

Лекция 7.а

Вътрешни и анонимни класове Част I

Основни теми

- **Вътрешни и анонимни класове**- синтаксис и приложения.,
 - ***public*** и ***private*** конструктори, скриване на source кода на приложението
 - обработка на събития с вътрешни и анонимни класове (***closure*** , ***callback*** конструкции)
 - Задачи

- 7a.1 Преговор на интерфейси
- 7a.2 Структура на интерфейс
- 7a.3 Множествено наследяване
- 7a.4 Наследяване при интерфейси
- 7a.5 Интерфейс или абстрактен клас
- 7a.6 Разширяване на интерфейс с използване на наследственост
- 7a.7 Вътрешни класове
- 7a.8 Външен клас и методи за връзка с вътрешни класове
- 7a.9 Вътрешни класов и преобразуване “нагоре” при наследственост за скриване на имплементацията
- 7a.10 Вътрешни класове в методи
- 7a.11 Вътрешен клас като връзка с външен клас
- 7a.12 Наследственост при вътрешни класове
- 7a.13 Closure и Callback
- 7a.14 Анонимни вътрешни класове

Задачи

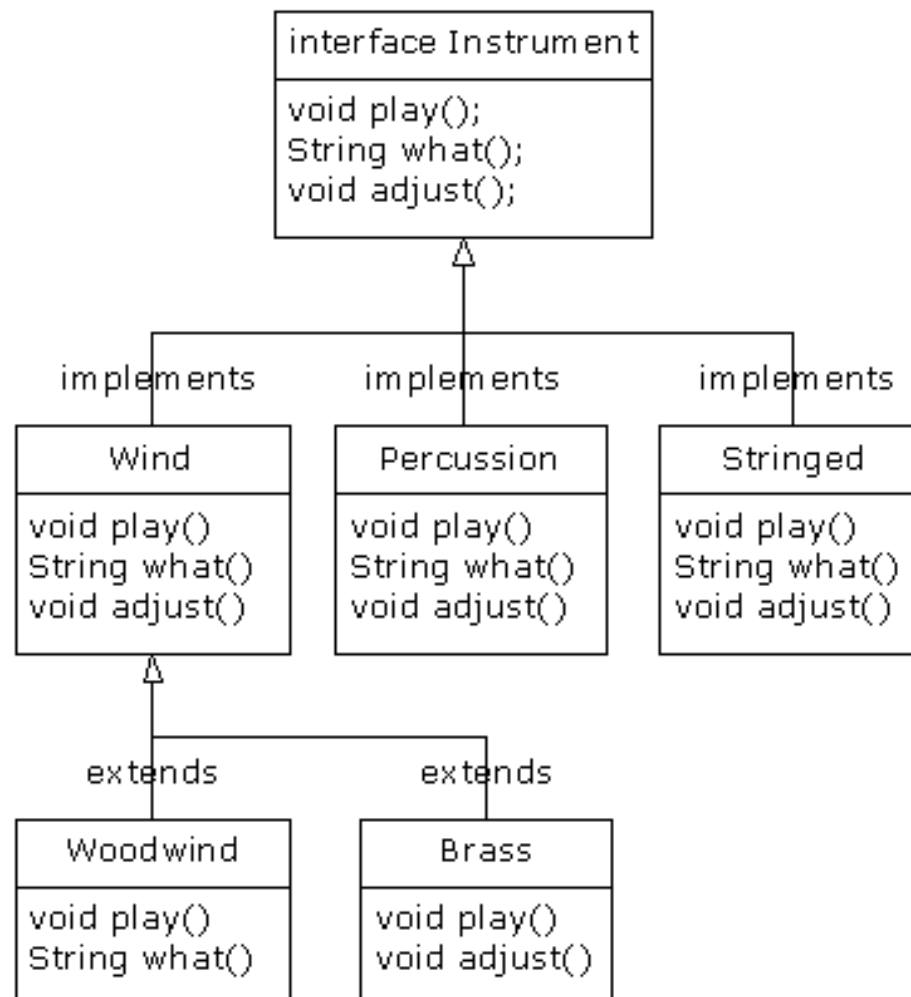
Литература:

***Bruce Eckel “Thinking in Java”, 2nd ed., Prentice Hall 2000
или българското ѝ издание “Да мислим на Java” том
1 и 2, SoftPress, 2001***

