## مقدمه

### تعریف شی گرا

شی گرا به مفهومی در برنامه نویسی اطلاق می شود که پیاده سازی آن مبتنی بر شی است. هر شی می تواند دارای یک سری خصوصیت و رفتار باشد که نیازهای کار با آن شی را مرتفع می سازد.

## مفاهیم شی گرا

شی گرایی (OO[1]) یک شیوه تفکر، یک شروش نگریستن به دنیا و دیدن همه انواع شی است.

پیش از شی گرایی شیوه های مختلفی برای برنامه نویسی وجود داشت و به مرور توسعه یافتند تا به شی گرایی رسیدیم.

زبان های رویه ای به برنامه نویس اجازه می دادند که برنامه را به تعدادی از اجزای مناسب جهت پردازش روی داده ها تقسیم کند. این رویه ها در واقع ساختار کل برنامه را تعیین می کردند. صدا کردن و فراخوانی این رویه ها اجرای برنامه ای ماژولار را به دنبال داشت. برنامه در انتها پس از اتمام فراخوانی فهرستی از رویه ها به پایان می رسید. طبیعت ذاتی برنامه نویسی رویه ای مشکلاتی را نیز به دنبال داشت. در برنامه نویسی رویه ای داده ها و رویه ها از یکدیگر جدا بودند و هیچگونه کپسوله سازی از داده ها وجود نداشت. این موضوع باعث می شود رویه ها ندانند چگونه به طور صحیح با داده ها برخورد کنند. از آنجا که هر رویه باید نحوه برخورد با داده را در خود نگاه دارد هر گونه تغییر در ساختار داده نیازمند تغییر در همه رویه هایی که به آن داده دسترسی داشتند می شد. به همین دلیل یک تغییر کوچک، موجب تغیرات سلسله مراتبی در دیگر رویه ها می شود.

برنامه نویسی ماژولار سعی در رفع مشکلات برنامه نویسی رویه ای داشت. در این شیوه داده ها و رویه ها با هم ترکیب می شوند. یک ماژول شامل داده و رویه هایی است که روی داده عمل می کنند. برای استفاده از یک ماژول به سادگی رابط آن ماژول فراخوانی می شود. ماژول تمام داده های درونی را از مابقی برنامه مخفی می کند. اما ماژول ها نیز مشکلات زیادی دارند. ماژول ها به راحتی قابل توسعه نیستند. همچنین نمی توان مبنای یک ماژول را روی ماژول دیگر قرار داد. یک ماژول نمی تواند نوع یک ماژول دیگر را به اشتراک بگذارد.

برنامه نویسی شی گرا با افزودن مفاهیمی چون وراثت و پلیمورفیسم به ماژول ها به وجود آمد. OOP برنامه را به تعدادی اشیاء سطح بالا تقسیم می کند. هر شی قسمتی از برنامه ای که باید حل شود را مدل می کند. در برنامه نویسی شی گرایی اشیاء با هم در تعامل هستند به نحوی که اجزای برنامه با یکدیگر برنامه را پیش می برند.

استفاده از اشیاء باعث می شود بتوان به مساله به صورت مفهومی نگاه کرد. بدین ترتیب می توان برنامه را به صورت طبیعی و با عبارات دنیای واقعی پیاده سازی نمود.

- ✓ پیاده سازی بیان چگونگی قسمتی از مساله است. در برنامه نویسی پیاده سازی به معنای نوشتن کد می باشد.

در شی گرایی برای نوشتن برنامه، دیگر غرق در جزییات پیاده سازی نمی شویم و تنها با تعدادی اشیاء سطح بالا کار خواهیم داشت که در سطوح پایین تر با یکدیگر در تعامل می باشند. به گونه ای که هر شی از بقیه سیستم ایزوله خواهد بود.

[1] Object Oriented

## مزایا و اهداف برنامه نویسی شی گرا

در برنامه نویسی شی گرا تلاش می شود تا نرم افزاری تولید شود که در برگیرنده مشخصه های زیر باشد:

۱- سادگی: می توان برنامه را به صورت مجموعه عبارات خود مساله توصیف نمود. بنابراین نیازی نیست که از نحوه کار قسمت های مختلف برنامه چیزی دانست و می توان تنها بر روی آنچه انجام می شود تمرکز نمود.

۲- پایداری: می توان قسمت هایی از برنامه را بدون آن که دیگر قسمت ها تحت تاثیر قرار بگیرند، تغییر داد. در واقع می توان اجزاء را ایزوله کرد و هر جزء را به طور مستقل آزمایش کرد و پس از آن با خیال راحت از آن جزء استفاده نمود.

۳- قابلیت استفاده مجدد: به راحتی می توان از کلاس های شی گرا مجددا استفاده نمود. از طریق وراثت می توان اشیاء موجود را توسعه داد. همچنین از طریق چندشکلی می توان کدهای عمومی و جامعی نوشت.

۴- قابلیت نگهداری: چرخه زندگی (life cycle) یک برنامه پس از ارائه آن به بازار به اتمام نمی رسد. معمولا بیشترین درصد زمان صرف نگهداری یک برنامه می شود. کدهای شی گرا با طراحی خوب قابلیت نگهداری خوبی دارند و درواقع باید به گونه ای نوشته شوند که دیگر برنامه نویسان به راحتی آن را بفهمند.

۵- قابلیت توسعه: شی گرا خواصی همچون وراثت، چندشکلی، جایگزینی و غیره را ارائه می کند که توسط آن ها به راحتی می توان کدها را توسعه داد.

۶- کاهش زمان پیاده سازی: خواصی چون پایداری، قابلیت استفاده مجدد و قابلیت توسعه چرخه زمانی توسعه نرم افزار را کوتاه می کنند. اجزای کوچک شده برنامه را می توان به صورت موازی توسط چند برنامه نویس پیش برد و زمان اتمام نرم افزار را کاهش داد.

## نکاتی درخصوص شی گرایی

۱- OOP زبانی ساده نیست: OOP فراتر از یک زبان و مجموعه ای از تعریف های مشخص است. OOP یک تفکر است که بتوان مساله را به شکل مجموعه ای از اشیا دید و از امکانات آن به درستی استفاده نمود. می توان با استفاده از زبان شی گرا، کدهای غیرشی گرا نوشت. استفاده درست از شی گرا نیاز به تمرین زیادی دارد.

۲- کدها باید به گونه ای نوشته شوند که قابلیت استفاده مجدد را داشته باشند. همچنین برنامه نویسی شی گرا در برنامه های پیچیده کاری گروهی است و استفاده از کدهای اصولی نوشته شده توسط سایر برنامه نویسان ضروری خواهد بود.

۳- OOP راهکار مناسب برای تمام مسائل برنامه نویسی نیست. باید همیشه از ابزار مناسب استفاده نمود.

۴- مخفی کاری در شی گرایی مناسب نیست. برنامه نویس باید کدهای خود را به اشتراک بگذارد و مستندات فنی مناسب تهیه کند.

## شروع برنامه نویسی شی گرا

| **تعریف شی** |
| --- |
| یک شی جزئی از برنامه است که رفتار و حالت را در خود کپسوله و نگهداری می کند. اشیاء به برنامه نویس کمک می کنند تا نرم افزار را با عبارات دنیای واقعی مدل سازی کنند. |

همانطور که دنیای واقعی از اشیاء ساخته شده است می توان گفت نرم افزارهای شی گرا نیز مبتنی بر اشیاء هستند. در زبان های برنامه نویسی شی گرای خالص همه چیز توسط اشیاء صورت می گیرد.

