МНОГОКРАТНО ИЗПОЛЗВАНЕ

ЛЕКЦИОНЕН КУРС "ООП(JAVA)"

проф. д-р Станимир Стоянов

СТРУКТУРА НА ЛЕКЦИЯТА

- Въведение
- Синтаксис на композиция
- Синтаксис на наследяване
- Комбинирано използване
- Преобразуване нагоре
- Ключова дума final

ВЪВЕДЕНИЕ

- Една от най-съществените характеристики на Java е многократното използване
  - Предоставя значително повече възможности от копиране и промяна на кода
- Решението на този проблем (както всичко друго в Java) е свързано с класовете
  - Многократното използване на код като създаване на нови класове
  - Вместо да ги създаваме от самото начало можем да използваме съществуващи класове
  - Грешките в тях са вече отстранени
- Цел: използване на класове без добавяне излишни неща в съществуващия код

МНОГОКРАТНО ИЗПОЛЗВАНЕ НА ИМПЛЕМЕНТАЦИЯТА

- Многократното използване на код е едно от най-големите предимства на ООП
- Най-прост начин за многократно използване
  - Създаване и използване на обект от един клас
  - Създаване на член-обект - поставяне на обект от съществуващ в нов клас

МНОГОКРАТНО ИЗПОЛЗВАНЕ НА ИМПЛЕМЕНТАЦИЯТА

- Композиция: новият клас може да бъде съставен от произволен брой и тип други обекти в произволна комбинация, необходима за постигане на желаната функционалност
  - Притежава голяма гъвкавост
  - Член-обектите обикновено са недостъпни за клиентите (private)
    - Така можем да променяме тези обекти без да влияем върху кода на клиентите
  - Освен това, можем да променяме член-обектите по време на изпълнение за промяна на динамичното поведение на програмите
- Наследяването не притежава тази гъвкавост, понеже компилаторът поставя ограничения по време на компилиране за класовете, създадени чрез наследяване
  - Понеже на наследяването се отделя голямо внимание, може да се помисли, че то може да се използва навсякъде
    - Понякога не е целесъобразно – може да доведе до трудни и прекалено сложни проекти
  - Вместо това, първо може да се разглежда композицията при създаване на нови класове, тъй като е по-просто и по-гъвкаво

ПОДХОДИ ЗА МНОГОКРАТНО ИЗПОЛЗВАНЕ

- Два подхода:
  - Композиция – използване обекти от съществуващи класове в новия клас
    - Използва се многократно функционалността на съществуващия код (не неговата форма)
  - Наследяване – новият клас се създава като тип на съществуващ клас
    - Използва се формата на съществуващ клас, като добавяме към нея код, без да променяме съществуващия клас
    - Компилаторът извършва по-голямата част от работата
    - Наследяването е базова концепция на ООП
- Синтаксисът и поведението на композицията и наследяването са сходни
  - Те представляват начини за създаване на нови типове от съществуващи

СИНТАКСИС НА КОМПОЗИЦИЯ

- В много ОО системи се използва композиция
- Тя се реализира просто като поставим референции на обекти в рамките на нови класове
- Така, новите класове могат да съдържат:
  - Обекти от съществуващи класове – посредством референции
  - Примитиви – дефинират се директно

ПРИМЕР

```
class WaterSource {
  private String s;
  WaterSource () {
    System.out.println("WaterSource ()");
    s = new String("Constructed");
  }
   public String toString () { return s; }
}

public class SprinklerSystem {
  private String valve1, valve2, valve3, valve4;
  WaterSource source;
  int i;
  float f;
  void print () {
    System.out.println("valve1 = " + valve1);
    System.out.println("valve2 = " + valve2);
    System.out.println("valve3 = " + valve3);
    System.out.println("valve4 = " + valve4);
    System.out.println("i = " + i);
    System.out.println("f = " + f);
    System.out.println("source = " + source);
  }
  public static void main(String[ ] args) {
    SprinklerSystem x = new SprinklerSystem ();
    x.print ();
  }
}
```

```
valve1 = null
valve2 = null
valve3 = null
valve4 = null
i = 0
f = 0.0
source = null
```

1. Какъв резултат?
2. Коментар?

Компилаторът не извиква автоматично конструктора по подразбиране на WaterSource за да извърши инициализацията на source

ПРИМЕР

```
class WaterSource {
  private String s;
  WaterSource () {
    System.out.println("WaterSource ()");
    s = new String("Constructed");
   }
   public String toString () { return s; }
}

public class SprinklerSystem {
  private String valve1, valve2, valve3, valve4;
  WaterSource source;
  int i;
  float f;
  void print () {
    System.out.println("valve1 = " + valve1);
    System.out.println("valve2 = " + valve2);
    System.out.println("valve3 = " + valve3);
    System.out.println("valve4 = " + valve4);
    System.out.println("i = " + i);
    System.out.println("f = " + f);
    System.out.println("source = " + source);
  }
  public static void main(String[ ] args) {
    SprinklerSystem x = new SprinklerSystem ();
    x.print ();
  }
}
```

3 Друга особеност?

toString(): специален метод

- Всеки непримитивен обект притежава този метод
- Извиква се в специални случаи
- Тук, компилаторът разбира, че се опитваме да добавим обект от клас String към обект от клас WaterSource – не вижда смисъл в това, понеже към обект от клас String може да се добави само обект от същия клас
- По тази причина компилаторът първо го конвертира в String обект, като извика toString() метода, и след това комбинира двата низа

ИНИЦИАЛИЗАЦИЯ НА РЕФЕРЕНЦИИ

- Примитивите, които са полета в клас се инициализират автоматично с 0
- Референциите към обектите се инициализират с null
  - Ако се опитваме да извикаме методи за някой от тях ще възникне изключение
- Компилаторът не създава обект по подразбиране за референциите
- Инициализацията на референциите може да се направи явно по следните начини:
  - В момента на дефиниране на обектите
    - Винаги ще бъдат инициализирани преди извикване на конструктора
  - В конструктора на класа
  - Непосредствено преди използване на обекта
    - Съществуват ситуации, при които обектите не трябва да бъдат задължително създавани

ПРИМЕР

1 Три възможности за инициализиране в примера?

```
class Soap {
  private String s;
  Soap () {
    System.out.println("Soap ()");
    s = new String ("Constructed");
   }
  public String toString () { return s; }
}

public class Bath {  private String
  s1 = new String("Happy"),
  s2 = "Happy",
  s3, s4;
  Soap castille;
  int i;
  float toy;
  Bath () {
   System.out.println("Inside Bath ()");
   s3 = new String ("Joy");
   i = 47;
   toy = 3.14f;
   castille = new Soap ();
  }
```