7a.1 Интерфейси- преговор

- Позволяват да се реализира множествоно онаследяване
- Може да се разглеждат като “чист” **abstract class**. Може да имат:
 - Имена на методи, списък с аргументи и тип на връщани данни, но без дефиниция на методите.
 - Данни, но те са неявно *static* и *final*
 - Всички методи са неявно *public*
- Дефинират се с ключовата дума *interface*
- След имплементирането на *interface*, класът реализирал методите на интерфейса може да се разширява по познатия начин

7a.1 Пример



. След реализирането на метод на интерфейс, тази реализация се онаследява!

```

1. // Interfaces.
2. import java.util.*;

3. interface Instrument {
4.     // Compile-time constant:
5.     int i = 5; // static & final
6.     // Cannot have method definitions:
7.     void play(); // Automatically public
8.     String what();
9.     void adjust();
10.}

11.class Wind implements Instrument {
12.    public void play() {
13.        System.out.println("Wind.play()");
14.    }
15.    public String what() { return "Wind"; }
16.    public void adjust() {}
17.}

18.class Percussion implements Instrument {
19.    public void play() {
20.        System.out.println("Percussion.play()");
21.    }
22.    public String what() { return "Percussion"; }
23.    public void adjust() {}
24.}

25.class Stringed implements Instrument {
26.    public void play() {
27.        System.out.println("Stringed.play()");
28.    }
29.    public String what() { return "Stringed"; }
30.    public void adjust() {}
31.}
// continues on the next slide

```

Е. Кръстев, OOP
Java, 2014.

Може да се види как метод ***tuneAll*** се изпълнява за произволен инструмент, който е произведен на ***interface Instrument***.

.

Е. Кръстев, OOP
Java, 2014.

```

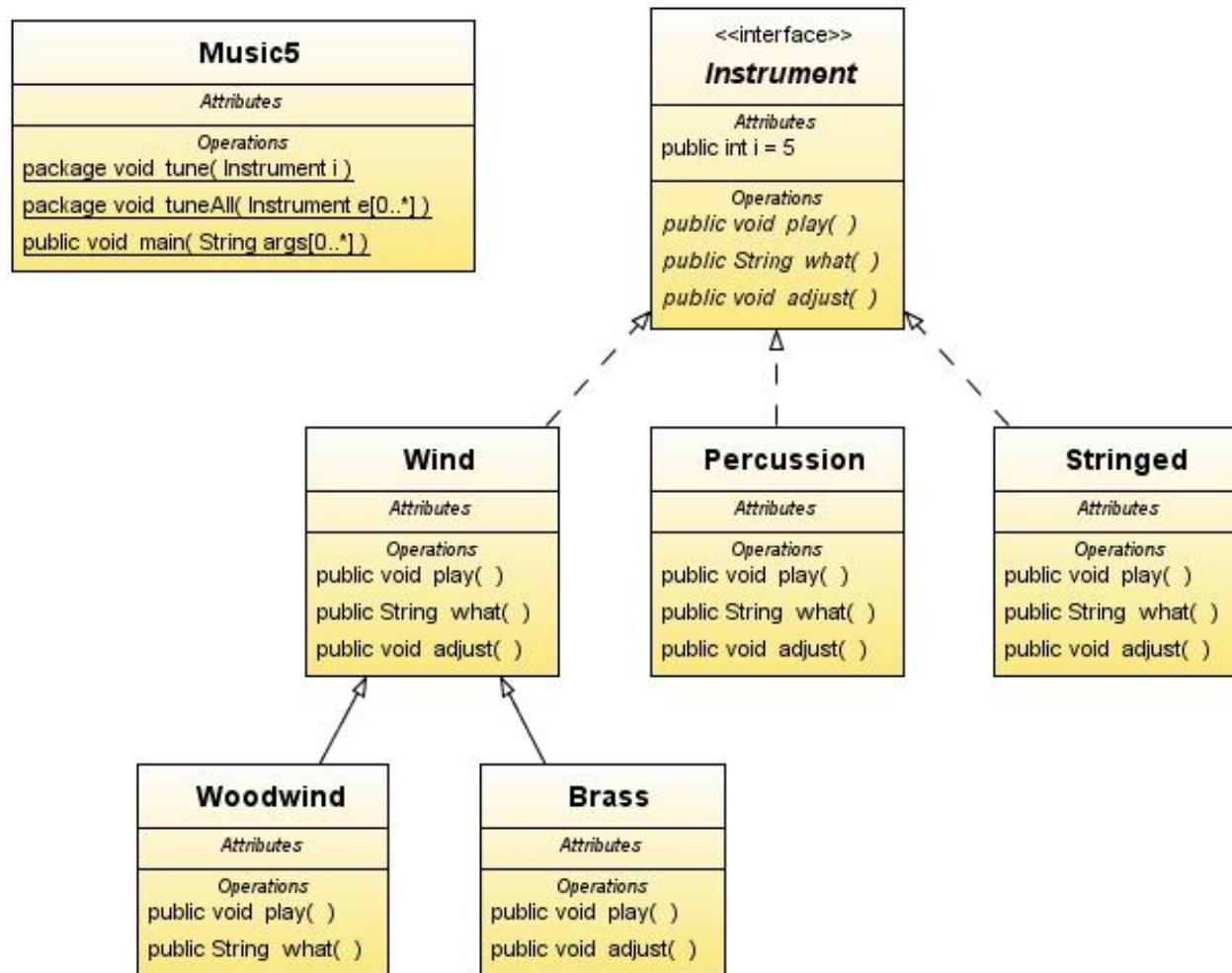
32.class Brass extends Wind {
33.    public void play() {
34.        System.out.println("Brass.play()");
35.    }
36.    public void adjust() {
37.        System.out.println("Brass.adjust()");
38.    }
39.}

40.class Woodwind extends Wind {
41.    public void play() {
42.        System.out.println("Woodwind.play()");
43.    }
44.    public String what() { return "Woodwind"; }
45.}

46.public class Music5 {
47.    // Doesn't care about type, so new types
48.    // added to the system still work right:
49.    static void tune(Instrument i) {
50.        // ...
51.        i.play();
52.    }
53.    static void tuneAll(Instrument[] e) {
54.        for(int i = 0; i < e.length; i++)
55.            tune(e[i]);
56.    }
57.    public static void main(String[] args) {
58.        Instrument[] orchestra = new Instrument[5];
59.        int i = 0;
60.        // Upcasting during addition to the array:
61.        orchestra[i++] = new Wind();
62.        orchestra[i++] = new Percussion();
63.        orchestra[i++] = new Stringed();
64.        orchestra[i++] = new Brass();
65.        orchestra[i++] = new Woodwind();
66.        tuneAll(orchestra);
67.    }
68.}