همانطور که در دنیای واقعی اشیا را دسته بندی می کنیم، در برنامه نویسی شی گرا نیز اشیا دسته بندی می شوند و این دسته بندی توسط کلاس انجام می شود. به طور مثال حیوانات به دسته های پستانداران، خزندگان و غیره تقسیم می شوند. این دسته بندی ها در شی گرایی توسط کلاس انجام می شود و هر حیوان (شی) از این دسته (کلاس ها) انتخاب می شود. مثلا حیوان (شی) سگ از دسته (کلاس) پستانداران می باشد.

| **تعریف کلاس** |
| --- |
| هر کلاس تمام مشخصه های مشترک از یک نوع شی را تعریف می کند. به عبارت دیگر کلاس شامل رفتارها (متد) و خواص یا صفات (attribute) یک شی می باشد. |

| **تعریف صفت (خصوصیت) (مشخصات) (attribute)** |
| --- |
| صفات، مشخصه های قابل مشاهده یک کلاس می باشند. رنگ چشم یا وزن نمونه هایی از صفات هستند. هر صفت می تواند دارای یک مقدار مشخص باشد که برای اشیای مختلف متفاوت است.<br>در برنامه نویسی های متداول از صفات با عنوان متغیر یاد می کردیم. |

| **تعریف رفتار** |
| --- |
| رفتار عملی است که توسط یک شی هنگام ارسال پیام یا درخواست به آن انجام می شود. رفتار می تواند توسط یک یا چند متد تعریف شده در کلاس صورت گیرد. |

| **ارتباط کلاس و شی** |
| --- |
| اشیاء نمونه هایی هستند که از کلاس نمونه برداری می شوند.<br>پس ابتدا یک کلاس با مجموعه ای خصوصیت و رفتار تعریف می شود (مانند کلاس پستانداران) سپس یک شی را به عنوان نمونه ای از آن کلاس تعریف می کنیم (مانند شی سگ به عنوان نمونه ای از کلاس پستانداران) |

## ساختار کلی تعریف کلاس

```
public class [نام کلاس] {
	// خطوط توضیحات
	// تعریف صفات
	[نوع دسترسی به صفت] [نوع داده ای صفت] [نام صفت]
	private int qty; // تعریف صفت از نوع داده صحیح با دسترسی خصوصی

	// تعریف متدها
	[نوع دسترسی به متد] [نوع داده خروجی متد] [نام متد](پارامترهای ورودی متد)  { ...........کد............ }
	public int method1(int a) {
		a = a*2;
		return a;
	}
}
```

مثال:

```
public class Item {
	private double unit_price;
	private doube discount;
	private int quantity;
	private String description;
	private String id;

	public Item (String id, String description, int quantity, double price) {
		this.id = id; //to access to the attribute of this class we use this.
		this.description = description;
		this.unit_price = price;
		if (quantity >=0) {
			this.quantity = quantity;
		} else {
			This.quantity = 0;
		}
	}

	public int getQuantity() {
		return this.quantity;
	}
```

```
	public int setQuantity(int quantity) {
		if (quantity >=0) {
			this.quantity = quantity;
		}
	}

	public String getProductID() {
		return this.id;
	}

	public String getDescription () {
		return this.description;
	}

	public String getDiscount() {
		return this.discount;
	}

	public void setDiscount(double Discount) {
		if(discount<=0) {
			this.discount = discount;
		} else {
			This.discount = 0.0;
		}
	}

}
```

در مثال بالا به متد Item دقت کنید

public Item (String id, String description, int quantity, double price

این متد دارای نام یکسان با نام کلاس می باشد به همین خاطر به آن سازنده گفته می شود. تفاوت سازنده با سایر متدها در این است که به محض اینکه یک شی ساخته می شود سازنده به طور خودکار اجرا می شود و نیازی به فراخوانی آن نمی باشد. معمولا مقداردهی های اولیه مورد نیاز هر شی را در تابع سازنده قرار می دهند.

| **تعریف سازنده** |
|---|
| سازنده به متدی گفته می شود که نام آن با نام کلاس یکی می باشد و به محض ساخته شدن شی به طور خودکار اجرا می شود. |

✓ از عبارت this برای دسترسی به صفات و متدهای شی جاری استفاده می شود.

متدهایی همچون getDiscount و getProductID و getQuantity که دسترسی به صفات شی را فراهم می کنند دست یابنده یا accessor گفته می شود. همچنین به متدهایی همچون setDiscount(double Discount) و setQuantity(int quantity) که صفات شی را مقدار دهی می کنند نیز تغییردهنده یا mutator می گویند. اصولا بهتر است دسترسی به صفات توسط accessor ها و mutator ها تامین شوند بدین ترتیب توسعه کلاس ها ساده تر خواهد بود.

| **دست یابنده (accessor)** |
| --- |
| به متدهایی گفته می شود که دسترسی به صفت خاصی از شی را فراهم می کنند. |

| **تغییر دهنده یا دگرگون کننده (mutator)** |
| --- |
| به متدهایی گفته می شود که امکان تغییر صفت خاصی از شی را فراهم می کنند و توسط آن ها می توان صفات را مقداردهی کرد. |

## ایجاد اشیاء

اصولا کلاس ها به عنوان یک دسته می باشند و نمی توان از آن ها به طور مستقیم استفاده نمود، برای این کار باید یک نمونه از آن کلاس ایجاد کرد به این نمونه ها شی گفته می شود.

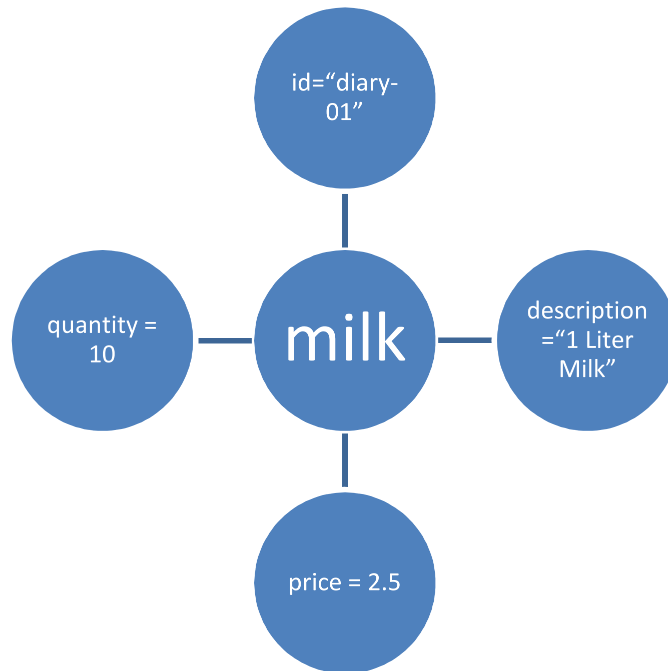
ساختار کلی ایجاد شی به شکل زیر می باشد:

| [نام کلاس] [نام شی] = new [نام کلاس](پارامترهای متد سازنده) |
| --- |

به طور مثال در زیر شی milk از کلاس Item که در بالا تعریف شده ساخته شده است:

```
Item milk = new Item(“diary-01”, “1 Liter Milk”, 10, 2.5 );
```

در مثال بالا شی با نام milk از کلاس Item ساخته شده است که توسط سازنده کلاس id برابر با diary-01 و description یا توضیحات برابر ”1 Liter Milk“ و quantity یا تعداد برابر ۱۰ و price یا قیمت برابر 2.5 قرار می گیرند.




## ارکان برنامه نویسی شی گرا

۱- کپسوله سازی (Encapsulation)

۲- وراثت (Inheritance)

۳- چندشکلی بودن (Polymorphism)

# کپسوله سازی

برای کپسوله سازی ابتدا برنامه به اجزای کوچک تر تقسیم می شوند و هر جزء کار خود را مستقل از سایر اجزاء انجام می دهد. کپسوله سازی با مخفی کردن جزییات پیاده سازی هر جزء این استقلال را ایجاد می کند.

وقتی کپسوله سازی اعمال شده باشد می توان هر موجودیت نرم افزاری را به عنوان یک جعبه سیاه درنظر گرفت. با استفاده از رابط خارجی جعبه سیاه می توان فهمید که چه کاری انجام می دهد بدون نیاز به دانستن این موضوع که جعبه سیاه چگونه کار خود را انجام میدهد.

## رابط

رابط تمام سرویس هایی که توسط قطعه کپسوله شده ارائه می شوند را لیست می کند. به عبارت دیگر رابط مشخص می کند که قطعه فوق چه کاری می تواند با شی انجام دهد و مانند کنترل پنل شی است.

