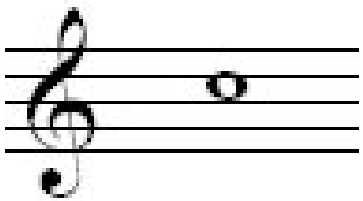
- حیوانات توانایی «حرکت کردن» دارند
- اما هر نوع حیوان، به شکلی حرکت می‌کند

پیغام (درخواست) حرکت

- می‌توانیم از یک نمونه (شیء) حیوان: رفتار «حرکت کردن» را فراخوانی کنیم
- مثلاً: متد «حرکت کن» را اجرا کنیم و «دو متر به راست» را به آن پاس کنیم
- آن حیوان چه می‌کند؟ بستگی به نوع حیوان دارد



مثال دیگر: شبیه‌سازی آلات موسیقی



- یک نت مشابه: صداهای متفاوت روی آلات مختلف

- رنگ صدای تار و کمانچه متفاوت است، حتی اگر یک آهنگ را بنوازند
  - این اشیاء رفتار متفاوتی هنگام فراخوانی دستور یکسان دارند

- رفتار یک ساز: به نوع آن وابسته است

چندریختی (Polymorphism)

- Poly ≈ many (چند) و Morph ≈ form, shape (شکل)
- امکان فراخوانی یک درخواست مشابه (واسط مشترک) در اشیائی از انواع مختلف
- واسط مشترک (درخواست یکسان) **اما رفتار متفاوت**
- رفتار شیء در قبال فراخوانی این درخواست، وابسته به نوع شیء خواهد بود
- واسط یکسان:
  - **animal.*move***(Direction d, double distane)
  - **instrument.*play***(int note)
- اما با پیاده سازی مختلف در زیرکلاسهای متفاوت
  - مثلاً پیاده‌سازی move در زیرکلاس Dog و Fish متفاوت است
  - رفتار dog.move(right, 3) و fish.move(right, 3) متفاوت خواهد بود

چندریختی (ادامه)

- چندریختی، سومین امکان مهم و حیاتی زبان‌های شیءگرا است
  - بعد از محصورسازی (Encapsulation) و وراثت (Inheritance)
- امکان چندریختی در زبان‌های شیءگرا:
  - متدی روی یک شیء فراخوانی می‌شود
  - نوع دقیق شیء در زمان اجرا مشخص می‌شود
  - در زمان اجرا رفتار دقیق این شیء (با توجه به نوع آن) معلوم می‌شود
- مثال:

```
Animal a ;
if(X) a = new Cat();
else  a = new Fish();
a.move("right", 3.0);
```

تغییر نوع به بالا و پایین
Upcasting & Downcasting

```
Child c = new Child();
Parent p = new Parent();
```

نکته

- فرض کنید Child زیرکلاسی از کلاس Parent باشد
- به یاد داشته باشید که همواره شیء کلاس Child شیء Parent نیز هست
  - رابطه **is-a**
- بنابراین این خطوط معتبر هستند:

```
p = c;
Parent p = new Child();
```

- اما این خطوط نامعتبر هستند:

```
c = p;
Child c = new Parent();
```

- مثال: `Animal a=new Dog();` صحیح ولی `Cat c=a;` غلط است
  - هر سگی یک حیوان است (ارجاع a هم قرار است به یک حیوان اشاره کند)
  - هر حیوانی لزوماً یک گربه نیست (ارجاع c قرار است به یک گربه اشاره کند)
- **تأکید**: درباره عملگر = صحبت می‌کنیم که «نوع» سمت چپ و راست آن متفاوت است

تغییر نوع به بالا (UpCasting)

- گاهی از یک شیء، به عنوان شیئی از نوع اَبَرکلاس استفاده می‌کنیم
- مثال:

```
Shape s = new Rectangle();
Circle c = new Circle();
Shape s =  c;
Animal a = new Dog();
Person p = new Student("Ali", 9430623);
```

- به این کار، «تغییر نوع به بالا» یا Upcasting می‌گویند
- تغییر نوع به بالا همواره معتبر است
- کامپایلر جلوی آن را نمی‌گیرد (خطای کامپایل ایجاد نمی‌شود)

تغییر نوع به پایین (DownCasting)

- اگر از یک شیء، به عنوان شیئی از نوع زیرکلاس استفاده می‌کنیم
- به این کار، «تغییر نوع به پایین» یا Downcasting می‌گویند

```
Shape   s = …
Circle   c  = s;
```

- تغییر نوع به پایین همواره معتبر نیست (گاهی معتبر و گاهی نامعتبر است)
- بنابراین کامپایلر جلوی آن را می‌گیرد (خطای کامپایل ایجاد می‌شود)
- مگر این که صراحتاً از عملگر «تغییر نوع» (Cast) استفاده شود

```
Shape s = new Circle();
Circle c = (Circle) s;
```

- در این صورت در زمان کامپایل خطایی گرفته نمی‌شود
- اما ممکن است منجر به خطا در زمان اجرا شود

```
Shape s = new Rectangle();
Circle c = (Circle) s;
```

خطا در زمان اجرا ←

رفتار چندریخت (Polymorphic Behavior)

- دیدیم که ممکن است ارجاعی از نوع اَبَرکلاس، به شیئی از نوع زیرکلاس اشاره کند
- مثل `Animal a=new Dog();` و یا `Person p = new Student();`
- اگر یک متد از چنین ارجاعی فراخوانی شود، متد اَبَرکلاس اجرا می‌شود یا متد زیرکلاس؟
  - مثلاً `a.move` متد `move` از `Animal` را اجرا می‌کند یا همین متد از `Dog` ؟
- **چندریختی** : نوع دقیق شیء تعیین کننده رفتار شیء است، نه نوع ارجاع آن

```
Shape    s = new Rectangle();
s.draw();
double d = s.getArea();

Circle c = new Circle();
s =  c;
s.draw();
d = s.getArea();
```