```

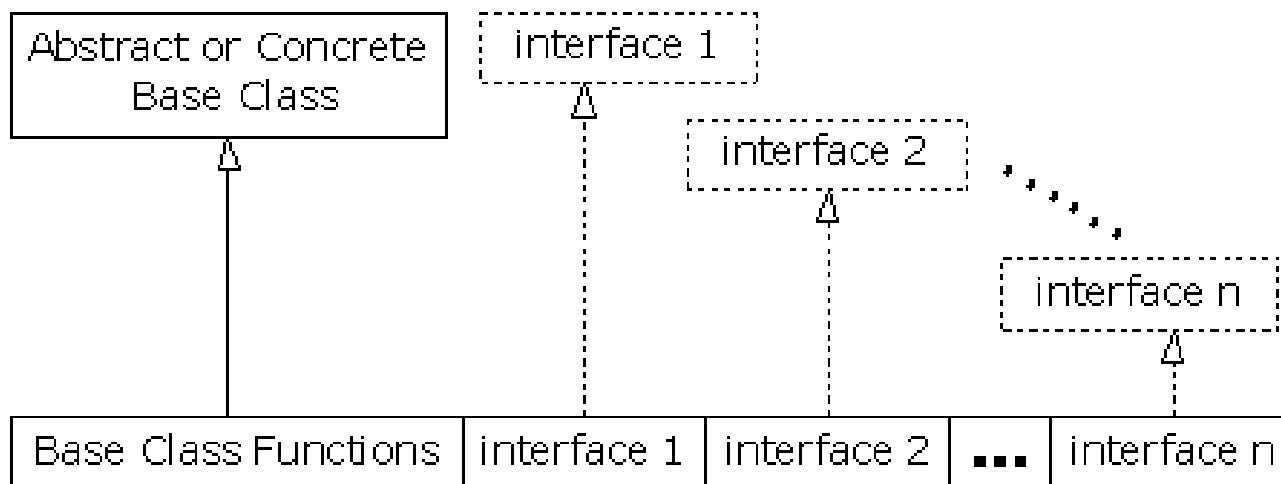
7a.1 UML diagram



7а.2 Структура на интерфейс

- След имплементирането на **interface**, тази имплементация става обикновен клас, който може да се разширява според релацията “обект *A* *е* обект *B*”
- При имплементацията *implement* на **interface**, методите на **interface** трябва да се *public* (те са *public по декларация в interface*)
- В противен случай се прави опит за предефиниране на тези методи с пакетен “*приятелски*” достъп и с това се намалява нивото на достъп при наследственост, което е забранено

7a.3 Множествено онаследяване



Всеки от интерфейсите се изброява със запетая след ключовата дума *implements*.