نکته جالب در این است که رابط نمی گوید جعبه سیاه فوق چگونه کار خود را انجام می دهد و پیاده سازی کد را از دید همه مخفی می کند. بدین ترتیب همیشه می توان پیاده سازی کد را تغییر داد و اصلاح نمود و مادامی که رابط بدون تغییر بماند هیچ مشکلی بوجود نخواهد آمد.

❄ به عنوان نمونه ای از یک رابط می توان API را مثال زد.

مثال:

```
public class log {
	public void debug (String message) {
		print ( “DEBUG” , message );
	}
	public void info (String message) {
		print ( “INFO” , message );
	}
	public void warning (String message) {
		print ( “WARNING” , message );
	}
	public void error (String message) {
		print ( “ERROR” , message );
	}
	public void fatal (String message) {
		print ( “FATAL” , message );
		Sysem.exit(0);
	}
	private void print (String message , String type ) {
		System.out.println ( type + “: “ + message );
	}
}
```

در مثال فوق رابط عمومی کلاس log شامل متدهای زیر است:

```
public void debug (String message)
public void info (String message)
public void warning (String message)
public void error (String message)
public void fatal (String message)
```

به جز این متدها بقیه موارد پیاده سازی محسوب می شوند که از دنیای خارج پنهان شده اند. در واقع رابط با بیرون ارتباط دارد اما چگونگی عملیات را کاملا پنهان کرده است.

نکته جالب توجه دیگر آن است که به طور مثال رابط debug نمی گوید که پیغام را لزوما روی صفحه نمایش چاپ می کند و عملکرد آن به پیاده سازی واگذار شده است و می تواند پیام را در صفحه نمایش یا یک فایل و یا یک پایگاه داده ذخیره نمود.

## سطوح دسترسی به متدها و خصوصیت ها

- public : اجازه دسترسی توسط تمام اشیاء
- private : اجازه دسترسی فقط داخلی (خود موجودیت)
- protected : اجازه دسترسی داخلی و کلاس های مشتق شده از آن

## مزایای کپسوله سازی

۱- با کپسوله سازی صحیح، اشیاء به هیچ برنامه خاصی وابسته نخواهند بود و می توان از آن ها در هر کجا بدون نیاز به بررسی یا تغییر استفاده نمود.

۲- امکان بهبود پیاده سازی شی به راحتی امکان پذیر خواهد بود و تا زمانی که رابط تغییر نکند، تمام تغییرات مخفی خواهند بود و تاثیری در مابقی برنامه نخواهد داشت و استفاده کنندگان شی به طور خودکار از تغییرات اعمال شده سود خواهند برد.

۳- استفاده از اشیاء کپسوله شده مانع تاثیرات ناخواسته روی مابقی برنامه خواهد شد.

## خصوصیات کپسوله سازی

۱- abstraction : (تجرید) پایه قابلیت استفاده مجدد است.

۲- مخفی کردن پیاده سازی کد

۳- تقسیم مسوولیت

### abstraction

abstraction یا تجرید فرآیندی است جهت ساده کردن یک مساله پیچیده. برای حل یک مساله لزومی ندارد که به همه جزییات توجه شود و می توان مساله را تنها با در نظر گرفتن جزییات مورد نیاز برای رسیدن به راه حل به مساله ای ساده تر تبدیل کرد.

❋ برای ساخت اشیاء و کلاس ها باید سعی شود به گونه ای طراحی شوند که مجموعه ای از مسائل را حل کنند نه یک مساله خاص را. مدل عام می تواند قابلیت استفاده مجدد از شی را گسترش دهد.

مثال: کلاس صف FIFO برای صف بانک، خط تولید محصول، غیره

نکات ایجاد تجرید موثر:

- تلاش برای دستیابی به یک روال کلی
- تمرکز برای رسیدن به مدل کلی باعث دور شدن از هدف اصلی (حل مساله) نشود.
- تعمیم دادن یک مساله و یافتن مدل کلی معمولا دشوار است و چندین بار تلاش را طلب می کند.
- باید همواره آماده رفع اشکالات روال کلی فوق در آینده را داشت.
- تکمیل یک روال کلی به زمان نیاز دارد.

## مخفی کردن پیاده سازی کد

مخفی کردن پیاده سازی دو مزیت مهم دارد:

۱- شی را در مقابل سایر کاربران حفاظت می کند.
۲- کاربران را از درگیری زائد با شی در امان می دارد.

### محافظت از اشیاء توسط ADT[۲] ها

ADT ها امکان تعرف انواع جدیدی از داده ها و و عملیات را از طریق مخفی کردن داده های درونی و ساختار داده فراهم می آورند.

ADT ها برنامه نویس را قادر می سازند انواع داده های جدیدی برای برنامه تعریف کند که استفاده از آن ها ایمن و بی خطر باشد. محدودیت ها و قید بندهای تعریف شده توسط نوع، شی را از ارتباطات غیرمنطقی و احتمالا مخرب باز می دارد. تعریف درست نوع مانع استفاده نابجا و ناصحیح از شی می شود.

ADT به برنامه نویس این امکان را می دهد که هر گاه نیاز به بیان ایده ای تازه باشد، واژه های برنامه نویسی جدیدی ایجاد کند.

[۲] Abstract Data Type

مثال: کلاس صف FIFO برای صف بانک، خط تولید محصول، غیره

برای این مثال یک ADT با نام Queue ایجاد می کنیم:

```
public interface Queue {
        public void insqueue ( Object obj );        // Insert element Into queue
        public Object rmqueue ();                    // Remove Element from queue
        public boolean isEmpty();                    // Check queue
        public Object checkfirst();                  // Check first Element
}
```

تمام جزییاتی همچون نحوه نگهداری داده ها در صف مخفی شده اند. همچنین رابط امکان هیچگونه دسترسی غیرمجاز به هیچ یک از داده های درونی اش را نمی دهد. تمام این جزییات مخفی شده اند.

در مثال بالا یک نوع داده جدید تعریف شده است. یک صف که می توان از آن در هر برنامه ای استفاده کرد. در اینجا برنامه نویس در سطح لیست ها و اشاره گرها کار نکرده و به حل مساله می اندیشد.

در مثال بالا نحوه پیاده سازی صف مخفی شده است. صف فوق می تواند با استفاده از لیست های پیوندی یا آرایه پیاده سازی شده باشد. اما ایجاد یک رابط کلی دردسرهای زیادی دارد. به طور مثال صفی از اعداد صحیح واضح است، هر عنصر یک عدد صحیح است. اما صفی از اشیا شرایط را سخت می کند و با بیرون کشیدن یک عنصر از صف احتمالا نتوان به راحتی نوع داده آن را تشخیص داد.

❋ یک نوع می تواند حاوی انواع دیگری نیز باشد.

## حفاظت از کاربران با اختفای کد

اختفای کد امکان طراحی انعطاف پذیرتری را فراهم می کند، زیرا کاربران را از اجبار به هماهنگی با روش پیاده سازی درونی محافظت می کند.

در واقع دو نوع کد برای اشیاء وجود دارد:

۱- کد با وابستگی محدود: از روش پیاده سازی کد اشیاء دیگر مستقل است.
۲- کد متصل یا وابسته: وابسته به روش پیاده سازی کد اشیای دیگر است.

❋ به عنوان نمونه ای از اشکالات کد وابسته، اگر خصوصیتی در رابط عمومی شی ظاهر شود هر کس که از آن خاصیت استفاده کند به وجود آن خاصیت وابسته می شود.

❋ وابستگی کد غیر قابل اجتناب است و نمی توان آن را کاملا حذف کرد. اما باید برای دستیابی به حداقل وابستگی درون شیء تلاش کرد. معمولا چنین وابستگی با نوشتن یک رابط خوب و کامل محدود می شود.