- **توجه کنید**: بخش‌هایی از برنامه مقابل، ظاهری یکسان ولی رفتاری متفاوت دارند
- واسطی یکسان که شکل‌های مختلف رفتار را ایجاد می‌کند
- به این وضعیت چندریختی می‌گویند

مثال

```
class Parent {
    public void f(){
        System.out.println("f() in Parent");
    }
}

public class Child extends Parent{
    public void f(){
        System.out.println("f() in Child");
    }

    public static void main(String[] args) {
        Parent p = new Parent();
        p.f();
        Child c = new Child();
        c.f();
        p = c;
        p.f();
    }
}
```

نکته

- امکان چندریختی، از عهده کامپایلر بر نمی‌آید
- دقت کنید: کامپایلر نمی‌داند یک ارجاع به شیئی از چه نوعی اشاره خواهد کرد

```
Animal a ;
if(X) a = new Cat();
else  a = new Fish();
a.move("right", 3.0);
```

- در **زمان اجرا** مشخص می‌شود:

  شیئی که یک متغیر به آن ارجاع می‌دهد و نوع (کلاس) این شیء

- بسیار مهم: برخی کارها در زمان اجرا و برخی در زمان کامپایل انجام می‌شوند
- **Compile time & Runtime**

کوییز

فرض کنید:

```
class Animal{}
class Cat extends Animal{}
class Dog extends Animal{}
```

```
Object o = new Object();
Animal a = new Animal();
Animal x = new Cat();
Cat c = new Cat();
Dog d = new Dog();
```

- کدام دستورات خطای کامپایل ایجاد می‌کنند؟
- کدام دستورات خطا در زمان اجرا ایجاد می‌کنند؟ (هر دستور را مستقل فرض کنید)

خطای کامپایل به دلیل Downcasting

```
o = a;
o = c;
```

```
a = o;
a = c;
```

```
c = o;
c = a;
c = d;
```

```
c = (Cat) x;
d = (Dog) x;
```

خطای ClassCastException در زمان اجرا

کاربرد چندریختی

کاربردهای چندریختی

- مثال از رفتار چندریختی در یک برنامه گرافیکی:
- در یک بازی فوتبال (یک برنامه گرافیکی) اشیاء مختلفی وجود دارند
  - اَبَرکلاس: Drawable و زیرکلاس‌ها: Ball ، Player ، Referee و ....
  - اشیاء: توپ، علی دایی، علی کریمی، فرزاد مجیدی، فنایی و ...
- همه این اشیاء، رفتار (عمل) draw (رسم کردن) را دارند
  - این متد در کلاس Drawable تعریف شده
  - وقتی این متد برای یک شیء فراخوانی شود، این شیء نمایش داده می‌شود
- ما به راحتی عمل draw() را روی هر یک از اشیاء صدا می‌زنیم
- و این اشیاء می‌دانند چطور خود را ترسیم کنند

اگر امکان چندریختی نداشتیم:

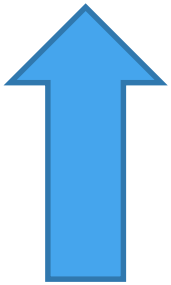
```
Player[] players = ...
Referee[] refs = ...
Ball ball = ...

for (Player player : players) {
    player.draw();
}
for (Referee ref : refs) {
    ref.draw();
}
ball.draw();
```

- اما در یک بازی فوتبال، صدها نوع شیء وجود دارد
- به ازای هر نوع، یک حلقه ایجاد کنیم؟!

با وجود امکان چندریختی

```
Drawable[] drawables = ...
for (Drawable drawable : drawables) {
    drawable.draw();
}
```

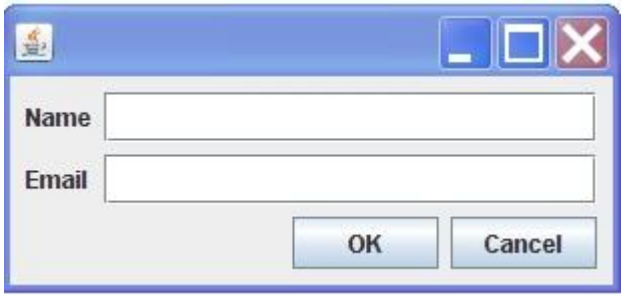


- یک آرایه از جنس ابرکلاس می‌سازیم
- همه اشیاء را در این آرایه قرار می‌دهیم
- متد draw را برای اعضای این آرایه فراخوانی می‌کنیم

```
for (Player player : players) {
    player.draw();
}
for (Referee ref : refs) {
    ref.draw();
}
ball.draw();
```

مثال دیگر

- فرض کنید یک برنامه با واسط کاربری گرافیکی داریم
- همه اشیاء از اَبَرکلاس Component ارث‌بری می‌کنند
  - Button, TextBox, Checkbox, …
- متد select در Component تعریف شده
- و در هر یک از زیرکلاس‌ها پیاده‌سازی خاصی دارد




```
Component selected = findComponent(x, y);
selected.select();
```

کلاس‌ها و متدهای انتزاعی
Abstract Classes & Methods

رفتارهای انتزاعی (Abstract)