Позволени са произволен брой интерфейси до които може да се извършва преобразуване нагоре.

Следва пример за това как конкретен клас може да наследи от няколко интерфейса и отделен клас.

```
// Multiple interfaces.
import java.util.*;

interface CanFight {
    void fight();
}

interface CanSwim {
    void swim();
}

interface CanFly {
    void fly();
}

class ActionCharacter {
    public void fight() {}
}

class Hero extends ActionCharacter implements CanFight, CanSwim, CanFly
{
    public void swim() {}
    public void fly() {}
}
```

Може да се види как **Hero** обединява конкретен клас **ActionCharacter** с интерфейсите **CanFight**, **CanSwim**, и **CanFly**.

При комбиниране на конкретен клас с интерфейси, конкретни клас се пише пръв

```
public class Adventure {

    static void makeTrouble(CanFight x) { x.fight(); }

    static void breakRecord(CanSwim x) { x.swim(); }

    static void tryFaster(CanFly x) { x.fly(); }

    static void makeMovie(ActionCharacter x) { x.fight(); }

    public static void main(String[] args)
    {
        Hero hero = new Hero();
        // Hero is an ActionCharacter
        // he also CanFight, CanSwim, CanFly,
        makeTrouble(hero); // Treat hero as a CanFight

        breakRecord(hero); // Treat hero as a CanSwim

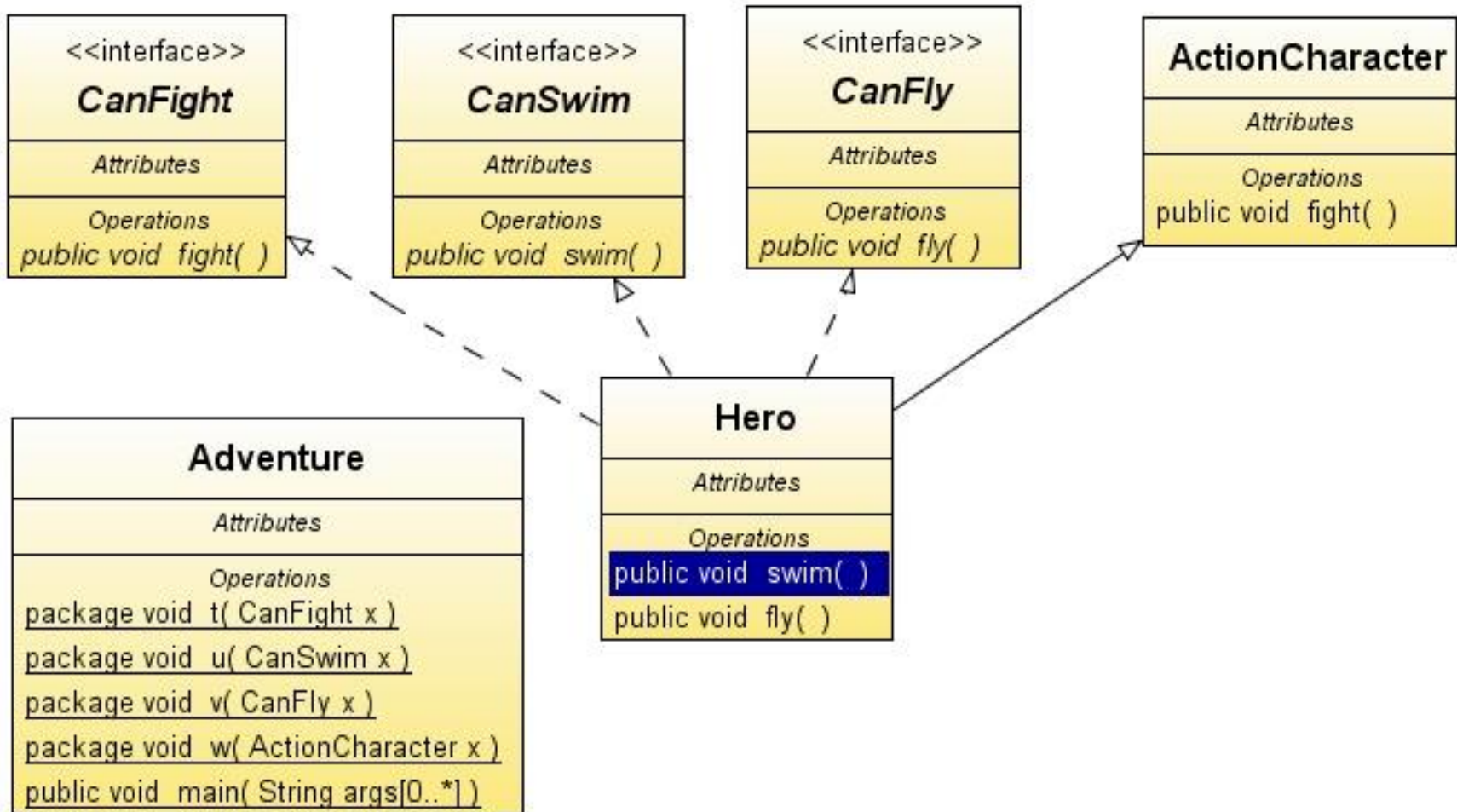
        tryFaster(hero); // Treat hero as a CanFly

        makeMovie(hero); // Treat hero as an ActionCharacter
    }
}
```

