

ПОЛИМОРФИЗЪМ

ЛЕКЦИОНЕН КУРС “ООП (JAVA)”



СТРУКТУРА НА ЛЕКЦИЯТА

- Въведение
- Преобразуване нагоре
- Късно (динамично) свързване
- Конструктори и полиморфизъм

НАСЛЕДЯВАНЕ

- Наследяване
 - При създаване на нов клас със сходна функционалност е удобно да клонираме съществуващ и след това да добавим допълнение и промени в клонинга
- Проблем на архитектурата
 - Отразява отношения между типове
 - Наследяването изразява сходството между типовете посредством концепцията за базови и производни типове
 - Базовият тип съдържа всички характеристики и поведения, общи за произведените от него типове
- Създаваме базови типове за представяне ядрото на разработваното приложение
 - За изразяване различните начини, по които може да се реализира това ядро, създаваме типове, производни на базовия

МНОГОКРАТНО ИЗПОЛЗВАНЕ НА ИНТЕРФЕЙСА

- Представяне на решението в същите условия като на задачата е огромно предимство
 - Не се нуждае от много междинни модели за преминаване от описанието на задачата към описание на решението
- При обектите, йерархията на типовете е основният модел
 - От описанието на системата в реалния свят директно преминаваме към описание на системата в код
- Когато наследяваме от съществуващ тип създаваме нов
 - Този нов тип съдържа не само всички членове на съществуващия тип
 - По-важно, дублира интерфейса на базовия клас
 - Т.е., всички съобщения, които могат да се изпращат на обекти от базовия клас, могат да се изпращат също на обекти от производния клас
 - Тъй като знаем типа на съобщенията, това означава, че производният клас е от същия тип като базовия клас
- Такъв тип еквивалентност (чрез наследяване) е основополагаща за разбиране значението на ООП

МНОГОКРАТНО ИЗПОЛЗВАНЕ НА ИНТЕРФЕЙСА

- Понеже базовият и производният клас притежават един и същ интерфейс, трябва да съществува някаква обща имплементация
 - Т.е., трябва да има код, който да се изпълни, когато обект получи съобщение
- Ако просто само наследим клас, без да правим нищо повече, методите на базовия клас се наследяват в производния клас
 - Т.е., производният клас има не само същия тип, но и същото поведение

РАЗГРАНИЧАВАНЕ

- Два начина за разграничаване новия произведен от съществуващия базов клас:
 - Нова функционалност
 - Добавяме нови функции към производния клас
 - Внимателно трябва да се прецени дали тази нова функционалност е необходима за решаване на проблема
 - Въпреки, че ключовата дума `extends` води до такова очакване
 - предефиниране (overriding)
 - Променяме поведението на съществуваща функция на базовия клас предефиниране
 - По-съществен начин за разграничаване
 - В производния клас създаваме нова дефиниция на съществуващата функция

ВЗАИМООТНОШЕНИЯ

- Трябва ли наследяването да предефинира само функциите на базовия клас?
 - Това би означавало, че производният тип е точно същият като базовия, тъй като има същия интерфейс
- Можем да заместим обект от производния клас с обект от базовия клас
 - „чисто“ заместване, познато като принцип на заместването
 - В този случай отношенията между базовия и производния клас са идентични
 - „един квадрат е фигура“
- Можем да добавяме нови елементи към интерфейса на производния клас
 - Този нов тип все още може да замести базови тип
 - Това заместване не е идентично, а сходно взаимоотношение

ВЗАИМОЗАМЕНЯЕМОСТ

- Когато работим с йерархия от обекти, в определени ситуации искаме да третираме един обект, не като обект от дадения тип, а като обект от базовия тип
 - Това ни дава възможност да пишем код, независещ от определени типове
 - Напр., всички фигури могат да бъдат изчертавани, изтривани, премествани, ...
 - Тези функции изпращат съобщение към фигурата обект без да се интересуват какво прави обектът със съобщението
 - Такъв код не се влияе от добавянето на нови типове
 - Добавянето на нови типове е най-често срещаният начин за разширяване на една обектно-ориентирана програма за обработка на нови ситуации
 - Напр., добавяне на нов тип фигура не променя функциите, работещи със сродни фигури

ПРОБЛЕМ

- Целият смисъл е в следното:
 - Когато се изпрати едно съобщение, програмистът не иска да знае кой код ще се изпълни
 - Обектът, получаващ съобщението, трябва да изпълни правилния код в зависимост от неговия специфичен тип
- Съществува обаче един проблем при разглеждане на обекти от произведен тип като обекти от техния базов тип
 - Компиляторът не може да знае по време на компилация точно кой код ще се изпълни
- Отговорът на този проблем е основна особеност на ООП
 - Компиляторът не може да извиква функции по традиционния начин

РАННО И КЪСНО СВЪРЗВАНЕ

- При императивните езици за програмиране се използва ранно свързване
 - Компиляторът генерира обръщение към определено име на функция
 - Програмата за свързване преобразува това обръщение в абсолютен адрес на кода, който трябва да се изпълни
- Обектно-ориентираният езици за програмиране използват концепцията за късно свързване
 - Когато се изпрати съобщение до обект, извикваният код не се определя до времето за изпълнение
 - Java използва специален код, който изчислява адреса на тялото на функцията, използвайки информация, съхранявана в самия обект

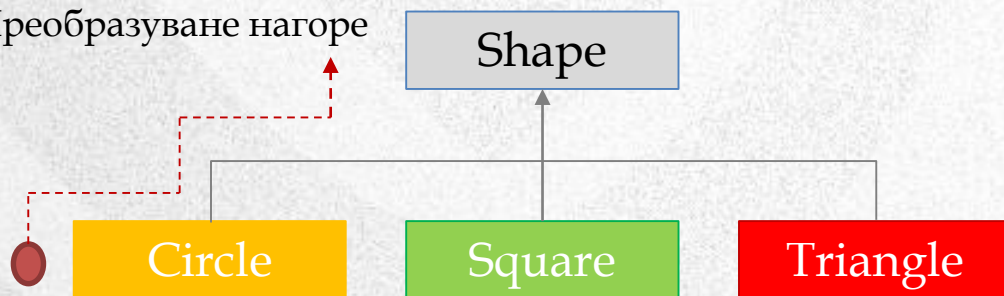
ПОЛИМОРФИЗЪМ ПО ПОДРАЗБИРАНЕ

- В някои езици за програмиране (в частност C++) трябва изрично да се посочи, че искаме една функция да притежава гъвкавостта на късното свързване
 - В тези езици: по подразбиране функциите не се свързват динамично
- В Java: по подразбиране - **динамично** (късно) свързване
 - Не е необходимо да използваме ключови думи за полиморфизъм

ПРИМЕР

Изпълнява се правилният код на draw() поради полиморфизма

Преобразуване нагоре



```
void doStuff(Shape s) {  
    s.erase();  
    // ...  
    s.draw();  
}
```

```
Circle c = new Circle();  
Triangle t = new Triangle();  
Line l = new Line();  
doStuff(c);  
doStuff(t);  
doStuff(l);  
...
```

- Кодът игнорира специфичните особености на типа
- Комуникира единствено с базовия клас
- Специфичната за типа информация се отделя от този код
- Така кодът става по-прост за създаване и по-лесен за разбиране
- Ако се добави нов тип (напр. Hexagon) чрез наследяване, кодът ще работи еднакво за новия тип (както за съществуващите)
- По този начин програмата става разширяема

КАКВО Е ПОЛИМОРФИЗЪМ?

- Полиморфизъм – основна възможност на един обектно-ориентиран език за програмиране
 - Предоставя друго измерение в отделянето на интерфейса от имплементацията с цел разделяне на това **какво** се прави, от това **как** се прави
 - Позволява:
 - Подобрена организация на кода и четливост
 - Също така създаване на разширяеми програми, които могат да се разширяват не само при първоначално създаване на проекта, но също при необходимост от добавяне на нови възможности
- Полиморфизмът е насочен към разделяне на типове
 - Полиморфното извикване на методи позволява даден тип да се разграничи от друг подобен тип, но ако и двата типа произлизат от един и същ базов тип
 - Тази разлика се проявява като разлика в поведението на методите, които могат да се извикват чрез базовия клас
- Полиморфизъм познат още като динамично свързване, късно свързване, свързване по време на изпълнение

ПРЕОБРАЗУВАНЕ НАГОРЕ

- При наследяването – един обект може да се използва като свой собствен тип или като обект от своя базов тип
 - Това се нарича преобразуване нагоре (upcasting)
- При това възниква проблем, познат като „стесняване“ на интерфейса

ПРИМЕР

```
class Note {  
    private int value;  
    private Note(int val) { value = val; }  
    public static final Note MIDDLE_C = new Note(0),  
                           C_SHARP = new Note(1),  
                           B_FLAT = new Note(2);  
}  
class Instrument {  
    public void play(Note n) {  
        System.out.println("Instrument.play()");  
    }  
}  
class Wind extends Instrument {  
    public void play(Note n) {  
        System.out.println("Wind.play()");  
    }  
}
```

```
public class Music {  
    public static void tune(Instrument i) {  
        // ...  
        i.play(Note.MIDDLE_C);  
    }  
    public static void main(String[ ] args) {  
        Wind flute = new Wind();  
        tune(flute); // Преобразуване нагоре  
    }  
}
```