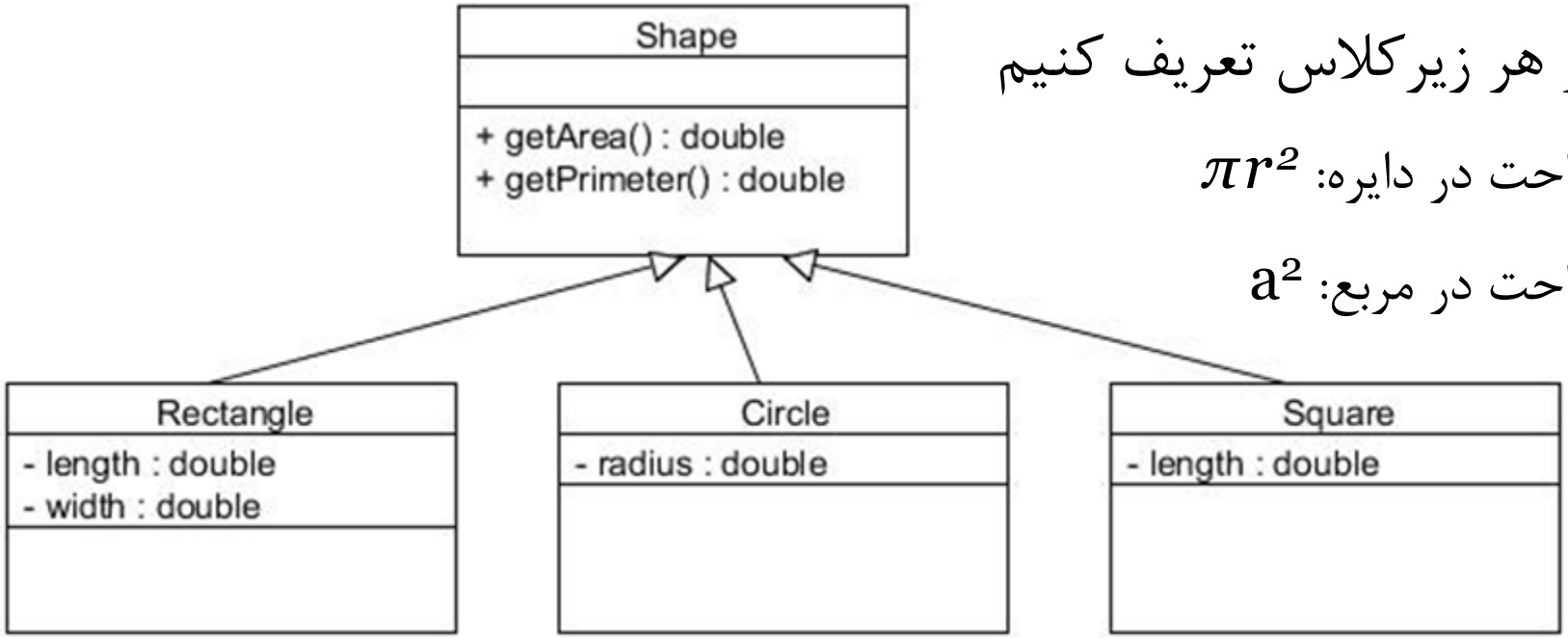
- سؤال: آیا هر حیوانی شنا می‌کند؟
  - خیر. همه نمونه‌های حیوان (اشیاء مختلف) این رفتار را ندارند
  - پس متد «شنا کردن» را در کلاس حیوان قرار نمی‌دهیم
- سؤال: آیا هر حیوانی حرکت می‌کند؟
  - بله. پس متد (رفتار) «حرکت کردن» برای کلاس حیوان وجود دارد
- اما چگونه می‌توانیم این متد را در کلاس حیوان پیاده کنیم؟
  - همه حیوانات حرکت می‌کنند، ولی نحوه انجام این رفتار در هر نوع حیوان متفاوت است
  - مثلاً ماهی‌ها برای جابجایی شنا می‌کنند، پرندگان می‌پرند، سگ‌ها می‌دوند و ...
- متد (رفتار) حرکت کردن برای کلاس «حیوان» **انتزاعی** (**abstract**) است

متد (رفتار) انتزاعی

- Abstract Method یا Abstract Behavior
  - متدی که برای همه اشیاء یک کلاس وجود دارد،
    اما جزئیات دقیق و پیاده‌سازی این متد در آن کلاس غیرممکن است
    و باید در هر زیرکلاس پیاده‌سازی شود
  - چنین متدی در اَبَرکلاس، متد انتزاعی خوانده می‌شود
  - متدی که دقیقاً قابل پیاده‌سازی باشد (انتزاعی نباشد) یک متد واقعی خوانده می‌شود
    - (Concrete method ≠ Abstract Method)
  - رفتار «حرکت کردن» درکلاس حیوان، یک متد **انتزاعی** است
  - رفتار «حرکت کردن» در کلاس سگ **واقعی** است (انتزاعی نیست)
  - رفتار setName در کلاس حیوان **واقعی** است (انتزاعی نیست)

مثال دیگری از متدهای انتزاعی

- متدهای **محاسبه محیط** و **محاسبه مساحت**
- هر شکل (Shape) امکان محاسبه محیط و مساحت دارد
- ولی این متدها را در کلاس Shape نمی‌توانیم پیاده‌سازی کنیم
- باید در هر زیرکلاس تعریف کنیم
  - مساحت در دایره: πr^2
  - مساحت در مربع: a^2



کلاس انتزاعی (Abstract Class)