Инациализация в момента на дефиниране

В конструктора

```
  void print () {
    // Късна инициализация:
    if (s4 == null)
      s4 = new String ("Joy");
    System.out.println ("s1 = " + s1);
    System.out.println ("s2 = " + s2);
    System.out.println ("s3 = " + s3);
    System.out.println ("s4 = " + s4);
    System.out.println ("i = " + i);
    System.out.println ("toy = " + toy);
                                  = " + castille);

  public static void main(String[ ] args) {
    Bath b = new Bath ();
    b.print ();
  }

}
```

Късна инициализация

ПРИМЕР

1 Какъв резултат?

```
class Soap {
  private String s;
  Soap () {
    System.out.println("Soap ()");
    s = new String ("Constructed");
   }
  public String toString () { return s; }
}

public class Bath {
  private String s1 = new String("Happy"),
                 s2 = "Happy",
                 s3, s4;
  Soap castille;
  int i;
  float toy;
  Bath () {
   System.out.println("Inside Bath ()");
   s3 = new String ("Joy");
   i = 47;
   toy = 3.14f;
   castille = new Soap ();
  }
  void print () {
    if (s4 == null)
       s4 = new String ("Joy");
    System.out.println ("s1 = " + s1);
    System.out.println ("s2 = " + s2);
    System.out.println ("s3 = " + s3);
    System.out.println ("s4 = " + s4);
    System.out.println ("i = " + i);
    System.out.println ("toy = " + toy);
    System.out.println ("castille = " + castille);
  }

  public static void main(String[ ] args) {
    Bath b = new Bath ();
    b.print ();
  }

}
```

```
Inside Bath ()
Soap ()
s1 = Happy
s2 = Happy
s3 = Joy
s4 = Joy
i = 47
toy = 3.14
castille = Constructed
```

ПРИМЕР

1 Друга особеност?

```
class Soap {
  private String s;
  Soap () {
    System.out.println("Soap ()");
    s = new String ("Constructed");
   }
  public String toString () { return s; }
}

public class Bath {
  private String  s1 = new String("Happy"),
                  s2 = "Happy",
                  s3, s4;
  Soap castille;
  int i;
  float toy;
  Bath () {
   System.out.println("Inside Bath ()");
   s3 = new String ("Joy");
   i = 47;
   toy = 3.14f;
   castille = new Soap ();
  }
```

```
  void print () {
    // Късна инициализация:
    if (s4 == null)
      s4 = new String ("Joy");
    System.out.println ("s1 = " + s1);
    System.out.println ("s2 = " + s2);
    System.out.println ("s3 = " + s3);
    System.out.println ("s4 = " + s4);
    System.out.println ("i = " + i);
    System.out.println ("toy = " + toy);
    System.out.println ("castille = " + castille);
  }

  public static void main(String[ ] args) {
    Bath b = new Bath ();
    b.print ();
  }
```

- Конструкторът се изпълнява преди да се извърши някоя от инициализациите.
- Ако не извършваме инициализация по време на дефиниция, все още нямаме гаранция, че няма да се изпрати съобщение към обект преди да е инициализиран.

СИНТАКСИС НА НАСЛЕДЯВАНЕ

- Наследяването е неотменна част от Java (и всички ОО езици изобщо)
- По принцип винаги извършваме наследяване, когато създаваме клас, дори явно да не наследяваме друг клас
  - Безусловно наследяваме стандартния базов клас на Java, наречен Object
- Синтаксисът на наследяването използва съвсем различна форма от композицията
  - Идеята е да покажем, че „Един нов клас е като някой съществуващ клас“
  - Ключова дума extends
  - Новият клас автоматично получава всички данни и методи на базовия клас

ПРИМЕР

1. Коментар? Примерът демонстрира различни свойства на наследяването
2. Резултат?

```
class Cleanser {
  private String s = new String("Cleanser");
  public void append(String a) { s += a; }
  public void dilute() { append(" dilute()"); }
  public void apply() { append(" apply()"); }
  public void scrub() { append(" scrub()"); }
  public void print() { System.out.println(s); }

  public static void main(String[] args) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    x.print();
  }

}
```

Cleanser dilute() apply() scrub()

```
public class Detergent extends Cleanser {
 //Промяна на метод
 public void scrub() {
    append(" Detergent.scrub()");
    super.scrub(); // Извикване версията на базовия клас
 }
 // Добавяне методи към интерфейса
 public void foam() { append(" foam()"); }
 // Тестване на новия клас

 public static void main(String[ ] args) {
    Detergent x = new Detergent();
    x.dilute();
    x.apply();
    x.scrub();
    x.foam();
    x.print();
    System.out.println("Testing base class:");
    Cleanser.main(args);
 }
```

Cleanser dilute() apply() Detegent.scrub() scrub() foam()
Testing base class:
Cleanser dilute() apply() scrub()

ПРИМЕР

```
class Cleanser {
  private String s = new String("Cleanser");
  public void append(String a) { s += a; }
  public void scrub() { append(" scrub()"); }
  public void print() { System.out.println(s); }

  public static void main(String[] args) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    x.print();
  }

}
```

1 Метод **append**: Слепване на низове с помощта на оператора „+=“

```
public class Detergent extends Cleanser {
  //Промяна на метод
  public void scrub() {
    ията на базовия клас
  }
  // Добавяне методи към интерфейса
  public void foam() { append(" foam()"); }
  // Тестване на новия клас

  public static void main(String[ ] args) {
    Detergent x = new Detergent();
    x.dilute();
    x.apply();
    x.scrub();
    x.foam();
    x.print();
    System.out.println("Testing base class:");
    Cleanser.main(args);
  }

}
```

ПРИМЕР

- Двата класа съдържат **main** методи – могат независимо един от друг да бъдат извикани от командния ред
- Може да се задават за всеки клас
- Такъв стил се препоръчва – тестовият код е обвит от клас
- (докато main методът е public, няма значение дали неговият клас е public)
- Когато приключим с тестването, не е необходимо да изтриваме main – можем да го оставим за бъдещо тестване

```
cl
1
  public void dilute() { append(" dilute()"); }
  public void apply() { append(" apply()"); }
  public void scrub() { append(" scrub()"); }
  public void print() { System.out.println(s); }

2 public static void main(String[] args) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    x.print();
  }

}
```

```
    append(" Detegent.scrub()");
    super.scrub(); // Извикване версията на базовия клас
  }
  // Добавяне методи към интерфейса
  public void foam() { append(" foam()"); }
  // Тестване на новия клас

2 public static void main(String[ ] args) {
    Detergent x = new Detergent();
    x.dilute();
    x.apply();
    x.scrub();
    x.foam();
    x.print();
    System.out.println("Testing base class:");
    Cleanser.main(args);
  }

}
```