Основната причина да се въведат интерфейси се илюстрира тук. Целта е да се преобразува до повече от един базов клас.

Друга причина е, тази както при abstract базов class- да не се даде възможност на клиент програмиста да създава обекти от такъв базов клас

UML diagram - Adventure



7a.4 Наследственост при интерфейси

Забележете, че:

- подписът на ***fight()*** в ***class ActionCharacter*** е като на същия метод в ***interface CanFight*** и ***class ActionCharacter***, и също че ***fight()*** не е реализиран в ***class Hero***.
- **Правилото** при ***interface*** е, че може да се онаследява от него, но тогава получавате друг (*както ще видим след малко*), ***interface***. Ако искате да създадете обект от нов тип трябва да се реализират всички методи на интерфейса.
- Макар ***class Hero*** да няма явна реализация на метод ***fight()***, тази дефиниция се онаследява от базовия клас ***ActionCharacter*** и това е достатъчно да се създават обекти от ***class Hero***.

7a.4 Наследственост при интерфейси

- В class *Adventure*, има четири метода , които взимат за аргументи *различни interface* и конкретен *class*. Когато един *Hero* обект се създава, той може да се предаде на всеки един от тези методи, което означава, че *Hero* обектът се преобразува нагоре до всеки от съответните *interface* -и.

7a.5 Интерфейси или Абстрактен клас

- Един **interface** дава предимствата на **abstract class** и ползата на **interface** проличава, ако е възможно да се използва **базов class** без да са необходими дефиниции за методи или конкретни стойности за данни то **винаги предпочитайте interface** вместо **abstract class**.
- В действителност, ако се знае за нещо, че ще служи за **базов клас**, то **първоначалния избор** трябва да е **interface**
- **Само когато е необходимо** да използвате дефиниция за **метод или данни** то ще смените този избор с **abstract class** или конкретен клас.

7a.6 Генерализиране(разширяване) на *interface* с inheritance

- Винаги може да се добави нова декларация към един *interface* посредством **наследственост**, а също и да се комбинират методите на няколко *interface* в нов *interface* посредством **наследственост**. В двата случая се получава нов *interface*, както е показано на следващия **пример**

```
// Extending an interface with inheritance.
```

```
interface Monster {
    void menace();
}

interface DangerousMonster extends Monster {
    void destroy();
}

interface Lethal {
    void kill();
}

class DragonZilla implements DangerousMonster {
    public void menace() {}
    public void destroy() {}
}

interface Vampire extends DangerousMonster, Lethal {
    void drinkBlood();
}

class HorrorShow {
    static void feed(Monster b) { b.menace(); }
    static void runAway(DangerousMonster danger) {
        danger.menace();
        danger.destroy();
    }
}
```

Обикновено, използваме **extends** с един единствен клас при създаване на нов производен клас, но понеже **interface** може да е съставен от повече от един **interface**, то **extends** може да изброява със запетая повече от един интерфейс.