- **کلاس انتزاعی**: کلاسی که هیچ شیئی مستقیماً از آن ایجاد نمی‌شود
  - اگر شیئی از جنس این کلاس است، باید از یکی از زیرکلاس‌هایش تولید شود
- به‌ویژه کلاس‌هایی که متد انتزاعی دارند، قطعاً کلاس انتزاعی هستند
  - چون کلاسی که متد انتزاعی دارد، تعریف برخی رفتارها را ندارد
  - این رفتارهای انتزاعی در زیرکلاس‌ها تعریف (واقعی) می‌شوند
- مثال: Shape یک کلاس انتزاعی است، زیرا متدهای انتزاعی دارد
  - متدهای محاسبه مساحت و محیط انتزاعی هستند
  - هیچ شیئی مستقیماً از نوع Shape ساخته نمی‌شود.
- مثال: Animal یک کلاس انتزاعی است (متد حرکت کردن انتزاعی است)
  - شیئی نوع Animal ایجاد نمی‌شود، اما از نوع سگ و گربه ایجاد می‌شود

نحوه تعریف کلاس‌ها و متدهای انتزاعی

- انتزاعی بودن یک کلاس یا متد باید توسط برنامه‌نویس تصریح شود
- این کار با کلیدواژه abstract انجام می‌شود
- متد انتزاعی، دارای بدنه نیست

```
abstract class Animal { ... }
public abstract void talk();
```

- اگر در کلاسی یک متد انتزاعی تعریف کنید، باید آن کلاس را هم انتزاعی کنید
  - در تعریف کلاس کلیدواژه abstract را اضافه کنید
- اگر از یک کلاس انتزاعی، کلاسی را ارث‌بری کنیم
  و همه متدهای انتزاعی اَبَرکلاس را در زیرکلاس تعریف (پیاده‌سازی) نکنیم:
  کلاس جدید هم انتزاعی است و باید با پیشوند abstract تعریف شود
- از کلاس انتزاعی نمی‌توانیم نمونه‌ای بسازیم (چرا؟!)
  - استفاده از new برای یک کلاس انتزاعی باعث خطای کامپایل می‌شود

نکته

- کلاسی که متد انتزاعی دارد: قطعاً باید به صورت انتزاعی تعریف شود
- کلاسی که متدهای انتزاعی به ارث برده است:
  - اگر همه متدهای انتزاعی که به ارث برده، پیاده‌سازی کند: واقعی می‌شود
  - اگر همه متدهای انتزاعی که به ارث برده، پیاده‌سازی نکند: انتزاعی می‌شود
- آیا می‌توانیم کلاسی که هیچ متد انتزاعی ندارد را انتزاعی تعریف کنیم؟
  - حتی اگر هیچ متد انتزاعی به ارث هم نبرده باشد؟
  - بله. طراح کلاس می‌تواند آن را انتزاعی تعریف کند
    - مثلاً برای جلوگیری از ایجاد شیء از این کلاس
    - و یا برای اجبار ایجاد زیرکلاس‌هایی از آن

```
abstract class Human{
 private String name;
 public String getName() {
  return name;
 }
}
```

مثال‌هایی از متدهای انتزاعی

مثال: متد انتزاعی

```
abstract class Animal {
 private String name;
 public void setName(String name) {
   this.name = name;
 }
 public String getName() {
   return name;
 }
 public abstract void talk();
}
```

```
class Cat extends Animal{
 public void talk() {
  System.out.println("Meww!!");
 }
}
```

مثال دیگر: سلسله مراتب شکل‌ها

```
public abstract class Shape {
   public abstract double getArea();
   public abstract double getPerimeter();
}
```

کلاس دایره

```
public class Circle extends Shape{
    private double radius;

    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }

    public double getArea() {
        return Math.pow(radius, 2) * Math.PI;
    }
    public double getPerimeter() {
        return 2 * radius * Math.PI;
    }
}
```

کلاس مستطیل

```
public class Rectangle extends Shape{
    private double width, length;

    public double getWidth() {
        return width;
    }
    public void setWidth(double width) {
        this.width = width;
    }
    public double getLength() {
        return length;
    }
    public void setLength(double length) {
        this.length = length;
    }
    public double getArea() {
        return length * width;
    }
    public double getPerimeter() {
        return 2 * (length + width);
    }
}
```

کلاس مربع

```
public class Square  extends Shape{
    private double length;

    public double getLength() {
        return length;
    }
    public void setLength(double length) {
        this.length = length;
    }
    public double getArea() {
        return length * length;
    }
    public double getPerimeter() {
        return 4 * length;
    }
}
```

سؤال

- اصلاً چرا متد انتزاعی را در تعریف کلاس بگنجانیم؟ چه فایده‌ای دارد؟
  - وقتی نمی‌توانیم بدنه آن را تعریف کنیم، خوب اصلاً آن را اعلان نکنیم
- پاسخ: اگر اَبَرکلاس شامل یک متد نباشد،
  نمی‌توانیم این متد را روی ارجاعی از نوع اَبَرکلاس فراخوانی کنیم
  - دچار خطای کامپایل می‌شویم
  - در نتیجه نمی‌توانیم از امکان چند ریختی استفاده کنیم
- مثال:

```
Animal a = new Cat();
a.move();
```

  - اگر کلاس Animal شامل متد move نباشد (انتزاعی یا واقعی):
  - نمی‌توانیم move را روی شیء a فراخوانی کنیم
    - حتی اگر در Cat تعریف شده باشد