ПРИМЕР

- Detergent.main явно извиква Cleanser.main
- Препредава му същите аргументи от околната среда (командния ред)
- Възможно е също да се предава произволен масив от низове

```
class Cleanser {
  private String s = new String("Cleanser");
1 public void append(String a) { s += a; }
  public void dilute() { append(" dilute()"); }
  public void apply() { append(" apply()"); }
  public void scrub() { append(" scrub()"); }
  public void print() { System.out.println(s); }

2 public static void main(String[] args) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    x.print();
  }

}
```

```
public class Detergent extends Cleanser {
  //Промяна на метод
  public void scrub() {
    append(" Detegent.scrub()");
    super.scrub(); // Извикване версията на базовия клас
  }
  // Добавяне методи към интерфейса
  public void foam() { append(" foam()"); }
  // Тестване на новия клас

2 public static void main(String[ ] args) {
    Detergent x = new Detergent();
    x.dilute();
    x.apply();
    x.scrub();
    x.foam();
    x.print();
    System.out.println("Testing base class:");
3   Cleanser.main(args);
  }

}
```

ПРИМЕР

- Особено важно: всички методи в Cleanser са public
- Ако клас от друг пакет е наследник на Cleanser, той има право на достъп само до public членовете
- Detergent няма проблеми

```
class Cleanser {
  private String s = new String("Cleanser");
1 public void append(String a) { s += a; }
  public void dilute() { append(" dilute()"); }
4 public void apply() { append(" apply()"); }
  public void scrub() { append(" scrub()"); }
  public void print() { System.out.println(s); }

2 public static void main(String[] args) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    x.print();
  }

}
```

```
public class Detergent extends Cleanser {
 //Промяна на метод
 public void scrub() {
    append(" Detegent.scrub()");
    super.scrub(); // Извикване версията на базовия клас
 }
 // Добавяне методи към интерфейса
 public void foam() { append(" foam()"); }
 // Тестване на новия клас

2 public static void main(String[ ] args) {
    Detergent x = new Detergent();
    x.dilute();
                                    esting base class:");
3   Cleanser.main(args);
  }

}
```

Основно правило при наследяване:

- Всички полета private
- Всички методи public

ПРИМЕР

- Тези методи са от интерфейса на Cleanser
- Detergent (понеже е производен на Cleanser) автоматично получава тези методи в своя интерфейс (дори и да не са явно дефинирани в Detergent)

```
class Cleanser {
  private String s = new String("Cleanser");
1 public void append(String a) { s += a; }
  public void dilute() { append(" dilute()"); }
4 public void apply() { append(" apply()"); }
  public void scrub() { append(" scrub()"); }
  public void print() { System.out.println(s); }

2 public static void main(String[] args) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    x.print();
  }

}
```

```
public class Detergent extends Cleanser {
 //Промяна на метод
 public void scrub() {
    append(" Detegent.scrub()");
    super.scrub(); // Извикване версията на базовия клас
 }
 // Добавяне методи към интерфейса
 public void foam() { append(" foam()"); }
 // Тестване на новия клас

2 public static void main(String[ ] args) {
    Detergent x = new Detergent();
    x.foam();
    x.print();
    System.out.println("Testing base class:");
3   Cleanser.main(args);
  }

}
```

Следователно, можем да мислим за наследяването като за повторно използване на интерфейса

ПРИМЕР

1 Защо? **Ще предизвика нежелана рекурсия**

- Възможно е да променим метод, използван в базовия клас (подобно на scrub())
- В този случай може да искаме да извикаме метода на базовия клас в рамките на новата версия на метода
- В scrub() не можем просто да извикаме scrub()

1 Решение?

Използваме ключовата дума super, която се отнася за „супер-клас“, от който е наследен дадения клас

```
class Cleanser {
  private String s = new String("Cleanser");
  public void dilute() { append(" dilute()"); }
  public void apply() { append(" apply()"); }
  public void scrub() { append(" scrub()"); }
  public void print() { System.out.println(s); }

2 public static void main(String[] args) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    x.print();
  }

}
```

```
public class Detergent extends Cleanser {
  //Промяна на метод
6 public void scrub() {
    append(" Detegent.scrub()");
    super.scrub(); // Извикване версията на базовия клас
  }

2 public static void main(String[ ] args) {
    Detergent x = new Detergent();
    x.dilute();
    x.apply();
    x.scrub();
    x.foam();
    x.print();
    System.out.println("Testing base class:");
3   Cleanser.main(args);
  }

}
```

ПРИМЕР

- Когато наследяваме не сме ограничени да използваме само методите на базовия клас
- Можем да добавяме нови методи, напр. foam()

```
class Cleanser {
  private String s = new String("Cleanser");
1 public void append(String a) { s += a; }
  public void dilute() { append(" dilute()"); }
4 public v 5 apply() { append(" apply()"); }
  public void scrub() { append(" scrub()"); }
  public void print() { System.out.println(s); }

2 public static void main(String[] args) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    x.print();
  }

}
```

```
public class Detergent extends Cleanser {
  //Промяна на метод
6 public void scrub() {
    append(" Detegent.scrub()");
    super.scrub(); // Извикване версията на базовия клас
  }
  // Добавяне методи към интерфейса
7 public void foam() { append(" foam()"); }
  // Тестване на новия клас

2 public static void main(String[ ] args) {
    Detergent x = new Detergent();
    x.dilute();
    x.apply();
    x.scrub();
    x.foam();
    x.print();
    System.out.println("Testing base class:");
3   Cleanser.main(args);
  }

}
```

ИНИЦИАЛИЗАЦИЯ НА БАЗОВИ КЛАСОВЕ

- При наследяването различаваме два класа
  - Базов
  - Производен
- Наследяването не е просто копиране на интерфейса на базовия клас
  - Когато създаваме обект на производен клас, той съдържа в себе си подобект на базовия клас
  - Този подобект е като, че сме създали обект на самия базов клас
  - Обектът на базовия клас е обвит от обекта на производния клас
- От съществено значение е подобектът на базовия клас да бъде коректно инициализиран
  - Само един гарантиращ това начин - инициализация в конструктора, като се извиква конструктора на базовия клас
  - Java автоматично вмъква в конструктора на производния клас извиквания към конструктора на базовия клас

ПРИМЕР

1 Какъв резултат?

```
class Art {
  Art() {
    System.out.println("Art constructor");
  }
}

class Drawing extends Art {
  Drawing() {
    System.out.println("Drawing constructor");
  }
}

public class Cartoon extends Drawing {
  Cartoon() {
    System.out.println("Cartoon constructor");
  }

  public static void main(String[ ] args ) {
    Cartoon x = new Cartoon();
  }

}
```

Art constructor
Drawing constructor
Cartoon constructor

Действието на конструктора е в посока от базовия клас „навън“, така че базовият клас се инициализира преди конструкторите на производния клас да могат да осъществяват достъп до него

Дори да не създадете конструктор за Cartoon(), компилаторът ще синтезира вместо нас такъв по подразбиране, който ще извиква конструктора на базовия клас

ДРУГ ПРИМЕР

```
public class Box {
   double width, height, depth;
   Box(Box ob) { // pass object to constructor
      width = ob.width;  height = ob.height;  depth = ob.depth;
   }
   Box(double w, double h, double d) {
      width = w; height = h; depth = d;
    }
   Box() {
      width = -1; // use -1 to indicate
      height = -1; // an uninitialized
      depth = -1; // box
    }
    Box(double len) {
       width = height = depth = len;
    }
   double volume() {
       return width * height * depth;
    }
}
```

ДРУГ ПРИМЕР

```
public class BoxWeight extends Box {

    double weight; // weight of box

    // constructor for BoxWeight
    BoxWeight(double w, double h, double d, double m) {
        width = w;
        height = h;
        depth = d;
        weight = m;
    }
}
```

ДРУГ ПРИМЕР

```
public class DemoBoxWeight {
    public static void main (String arg[]) {
        BoxWeight mybox1 = new BoxWeight(10,20,15,34.3);
        BoxWeight mybox2 = new BoxWeight(2,3,4,0.076);
        double vol;

        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        System.out.println("Weight of mybox1 is " + mybox1.weight);
        System.out.println();

        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
        System.out.println("Weight of mybox2 is " + mybox2.weight);
    }
}
```

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3

Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
```

ДРУГ ПРИМЕР

1 Какъв резултат?

```
public class RefDemo {
   public static void main (String arg[]) {
        BoxWeight weightbox = new BoxWeight(3,5,7,8.37);
        Box plainbox = new Box();
        double vol;

        vol = weightbox.volume();
        System.out.println("Volume of weightbox is " + vol);
        System.out.println("Weight of weightbox is " + weightbox.weight);
        System.out.println();

        // assign BoxWeight reference to Box reference
        plainbox = weightbox;

        vol = plainbox.volume();
        System.out.println("Volume of plainbox is " + vol);

        System.out.println("Weight of plainbox is " + plainbox.weight);
   }
```

ДРУГ ПРИМЕР

```
public class RefDemo {
   public static void main (String arg[]) {
        BoxWeight weightbox = new BoxWeight(3,5,7,8.37);
        Box plainbox = new Box();
        double vol;

        vol = weightbox.volume();
        System.out.println("Volume of weightbox is " + vol);
        System.out.println("Weight of weightbox is " + weightbox.weight);
        System.out.println();

        // assign BoxWeight reference to Box reference
        plainbox = weightbox;

        vol = plainbox.volume(); // OK, volume() defined in Box
        System.out.println("Volume of plainbox is " + vol);

        /* The following statement is invalid because plainbox does not define a weight member. */
        // System.out.println("Weight of plainbox is " + plainbox.weight);
   }
}
```

ДРУГ ПРИМЕР

1 Какъв резултат?

```
public class RefDemo {
   public static void main (String arg[]) {
        BoxWeight weightbox = new BoxWeight(3,5,7,8.37);
        Box plainbox = new Box();
        double vol;

        vol = weightbox.volume();
        System.out.println("Volume of weightbox is " + vol);
        System.out.println("Weight of weightbox is " + weightbox.weight);
        System.out.println();

        // assign BoxWeight reference to Box reference
        // plainbox = weightbox;
        weightbox = plainbox;

        vol = plainbox.volume();
        System.out.println("Volume of plainbox is " + vol);

        System.out.println("Weight of plainbox is " + plainbox.weight);
   }
}
```

ДРУГ ПРИМЕР

1 Какъв резултат?

```
public class RefDemo {
   public static void main (String arg[]) {
        BoxWeight weightbox = new BoxWeight(3,5,7,8.37);
        Box plainbox = new Box();
        double vol;

        vol = weightbox.volume();
        System.out.println("Volume of weightbox is " + vol);
        System.out.println("Weight of weightbox is " + weightbox.weight);
        System.out.println();

        // assign BoxWeight reference to Box reference
        // plainbox = weightbox;
        //weightbox = plainbox;
        weightbox = (BoxWeight) plainbox;

        vol = plainbox.volume();
        System.out.println("Volume of plainbox is " + vol);
        System.out.println("Weight of plainbox is " + plainbox.weight);
```

OK

КОНСТРУКТОРИ С АРГУМЕНТИ

- В предишния пример се използват конструктори по подразбиране
  - Без аргументи
  - Не създават проблеми на компилатора
- Ако искаме да използваме конструктор на базов клас с аргументи е необходима ключовата дума super
  - Освен това, извикването на конструктора на базовия клас трябва да бъде първият оператор в конструктора на производния клас
  - Ако не сме го направили компилаторът ще ни напомни

ПРИМЕР

1 Какъв резултат?

```
class Game {
   Game(int i) {
      System.out.println("Game constructor");
    }
}

class BordGame extends Game {
   BordGame(int i) {
     super(i);
     System.out.println("BordGame constructor");
   }
}

public class Chess extends BordGame {
    Chess() {
      super(11);
      System.out.println("Chess constructor");
   }
   public static void main(String[ ] args) {
      Chess x = new Chess();
   }
}
```

```
Game constructor
BordGame constructor
Chess constructor
```

ПРИМЕР

1 Какъв резултат?