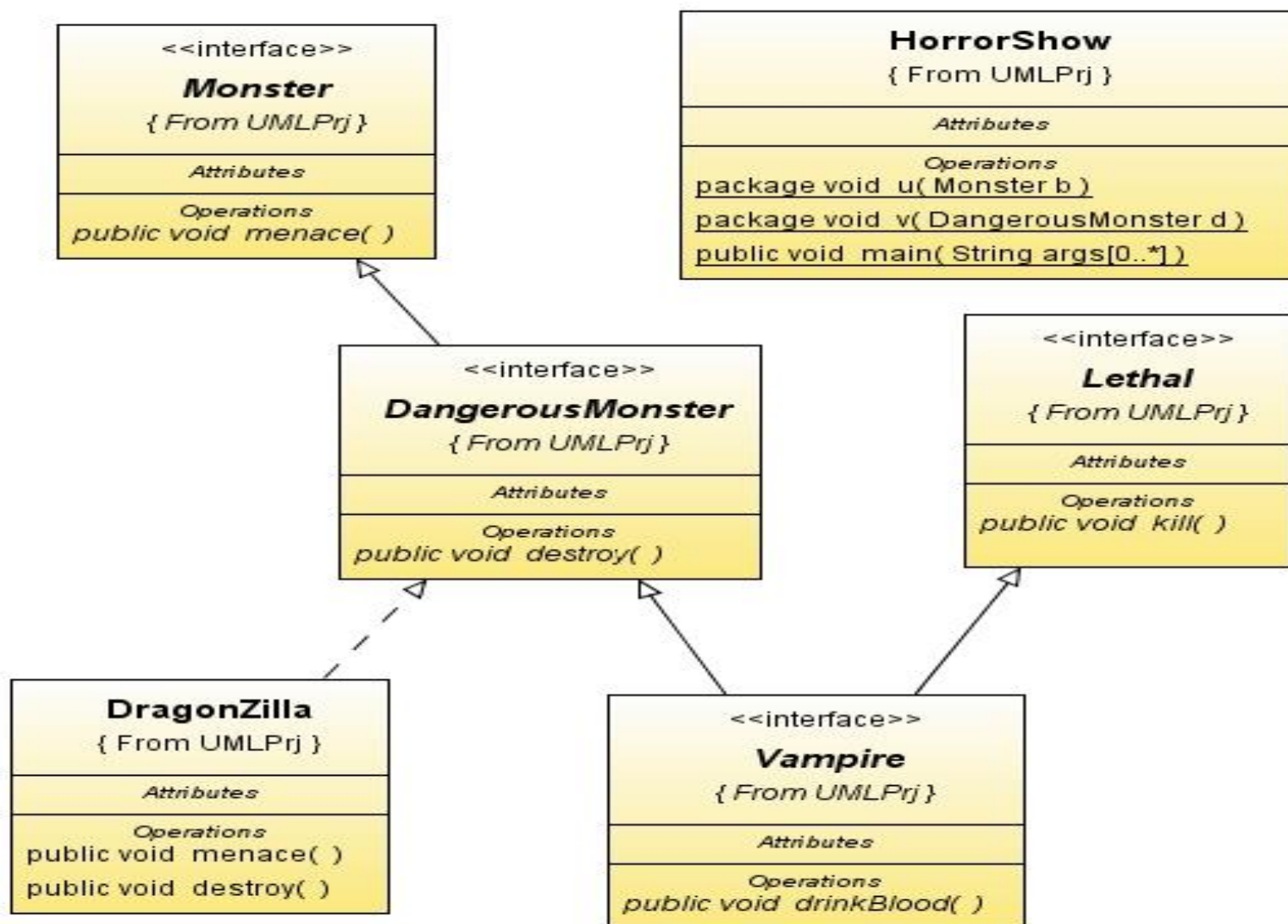
Е. Кръстев, OOP
Java, 2014.

```
public static void main(String[] args) {  
    DragonZilla babyZilla = new DragonZilla();  
    feed(babyZilla );  
    runAway(babyZilla );  
}  
}
```

DangerousMonster е интерфейс произведен на **Monster** и създава нов **interface**. Този интерфейс е реализиран в клас **DragonZilla**

Респективно обектите на клас **DragonZilla** могат да се преобразуват нагоре (в статичните методи **feed** и **runAway**) до интерфейс **DangerousMonster** и интерфейс **Monster**