مثال اختفاء:

```
public class Customer {
	//... Various customer methods
	Public Item [] items; //this array hold any selected items
}

public static void main (String [] args ) {
	Customer customer = new Customer();
	//...Select some items
	double total=0;
	for (int i=0 ; i<customer.items.length ; i++ ) {
		Item item = customer.items[i];
		Total = total + item.getAdjustedTotal();
	}
}
```

در مثال فوق اگر پیاده سازی تغییر کند تمام کدهایی که به آرایه Item ها دسترسی دارند باید تغیر کنند. پس آزادی مناسبی برای بهبود عملکرد شی وجود ندارد.

در این مثال باید دسترسی به آرایه عناصر Item خصوصی شود و دسترسی به این آرایه از طریق توابع مناسب فراهم شود.

## تقسیم مسوولیت

مخفی کردن پیاده سازی تنها یک قدم به سوی نوشتن کد مستقل است. برای داشتن کد مستقل باید محدوده مسوولیت و تقسیم وظایف صحیحی در برنامه اعمال شود.

تقسیم مسوولیت یعنی این که هر شی باید یک کار و وظیفه را انجام دهد. در واقع توابع و متغیرهایی که یک شی را تشکیل می دهند نباید بی ارتباط باشند و باید از لحاظ مفهومی ارتباط محکمی با یکدیگر داشته باشند، و با هم در جهت انجام یک مسوولیت واحد کار کنند.

مثال: نمونه از یک کد بد شی گرا که تقسیم مسوولیت ها به درستی انجام نشده است در زیر آمده است.

```
public class BadItem {
	private double unit_price;
	private double adjusted_price;
	private double discount; //a percentage discount to apply to the price
	private int quantity;
	private string description;
	private string id;

	public BadItem(String id, String description, int quanity, double price) {
		this.id = id;
		this.desc ription = description;
		if (quantity >=0) {
```

```
                        this.quantity = quantity;
                }
                else {
                        this.quantity = 0;
                }
                this.unit_price = price;
        }
        Public double getUnitPrice() {
                return unit_price;
        }
        // applies a percentage discount to the price
        Public void setDiscount(double discount) {
                if(discount <=1) {
                        this.discount = discount;
                }
        }
        public double getDiscount() {
                return this.discount;
        }
        public int getQuantity() {
                return this.quantity;
        }
        public string getProductId() {
                return this.id;
        }
        public string getDescription() {
                return this.description;
        }
        public double getAdjustedPrice() {
                return this.adjusted_price;
        }
        public void setAdjustedPrice() {
                this.adjusted_price = price;
        }
}
```

مشخص است که BadItem مسوول محاسبه قیمت تعدیل شده نیست و برای محاسبه آن نیاز به انجام محاسبات مانند زیر می باشد:

```
public static void main(String [] args) {
        //create the items
        BadItem milk = new BadItem("diary-011","1 Gallon Milk", 2, 2.50);
        // apply coupons
        milk.setDiscount(0.15);
        //get adjusted price
        double milk_price = milk.getQuantity() * milk.getUnitPrice();
        double milk_discount = milk.getDiscount() * milk_price;
        milk.setAdjustedPrice(milk_price – milk_discount);
        system.out.println("Your milk costs:\t $" + milk.getAdjustedPrice() );
}
```

در اینجا برای محاسبه قیمت نهایی لازم است چندین تابع صدا زده شوند و مسوولیت از دست شی خارج شده و به دست کاربر می افتد.

اگر فقط یک پیام برای شی ارسال شود و شی خود بتواند کاری که به او محول شده را انجام دهد، یک کار شی گرای واقعی انجام شده است.

چنانچه مسوولیت های شی زیاد باشند، پیاده سازی آن گیج کننده خواهد بود. همچنین مدیریت و توسعه آن نیز مشکل خواهد بود، و برای تغییر یک مسوولیت باید ریسک تغییر ناخواسته در مسوولیت دیگر را نیز متحمل شد.

اگر یک شی بیش از حد بزرگ شود، خود به خود به یک برنامه مبدل می شود و در دام روال گرایی می افتد. یعنی کپسوله سازی برنامه نویس هیچ سودی نداشته است.

تعریف کپسوله سازی موثر: تجرید مساله به علاوه مخفی کردن پیاده سازی به همراه تقسیم مسوولیت مناسب.

## نکته های کپسوله سازی

۱- تعمیم بیش از حد ممکن است فقط باعث از دست رفتن زمان بشود. نوشتن کلاسی که همه کاربران را راضی کند و همه مسائل را پوشش دهد تقریبا غیر ممکن است. پس اولویت حل مساله پیش رو می باشد.
۲- نباید یک کلاس را بیش از حد مساله پیش رو پیچیده کرد و سعی بر حل تمام مسائل داشت، قابلیت تعمیم برای این است که در آینده بتوان شی را برای سایر کاربردها تعمیم داد.
۳- تعمیم واقعی به مرور زمان و در عمل شکل می گیرد.
۴- ADT ها مستقیما قابل مقایسه با کلاس نبوده و خواص وراثت و چندشکلی را ندارند.
۵- فقط متدهایی که قرار است از بیرون کلاس استفاده شوند باید به عنوان رابط عمومی تعریف شوند.
۶- رابط شی باید رفتاری سطح بالاتر از پیاده سازی داشته باشد.

## مثال های عملی

مساله ۱: سازنده ها دو نوع هستند بدون آرگومان و با آرگومان

```
public class DoubleKey() {
        private string key1, key2;
        // no args constructor
        public DoublKey() {
                this.key1 = “key1”;
                this.key2 = “key2”;
        }
        // constructor with arguments
        public DoublKey(String key1, String key2) {
                this.key1 = key1;
                this.key2 = key2;
        }
        //accessor (getter)
        public string getKey1() {
                return this.key1;
        }
        //mutator (setter)
        public string setKey1(String key1) {
                this.key1 = key1;
        }
        //accessor (getter)
        public string getKey2() {
                return this.key2;
        }
        //mutator (setter)
        public string setKey2(String key2) {
                this.key2 = key2;
        }
```

مساله ۲: کلاسی برای حساب بانکی با شرایط زیر بنویسید.

- نام کلاس باید Account باشد.
- کلاس شامل دو تابع سازنده است:

```
public Account()
public Account(double initial_deposit)
```

سازنده بدون آرگومان موجودی اولیه را برابر صفر و سازنده دومی مقدار اولیه موجودی حساب را برابر initial_deposit قرار می دهد.

- کلاس باید شامل متدهای زیر باشد:

- public void depositFunds(double funds) جهت واریز پول به حساب
- public void withdrawFunds(double funds) جهت دریافت پول از حساب (اگر funds بیش از موجودی حساب باشد تنها به اندازه موجودی از حساب برداشته شود.) خروجی این متد مقدار پولی است که از حساب برداشته شده است.
- public double getBalance() جهت گرفتن موجودی حساب

پاسخ مساله ۲:

```
public class Account() {
	//private data
	private double balance;
	//constructor
	public Account() {
		this.balance = 0;
	}

	public Account(double initial_deposit) {
		this.balance = 0;
	}

	public void depositFunds(double funds) {
		this.balance += funds ;
	}

	public void withdrawFunds(double funds) {
		if(funds>this.balance) {
			funds = this.balance;
		}
		this.balance -= funds;
		return funds;
	}

	public double getBalance() {
		return this.balance;
	}
```

مساله ۳ کارت بازی:

بازی های ورق همراه با یک میز استاندارد بازی می شود پس بهترین جا برای شروع همان میز است. یک میز استاندارد شامل ۵۲ کارت است. پس از بر زدن می توان ورق را از هر جای میز برداشت. همچنین می توان ورق را در هر موقعیتی از میز قرار داد. باقی برداشت ها می تواند با برداشت ورق از هر نقطه از میز به صورت ویژه صورت گیرد.

کارت ها نیز ساختار مشترکی دارند. همه کارت ها در یکی از چهار مجموعه خشت، گشنیز، پیک و دل قرار می گیرند. هر کارت همچنین مقداری دارد: ۲ تا ۱۰، شاه، بی بی، سرباز و آس.

کارت ها همچنین دارای یک وضعیت هستند: رو به بالا یا رو به پایین.