```
class Game {
   Game(int i) {
      System.out.println("Game constructor");
    }
}

class BordGame extends Game {
   BordGame(int i) {
     // super(i);
     System.out.println("BordGame constructor");
   }
}

public class Chess extends BordGame {
    Chess() {
      super(11);
      System.out.println("Chess constructor");
   }
   public static void main(String[ ] args) {
      Chess x = new Chess();
   }
}
```

Implicit super constructor BordGame() is undefined. Must explicitly invoke another constructor

КОМБИНИРАНЕ КОМПОЗИЦИЯ И НАСЛЕДЯВАНЕ

- В много случаи е целесъобразно композицията и наследяването да се използват едновременно
- В тези случаи е целесъобразно създаване на по-сложни класове като се използват наследяване и композиция
  - Заедно с необходимата инициализация с помощта на конструктори

ПРИМЕР

```
class Plate {
   Plate(int i) {
      System.out.println("Plate constructor");
   }
}
class DinnerPlate extends Plate {
   DinnerPlate(int i) {
      super(i);
      System.out.println("DinnerPlate constructor");
   }
}
class Utensil {
    Utensil(int i) {
       System.out.println("Utensil constructor");
    }
}
class Spoon extends Utensil {
    Spoon(int i) {
       super(i);
       System.out.println("Spoon constructor");
    }
}
class Fork extends Utensil {
    Fork(int i) {
       super(i);
       System.out.println("Fork constructor");
    }
}
```

```
class Knife extends Utensil {
    Knife(int i) {
       super(i);
       System.out.println("Knife constructor");
    }
}
class Custom {
    Custom(int i) {
       System.out.println("Custom constructor");
    }
}
public class PlaceSetting extends Custom {
    Spoon sp;    Fork frk;
    Knife kn;    DinnerPlate pl;
    PlaceSetting(int i) {
       super(i + 1);
       sp = new Spoon(i + 2);
       frk = new Fork(i + 3);
       kn = new Knife(i + 4);
       pl = new DinnerPlate(i + 5);
       System.out.println("PlaceSetting constructor");
    }
    public static void main(String[ ] args) {
       PlaceSetting x = new PlaceSetting(9);
    }
}
```

ПРИМЕР

1 Какъв резултат?

```
class Plate {
   Plate(int i) {
      System.out.println("Plate constructor");
   }
}
class DinnerPlate extends Plate {
   DinnerPlate(int i) {
      super(i);
      System.out.println("DinnerPlate constructo
   }
}
class Utensil {
    Utensil(int i) {
       System.out.println("Utensil constructor");
    }
}
class Spoon extends Utensil {
    Spoon(int i) {
       super(i);
       System.out.println("Spoon constructor");
    }
}
class Fork extends Utensil {
    Fork(int i) {
       super(i);
       System.out.println("Fork constructor");
    }
}
```

```
class Knife extends Utensil {
    Knife(int i) {
       super(i);
       System.out.println("Knife constructor");
    }
}
class Custom {
                                {
                                .println("Custom constructor");
                                eSetting extends Custom {
                                ork frk;
                                DinnerPlate pl;
                                nt i) {
                                );
                                poon(i + 2);
                                Fork(i + 3);
                                Knife(i + 4);
       pl = new DinnerPlate(i + 5);
       System.out.println("PlaceSetting constructor");
    }
    public static void main(String[ ] args) {
        PlaceSetting x = new PlaceSetting(9);
    }
}
```

Custom constructor
Utensil constructor
Spoon constructor
Utensil constructor
Fork constructor
Utensil constructor
Knife constructor
Plate constructor
DinnerPlate constructor
PlaceSetting constructor

КОМПОЗИЦИЯ ВМЕСТО НАСЛЕДЯВАНЕ

- Както композицията, така също наследяването позволяват поставяне подобекти в нов клас
- Въпроси:
  - Разлика между двата подхода?
  - Кога кой подход е предпочитан?
- Обикновено композицията се използва когато искаме да включим възможностите на съществуващ клас в новия клас, но не и неговия интерфейс
  - Вграждаме обект, който можем да използваме за имплементиране функционалност в новия клас
  - Потребителят на новия клас вижда само интерфейса на новия клас (не интерфейса на вградения обект)
  - За постигане на този ефект в новия клас се вграждат private обекти на съществуващи класове
- В определени случаи има смисъл да позволим на потребителя на новия клас директен достъп до композицията, т.е. да направим член-обектите public
  - Когато потребителят знае, че сме обединили няколко части в едно, това улеснява разбирането на интерфейса на новия клас

ПРИМЕР

1 Нарушава ли публичната композиция принципите на ООП? не

```
class Engine {
   public void start() { }
   public void rev() { }
   public void stop() { }
}
class Wheel {
   public void inflate (int psi) { }
}
class Window {
   public void rollup() { }
   public void rolldown() { }
}
class Door {
   public Window window = new Window() ;
   public void open() { }
   public void close() { }
}
```

```
class Car {
   public Engine engine = new Engine() ;
   public Wheel[] wheel = new Wheel[4] ;
   public Door left = new Door(),
          right = new Door(); // с 2 врати
   public Car() {
      for(int i = 0; i < 4; i++)
      wheel[i] = new Wheel();
   }
  public static void main(String[] args) {
    Car car = new Car();
    car.left.window.rollup();
    car.wheel[0].inflate(72);
  }
}
```

КОМПОЗИЦИЯ & НАСЛЕДЯВАНЕ

- Когато извършваме наследяване създаваме специална версия на базовия клас
  - Това означава, че специализираме клас с общо предназначения за някаква конкретна необходимост
  - В примера, не е необходимо наследяване
- Зависимостта „е“ се изразява посредством наследяване
- Зависимостта „има“ се изразява посредством композиция

ПРИЯТЕЛСКИ ЗА НАСЛЕДЕНИ

- Ключовата дума protected е свързана с наследяването
- В определени случаи искаме да направим нещо скрито за външния свят, но същевременно да позволяваме достъп за членовете на производни класове
- Най-доброто решение е член-данните да остават private
  - Винаги трябва да запазим правото си за промяна на базовата имплементация
- Можем обаче, да позволим контролиран достъп до наследниците на базовия клас чрез protected методи

ПРИМЕР

```
import java.util.*;

class Villian {
   private int i;
   protected int read() { return i; }
   protected void set(int ii) { i = ii; }
   public Villian(int ii) { i = ii; }
   public int value(int m) { return m*i; }
}

public class Orc extends Villian {
   private int j;
   public Orc(int jj) { super(jj); j = jj; }
   public void change(int x) { set(x); }
}
```

change() има достъп до set(), понеже е protected

ПОСТЕПЕННО НАРАСТВАЩА РАЗРАБОТКА

- Разработването на програми е постепенен процес
  - Независимо от предхождащите анализи когато започваме един проект нямаме отговор на всички въпроси
  - Изгледите за успех са по-големи, когато проектът „нараства“ еволюционно, вместо конструиране изведнъж
- Едно от предимствата на наследяването е поддръжка на постепенно нарастване на разработките
- Позволява включване на нов код, без това да води до грешки във вече съществуващ код
  - Също изолира нови грешки във вече съществуващ код

ПОСТЕПЕННО НАРАСТВАЩА РАЗРАБОТКА

- Класовете са ясно разделени
  - Не се нуждаем от изходния код на методите за да ги използваме
- Въпреки, че техниката на наследяване може да бъде полезна при извършване на експерименти, след като разработката се стабилизира е необходимо преразглеждане на йерархията от класове
  - Целта е оптимизиране на архитектурата
  - Основен замисъл на наследяването „Този нов клас е тип на този стар клас“
  - Програмата трябва да се занимава със създаване и обработка на обекти от различни типове, за да изрази даден модел от гледна точка на приложната област на задачата

ПРЕОБРАЗУВАНЕ НАГОРЕ

- Най-важният аспект на наследяването не е доставянето на методи за новия клас
- Съществена е зависимостта между новия и базовия клас
  - Обобщена като: „Новият клас е **тип** на съществуващ клас“
  - Не е начин за обясняване на наследяването, а се поддържа директно от езика
  - Всеки обект от производния клас е също обект от базовия клас
- Наследяването означава, че всички методи на базовия клас са достъпни и в производния клас
  - Всяко съобщение, което можем да изпратим към базовия клас, може да бъде изпратено и към производния

ПРИМЕР

1 Възможно ли е?

```
import java.util.*;

class Instrument {
   public void play() { }
1  static void tune(Instrument i) {
      // ...
      i.play();
   }
}

// Обектите на Wind са инструменти,
// понеже имат същия интерфейс
class Wind extends Instrument {
   public static void main(String[] args) {
      Wind flute = new Wind();
1      Instrument.tune(flute); // Преобразуване нагоре
   }

}
```

- Независимо от това, че методът tune() е дефиниран за обекти от клас Instrument, той може да се извика с референция към обект от тип Wind
- Понеже Java е взискателен по отношение на проверка на типовете, на пръв поглед това изглежда необичайно
- Но, посредством наследяването обект от класа Wind по същество представлява също обект от класа Instrument

2 Реализация „е“?

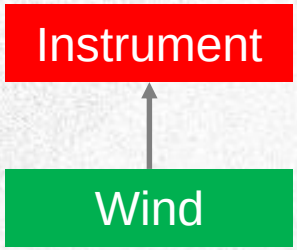
- Преобразуване на референцията към клас Wind в референция към клас Instrument

Нарича се преобразуване нагоре (upcast)

ЗАЩО ПРЕОБРАЗУВАНЕ НАГОРЕ?

1 Проблемно ли е „преобразуването нагоре“? не

- Историческа причина за термина: начин, по който са били чертани диаграмите, представящи наследяването
  - Преобразуването се извършва в посока нагоре на диаграмата на наследяване
  - Често се нарича „преобразуване нагоре“



2 Защо?

- Преминаваме от по-специфичен към по-общ тип
- Производният клас е супер-множество на базовия клас, т.е. може да съдържа повече методи от базовия клас (минимум тях)
- Единственото нещо, което може да се случи на интерфейса на класа е загуба, а не придобиване на методи
- Компилаторът позволява преобразуване нагоре без никакви явни преобразования или друга специална нотация

Възможно е също преобразуване надолу (downcast)

КОМПОЗИЦИЯ, НАСЛЕДЯВАНЕ (ОБОБЩЕНИЕ)

- В ООП:
  - Създаваме и използваме код просто като пакетираме данни и методи заедно в един клас и използваме неговите обекти
  - Също, използваме вече съществуващи обекти за създаване нови посредством композиция
  - Въпреки, че отдаваме голямо значение на наследяването, то се използва по-рядко
- Композиция или наследяване?
  - При необходимост от преобразуване нагоре: наследяване

ИЗПОЛЗВАНЕ НА FINAL

- В зависимост от контекста, ключовата дума final има в Java различни значения
  - По принцип указва елементи, които не могат да бъдат променяни
- Предотвратяване извършването на промени поради две причини:
  - Дизайн
  - Ефективност
- Двете причини значително се различават
  - Поради което final може да се използва неправилно

FINAL ДАННИ

- Указание за компилатора, че част от данните са постоянни (константи)
- Константите могат да бъдат:
  - Константи-примитиви (константи по време на компилиране)
  - Константи-референции (константи по време на изпълнение)
- С константите по време на компилиране могат да се правят изчисления от компилатора, като така се подобрява времето за изпълнение
  - Това могат да бъдат примитиви, които се представят с ключовата дума final
  - При дефиниране на такива константи трябва да се зададе стойност
  - За поле, което е едновременно static и final има определено място в паметта и то не може да се променя

FINAL РЕФЕРЕНЦИИ

- Константи по време на изпълнение са референции, които се инициализират по време на изпълнение
  - final за референции (не за примитиви)
  - Правят референциите константи
    - След като референцията бъде инициализирана, не може да бъде променяна да сочи към друг обект
    - Самият обект обаче, може да бъде променян
    - Java не доставя възможност произволен обект да стане константа

ПРИМЕР

```
class Value {
   int i = 1;
}

public class FinalData {
  // Могат да бъдат константи по време на компилиране
  final int i1 = 9;
  static final int VAL_TWO = 99;
  // Типична public константа
  public static final int VAL_THREE = 39;
  //
  fi
  st

  V
  fi
  static final Value v3 = new Value();
  // Масиви
  final int[ ] a = { 1, 2, 3, 4, 5, 6 };

  public void print(String id) {
     System.out.println(id + ": " + "i4 = " + i4 + ", i5 = " + i5);
  }
```

```
  public static void main(String[ ] args) {
    FinalData fd1 = new FinalData();
     //! fd1.i1++; // Грешка: не може да се променя стойността
     fd1.v2.i++; // Обектът не е константен
     fd1.v1 = new Value(); // OK - не е final
     for(int i = 0; i < fd1.a.length; i++)
       fd1.a[i]++; // обектът не е константен
     //! fd1.v2 = new Value(); // Грешка: не може
     //! fd1.v3 = new Value(); // да се променя референцията
     //! fd1.a = new int[3];
  }

}
```

- i1, VAL_TWO: final примитиви със стойности по време на компилиране
- VAL_TREE: типичен начин за дефиниране на такива константи – static (само една), final (константа)
- Такива константи се именуват с главни букви, думите разделени с подчертаващо тире

ПРИМЕР

1 Каква?

```
class Value {
   int i = 1;
}

public class FinalData {
  // Могат да бъдат константи по вр
  final int i1 = 9;
1 static final int VAL_TWO = 99;
  // Типична public константа
  public static final int VAL_THREE = 39;
  // Не могат да бъдат константи по време на компилиране
2 final int i4 = (int) (Math.random()*20);
  static final int i5 = (int) (Math.random()*20);

  Value v1 = new Value();
  final Value v2 = new Value();
  static final Value v3 = new Value();
  // Масиви
  final int[ ] a = { 1, 2, 3, 4, 5, 6 };

  public void print(String id) {
     System.out.println(id + ": " + "i4 = " + i4 + ", i5 = " + i5);
  }
```

```
public static void main(String[ ] args) {
   FinalData fd1 = new FinalData();
    //! fd1.i1++; // Грешка: не може да се променя стойността
    fd1.v2.i++; // Обектът не е константен
                                                          ията

    fd1.print("fd1");
    System.out.println("Creating new FinalData");
    FinalData fd2 = new FinalData();

}
```

- i5 не се променя при създаването на втори обект на класа FinalData
- Стойността е static и се инициализира един единствен път, а не всеки път, когато се създава нов обект

- i4, i5: само защото са обявени за final не означава, че стойността им е позната по време на компилация
- Разликата между двете константи: не-static и static

ПРИМЕР

```
class Value {
   int i = 1;
}

public class FinalData {
  // Могат да бъдат константи по време на компилиране
  final int i1 = 9;
1 static final int VAL_TWO = 99;
  // Типична public константа
  public static final int VAL_THREE = 39;
  // Не могат да бъдат константи по време на компилиране
2 static final int i5 = (int) (Math.random()*20);

  Value v1 = new Value();
3 final Value v2 = new Value();
  static final Value v3 = new Value();
  // Масиви
  final int[ ] a = { 1, 2, 3, 4, 5, 6 };

  public void print(String id) {
     System.out.println(id + ": " + "i4 = " + i4 + ", i5 = " + i5);
  }
```

- v1, v2, v3: демонстрират значението на final референциите

```
  public static void main(String[ ] args) {
     FinalData fd1 = new FinalData();
      //! fd1.i1++; // Грешка: не може да се променя стойността
      fd1.v2.i++; // Обектът не е константен
      fd1.v1 = new Value(); // OK - не е final
      for(int i = 0; i < fd1.a.length; i++)
         fd1.a[i]++; // обектът не е константен
      //! fd1.v2 = new Value(); // Грешка: не може
      //! fd1.v3 = new Value(); // да се променя референцията
      //! fd1.a = new int[3];

                                    );
      System.out.println("Creating new FinalData");
      FinalData fd2 = new FinalData();
      fd1.print("fd1");
      fd2.print("fd2");

   }

}
```

ПРИМЕР

```
class Value {
   int i = 1;
}

public class FinalData {
  // Могат да бъдат константи по време на компилиране
  final int i1 = 9;
1 static final int VAL_TWO = 99;
  // Типична public константа
  public static final int VAL_THREE = 39;
  // Не мог
2 final int i4
  static fina

  Value v1 =
3 final Value v2 = new Value();
  static final Value v3 = new Value();
  // Масиви
  final int[ ] a = { 1, 2, 3, 4, 5, 6 };

  public void print(String id) {
     System.out.println(id + ": " + "i4 = " + i4 + ", i5 = " + i5);
  }
```

```
  public static void main(String[ ] args) {
    FinalData fd1 = new FinalData();
     //! fd1.i1++; // Грешка: не може да се променя стойността
4    fd1.v2.i++; // Обектът не е константен
     fd1.v1 = new Value(); // OK - не е final
     for(int i = 0; i < fd1.a.length; i++)
       fd1.a[i]++; // обектът не е константен
4  //! fd1.v2 = new Value(); // Грешка: не може
   //! fd1.v3 = new Value(); // да се променя референцията
   //! fd1.a = new int[3];
     fd2.print("fd2");

  }

}
```

- v2: само понеже е final не означава, че не може да се променя реферираният от нея обект
- v2, v3: обаче, не могат да се обвържат повторно с нови обекти, понеже са final – това означава final за референции

ПРИМЕР

1 Защо при масива също грешка?

Масивите се представят също чрез референции

```
class Value {
   int i = 1;
}

public class FinalData {
  // Могат да бъдат константи по време на компилиране
  final int i1 = 9;
1 static final int VAL_TWO = 99;
  // Типична public константа
  public static final int VAL_THREE = 39;
  // Не могат да бъдат константи по време на компилиране
2 final int i4 = (int) (Math.random()*20);
  static final int i5 = (int) (Math.random()*20);

  Value v1 = new Value();
3 final Value v2 = new Value();
  static final Value v3 = new Value();
  // Масиви
  final int[ ] a = { 1, 2, 3, 4, 5, 6 };

  public void print(String id) {
     System.out.println(id + ": " + "i4 = " + i4 + ", i5 = " + i5);
  }
```

```
  public static void main(String[ ] args) {
    FinalData fd1 = new FinalData();
     //! fd1.i1++; // Грешка: не може да се променя стойността
4    fd1.v2.i++; // Обектът не е константен
     fd1.v1 = new Value(); // OK - не е final
     for(int i = 0; i < fd1.a.length; i++)
       fd1.a[i]++; // обектът не е константен
4   //! fd1.v2 = new Value(); // Грешка: не може
     //! fd1.v3 = new Value(); // да се променя референцията
5    //! fd1.a = new int[3];

    fd1.print("fd1");
    System.out.println("Creating new FinalData");
    FinalData fd2 = new FinalData();
    fd1.print("fd1");
    fd2.print("fd2");

  }

}
```

ПРИМЕР

1 Резултат?