UML diagram - Monster



7a.8 Инициализиране на константи в интерфейс

- Данни дефинирани в интерфейс са неявно *static* и *final*.
- Те не могат да се оставят “неинициализирани,” но могат да се инициализират с неконстантни изрази

```
// Initializing interface fields with
// non-constant initializers.
import java.util.*;

public interface RandVals {
    int rint = (int) (Math.random() * 10);
    long rlong = (long) (Math.random() * 10);
    float rfloat = (float) (Math.random() * 10);
    double rdouble = Math.random() * 10;
}

// using the interface data members

public class TestRandVals
{
    public static void main(String[] args)
    { System.out.println(RandVals.rint);
      System.out.println(RandVals.rlong);

System.out.println(RandVals.rfloat);
      System.out.println(RandVals.rdouble);
    }
}
```

Понеже **данните на интерфейса са static**, те се инициализират при зареждането на класа, което става при първия път на използване на данните

вижте примера

Е. Кръстев, OOP
Java, 2014.

7а.8 Изброим тип данни

- `enum` ключова дума за тип данни
- **Спадат към референтен** тип- обектите им се реферират с променливи
- Служи за дефиниране на списък от константи, представени общо с уникални, лесно читаеми имена
 - **Декларират** се с `enum` ключова дума
 - Списък от константи разделени със запетая след `enum`
 - Всяка константа има **име**, а може и да има **аргументи**
 - Декларират `enum class` със следните ограничения:
 - `enum` е тип, който неявно е **final**
 - `enum` константите са също **static**
 - Опит за създаване на обект от `enum` тип посредством **new** дава **грешка** при компилация
 - `enum` константи се използват **като всички други константи**
 - `enum` **конструктор**
 - Може да има допълнителни дефиниции т.е. да се използва като конструктор за общо ползване

7a.8 Изброим тип данни...

- Най-прост вид-служат за дефиниране на група от константи.

```
enum Status{ CONTINUE, WON, LOST}; // static!
```

```
// .... in some method
```

```
Status mode = Status.WON;
```

```
// ...change it
```

```
mode = Status.LOST;
```


7a.8 Изброим тип данни...

- Най- прост вид- служат за дефиниране на група от константи.

```
// class data member
enum Status{ CONTINUE, WON, LOST};
void someMethod()
{
    // ...
    Status mode = Status.WON;
    // ...
    mode = Status.LOST;
}
```

Важно: *enum* константите не могат да са локални!

7a.8 Изброим тип данни...

- Могат да се използват със `switch()`

```
enum Answer{ OK, NO};  
boolean  someTest(Answer ask)  
{  
    switch (ask)  
    {  
        case OK :  
            return true;  
        case NO :  
            return false;  
    }  
    return false;  
}
```

Outline

Book.java

(1 от 2)

```
1 // Fig. 8.10: Book.java
2 // Declaring an enum type with constructor and explicit instance fields
3 // and accessors for these field
```

```
4
5 public enum Book
6 {
7     // declare constants of enum type
8     JHTP6( "Java How to Program 6e", "2005" ),
9     CHTP4( "C How to Program 4e", "2004" ),
10    IW3HTP3( "Internet & World wide Web How to Program 3e", "2004" ),
11    CPPHTP4( "C++ How to Program 4e", "2003" ),
12    VBHTP2( "Visual Basic .NET How to Program 2e", "2002" ),
13    CSHARPHTP( "C# How to Program", "2002" );
14
15    // instance fields
16    private final String title; // book title
17    private final String copyrightYear; // copyright year
18
19    // enum constructor
20    Book( String bookTitle, String year )
21    {
22        title = bookTitle;
23        copyrightYear = year;
24    } // end enum Book constructor
25
```

Декларира 6 enum
константи

Имена на enum
константи

Структура от
аргументи на enum
константи, които се
предават на
конструктора на
enum

Декларираме клас данни, съответстват
по брой и тип на аргументите на
константите

Дефинира конструктора за
enum Book



Outline

Book.java

(2 от 2)

```
26 // accessor for field title
27 public String getTitle()
28 {
29     return title;
30 } // end method getTitle
31
32 // accessor for field copyrightYear
33 public String getCopyrightYear()
34 {
35     return copyrightYear;
36 } // end method getCopyrightYear
37 } // end enum Book
```



7a.8 Изброим тип данни...

- Всяка *enum* константа е обект от тип *Book* и има свое копие от клас данните *title* and *copyrightYear* (константни и статични!)
- **Конструкторът** (редове 20- 24) взима два *String* параметъра, първият задава заглавието на книга, а вторият, авторските права на книгата.
- Редове 22- 23 присвояват тези променливи на клас данните.
- Редове 27-36 декларират GET методи за *title* и *copyrightYear*
- *enum* типът позволява да обхождаме константите.