مثال: حیوانات

```
abstract class Animal {
 private String name;
 public Animal(String name) {
  this.name = name;
 }
 public String getName() {
  return name;
 }
 public abstract void talk();
}
```

```
Animal[] animals = {
 new Cat("Maloos"),
 new Cat("loos"),
 new Dog("Fido")};
for (Animal a : animals) {
 System.out.println(a.getName());
 a.talk();
}
```

```
class Cat extends Animal{
 public Cat(String name) {
  super(name);
 }
 @Override
 public void talk() {
  System.out.println("Mew!");
 }
}
```

```
class Dog extends Animal{
 public Dog(String name) {
  super(name);
 }
 @Override
 public void talk() {
  System.out.println("Hop!");
 }
}
```

کوییز

کوییز

- به نظر شما در هر موضوع زیر، کدام کلاس انتزاعی تعریف شود:
  - حساب بانکی، حساب سپرده کوتاه‌مدت، حساب سپرده بلندمدت، حساب قرض‌الحسنه
    - فرض: غیر از حساب‌های کوتاه‌مدت، بلندمدت و قرض‌الحسنه هیچ نوع حساب دیگری نداریم
  - کارمند دانشگاه، استاد
    - فرض: غیر از استاد، انواع دیگری از کارمندان نیز در دانشگاه هستند (کارشناس، راننده و ...)

سؤال:

```
abstract class Animal {
 public abstract void move();
 private String name;
 public String getName() {return name;}
 public Animal(String name) {this.name = name;}
}
class Cat extends Animal {
 @Override
 public void move() {System.out.println("Jump");}
 public Cat(String name) { super(name); }
}
class Dog extends Animal {
 @Override
 public void move() {System.out.println("Run");}
 public Dog(String name) {super(name);}
}
Animal a;
a = new Cat("Maloos");
System.out.println(a.getName());
a.move();
a = new Dog("Fido");
System.out.println(a.getName());
a.move();
```

خروجی:

```
Maloos
Jump
Fido
Run
```

تمرین عملی

تمرین عملی

- تمرین کوتاه و ساده:
- کلاس Shape و Rectangle را تعریف کنید
  - متدهای انتزاعی در Shape ایجاد کنید
  - به محض ایجاد متد انتزاعی، کلاس هم باید انتزاعی شود
  - اما کلاسی که متد انتزاعی ندارد هم می‌تواند انتزاعی باشد
  - اگر متدهای انتزاعی که به ارث رسیده‌اند را پیاده نکنیم:
    - باید کلاسمان را انتزاعی کنیم

متدها و کلاس‌های final

کلاس‌ها و متدهای final

- قبلاً متغیرهای ثابت را دیدیم (final variable) :
  - مقدار آن‌ها قابل تغییر نیست
- از کلیدواژه final می‌توانیم برای تعریف یک کلاس یا متد هم استفاده کنیم

```
final class Dog extends Animal{
  ...
}
```

```
class Dog extends Animal{
  @Override
  public void talk() { ... }
  public final void bark() { ... }
}
```

معنی کلاس‌ها و متدهای final

- **کلاس final** : ارث‌بری از کلاس غیرمجاز می‌شود
  - هیچ کلاسی نمی‌تواند یک کلاس final را extend کند
  - معنای کلاس final ، نهایی است و قابل تغییر و گسترش نیست
  - کلاس final هیچ زیرکلاسی نخواهد داشت
  - ایجاد زیرکلاس برای یک کلاس final : ایجاد خطای کامپایل

- **متد final:** چنین متدی را در زیرکلاس نمی‌توانیم override کنیم
  - معنای متد final ، نهایی است: قابل تغییر نیست
  - هیچ زیرکلاسی نمی‌تواند تعریف یک متد final را لغو (override) کند
  - لغو (override) کردن یک متد final : ایجاد خطای کامپایل

چرا final ؟

- تعریف متد یا کلاس به صورت final : یک تصمیم طراحی است
  - با این کار طراح کلاس اجازه نمی‌دهد دیگران معنی کلاس یا متد را تغییر دهند
- اگر طراح چنین هدفی داشته باشد، متد یا کلاس موردنظر را final می‌کند
- مثال: کلاس String ، یک کلاس final است
  - نمی‌توانیم آن را extend کنیم
- در مورد متدها و کلاس‌های final : چندریختی وجود ندارد
  - چون معنی نهایی مشخص است

```
class Dog extends Animal{
 public final void bark() { ... }
}
```

```
Dog d = ...;
d.bark();
```

مروری بر کلیدواژه final

- متد ثابت (final method)
  - override نمی‌شوند
- کلاس ثابت (final class)
  - کلاس زیرمجموعه‌ای ندارند
- متغیر ثابت (final)
  - انواع داده اولیه (prmitive) : مقادیر ثابت
  - اشیاء : ارجاع‌های ثابت
  - تفاوت متغیرهای ثابت (final) و متغیرهای تغییرناپذیر (immutable)

انقیاد پویا و ایستا
Dynamic & Static Binding

انقیاد (Binding)

```
Animal a ;
if(X) a = new Cat();
else  a = new Fish();
a.move("right", 3.0);
```

- مثلاً در کد روبرو باید مشخص شود
  دقیقاً کدام متد move فراخوانی می‌شود
  - متدی که در Animal تعریف شده؟ یا متدی که در Cat یا در Dog تعریف شده؟
- **انقیاد متد** (**method binding**): تعیین متدی که فراخوانی شده است
- بدیهی است که برای اجرای یک متد باید این کار (binding) انجام شود
- گاهی این کار به سادگی در زمان کامپایل ممکن است
  - مثلاً وقتی یک متد خصوصی (private) را فراخوانی می‌کنیم
- گاهی این کار در زمان اجرا قابل انجام است
  - مثلاً وقتی از چندریختی استفاده می‌کنیم (مثل کد فوق)

انقیاد در زمان کامپایل

- Compile-time binding یا Static binding یا Early binding
  - زمانی که ابهامی در تشخیص متدی که فراخوانی شده، وجود ندارد
  - کامپایلر به سادگی می‌فهمد دقیقاً چه متدی فراخوانی شده
  - مثال: فراخوانی f() در زمان کامپایل مقید می‌شود

```
class StaticBinding{
 private void f(){...}
 public void g(){
  f();
  ...
 }
}
```

  - متدهای static یا private یا final در زمان کامپایل bind می‌شوند
  - چرا؟

انقیاد در زمان اجرا

- Late binding یا Dynamic binding یا Runtime binding
  - گاهی کامپایلر نمی‌تواند متد فراخوانی شده را تشخیص دهد
  - مثلاً وقتی متدی را روی ارجاعی از نوع اَبَرکلاس فراخوانی می‌کنیم
  - مثال:

```
Animal a ;
if(X) a = new Cat();
else  a = new Fish();
a.move("right", 3.0);
```

  - متد واقعی در زمان اجرا bind می‌شود
    - زیرا ممکن است override شده باشد

کوییز

فرض کنید:

```
class Parent{
	public void f(){
		System.out.println("f() in Parent");
	}
}
class Child extends Parent{
	public void f(){
		System.out.println("f() in Child");
	}
}
public class SomeClass {
	public void method(Parent p){
		System.out.println("method(Parent)");
	}
	public void method(Child p){
		System.out.println("method(Child)");
	}
}
```

خروجی این قطعه کد چیست؟

```
Child child = new Child();

Parent parent = new Parent();

Parent parentRefToChild = new Child();


parent.f();

child.f();

parentRefToChild.f();
```

Output:

```
f() in Parent
f() in Child
f() in Child
```

خروجی این قطعه کد چیست؟

```
SomeClass square = new SomeClass();
square.method(parent);
square.method(child);
square.method(parentRefToChild);
```

Output:

```
method(Parent)
method(Child)
method(Parent)
```

- نکته مهم:
  - رفتار چندریختی برای ارجاع
    - ارجاع قبل از نقطه
  - نه برای پارامترها

چند نکته مهم درباره چندریختی

نکته : Overloading

```
public class SomeClass {
      public void method(Parent p){
            System.out.println("method(Parent)");
      }
      public void method(Child p){
            System.out.println("method(Child)");
      }
}
```

- ***method()*** در SomeClass ، overload شده است
  - دو متد مستقل که هر دو حاضر و قابل استفاده هستند

Override یا Overload؟

```
class SomeClass {
      public void method(Parent p){
            System.out.println("method(Parent)");
      }
}
class SomeSubClass extends SomeClass{
      public void method(Child p){
            System.out.println("method(Child)");
      }
}
```

- ***method()*** در SomeSubClass ، Overload شده است
  - Override نشده است
  - دو متد مستقل: متد دوم معنی اولی را لغو نکرده است

خروجی این برنامه چیست؟

```
SomeSubClass ref = new SomeSubClass();
ref.method(parent);
ref.method(child);
ref.method(parentRefToChild);
```

Output:

method(Parent)

method(Child)

method(Parent)

سؤال

- وقتی متد equals() را برای یک کلاس تعریف می‌کنیم
  - در واقع در حال override کردن این متد هستیم
  - زیرا این متد در کلاس Object تعریف شده است:

```
public boolean equals(Object obj) { return (this == obj); }
```

- چرا در هنگام تعریف متد equal ، object را به عنوان پارامتر پاس می‌کنیم؟
- مثلاً کلاس Person متد equals ای به این شکل دارد:

```
public boolean equals(Object obj) {…}
```

- اما نه به این شکل: `public boolean equals(Person obj) {…}`
- **چرا؟!**
  - زیرا در شکل دوم، این متد overload می‌شود. در حالی که باید override شود

سؤال

- چرا متدهای انتزاعی (abstract) را تعریف می‌کردیم؟
- پاسخ:
- تا بتوانیم در رفتارهای چندریختی از آن‌ها استفاده کنیم
- وگرنه خطای کامپایل می‌گرفتیم

- مثلاً در برنامه روبرو:

```
Animal a ;
if(X) a = new Cat();
else  a = new Fish();
a.move("right", 3.0);
```

- اگر متد move در کلاس Animal تعریف نشده باشد: خطای کامپایل ایجاد می‌شود
- اگر بدنه این متد در Animal قابل تعریف نیست: باید به صورت انتزاعی تعریف شود

اطلاعات نوع داده در زمان اجرا
Runtime Type Identification

عملگر instanceof

`a instanceof Type`

```
Parent a = ...;
if(a instanceof Child){...}
```

- یک شیء (a) و یک کلاس (Type) می‌گیرد
  - اگر a نمونه‌ای از کلاس Type باشد، true برمی‌گرداند

```
Animal x = ...
if(x instanceof Cat){...}
else if(x instanceof Dog){...}
```

- کلاس Child زیرکلاس Parent باشد
  - اگر Child همان کلاس یا اَبَرکلاس Parent باشد: همیشه true، مگر...
  - اگر Child اَبَرکلاس، زیرکلاس یا خود Parent نباشد: خطای کامپایل (همیشه غلط)
- این بررسی در زمان اجرا انجام می‌شود
- قبل از هر تغییر نوع به پایین (Downcast)، بررسی نوع انجام دهید

```
Animal x = ...
if(x instanceof Cat){Cat c = (Cat)x; c.mew();}
```

- نکته: اگر ارجاع موردنظر null باشد، این عملگر false برمی‌گرداند

توضیح بیشتر:
در مبحث Reflection

شیء کلاس و متد getClass

- اولین بار که از یک کلاس استفاده می‌کنیم، این کلاس در حافظه بارگذاری می‌شود
- Dynamic Loading
- اطلاعات مربوط به این کلاس، در شیئی با نام «**شیء کلاس**» در حافظه جای می‌گیرد
- Class Object
- مثلاً یک شیء در حافظه اطلاعات کلاس String و شیء دیگری، اطلاعاتی درباره کلاس Person را نگهداری می‌کند
- هر شیئی، یک ارجاع به «**شیء کلاس**» خودش دارد
- این ارجاع با کمک متد **getClass()** برمی‌گردد
- متد getClass در Object پیاده‌سازی شده و final است

```
Animal a = new Dog("Fido");
String s = a.getClass().getSimpleName();
s = a.getClass().getName();
```

Metaspace و Permanent Generation

هر «شیء کلاس» در حافظه جای می‌گیرد. بخشی از حافظه مسؤول نگهداری این اشیاء است

قبل از نسخه ۸ جاوا

- اطلاعات کلاس‌ها (شیء کلاس‌ها) در بخشی به نام **PermGen** ذخیره می‌شود
- اگر پروژه بسیار بزرگی داشته باشیم، ممکن است این فضا پر شود و خطا ایجاد شود

OutOfMemoryError

- برنامه‌ای که کلاس‌های زیادی (کتابخانه‌ها و JAR های متنوع) را استفاده و بارگذاری کند

- حجم حافظه PermGen قابل تنظیم است:

java **-XX:MaxPermSize**=512m MyClass

بعد از جاوا ۸

اطلاعات مربوط به کلاس‌ها در **Metaspace** نگهداری می‌شود

- **PermGen حذف شده است**
- برخی از مشکلات و دردسرها هم از بین رفته: دیگر **نیازی به تنظیم PermSize نیست**

یادآوری: تنظیم اندازه حافظه Heap با کمک Xms– و Xmx–

کوییز

خروجی این برنامه چیست؟

پاسخ صحیح:

```
Animal a = new Cat("Maloos");
System.out.println(a instanceof Object);
System.out.println(a instanceof Animal);
System.out.println(a instanceof Cat);
System.out.println(a instanceof Dog);
System.out.println(a.getClass().getName());
```

true

true

true

false

```
ir.javacup.polymorphism.Cat
```

تمرین عملی

تمرین عملی

- کلاس Component
  - با ویژگی‌های طول و عرض
  - متد انتزاعی show
  - زیرکلاس‌های Button و TextBox
  - ایجاد و استفاده از آرایه Componentp[]
- تمرین در زمینه کلیدواژه final
  - متد، کلاس یا متغیر ثابت
- تولید و مشاهده متن متد equals که توسط eclipse ایجاد می‌شود

جمع‌بندی

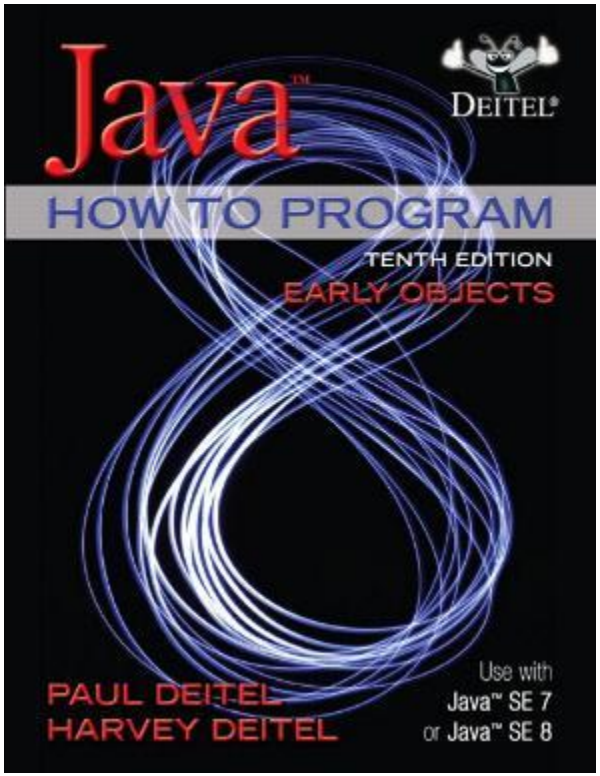
جمع‌بندی

- مفهوم چندریختی (Polymorphism)
- کاربرد چندریختی
- کلاس‌ها و متدهای انتزاعی (Abstract)
- اعضای final (کلاس‌ها، متدها و متغیرهای ثابت)
- انقیاد پویا (Dynamic Binding)
- اطلاعات نوع داده شیء در زمان اجرا
  - عملگر instanceof و متد getClass()

مطالعه کنید

- فصل ۱۰ کتاب دایتل

Java How to Program (Deitel & Deitel)



10- Object-Oriented Programming: **Polymorphism** and Interfaces

- تمرین‌های همین فصل از کتاب دایتل

تمرین

- کلاس Person و زیرکلاس‌های Student و Teacher را پیاده‌سازی کنید
- برای هر یک، متد toString مناسب و متفاوتی ایجاد کنید
- یک آرایه شامل تعدادی شیء از همه این انواع بسازید
- Person[] people = ...
- در یک حلقه، متد toString را برای همه فراخوانی و چاپ کنید
- برای هر کلاس متد equals مناسب پیاده‌سازی کنید
  - از این متد در حالت‌های مختلف استفاده کنید و آن را آزمایش کنید
- مشخصات هر کلاس، متد و متغیر را با دقت انتخاب کنید
  - سطح دسترسی؟ final ؟ abstract ؟ استاتیک؟ ...

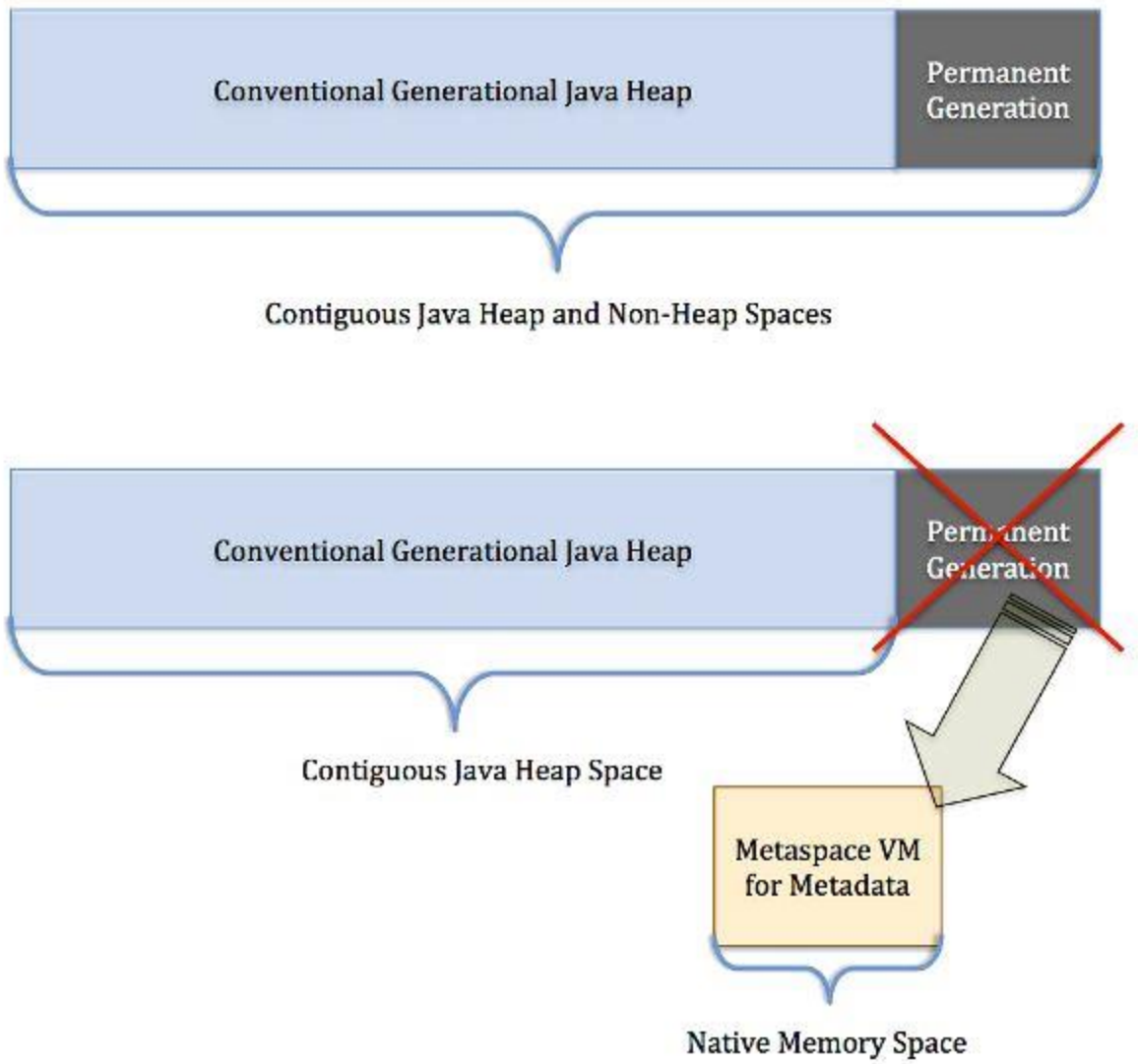
جستجو کنید و بخوانید

- متدهای مجازی (virtual)
  - در برخی زبانها (مانند C++) می‌توانیم مکانیزم انقیاد متد را مشخص کنیم
  - در صورتی از انقیاد پویا برای یک متد استفاده می‌شود که با کلیدواژه *virtual* تعریف شود
- گاهی برنامه‌نویسان برای افزایش کارایی، برخی متدها را final می‌کنند
  - اما این کار توصیه نمی‌شود
- به تفاوت Dynamic Loading و Dynamic Binding دقت کنید
- درباره کلاس شیء (Class Object)
  - نوع شیئی که getClass() برمی‌گرداند چیست؟ این نوع چه متدهایی دارد؟
  - بخش‌های مختلف حافظه در یک برنامه جاوا چه هستند؟
    - Young, Tenured, and Permanent Generation

پایان

سایر مطالب

از PermGen تا Metaspace



مثال جالب

- انتزاعی کردن متدی که به ارث رسیده:

```
abstract class Animal {
   public abstract void move();
   public abstract String toString();
}
```

تاریخچه تغییرات

نسخه	تاریخ	توضیح
۱.۰.۰	۱۳۹۴/۳/۲۶	نسخه اولیه ارائه آماده شد
۱.۱.۰	۱۳۹۴/۳/۲۶	بخش متدها و کلاس‌های انتزاعی (از موضوع وراثت) به این فایل منتقل شد