```
class Value {
   int i = 1;
}

public class FinalData {
  // Могат да бъдат константи по време на компилиране
  final int i1 = 9;
  static final int VAL_TWO = 99;
  // Типична public константа
  public static final int VAL_THREE = 39;
  // Не могат да бъдат константи по време
компилиране
  final int i4 = (int) (Math.random()*20);
  static final int i5 = (int) (Math.random()*20);

  Value v1 = new Value();
  final Value v2 = new Value();
  static final Value v3 = new Value();
  // Масиви
  final int[ ] a = { 1, 2, 3, 4, 5, 6 };

  public void print(String id) {
     System.out.println(id + ": " + "i4 = " + i4 + ", i5 = " + i5);
  }
```

```
  public static void main(String[ ] args) {
    FinalData fd1 = new FinalData();
     //! fd1.i1++; // Грешка: не може да се променя   стойността
     fd1.v2.i++; // Обектът не е константен
     fd1.v1 = new Value(); // OK - не е final
     for(int i = 0; i < fd1.a.length; i++)
       fd1.a[i]++; // обектът не е константен
     //! fd1.v2 = new Value(); // Грешка: не може
     //! fd1.v3 = new Value(); // да се променя референцията
                                 new int[3];

                                 d1");
                                 println("Creating new FinalData");
                                 d2 = new FinalData();
                                 d1");
     fd2.print("fd2");

  }

}
```

```
fd1: i4 = 4, i5 = 13
Creating new FinalData
fd1: i4 = 4, i5 = 13
fd2: i4 = 1, i5 = 13
```

ПРИМЕР

1 Без коментара при fd1.v2

```
class Value {
   int i = 1;
}

public class FinalData {
  // Могат да бъдат константи по време на компилиране
  final int i1 = 9;
  static final int VAL_TWO = 99;
  // Типична public константа
  public static final int VAL_THREE = 39;
  // Не могат да бъдат константи по време на
компилиране
  final int i4 = (int) (Math.random()*20);
  static final int i5 = (int) (Math.random()*20);

  Value v1 = new Value();
  final Value v2 = new Value();
  static final Value v3 = new Value();
  // Масиви
  final int[ ] a = { 1, 2, 3, 4, 5, 6 };

  public void print(String id) {
     System.out.println(id + ": " + "i4 = " + i4 + ", i5 = " + i5);
  }
```

```
  public static void main(String[ ] args) {
    FinalData fd1 = new FinalData();
     //! fd1.i1++; // Грешка: не може да се променя  стойността
     fd1.v2.i++; // Обектът не е константен
     fd1.v1 = new Value(); // OK - не е final
     for(int i = 0; i < fd1.a.length; i++)
       fd1.a[i]++; // обектът не е константен
6    fd1.v2 = new Value(); // Грешка: не може
     //! fd1.v3 = new Value(); // да се променя референцията
     //! fd1.a = new int[3];
The final field FinalData.v2 cannot be assigned
     fd1.print("fd1");
     System.out.println("Creating new FinalData");
     FinalData fd2 = new FinalData();
     fd1.print("fd1");
     fd2.print("fd2");

  }

}
```

FINAL АРГУМЕНТИ

- Java позволява да се използват final аргументи
- Декларираме ги като такива в списъка с аргументи
- Това означава, че вътре в метода не може да се променя това, към което сочи референцията на аргументите

ПРИМЕР

```
class Gizmo {
   public void spin() { }
}

public class FinalArguments {
   void with(final Gizmo g) {
   //! g = new Gizmo(); // забранено действие - g е final
   }
   void without(Gizmo g) {
       g = new Gizmo(); // OK - g не е final
       g.spin();
   }
   // void f(final int i) { i++; } // не може да се промени
   // един final примитив може само да се чете
   int g(final int i) { return i + 1; }

   public static void main(String[ ] args) {
       FinalArguments bf = new FinalArguments();
       bf.without(null);
       bf.with(null);
   }

}
```

Можем да четем аргументите, но не можем да ги променяме

FINAL МЕТОДИ

- Две причини за тяхното използване
  - „Заключваме“ метода за да предотвратим евентуална промяна от наследен клас
  - Ефективност
    - Компилаторът прави всички извиквания към този метод вмъкнати (inline)
- final и private
  - Всички private методи в един клас са final
  - Понеже нямаме достъп до private методите, не можем да го предефинираме
  - Можем да добавим спецификатора final към private
    - Няма да предаде никакво допълнително значение

ПРИМЕР

```
class WithFinals {
   // Ефектът е същият все едно е използвана final
   private final void f() {
      System.out.println("WithFinals.f()");
   }
   // Автоматично се приема за final
   private void g() {
      System.out.println("WithFinals.g()");
   }
}

class OverridingPrivate extends WithFinals {
   private final void f() {
      System.out.println("OverridingPrivate.f()");
   }
   private void g() {
      System.out.println("OverridingPrivate.g()");
   }
}
```

```
class OverridingPrivate2 extends OverridingPrivate {
   public final void f() {
      System.out.println("OverridingPrivate2.f()");
   }
   public void g() {
      System.out.println("OverridingPrivate2.g()");
   }
}

public class FinalOverridingIllusion {
    public static void main(String[ ] args) {
      OverridingPrivate2 op2 = new OverridingPrivate2();
      op2.f();
      op2.g();
      // Можем да извършим преобразуване нагоре
      OverridingPrivate op = op2;
      // но не можем да извикаме следните методи:
      //! op.f();
      //! op.g();
      // same here
      WithFinals wf = op2;
      //! wf.f();
      //! wf.g();

    }
}
```

```
OverridingPrivate2.f()
OverridingPrivate2.g()
```

FINAL КЛАСОВЕ

- Заявяваме, че не искаме да се наследява от този клас
  - Поради някаква причина структурата на класа е такава, че никога не възниква необходимост за извършване на промени
  - Също поради причини, свързани с безопасност или сигурност, не искаме да създаваме подкласове
  - Също по причини, свързани с по-голяма ефективност
    - Всяко действие, свързано с този клас е колкото се може по-ефективно

ПРИМЕР

```
class SmallBrain { }

final class Dinosaur {
  int i = 7;
  int j = 1;
  SmallBrain x = new SmallBrain();
  void f() { }
}

//! class Further extends Dinosaur { }
// Грешка: не можем да наследим final class Dinosaur

public class Jurassic {
  public static void main(String[ ] args) {
    Dinosaur n = new Dinosaur();
    n.f();
    n.i = 40;
    n.j++;
  }

}
```

БЛАГОДАРЯ ЗА ВНИМАНИЕТО!

КРАЙ “МНОГОКРАТНО ИЗПОЛЗВАНЕ”