7a.8 Изброим тип данни...

- **static** метод **values()**
 - Генерира се от компилаторът за всеки **enum** тип
 - Връща масив **an array of the enum's constants in the order in which they were declared**
- **static method ordinal**
 - Returns the sequential number of an **enum** constant
- **static** метод **range()** от **class EnumSet**
 - Взима два аргумента- **първия** и **последния** в желан **интервал** от **enum** константи
 - Връща **EnumSet** съдържащ константите в желания интервал, включително двата крайни обекта
 - Специализирана **for** кованда може да обходи множество **EnumSet** също както масив

7a.8 Изброим тип данни...

- You need this constructor to be with **package** access, because enums define a finite set of values (for example **JHTP6**, **CHTP4**, ..., **CSHARPHTP**). If the constructor was public people could potentially create more values (for example **invalid/undeclared** values such as **XX**, **KK**, etc). This would extend the set of initially declared values.
- The constructor for an **enum** type must be **package-private** or **private** access. It automatically creates the constants that are defined at the beginning of the enum body.
- You cannot invoke an enum constructor yourself.

Outline

EnumTest.java

(1 от 2)

```

1 // Fig. 8.11: EnumTest.java
2 // Testing enum type Book.
3 import java.util.EnumSet;
4
5 public class EnumTest
6 {
7     public static void main( String args[] )
8     {
9         System.out.println( "All books:\n" );
10
11         // print all books in enum Book
12         for ( Book book : Book.values() )
13             System.out.printf( "%-10s%-45s%\n", book,
14                               book.getTitle(), book.getCopyrightYear() );
15
16         System.out.println( "\nDisplay a range of enum constants:\n" );
17
18         // print first four books
19         for ( Book book : EnumSet.range( Book.JHTP6, Book.CPPHTP4 ) )
20             System.out.printf( "%-10s%-45s%\n", book,
21                               book.getTitle(), book.getCopyrightYear() );
22     } // end main
23 } // end class EnumTest

```

Специализиран **for** цикъл обхожда всяка от enum константите в масива върнат от метод value()

Връща името на текущата **enum** константа

Специализиран **for** цикъл обхожда всяка от enum константите МНОЖЕСТВОТО EnumSet върнато от метод range

Извежда title и copyrightYear на текущо обработваната enum КОНСТАНТА



Outline

EnumTest.java

(2 от 2)

All books:

JHTP6	Java How to Program 6e	2005
CHTP4	C How to Program 4e	2004
IW3HTP3	Internet & World Wide Web How to Program 3e	2004
CPPHTP4	C++ How to Program 4e	2003
VBHTP2	Visual Basic .NET How to Program 2e	2002
CSHARPHTP	C# How to Program	2002

Display a range of enum constants:

JHTP6	Java How to Program 6e	2005
CHTP4	C How to Program 4e	2004
IW3HTP3	Internet & World Wide Web How to Program 3e	2004
CPPHTP4	C++ How to Program 4e	2003



Обичайна грешка при програмиране 8.6


Синтактична грешка при *enum* декларация е да се декларират *enum* константите след *enum* конструкторите, данните или методите в *enum* декларацията.

```
1 public enum ConvertableEnum { // convert int to enum
2     POSITIVE, NEGATIVE, EITHER, UNDEFINED;
3
4     public static ConvertableEnum convertIntToEnum(int i) {
5         return values()[i]; // values() converts enum to an array
6     }
7
8     public static ConvertableEnum convertIntToEnumWithException(int i) {
9         try {
10             return values()[i];
11         } catch (ArrayIndexOutOfBoundsException e) {
12             return UNDEFINED;
13         }
14     }
15
16     public static ConvertableEnum convertIntToEnumWithOrdinal(int i) {
17         for (ConvertableEnum current : values()) {
18             if (current.ordinal() == i) { // Using ordinal()!!
19                 return current;
20             }
21         }
22
23         return UNDEFINED;
24     }
25 }
```

ConvertableEnum.java



values() converts **enum** to an array



ordinal() returns the sequential
number of an **enum** constant



7a.9 Вътрешни класове

- В редица случай е удачно **да се вложи дефиницията на клас в дефиницията на друг class**. Това се нарича **вътрешен (*inner*) class**.
- Вътрешният клас е незаменим понеже **позволява да се групират класове**, които логически са заедно, а също и **за управление на достъпа помежду** им..
- Вътрешните класове **са различни от композиция на данни**.

7a.9 Вътрешни класове

- