ПОЛИМОРФИЗЪМ

ЛЕКЦИОНЕН КУРС "ООП (JAVA)"

Distributed @learning center

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 същият 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, ще изпълни правилния код в зависимост от неговия специфичен тип
- Съществува обаче един проблем при разглеждане на обекти от производен тип като обекти от техния базов тип
  - Компилаторът не може да знае по време на компилация точно кой код ще се изпълни
- Отговорът на този проблем е основна особеност на ООП
  - Компилаторът не може да извиква функции по традиционния начин

РАННО И КЪСНО СВЪРЗВАНЕ

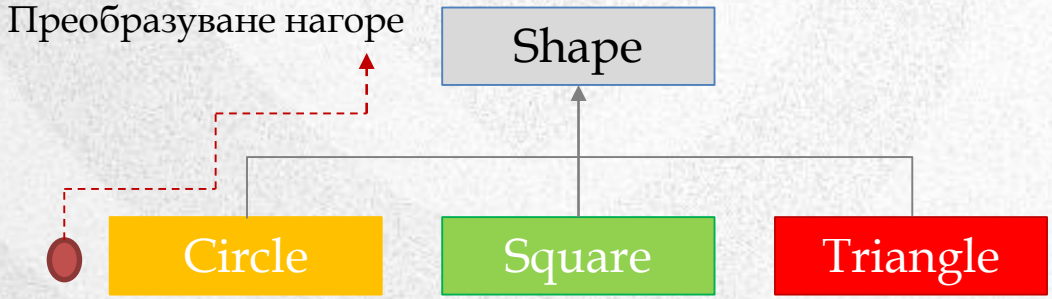
- При императивните езици за програмиране се използва ранно свързване
  - Компилаторът генерира обръщение към определено име на функция
  - Програмата за свързване преобразува това обръщение в абсолютен адрес на кода, който трябва да се изпълни
- Обектно-ориентираните езици за програмиране използват 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
  - По подразбиране в тези езици е функциите да не се свързват динамично
- В Java динамичното (късно) свързване е по подразбиране
  - Не е необходимо да помним, че трябва да добавяме някакви ключови думи за да получим полиморфизъм

ПРИМЕР

Изпълнява се правилният код на draw() поради полиморфизма




```
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 MID
                                  C_S
                                  B_F
}
class Instrument {
   public void play(Note n) {
      System.out.println("Instru
   }
}
class Wind extends Instrume
  public void play(Note n) {
     System.out.println("Wind.
  }
}
public class Music {
1  public static void tune(Instrument i) {
      // ...
      i.play(Note.MIDDLE_C);
   }
   public static void main(String[ ] args) {
      Wind flute = new Wind();
1      tune(flute); // Преобразуване нагоре
   }
}
```

- 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

ПРИМЕР

1 Какъв 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()");
  }
}
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
   }
}
```

Wind.play()

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 {
1   public static void tune(Wind i) {
      i.play(Note.MIDDLE_C);
    }
1   public static void tune(Stringed i) {
      i.play(Note.MIDDLE_C);
    }
1   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);
    }

}
```

Необходимост от писане на нови методи за всеки един нов клас

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

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

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 сочи към 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
  - Извършва се от компилатора или специализирана свързваща програма
  - В процедурните езици този подход няма алтернатива (напр., 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
- В Java се използва само късно свързване
  - Изключение final методи
  - Късното свързване става автоматично (програмистът не трябва да мисли за него)

ПРИНЦИПНА РЕАЛИЗАЦИЯ НА КС

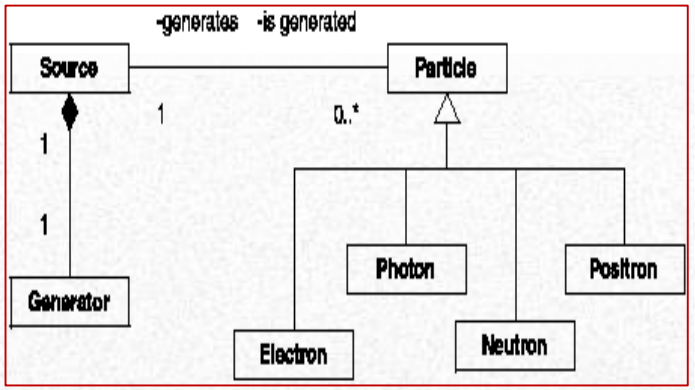
- Реализация на късното свързване, т.е. определяне на актуалния тип на обекта, за който ще се извика методът
- Използват се така наречените V*irtual Method Tables* (VMT)
  - Скрити елементи на полиморфните класове (не се декларират от програмиста, а се добавят автоматично от компилатора)
  - Всеки полиморфен клас съдържа точно една VMT
  - Една VMT е масив, съдържащ указатели към всички виртуални методи на класа
  - Всеки изведен клас има собствена VMT, съдържаща указатели към всички виртуални методи (дори на тези, които не са пренаписани в този клас)
  - Указателите във VMT на изведения клас са в същия ред като указателите към кореспондиращите методи в базовия клас
- Всеки обект от полиморфен клас съдържа още един скрит елемент – указател към VMT
- Този елемент се съхранява на едно и също място във всички обекти
  - В началото

ИЗВИКВАНЕ НА МЕТОДИ ПРИ КС

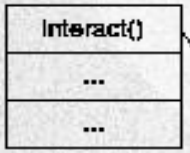
Извикване на методи се извършва по следния начин:

- Програмата взема инстанцията за която е извикан метода
- В инстанцията тя намира указателя към VMT
- Във VMT тя намира указателя към метода, който ще се извика. Във всички VMTs този указател се намира на една и съща позиция
  - Напр., указателят към Interact() метода е на първо място във VMT на Particle клас и във VMTs на всички класове, изведени от Particle
- Програмата използва този указател за извикване на метода на актуалната инстанция на актуалния клас

СХЕМА НА РЕАЛИЗАЦИЯ НА КС

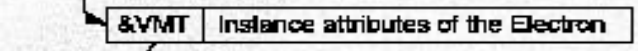


The Particle VMT

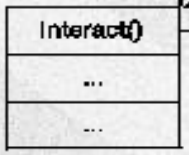



```
void
Particle::Interact(...)
{ // Code of the method
  // in the Particle class
}
```

```
Particle *pp1 = new Electron;
```

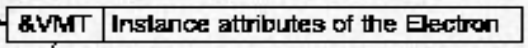


The Electron VMT

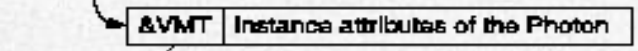



```
void
Electron::Interact(...)
{ // Code of the method
  // in the Electron class
}
```

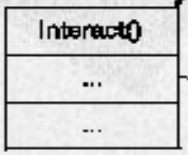
```
Particle *pp2 = new Electron;
```




```
pp2 = new Photon;
```



The Photon VMT




```
void Photon::Interact(...)
{ // Code of the method
  // in the Photon class
}
```

ГЕНЕРИРАНЕ НА КОРЕКТНО ПОВЕДЕНИЕ

- След като вече знаем, че всяко свързване на методи 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 данните към даден обект и го оставяме сам да извърши правилното действие

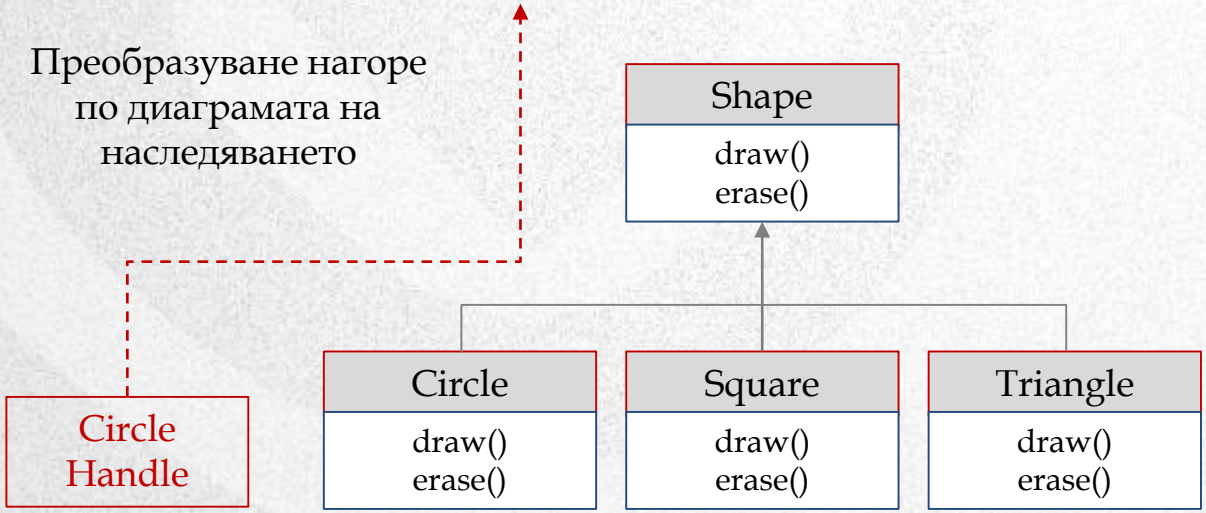
ПРИМЕР

1 Коментар?