کارت ها و میزهای بازی کار خاصی انجام نمی دهند. در واقع این بازیگر است که همه کارها از بر زدن تا بازی کردن را انجام می دهد.

در یک برنامه ساده، کارت باید نحوه نمایش خود را بداند. میز بازی نیز باید کارت ها را ساخته و درخود نگاه دارد. نهایتا بازیکن نیز باید بداند چگونه کارت ها را بکشد و از آن ها در بازی استفاده کند.

حال کلاس هایی که نمایانگر کارت ها، میز کارت ها و بازیکن هستند را طراحی نمایید.

پاسخ مساله ۳:

```
public class Card {
        private int rank;
        privat int suit;
        private boolean face_up;

        //suits
        public static final int HEARTS = 3;
        public static final int DIAMONDS = 4;
        public static final int CLUBS = 5;
        public static final int SPADES = 6;
        //values
        public static final int TWO = 2;
        public static final int THREE = 3;
        public static final int FOUR = 4;
        public static final int FIVE = 5;
        public static final int SIX = 6;
        public static final int SEVEN = 7;
        public static final int EIGHT = 8;
        public static final int NINE = 9;
        public static final int TEN = 10;
        public static final int JACK = 74;
        public static final int QUEEN = 81;
        public static final int KING = 75;
        public static final int ACE = 65;

        // create a new card – only use the constants to initialize
        Public Card(int suit, int rank) {
                // in a real program arguments should be validate
                this.suit = suit;
                this.rank = rank;
        }
```

```
        public int getSuit() {
                return this.suit;
        }
        public int getRank() {
                return this.rank;
        }
        public void faceDown() {
                this.face_up = false;
        }
        public void faceUp() {
                this.face_up = true;
        }
        public void isfaceUp() {
                return this.face_up;
        }

        public string display() {
                String display;
                If (this.rank > 10) {
                        display = String.valueOf ( (char) rank );
                } else {
                        display = String.valueOf ( rank );
                }
                switch (suit) {
                        case DIAMONDS:
                                return display + String.valueOf ( (char) DIAMONDS );
                        case HEARTS:
                                return display + String.valueOf ( (char) HEARTS);
                        case SPADES:
                                return display + String.valueOf ( (char) SPADES);
                        default:
                                return display + String.valueOf ( (char) CLUBS );
                }
        }
}
```

زمانی که نمونه ای از کلاس Card ایجاد شود دیگر نمی توان مقدار آن را تغییر داد. این گونه اشیاء را غیر قابل تغییر (immutable) می نامند.

کلاس Deck میز بازی است و مسوول ایجاد کارت ها توسط کلاس Card و فراهم کردن امکانات دسترسی به آن ها می باشد.

```
public class Deck {
        private java.util.LinkedList deck;

        public Deck() {
                buildCards();
        }

        public Card get(int index) {
```

```
		if (index < deck.size() ) {
			return (Card) deck.get( index );
		}
		return null;
	}

	public void replace( int index, Card card) {
		deck.set(index, card);
	}

	public int size() {
		return deck.size();
	}

	public Card removeFromFront() {
		if (deck.size() >0 ) {
			Card card = (Card) deck.removeFirst();
			return card;
		}
		return null;
	}

	public void buildCards() {
		deck = new java.util.LinkedList();

		deck.add( new Card(Card.CLUBS, Card.TWO) );
		deck.add( new Card(Card.CLUBS, Card.THREE) );
		deck.add( new Card(Card.CLUBS, Card.FOUR) );
		deck.add( new Card(Card.CLUBS, Card.FIVE) );
		deck.add( new Card(Card.CLUBS, Card.SIX) );
		deck.add( new Card(Card.CLUBS, Card.SEVEN) );
		deck.add( new Card(Card.CLUBS, Card.EIGHT) );
		deck.add( new Card(Card.CLUBS, Card.NINE) );
		deck.add( new Card(Card.CLUBS, Card.TEN) );
		deck.add( new Card(Card.CLUBS, Card.JACK) );
		deck.add( new Card(Card.CLUBS, Card.QUEEN) );
		deck.add( new Card(Card.CLUBS, Card.KING) );
		deck.add( new Card(Card.CLUBS, Card.ACE) );

		deck.add( new Card(Card.SPADES, Card.TWO) );
		deck.add( new Card(Card.SPADES, Card.THREE) );
		deck.add( new Card(Card.SPADES, Card.FOUR) );
		deck.add( new Card(Card.SPADES, Card.FIVE) );
		deck.add( new Card(Card.SPADES, Card.SIX) );
		deck.add( new Card(Card.SPADES, Card.SEVEN) );
		deck.add( new Card(Card.SPADES, Card.EIGHT) );
		deck.add( new Card(Card.SPADES, Card.NINE) );
		deck.add( new Card(Card.SPADES, Card.TEN) );
		deck.add( new Card(Card.SPADES, Card.JACK) );
		deck.add( new Card(Card.SPADES, Card.QUEEN) );
		deck.add( new Card(Card.SPADES, Card.KING) );
```

```
		deck.add( new Card(Card.SPADES, Card.ACE) );

		deck.add( new Card(Card.HEARTS, Card.TWO) );
		deck.add( new Card(Card.HEARTS, Card.THREE) );
		deck.add( new Card(Card.HEARTS, Card.FOUR) );
		deck.add( new Card(Card.HEARTS, Card.FIVE) );
		deck.add( new Card(Card.HEARTS, Card.SIX) );
		deck.add( new Card(Card.HEARTS, Card.SEVEN) );
		deck.add( new Card(Card.HEARTS, Card.EIGHT) );
		deck.add( new Card(Card.HEARTS, Card.NINE) );
		deck.add( new Card(Card.HEARTS, Card.TEN) );
		deck.add( new Card(Card.HEARTS, Card.JACK) );
		deck.add( new Card(Card.HEARTS, Card.QUEEN) );
		deck.add( new Card(Card.HEARTS, Card.KING) );
		deck.add( new Card(Card.HEARTS, Card.ACE) );

		deck.add( new Card(Card.DIAMONDS, Card.TWO) );
		deck.add( new Card(Card.DIAMONDS, Card.THREE) );
		deck.add( new Card(Card.DIAMONDS, Card.FOUR) );
		deck.add( new Card(Card.DIAMONDS, Card.FIVE) );
		deck.add( new Card(Card.DIAMONDS, Card.SIX) );
		deck.add( new Card(Card.DIAMONDS, Card.SEVEN) );
		deck.add( new Card(Card.DIAMONDS, Card.EIGHT) );
		deck.add( new Card(Card.DIAMONDS, Card.NINE) );
		deck.add( new Card(Card.DIAMONDS, Card.TEN) );
		deck.add( new Card(Card.DIAMONDS, Card.JACK) );
		deck.add( new Card(Card.DIAMONDS, Card.QUEEN) );
		deck.add( new Card(Card.DIAMONDS, Card.KING) );
		deck.add( new Card(Card.DIAMONDS, Card.ACE) );
	}
}
```

کلاس Dealer مسوولیت پخش کردن کارت های بازیکن و میز را برعهده دارد.

```
public class Dealer() {
	private Deck deck;

	public Dealer(Deck d) {
		deck = d ;
	}

	public void shuffle() {
		//randomize the card array
		int num_cards = deck.size();
		for (int i=0 ; i<num_cards; i++) {
			int index = (int) (Math.random() * num_cards );
			Card card_i = (Card) deck.get(i);
			Card card_index = (Card) deck.get(index);
			deck.replace(i, card_index);
```

```
                        deck.replace(index, card_i);
                }
        }

        public Card dealCard() {
                if (deck.size() > 0) {
                        return deck.removeFromFront();
                }
                Return null;
        }
}
```

هر سه کلاس به نحوی مسوولیت پذیری را بر عهده گرفته و نحوه پیاده سازی خود را مخفی نگاه می دارند. به نکات زیر دقت کنید:

- هیچ چیزی بیانگر این نیست که کلاس Deck از LinkedList برای پیاده سازی استفاده کرده است.
- کلاس Card تعدادی ثابت را تعریف کرده و آزاد است به هر نحو ممکن از این ثابت ها استفاده کند. حتی آزاد است که در صورت نیاز مقدار این ثابت ها را تغییر دهد.
- متد buildCards() از کلاس Deck نحوه مخفی کردن جزییات پیاده سازی را به خوبی نشان می دهد. به راحتی در اینجا می توان کارت ها را با یک حلقه for مقداردهی کرد.
- توجه شود که برنامه نباید به مقادیری که به عنوان ثابت تعریف شده اند وابسته باشد. برای فراخوانی ثابت ها به راحتی می توان به شکل Card.TWO یا Card.THREE از آن ها استفاده نمود. در زمان لازم کلاس Card مقدار واقعی را جایگزین خواهد نمود.

# وراثت

وراثت، مکانیزمی است برای مشتق کردن یک کلاس تازه از یک کلاس موجود که کلاس جدید تمام خواص و رفتارهای کلاس قبلی را به ارث می برد.

برای مثال کلاس زیر را در نظر بگیرید:

```
public class Employee {
	private String first_name;
	private String last_name;
	private String wage;

	public Employee ( String first_name, String last_name, double wage) {
		this.first_name = first_name;
		this.last_name = last_name;
		this.wage = wage;
	}
	public double getWage() {
		return this.wage;
	}
	public String getFirstName() {
		return this.first_name;
	}
	public String getLastName() {
		return this.last_name;
	}
}
```

این کلاس می تواند در یک سیستم مدیریت حقوق و دستمزد استفاده شود. حال چنانچه در این برنامه نیاز به تعریف کارمند شریک در سود باشد، این کارمند علاوه بر حقوق پایه (که می تواند صفر نیز باشد) سهمی از فروش یا سود نیز دارد. بقیه ویژگی های این کارمند همانند کارمند معمولی می باشد.

برای تعریف کلاس جدید برای کارمند شریک در سود راه کار بد کپی کردن مجدد کلاس Employee با نام جدید و افزودن کدی برای تعیین میزان سود می باشد. در این حالت در صورت نیاز به هر تغییری باید هر دو کلاس را تغییر داد.

راهکار مناسب تر استفاده از وراثت به شیوه زیر است:

```
public class CommissionedEmployee extends Employee {
	private double commission; //the commission per unit
	private int units; //keep track of the number of units sold

	public CommissionedEmployee ( String first_name, String last_name, double wage, double
commission) {
		super (first_name, last_name, wage); // call the original constructor
		this.commission = commission;
	}
	public double calculatePay() {
		return getWage() + (this.commission * this.units);
	}
	public void addSales (int units) {
		this.units = this.units + units ;
	}
	public void resetSales (int units) {
		this.units = 0;
	}
}
```

در مثال اخیر، CommissionedEmployee از کلاس Employee مشتق شده است در نتیجه تمام صفات و متدها همچون first_name, last_name, wage, getFirstNAme, getLastName را به ارث می برد.

قطعه کد زیر نمونه ای جهت استفاده از کلاس تعریف شده در بالا می باشد:

```
public static void main (String [] args) {
	CommissionedEmployee c = new CommissionedEmployee(“John”,”Doe”,1000,10); //define
object
	c.addSales(5);
	System.out.println(“First Name: “ + c.getFirstName() );
	System.out.println(“Last Name: “ + c.getLastName() );
	System.out.println(“Base Pay: “ + c.getWage() );
	System.out.println(“Total Pay: “ + c.calculatePay() );
}
```

توجه: کلاس فرزند (زیرکلاس) می تواند هر رفتاری از کلاس والد را که مناسب نمی بیند دوباره برای خود تعریف کند. ارزش جایگزینی (override) در این است که می توان بدون نیاز به تغییر کلاس اصلی، عملکرد شیء را تغییر داد. لذا می توان کد آزموده شده و مطمئن را بدون تغییر حفظ نمود.

توجه: اصولا زمانی از وراثت استفاده می کنیم که هر دو کلاس از نوع یکسانی باشند.

سلسله مراتب وراثتی: عبارت است از ساختار درختی شکل روابطی که بین کلاس هایی که در وراثت سهیم هستند بوجود می آید.

نکته: وقتی یک کلاس از یک کلاس والد مشتق می شود، کلاس فرزند تمام صفات و رفتارهایی که ممکن است کلاس والد از کلاس های دیگر به ارث برده باشد را خود به خود دریافت می کند.

نکته: کلاس مشتق شده (فرزند) فقط می تواند طرز کار را تغییر دهد و یا کارایی های جدید ایجاد کند، اما هیچگاه قادر نخواهد بود کارایی را کم کند.

نکته: اگر در شرایطی لازم باشد که کلاس مشتق شده (فرزند) برخی رفتارها یا کارایی های والد را حذف کند می توان نتیجه گیری کرد که این کلاس باید قبل از کلاس والد در سلسله مراتب وراثت ظاهر شود.

نکته: کلاس ها می توانند تنها از یک والد مشتق شوند. البته برخی از زبان ها اجازه اشتقاق از چند کلاس را می دهند، این عمل را وراثت چندگانه می نامند.

نکته: اکثر زبان های برنامه نویسی در وراثت اجازه جایگزینی صفات یا خواص را نمی دهند و تنها امکان جایگزینی رفتار یا متد را می دهند.

نکته: کلاس ریشه یا کلاس پایه، کلاسی است که در سلسله مراتب وراثت بالاترین جایگاه را به خود اختصاص داده است.

نکته: کلاس برگ، کلاسی است که هیچ کلاس از آن مشتق نمی شود.

مساله۱ : الف) کلاسی برای تعریف نقاط دوبعدی طراحی کنید. ب) کلاسی برای تعریف نقاط ۳ بعدی که از کلاس قبلی مشتق شده باشد طراحی کنید.

```
public class TwoDimensionalPoint {
	private double x_coord;
	private double y_coord;

	public TwoDimensionalPoint(double x, double y) {
		this.setXCoordinate(x);
		this.setYCoordinate(y);
	}
	public double getXCoordinate() {
		return this.x_coord;
	}
	public double getYCoordinate() {
		return this.y_coord;
	}
	public void setXCoordinate(double x) {
		this.x_coord = x;
```

```
	}
	public void setYCoordinate(double y) {
		this.y_coord = y;
	}
	public String toString() {
		return “X coordinate is: “ + this. getXCoordinate() + “\n” +
			“Y coordinate is: “ + this. getYCoordinate() ;
	}
}
```

و حال کلاس مختصات سه بعدی

```
public class ThreeDimensionalPoint extends TwoDimensionalPoint {
	private double z_coord;

	public ThreeDimensionalPoint(double x, double y, double z) {
		super (x, y, z); //initialize the inherited attributes by calling the parent constructor.
		this.setZCoordinate(x);
	}
	public double getZCoordinate() {
		return this.z_coord;
	}
	public void setZCoordinate(double z) {
		this.z_coord = z;
	}
	public String toString() {
		return   “X coordinate is: “ + this. getXCoordinate() + “\n” +
			“Y coordinate is: “ + this. getYCoordinate() + “\n” +
			“Z coordinate is: “ + this. getZCoordinate() ;
	}
}
```

قطعه کد زیر می تواند نمونه ای از کارکرد کلاس های بالا را نشان دهد:

```
public static void main(String [] args) {
	TwoDimensionalPoint two = new TwoDimensionalPoint(1,2);
	ThreeDimensionalPoint three = new ThreeDimensionalPoint(1,2,3);
	System.out.println(two.toString());
	System.out.println(three.toString());
}
```

بهتر است بار دیگر جهت یادآوری، سطوح دسترسی به متدها و خصوصیت ها را مرور کنیم:

- public : اجازه دسترسی توسط تمام اشیاء
- private : اجازه دسترسی فقط داخلی (خود موجودیت)

– protected : اجازه دسترسی داخلی و کلاس های مشتق شده از آن

نکته: صفات و متدهایی که استفاده از آن ها فقط برای کلاس معنی دار است را اختصاصی نگاه دارید.