ПРИМЕР

- Методът приема референция към класа `Instrument`, но и всичко произлизащо от `Instrument`
- В метода `main()` референция към `Wind` се предава към метода `tune()`, без да е необходимо преобразуване
- Допустимо понеже интерфейсът на `Instrument` съществува в `Wind` (поради наследяването)
- Преобразуването нагоре може да „стесни“ този интерфейс, но не по-малък от пълния интерфейс на `Instrument`

```
class Instrument {  
    public void play(Note n) {  
        System.out.println("Instrument.play()");  
    }  
}  
class Wind extends Instrument {  
    public void play(Note n) {  
        System.out.println("Wind.play()");  
    }  
}
```

```
public class Music {  
    public static void tune(Instrument i) {  
        // ...  
        i.play(Note.MIDDLE_C);  
    }  
    public static void main(String[ ] args) {  
        Wind flute = new Wind();  
        tune(flute); // Преобразуване нагоре  
    }  
}
```

ПРЕНЕБРЕГВАНЕ НА ТИПА НА ОБЕКТА

- Защо трябва умишлено да се пренебрегва (забравя) типа?
 - Това става, когато извършваме преобразуване нагоре
- Ще бъде много по-недвусмислено, ако методът `tune()` просто приеме като аргумент референция към `Wind`
 - Това води до съществен момент:
 - Трябва да пишем нови методи `tune()` за всеки нов тип от класа `Instrument`, добавян в нашата система

ПРИМЕР

```
class Note {
    private int value;
    private Note(int val) { value = val; }
    public static final Note
        MIDDLE_C = new Note(0),
        C_SHARP = new Note(1),
        B_FLAT = new Note(2);
}.

class Instrument {
    public void play(Note n) {
        System.out.println("Instrument.play()");
    }
}

class Wind extends Instrument {
    public void play(Note n) {
        System.out.println("Wind.play()");
    }
}

class Stringed extends Instrument {
    public void play(Note n) {
        System.out.println("Stringed.play()");
    }
}
```

```
class Brass extends Instrument {
    public void play(Note n) {
        System.out.println("Brass.play()");
    }
}

public class Music2 {
    public static void tune(Wind i) {
        i.play(Note.MIDDLE_C);
    }
    public static void tune(Stringed i) {
        i.play(Note.MIDDLE_C);
    }
    public static void tune(Brass i) {
        i.play(Note.MIDDLE_C);
    }
    public static void main(String[] args) {
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        Brass frenchHorn = new Brass();
        tune(flute); // без преобразуване нагоре
        tune(violin);
        tune(frenchHorn);
    }
}
```

Wind.play()
Stringed.play()
Brass.play()

Необходимост
от писане на
нови методи
за всеки един
нов клас

ПРОБЛЕМ

- Това работи, но има едно голямо неудобство
 - За всеки клас трябва да пишем специфични за типа методи
- Не е ли по-добре, ако напишем един единствен метод, който приема като аргумент базовия клас, а не някой от неговите производни класове?
 - Т.е., не е ли по-добре просто да забравим, че съществуват производни класове и да пишем кода на програмата само спрямо базовия клас?
- Полиморфизмът ни позволява да направим ТОЧНО ТОВА

ПРИМЕР

? Резултат

```
class Note {  
    private int value;  
    private Note(int val) { value = val; }  
    public static final Note MIDDLE_C = new Note(0),  
                           C_SHARP = new Note(1),  
                           B_FLAT = new Note(2);  
}  
class Instrument {  
    public void play(Note n) {  
        System.out.println("Instrument.play()");  
    }  
}  
class Wind extends Instrument {  
    public void play(Note n) {  
        System.out.println("Wind.play()");  
    }  
}
```

Wind.play()

```
public class Music {  
    public static void tune(Instrument i) {  
        // ...  
        i.play(Note.MIDDLE_C);  
    }  
    public static void main(String[ ] args) {  
        Wind flute = new Wind();  
        tune(flute); // Преобразуване нагоре  
    }  
}
```


ПРОБЛЕМ

```
public static void tune(Instrument i) {  
    // ...  
    i.play(Note.MIDDLE_C);  
}
```

- Методът tune() получава референция към Instrument
- Как компилаторът може да разбере, че тази референция в случая е от тип Wind, а не към Brass или Stringed
 - **Не може!**

За да разберем как функционира това трябва да се запознаем с начина на свързване на обектите (binding)

СВЪРЗВАНЕ

- Свързване: процес на връзка с тялото на методи при тяхното извикване
- Два принципни подхода за реализиране:
 - Предварително (ранно) свързване – извършва се преди стартиране на програмата
 - Късно свързване (late, dynamic, run-time binding) – извършва се по време на изпълнение на програмата

РАННО СВЪРЗВАНЕ

- Ранно свързване (early binding)
 - Извършва се от компилатора или специализирана свързваща програма
 - В процедурните езици този подход няма алтернатива
 - Напр.,
 - Pascal
 - C

КЪСНО СВЪРЗВАНЕ

- Базира се на **типа на обекта**
 - Не на типа на референцията на обекта
- За реализацията е необходим механизъм за определяне типа на обектите и извикване на подходящите методи
 - Т.е. компилаторът не знае типа на обектите
 - Механизмът открива тялото на съответния метод и осъществява връзка с него
 - Различен при различните ОО езици за програмиране
 - Обща идея: по някакъв начин да се съхрани информация за типа на обектите

СВЪРЗВАНЕ В JAVA

- По подразбиране в Java се използва късно свързване
 - Свързването става **автоматично**
 - Програмистът **не трябва** да мисли за него
 - Но, трябва да го **разбира**
- Изключения:
 - final методи
 - static методи

МЕТОДИ В JAVA

- Програмният език на Java предоставя два основни вида методи
 - Методи на обекти (инстанции)
 - Методи на класове (статични)
- Разлики между тези два вида методи:
 - Методите на обекти **ИЗИСКВАТ** инстанция, преди да могат да бъдат извикани
 - Методите на класове **НЕ ИЗИСКВАТ**
 - Методите на обекти използват **късно** (динамично) свързване
 - Методите на класове използват **ранно** (статично) свързване

ИЗВИКВАНЕ НА МЕТОДИ В JAVA

- Когато Java Virtual Machine (JVM) извиква метод на клас, тя избира метода на основата на **типа на референцията** на обекта
 - Известен при компилиране
- Когато Java Virtual Machine (JVM) извиква метод на обект, избира метода на основата на **типа на обекта**
 - Известен само по време на изпълнение
- Следователно – при генерирането си обектите **“ПОМНЯТ”** като какви са създадени

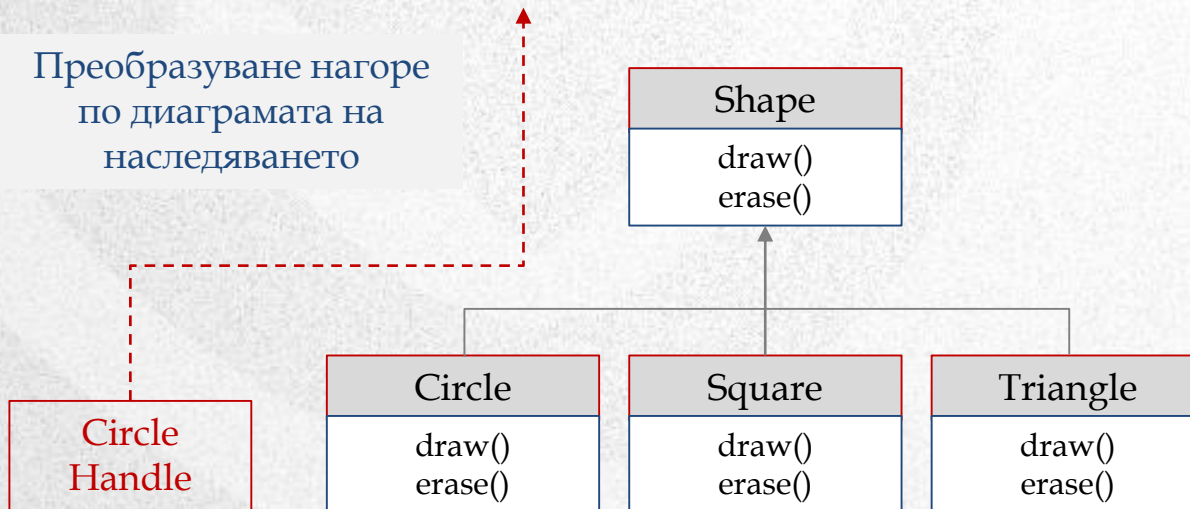
ГЕНЕРИРАНЕ НА КОРЕКТНО ПОВЕДЕНИЕ

- След като вече знаем, че свързване на методи на обекти в Java става полиморфно посредством динамично (късно) свързване, можем да пишем кода си спрямо базовия клас
 - И ще сме сигурни, че производните класове ще работят коректно, използвайки същия код
- Т.е., изпращаме съобщения (извикваме методи) към даден обект и го оставяме **сам** да извърши **правилното** действие

ПРИМЕР

1 Коментар?

Преобразуване нагоре
по диаграмата на
наследяването



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

- Не е грешка!
- Компиляторът приема тази конструкция въпреки различните типове
- Понеже по принципа на наследяването „**Circle е Shape**“

2 Кой метод се извиква в действителност?

По силата на късното свързване (полиморфизъм) се извиква правилния метод **Circle.draw()**!

```
s.draw();
```

Извикване на метод, дефиниран в базовия клас и предефиниран в производните класове

ПРИМЕР

```
class Shape {
    void draw() {}
    void erase() {}
}
class Circle extends Shape {
    void draw() {
        System.out.println("Circle.draw()");
    }
    void erase() {
        System.out.println("Circle.erase()");
    }
}
class Square extends Shape {
    void draw() {
        System.out.println("Square.draw()");
    }
    void erase() {
        System.out.println("Square.erase()");
    }
}
class Triangle extends Shape {
    void draw() {
        System.out.println("Triangle.draw()");
    }
    void erase() {
        System.out.println("Triangle.erase()");
    }
}
```

```
public class Shapes {
    public static Shape randShape() {
        switch((int) (Math.random() * 3)) {
            default: case 0: return new Circle();
            case 1: return new Square();
            case 2: return new Triangle();
        }
    }
    public static void main(String[] args) {
        Shape[] s = new Shape[9];
        // Запълване на масива с различни фигури
        for(int i = 0; i < s.length; i++)
            s[i] = randShape();
        // Полиморфно извикване на методи
        for(int i = 0; i < s.length; i++)
            s[i].draw();
    }
}
```

ПРИМЕР

```
class Shape {
    void draw() {}
    void erase() {}
}
class Circle extends Shape {
    void draw() {
        System.out.println("Circle.draw()");
    }
    void erase() {
        System.out.println("Circle.erase()");
    }
}
class Square extends Shape {
    void draw() {
        System.out.println("Square.draw()");
    }
    void erase() {
        System.out.println("Square.erase()");
    }
}
class Triangle extends Shape {
    void draw() {
        System.out.println("Triangle.draw()");
    }
    void erase() {
        System.out.println("Triangle.erase()");
    }
}
```

- В базовия клас Shape се задава общия интерфейс за всички класове, производни на Shape
- Т.е., всички фигури могат да бъдат изчертавани и изтриване
- Производните класове предефинират тези методи като предоставят уникално поведение за всеки специфичен тип фигура

```
switch((int) (Math.random() * 3)) {
    default: case 0: return new Circle();
    case 1: return new Square();
    case 2: return new Triangle();
}
public static void main(String[] args) {
    Shape[] s = new Shape[9];
    // Запълване на масива с различни фигури
    for(int i = 0; i < s.length; i++)
        s[i] = randShape();
    // Полиморфно извикване на методи
    for(int i = 0; i < s.length; i++)
        s[i].draw();
}
```

ПРИМЕР

```
class Shape {  
    void draw() {}  
    void erase() {}  
}  
class Circle extends Shape {  
    void draw() {  
        System.out.println("Circle.draw()");  
    }  
    void erase() {  
        System.out.println("Circle.erase()");  
    }  
}  
class Square extends Shape {  
    void draw() {  
        System.out.println("Square.draw()");  
    }  
    void erase() {  
        System.out.println("Square.erase()");  
    }  
}
```

```
public class Shapes {  
    public static Shape randShape() {  
        switch((int) (Math.random() * 3)) {  
            default: case 0: return new Circle();  
            case 1: return new Square();  
            case 2: return new Triangle();  
        }  
    }  
    public static void main(String[] args) {  
        Shape[] s = new Shape[9];  
    }  
}
```

- Класът Shapes съдържа static метод randShape()
- При всяко извикване генерира референция към произволно избран обект от класа Shape
- При всяко извикване извършва преобразуване нагоре на референция на някой от типовете Circle, Square, Triangle
 - Т.е., никога не можем да видим специфичния тип – винаги връща референция към Shape

ПРИМЕР

```
class Shape {
    void draw() {}
    void erase() {}
}
class Circle extends Shape {
    void draw() {
        System.out.println("Circle.draw()");
    }
    void erase() {
        System.out.println("Circle.erase()");
    }
}
class Square extends Shape {
    void draw() {
        System.out.println("Square.draw()");
    }
    void erase() {
        System.out.println("Square.erase()");
    }
}
class Triangle extends Shape {
    void draw() {
        System.out.println("Triangle.draw()");
    }
    void erase() {
        System.out.println("Triangle.erase()");
    }
}
```

- Методът `main()` съдържа масив от референции към `Shape`, който се запълва чрез извиквания на метода `randShape()`
- На този етап знаем, че имаме фигури и нищо по-конкретно за тях (както и компилаторът)

```
public class Shapes {
    public static Shape randShape() {
        switch((int) (Math.random() * 3)) {
            default: case 0: return new Circle();
            case 1: return new Square();
            case 2: return new Triangle();
        }
    }
    public static void main(String[] args) {
        Shape[] s = new Shape[9];
        // Запълване на масива с различни фигури
        for(int i = 0; i < s.length; i++)
            s[i] = randShape();
        // Полиморфно извикване на методи
        for(int i = 0; i < s.length; i++)
            s[i].draw();
    }
}
```

ПРИМЕР

- Когато обаче извикаме метода `draw()` за всеки елемент от този масив, се изпълнява съответният за всеки тип фигура метод `draw()`
- Резултатите демонстрират това
- Всички извиквания на `draw()` се реализират чрез **ДИНАМИЧНО СВЪРЗВАНЕ**

```
class Shape {
    void draw() {}
    void erase() {}
}
class Circle extends Shape {
    void draw() {
        System.out.println("Circle.draw()");
    }
    void erase() {
        System.out.println("Circle.erase()");
    }
}
class Square extends Shape {
    void draw() {
        System.out.println("Square.draw()");
    }
    void erase() {
        System.out.println("Square.erase()");
    }
}
class Triangle extends Shape {
    void draw() {
        System.out.println("Triangle.draw()");
    }
    void erase() {
        System.out.println("Triangle.erase()");
    }
}
```

```
public class Shapes {
    public static Shape randShape() {
        switch((int) (Math.random() * 3)) {
            default: case 0: return new Circle();
            case 1: return new Square();
            case 2: return new Triangle();
        }
    }
    public static void main(String[] args) {
        Shape[] s = new Shape[9];
        // Запълване на масива с различни фигури
        for(int i = 0; i < s.length; i++)
            s[i] = randShape();
        // Полиморфно извикване на методи
        for(int i = 0; i < s.length; i++)
            s[i].draw();
    }
}
```

ПРИМЕР

?

Резултат

```
class Shape {  
    void draw() {}  
    void erase() {}  
}  
class Circle extends Shape {  
    void draw() {  
        System.out.println("Circle.draw()");  
    }  
    void erase() {  
        System.out.println("Circle.erase()");  
    }  
}  
class Square extends Shape {  
    void draw() {  
        System.out.println("Square.draw()");  
    }  
    void erase() {  
        System.out.println("Square.erase()");  
    }  
}
```

```
public class Shapes {  
    public static Shape randShape() {  
        switch((int) (Math.random() * 3)) {  
            default: case 0: return new Circle();  
            case 1: return new Square();  
            case 2: return new Triangle();  
        }  
    }  
    public static void main(String[] args) {  
        Shape[] s = new Shape[9];  
        // Запълване на масива с различни фигури  
        for(int i = 0; i < s.length; i++)  
            s[i] = randShape();  
        // Полиморфно извикване на методи  
        for(int i = 0; i < s.length; i++)  
            s[i].draw();  
    }  
}
```

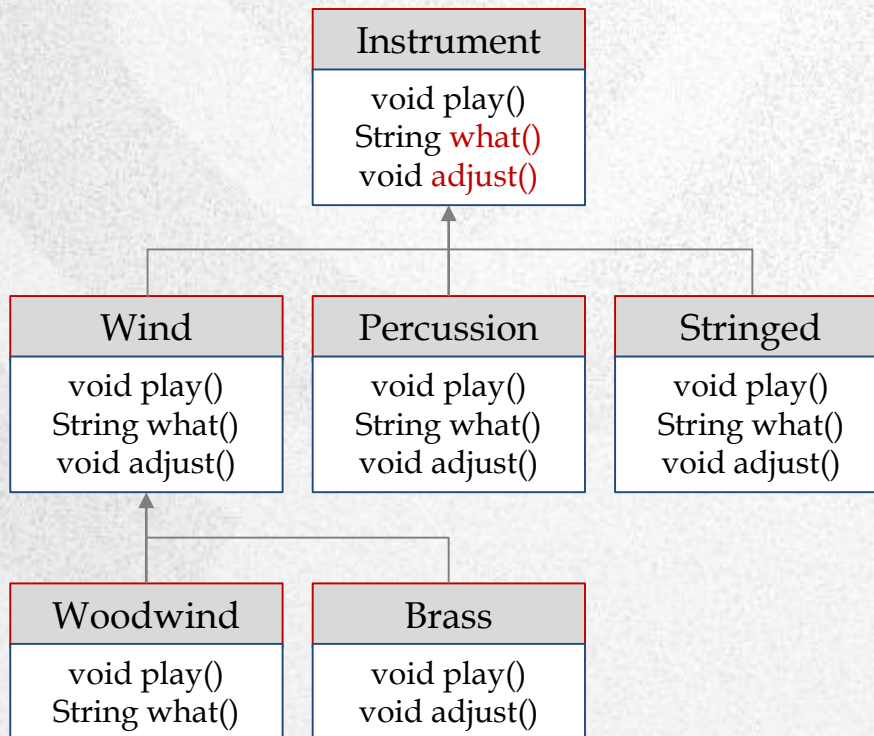
1
Square.draw()
Circle.draw()
Triangle.draw()
Triangle.draw()
Circle.draw()
Circle.draw()
Triangle.draw()
Square.draw()
Triangle.draw()

2
Triangle.draw()
Square.draw()
Square.draw()
Square.draw()
Triangle.draw()
Square.draw()
Square.draw()
Circle.draw()
Circle.draw()

РАЗШИРЯЕМОСТ

- Благодарение на полиморфизма можем да добавяме към системата нови типове, без да променяме метода `tune()`
- В една добре проектирана ОО програма повечето или всички методи ще следват модела на `tune()` и ще комуникират само с интерфейса на базовия клас
 - Такава програма е разширяема – можем да добавяме нова функционалност като наследяваме нови типове данни от общия базов клас
 - Методът, който манипулира интерфейса на базовия клас, не се нуждае от никаква промяна с цел „нагаждане“ към новите класове

РАЗШИРЕН ПРИМЕР



- Към примера с инструменти добавяме нови методи, както и нови класове
- Всички тези нови класове работят правилно със стария, непроменен метод `tune()`

РАЗШИРЕН ПРИМЕР

```
class Instrument {  
    public void play() {  
        System.out.println("Instrument.play()");  
    }  
    public String what() { return "Instrument"; }  
    public void adjust() { }  
}
```

```
class Wind extends Instrument {  
    public void play() {  
        System.out.println("Wind.play()");  
    }  
    public String what() { return "Wind"; }  
    public void adjust() { }  
}
```

```
class Percussion extends Instrument {  
    public void play() {  
        System.out.println("Percussion.play()");  
    }  
    public String what() { return "Percussion"; }  
    public void adjust() { }  
}
```

```
class Stringed extends Instrument {  
    public void play() {  
        System.out.println("Stringed.play()");  
    }  
    public String what() { return "Stringed"; }  
    public void adjust() { }  
}
```

```
class Brass extends Wind {  
    public void play() {  
        System.out.println("Brass.play()");  
    }  
    public void adjust() {  
        System.out.println("Brass.adjust()");  
    }  
}
```

```
class Woodwind extends Wind {  
    public void play() {  
        System.out.println("Woodwind.play()");  
    }  
    public String what() { return "Woodwind"; }  
}
```


РАЗШИРЕН ПРИМЕР

```
class Instrument {  
    public void play() {  
        System.out.println("Instrument.play()");  
    }  
    public String what() { return "Instrument"; }  
    public void adjust() {}  
}
```

```
class Wind extends Instrument {  
    public void play() {  
        System.out.println("Wind.play()");  
    }  
    public String what() { return "Wind"; }  
    public void adjust() {}  
}
```

```
class Percussion extends Instrument {  
    public void play() {  
        System.out.println("Percussion.play()");  
    }  
    public String what() { return "Percussion"; }  
    public void adjust() {}  
}
```

```
class Stringed extends Instrument {  
    public void play() {  
        System.out.println("Stringed.play()");  
    }  
    • Новите методи са:
```

- what(), който връща референция към String с описание на класа
- adjust(), който предоставя начин за настройка на всеки един инструмент

```
    public void adjust() {  
        System.out.println("Stringed.adjust()");  
    }  
}
```

```
class Woodwind extends Wind {  
    public void play() {  
        System.out.println("Woodwind.play()");  
    }  
    public String what() { return "Woodwind"; }  
}
```

РАЗШИРЕН ПРИМЕР

```
public class Music3 {  
    // Без зависимост от типа, така че нови типове, добавени към  
    // системата, работят правилно  
    static void tune(Instrument i) {  
        // ...  
        i.play();  
    }  
    public static void tuneAll(Instrument[] e) {  
        for(int i = 0; i < e.length; i++)  
            tune(e[i]);  
    }  
    public static void main(String[] args) {  
        Instrument[] orchestra = new Instrument[5];  
        int i = 0;  
        // Преобразуване нагоре по време на добавяне в масива:  
        orchestra[i++] = new Wind();  
        orchestra[i++] = new Percussion();  
        orchestra[i++] = new Stringed();  
        orchestra[i++] = new Brass();  
        orchestra[i++] = new Woodwind();  
        tuneAll(orchestra);  
    }  
}
```

РАЗШИРЕН ПРИМЕР

```
public class Music3 {  
    // Без зависимост от типа, така че нови типове, добавени към  
    // системата, работят правилно  
    static void tune(Instrument i) {  
        // ...  
        i.play();  
    }  
    public static void tuneAll(Instrument[] e) {  
        for(int i = 0; i < e.length; i++)  
            tune(e[i]);  
    }  
    public static void main(String[] args) {  
        Instrument[] orchestra = new Instrument[5];  
        int i = 0;  
        // Преобразуване нагоре по време на доба  
        orchestra[i++] = new Wind();  
        orchestra[i++] = new Percussion();  
        orchestra[i++] = new Stringed();  
        orchestra[i++] = new Brass();  
        orchestra[i++] = new Woodwind();  
        tuneAll(orchestra);  
    }  
}
```

- Методът `tune()` е напълно независим от всички промени в кода и продължава да работи коректно
- Точно това трябва да доставя полиморфизма
- Промяната в кода не влияе върху части от програмата, които не трябва да бъдат засягани

РАЗШИРЕН ПРИМЕР

```
public class Music3 {  
    // Без зависимост от типа, така че нови типове, добавени към  
    // системата, работят правилно  
    static void tune(Instrument i) {  
        // ...  
        i.play();  
    }  
    public static void tuneAll(Instrument[] e) {  
        for(int i = 0; i < e.length; i++)  
            tune(e[i]);  
    }  
    public static void main(String[] args) {  
        Instrument[] orchestra = new Instrument[5];  
        int i = 0;  
        // Преобразуване нагоре по време на доба  
        orchestra[i++] = new Wind();  
        orchestra[i++] = new Percussion();  
        orchestra[i++] = new Stringed();  
        orchestra[i++] = new Brass();  
        orchestra[i++] = new Woodwind();  
        tuneAll(orchestra);  
    }  
}
```

- Методът `tune()` е напълно независим от всички промени в кода и продължава да работи коректно
- Точно това трябва да доставя полиморфизма
- Промяната в кода не влияе върху части от програмата, които не трябва да бъдат засягати

ОБОБЩЕНИЕ

Полиморфизмът е една от най-важните техники, които позволяват на програмиста да отделя нещата, които трябва да се променят, от тези, които трябва да останат същите (непроменяеми)

КОНСТРУКТОРИ & ПОЛИМОРФИЗЪМ

- В предишни лекции също са разглеждани конструктори
 - Тук във връзка с полиморфизма
- Въпреки, че конструкторите **не са полиморфни**, съществено е да се разбере начина, по който конструкторите работят в сложни йерархии с използване на полиморфизъм

КОНСТРУКТОРИ & ПОЛИМОРФИЗЪМ

- Конструктор на базов клас се извиква винаги от конструктор на производен клас, променяйки йерархията на наследяване така, че се извиква конструктор на всеки базов клас
 - Има смисъл, понеже конструкторът има специално предназначение – да установи дали обектът е построен коректно
 - Само конструкторите имат достъп до собствените си член-обекти
 - По тази причина е съществено тяхното извикване при инициализация
 - В противен случай обектът не може да бъде инициализиран напълно
 - По тази причина компилаторът извиква конструктор за всяка част на производния клас

ПРИМЕР

```
class Meal {  
    Meal() { System.out.println("Meal()"); }  
}  
class Bread {  
    Bread() { System.out.println("Bread()"); }  
}  
class Cheese {  
    Cheese() { System.out.println("Cheese()"); }  
}  
class Lettuce {  
    Lettuce() { System.out.println("Lettuce()"); }  
}  
class Lunch extends Meal {  
    Lunch() { System.out.println("Lunch()"); }  
}  
class PortableLunch extends Lunch {  
    PortableLunch() {  
        System.out.println("PortableLunch()");  
    }  
}
```

```
class Sandwich extends PortableLunch {  
    Bread b = new Bread();  
    Cheese c = new Cheese();  
    Lettuce l = new Lettuce();  
    Sandwich() {  
        System.out.println("Sandwich()");  
    }  
  
    public static void main(String[] args) {  
        new Sandwich();  
    }  
}
```

ПРИМЕР

```
class Meal {  
    Meal() { System.out.println("Meal()"); }  
}  
class Bread {  
    Bread() { System.out.println("Bread()"); }  
}  
class Cheese {  
    Cheese() { System.out.println("Cheese()"); }  
}  
class Lettuce {  
    Lettuce() { System.out.println("Lettuce()"); }  
}  
class Lunch extends Meal {  
    Lunch() { System.out.println("Lunch()"); }  
}  
class PortableLunch extends Lunch {  
    PortableLunch() {  
        System.out.println("PortableLunch()");  
    }  
}
```

?

Нива на наследяване

4

?

Член-обекта

3

```
class Sandwich extends PortableLunch {  
    Bread b = new Bread();  
    Cheese c = new Cheese();  
    Lettuce l = new Lettuce();  
    Sandwich() {  
        System.out.println("Sandwich()");  
    }  
  
    public static void main(String[] args) {  
        new Sandwich();  
    }  
}
```


ПРИМЕР

?

Результат

```
Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Sandwitch()
```

```
class Meal {
    Meal() { System.out.println("Meal()"); }
}
class Bread {
    Bread() { System.out.println("Bread()"); }
}
class Cheese {
    Cheese() { System.out.println("Cheese()"); }
}
class Lettuce {
    Lettuce() { System.out.println("Lettuce()"); }
}
class Lunch extends Meal {
    Lunch() { System.out.println("Lunch()"); }
}
class PortableLunch extends Lunch {
    PortableLunch() {
        System.out.println("PortableLunch()");
    }
}
```

```
class Sandwich extends PortableLunch {
    Bread b = new Bread();
    Cheese c = new Cheese();
    Lettuce l = new Lettuce();
    Sandwich() {
        System.out.println("Sandwitch()");
    }

    public static void main(String[] args) {
        new Sandwich();
    }
}
```

ИЗВИКВАНЕ НА КОНСТРУКТОРИ

- Редът на извикване на конструктори за един сложен обект е следният:
 - Рекурсивно извикване на конструктора на базовия клас
 - Първо се конструира коренът на йерархичното дърво
 - Следван от първото ниво на производните класове
 - Така, докато се стигне до класовете от най-ниското ниво на йерархията
 - Членовете се инициализират по реда на декларирането им
 - Изпълнява се останалият код на конструктора на производния клас

БЛАГОДАРЯ ЗА ВНИМАНИЕТО!

КРАЙ “ПОЛИМОРФИЗЪМ”