Shape s = new Circle();

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

Преобразуване нагоре по диаграмата на наследяването



2 Кой метод се извиква в действителност?

s.draw();

По силата на късното свързване (полиморфизъм) се извиква правилния метод Circle.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 RandomShapeGenerator {
  private Random rand = new Random();
  public Shape randShape() {
    switch(rand.nextInt(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

- 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

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.o
  }
}
```

```
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();
    }
  }
```

- 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.o
  }
  void erase()
      System.o
  }
}
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()");
                                          ();
        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();
   }
}
```

- 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)

ПРИМЕР

```
class Shape {
  void draw() { }
  void erase() { }
}
class Circle extends Shape {
  void draw() {
      System.o
  }
  void erase()
      System.o
  }
}
class Square
  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 static void main(String
     Shape[] s = new Shape[9];
     // Запълване на масива с р
     for(int i = 0; i < s.length; i+
        s[i] = randShape();
     // Полиморфно извикван
     for(int i = 0; i < s.length; i+
        s[i].draw();
  }
}
```

- 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

1

```
Square.draw()
Circle.draw()
Triangle.draw()
Triangle.draw()
Circle.draw()
Circle.draw()
Triangle.draw()
Square.draw()
Triangle.draw()
```

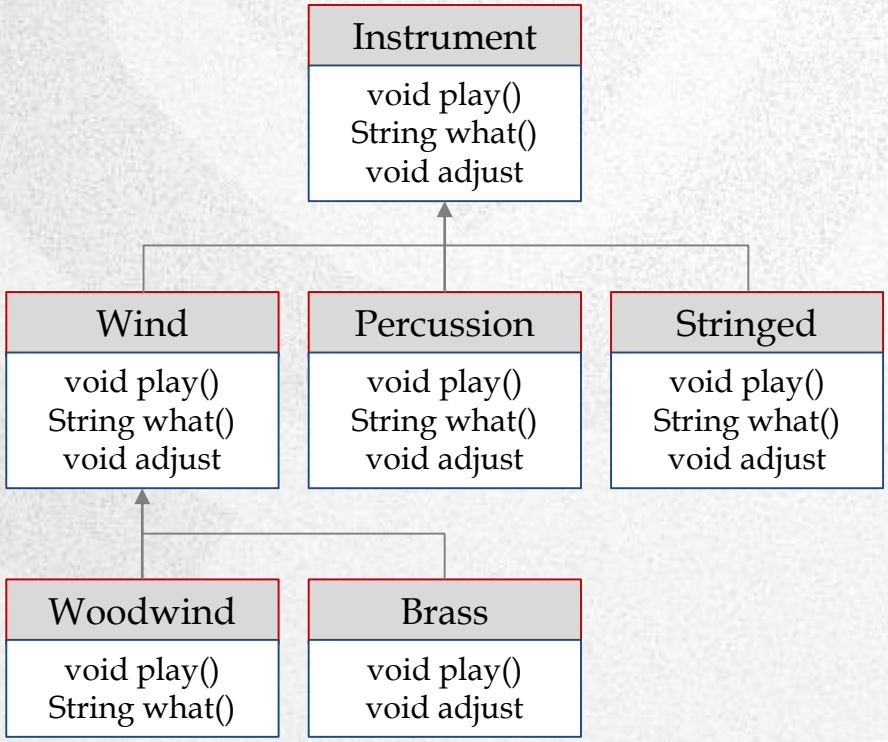
2

```
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

ПРИМЕР



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

ПРИМЕР

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

class Wind ext
  public void p
    System.ou
  }
  public String
  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() { }
```

```
    System.out.println("Brass.adjust()");
  }
}

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

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

ПРИМЕР

```
public class Music3 {
// Без зависимост от типа, така че
// нови типове, добавени към системата, работят правилно

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

- В метода main(), при вмъкване на елемент в масива Instrument, автоматично се извършва преобразуване нагоре до класа Instrument

```
    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);
    }
}
```

2

ПРИМЕР

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

```
        i.play();
    }
```

- 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

```
        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);
    }
}
```

2

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
- 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 членове – 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

1. Колко нива на наследяване? 4
2. Колко член-обекта? 3

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

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

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

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

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
  - Извиква се конструкторът на производния клас

БЛАГОДАРЯ ЗА ВНИМАНИЕТО!

КРАЙ “ПОЛИМОРФИЗЪМ”

Distributed @learning center