نکته: صفات و متد اختصاصی را برای کاربردهای احتمالی، در سطح محافظت شده قرار ندهید.

شاید در آینده لازم باشد به تعریف کلاس مراجعه و سطوح دسترسی را تغییر داد، اما در عوض طراحی استوارتری خواهید داشت.

مساله ۲: کلاس پایه زیر را در نظر بگیرید:

```
public class MoodyObject {
	//return the mood
	protected String getMood() {
		return “moody”;
	}
	public void queryMood() {
		System.out.println(“I feel “ + this.getMood() + “ today!”);
	}
}
```

دو زیرکلاس ایجاد کنید با نام های sadObject و happyObject هر دو زیرکلاس باید متد getMood() را جایگزین نمایند تا پیغام های مناسب وضعیتشان چاپ گردد. کلاس sadObject باید متد public void cry() را تعریف نماید به طوری که پیغام ”nonono“ را در خط فرمان چاپ کند و کلاس happyObject باید متد public void laugh را تعریف نماید به طوری که پیغام ”hahaha“ را در خط فرمان چاپ کند.

پاسخ مساله۲:

```
public class happyObject extends MoodyObject {
	protected String getMood() {
		return “happy”;
	}
	public void laugh() {
		System.out.println(“hahaha”);
	}
}
```

```
public class sadObject extends MoodyObject {
	protected String getMood() {
		return “sad”;
	}
	public void cry() {
		System.out.println(“nonono”);
	}
}
```

مساله ۳: کلاس Employee را به شکل زیر در نظر بگیرید.

```
public class Employee {
	private String first_name;
	private String last_name;
	private String wage;

	public Employee ( String first_name, String last_name, double wage) {
		this.first_name = first_name;
		this.last_name = last_name;
		this.wage = wage;
	}
	public double getWage() {
		return this.wage;
	}
	public String getFirstName() {
		return this.first_name;
	}
	public String getLastName() {
		return this.last_name;
	}
	public String printPayCheck() {
		String full_name = this.last_name + ", " + this.first_name ;
		Return ("Pay: " + full_name + " $" + this.calculatePay() );
	}
}
```

از کلاس Employee می توان به عنوان کلاس پایه برای کلاس های HourlyEmployee, CommissionedEmployee, SalariedEmployee استفاده نمود. هر زیرکلاس متد calculatePay() مخصوص خود را دارد اما کلاس Employee هیچ قانونی جهت محاسبه پرداخت ندارد. اگر این قسمت را حذف کنیم چندان جالب نیست زیرا کلاس پایه رفتار کلی یک کارمند را مدل نخواهد کرد. راه حل دیگر نوشتن متدی ساده مثلا بازگرداندن مقدار دستمزد است. ممکن است برنامه نویسان دیگر این متد را بازنویسی نکنند و مشکلاتی رخ دهد. بهتین کار استفاده از کلاس مجرد (abstraction) است. سپس متد calculatePay() را نیز به صورت مجرد تعریف می کنیم.

کلاس مجرد: کلاس پایه ای است که نمی توان از آن نمونه گرفت و شی تعریف نمود.

متد مجرد: به متدی گفته می شود که تعریف شده ولی پیاده سازی نشده است. فقط در کلاس های مجرد می توان متد مجرد داشت. وظیفه کلاس های فرزند است که متد مجرد را پیاده سازی کنند.

حال کلاس Employee را به صورت مجرد بازنویسی می کنیم:

```
public abstract class Employee {
	...
	public abstract double calculatePay();
	// rest of the code remains the same.
}
```

حال زیرکلاس HourlyEmployee را به صورت زیر تعریف می کنیم:

```
public class HourlyEmployee extends Employee {
	private int hours; //kep track of the # of hours worked
	public HourlyEmployee(String first_name, String last_name, double wage) {
		super(first_name, last_name, wage); //call the original constructor
	}
	public double calculatePay() {
		return getWage() * this.hours;
	}
	public void addHours(int hours) {
		this.hours = this.hours + hours ;
	}
	public void resetHours() {
		this.hours = 0 ;
	}
}
```

مساله ۳: مساله اول را به صورت کلاس مجرد بازنویسی کنید.

مساله ۴: مثال حساب بانکی در قسمت کپسوله سازی را با استفاده از کلاس تجرید بازنویسی کنید به گونه ای که مشتریان بتوانند چندین نوع حساب بانکی به شرح زیر داشته باشند.

- حساب پس انداز:
  - به نرخ سود بانکی به موجودی حساب اضاف شود.
  - امکان برداشت بیش از موجودی وجود ندارد.
- حساب جاری:
  - نمی توان بیش از موجودی پول برداشت کرد.
  - بانک به ازای هر نقل و انتقال کارمزد دریافت می کند.
  - سودی به موجودی اضافه نمی شود.
- حساب اعتباری:
  - می توان بیش از موجود چول برداشت کرد.
  - اگر موجودی منفی شود (اعتبار اخذ شود) کارمزد بانک از موجودی کسر می شود.
  - اگر موجودی مثبت باشد کارمزدی وجود ندارد.

توجه شود که در اینجا هدف نوشتن یک سیستم جامع حسابداری نمی باشد پس تنها روی وراثت و استفاده از تجرید تمرکز شود.

# چندشکلی بودن (Polymorphism)

امکان استفاده از انواع مختلف داده از طریق یک واسط را می گویند.

چندشکلی یا چندریختی انواع مختلفی دارد که در اینجا به برخی از آن ها می پردازیم:

## Overloading یا سربارگذاری یا چندشکلی بودن ویژه (ad-hoc)

این روش تغییر در سطح متدها یا عملگرها می باشد.

به طور مثال در دستور ;system.out.print(para) می توان به جای para انواع مختلف رشته ای (string) یا عددی (integer) را قرار داد.

به عنوان نمونه دیگر عملگر + در جاوا هم برای جمع کردن دو عدد به کار می رود و هم برای به هم چسباندن دو رشته کاربرد دارد و خود عملگر بر اساس نوع متغیرهای اطرافش می تواند عملکرد صحیحی را از خود بروز دهد، بدون آن که نیازی باشد توضیحات اضافه ای به آن داد. به عبارت دیگر اگر دو عدد را با علامت + با هم جمع کنیم نتیجه حاصل جمع آن ها خواهد بود و اگر دو عبارت رشته ای را با هم + کنیم حاصل رشته ای خواهد بود متشکل از هر دو رشته که به هم چسبیده اند.

## Overriding یا جایگزینی

در این شیوه می توان داخل یک کلاس متدهای متعددی داشت و سپس در کلاس هایی که از آن کلاسه پایه وراثت می گیرند برخی از متدها را جایگزین نمود (دوباره تعریف نمود) بدین ترتیب متد فوق در کلاس والد به یک شیوه عمل می کند و در کلاس فرزند به شیوه ای دیگر. این عمل را می توان در چندین کلاس فرزند نیز انجام داد و بدین ترتیب شیوه های متعدد متفاوتی از اجرای متد فوق بدست آورد. در این صورت گفته می شود متد فوق override شده است.

مثالی از عمل جایگزینی را پیش از این در کلاس MoodyObject به کار بردیم.

به عنوان یک مثال دیگر فرض کنید می خواهیم یک کلاس گرافیکی برای رسم اشیای گرافیکی ایجاد کنیم. این مثال را به شرح زیر تعریف می کنیم.

مساله:

می خواهیم کلاسی با نام Graphic داشته باشیم که بتوانیم برای هر شکل گرافیکی همچون دایره، مثلث و مستطیل با استفاده از متد Draw آن شکل را ترسیم کنیم.

یک شیوه بد ساخت کلاس فوق به صورت زیر است:

```
public class Graphic {
   public void DrawCircle()    {
```

```
    //
  }

  public void DrawRectangle()   {
    //
  }

  public void DrawTriangle()   {
    //
  }

  // and other methods for drawing anything else
}
```

حال این کلاس را با استفاده از مفاهیم وراثت و چندشکلی جایگزینی بازنویسی کنید به گونه ای که برای هر شکل یک کلاس داشته باشیم.

پاسخ :

```
public class Graphic {
  public void Draw()
  {
    //
  }
}

public class Circle extends Graphic {
  public void Draw()    {
    // codes for drawing circle
  }
}

public class Rectangle extends Graphic {
  public void Draw()    {
    // codes for drawing rectangle
  }
}

public class Triangle extends Graphic {
  public void Draw()    {
    // codes for drawing rectangle
  }
}
```

تمرین: کلاس فوق را با استفاده از مفاهیم وراثت، چندشکلی جایگزینی و تجرید بازنویسی کنید.

## چندشکلی بودن درونی (چندشکلی بودن خالص)

همه زبان ها چندشکلی بودن را پشتیبانی نمی کنند. وراثت زمینه را برای برخی انواع چندشکلی فراهم می کند.

کلاس های زیر را درنظر بگیرید:

```
public class ParentObject {
	public String speak() {
		return “This is a parent object”;
	}
}

public class FirstChildObject extends ParentObject {
	public String speak() {
		return “This is child 1”;
	}
}

public class SecondChildObject extends ParentObject {
	public String speak() {
		return “This is child 2”;
	}
}

public class ThirdChildObject extends ParentObject {
	public String speak() {
		return “This is child 3”;
	}
}

public class FourthChildObject extends ParentObject {
	public String speak() {
		return “This is child 4”;
	}
}
```

این کلاس های یک سلسله مراتب وراثت یکدست هستند. کلاس پایه یک روال speak() تعریف کرده و هر زیرکلاس آن را دوباره برای خود تعریف کرده است.

```
public static void main(String [] args) {
	ParentObject parent = new ParentObject();
	FirstChildObject child1 = new FirstChildObject ();
	SecondChildObject child2 = new SecondChildObject ();
	ThirdChildObject child3 = new ThirdChildObject ();
	FourthChildObject child4 = new FourthChildObject ();

	ParentObject [] parents = new ParentObject();
	parents[0] = parent;
	parents[1] = child1;
	parents[2] = child2;
```

```
        parents[3] = child3;
        parents[4] = child4;

        System.out.println("parents[0] speaks: "+ parents[0].speak());
        System.out.println("parents[1] speaks: "+ parents[1].speak());
        System.out.println("parents[2] speaks: "+ parents[2].speak());
        System.out.println("parents[3] speaks: "+ parents[3].speak());
        System.out.println("parents[4] speaks: "+ parents[4].speak());
}
```

با آن که آرایه حاوی عناصر ParentObject می باشد، رفتار هر عنصر در صدا کردن روال speak() متفاوت است. حال روال زیر را درنظر بگیرید:

```
public void makeSpeak(FirstChildObject obj) {
        System.out.print (obj.speak());
}
```

فرض کنید برای سه نوع فرزند دیگر نیز کد مشابهی داشته باشیم. اما تمام این فرزندان به هم مربوطه هستند چون از یک کلاس واحد مشتقد شده اند. با استفاده از جانشین پذیری و جندشکلی بودن می توان برای تمام انواع ParentObject روال واحدی را نوشت:

```
public void makeSpeak(ParentObject obj) {
        System.out.print (obj.speak());
}
```

جانشین پذیری امکان پاس کردن تمان انواع ParentObject به روال را فراهم می کند.

خاصیت مهم چندشکلی بودن درونی، کاهش کدنویسی مورد نیاز است.

حسن دیگر ParentObject در این است که به نظر می رسد متغیرهای از نوع ParentObject رفتارهای بسیار متعددی را از خود نشان می دهند. پیغام نماش داده شده توسط makeSpeak() بر اساس نوع ورودی آن تغییر خواهد کرد.

مساله۱: کلاس گزارش دهی را به گونه ای باز نویسی کنید که نحوه ثبت گزارش فقطدر کلاس های فرزند مشخص شود.

```
public class BaseLog {
        //Some useful constants
        private final static String DEBUG = "DEBUG";
        private final static String INFO = "INFO";
        private final static String WARNING = "WARNING";
        private final static String ERROR = "ERROR";
        private final static String FATAL = "FATAL";

        public void debug (String message) {
                log (DEBUG, message);
        }
        public void info (String message) {
                log ( INFO, message );
```

```
        }
        public void warning (String message) {
                log (WARNING , message );
        }
        public void error (String message) {
                log (ERROR, message );
        }
        public void fatal (String message) {
                log (FATAL , message );
                Sysem.exit(0);
        }

        //let subclasses define how and where to write log to.
        protected abstract void log (String type, String message);
}
```

حال یک کلاس فرزند از کلاس BaseLog می نویسیم که بتواند گزارشات را روی صفحه نمایش نشان دهد.

```
public class ScreenLog extends BaseLog {
        private void log (String type , String message ) {
                System.out.println ( type + ": " + message );
        }
}
```

و کلاس فرزند بعدی باید بتواند گزارش ها را روی فایل چاپ کند.

```
public class FileLog extends BaseLog {
        private java.io.PrintWriter pw;

        public FileLog(String filename) {
                // ... create and open new file (pw)
        }
        private void log (String type , String message ) {
                pw.println ( type + ": " + message );
        }
        public void close() {
                pw.close();
        }
}
```

در مثال بالا با توجه به اینکه هدف برنامه نویسی نبوده و صرفا استفاده از مفاهیم شی گرایی بوده است برخی قسمت های مربوط به کدنویسی کار با فایل ها حذف شده است.

## چندشکلی بودن پارامتری

روال زیر را در نظر بگیرید:

```
int add (int a, int b)
```

این روال دو عدد صحیح را با هم جمع می کند. برای جمع اعداد حقیقی یا ماتریس ها باید روال های دیگری نوشت:

```
int add_real (real a, real b)
```

```
int add_matrix (matrixt a, matrix b)
```

یک روش استفاده از چندشکلی بودن درونی است. می توان نوعی به نام Addable تعریف کرد که بداند چگونه یک متغیر از نوع خود را به دیگری بیافزاید.

```
public abstract class Addable {
        public Addable add (Addable);
}
```

و روال جدید مانند زیر خواهد بود:

```
Addable add_addable (Addable a, Addable b)
        Return a.add(b);
```

با این کار باید فقط یک روال برای افزودن مقادیر نوشته شود. البته روال فقط برای آرگومان های از نوع Addable کار می کند. و Addbale ها نیز باید از یک نوع واحد باشند. البته مساله هنوز باقی است و باید برای هر نوع داده که با Addable متفاوت باشد روال جداگانه ای نوشت. با استفاده از روش پارامتری داریم:

```
add ([T] a, [T] b): [T]
```

در اینجا [T] آرگومانی مانند a, b است که نوع داده آن ها را مشخص می کند. با این شیوه تعریف روال، تعریف انواع داده تا زمان اجرا به تاخیر می افتد. درون روال می توان به صورت زیر باشد:

```
[T] add ([T] a, [T] b)
        return a+b;
```

در چنین حالتی هر آرگومانی که پاس می شود باید اپراتور + را برای خود تعریف کرده باشد.

## مشکلات چند شکلی

۱. بالا فرستادن رفتارها بیش از حد در سلسله مراتب وراثت می تواند باعث تضعیف سلسله مراتب شود.
۲. روال های چندشکلی کارایی کمتری نسبت به روال هایی که دقیقا آرگومان های خود را می شناسند دارند.
۳. هرچند که می توان یک زیرکلاس را به روالی که برای والد طراحی شده پاس داد (چندشکلی بودن درونی)، اما با این حال روال نمی تواند از قابلیت های جدیدی که زیرکلاس به خود افزوده استفاده کند.
۴. توجه شود که هر زبانی چندشکلی بودن را به صورتی وتفاوت پیاده سازی می کند. به طور مثال بیشتر زبان ها چندشکلی بودن پارامتری را تا حدی پشتیبانی می کنند و تعداد کمی از آن ها چندشکلی بودن پارامتری را پشتیبانی می کنند به طور مثال جاوا چندشکلی بودن پارامتری را پشتیبانی نمی کند.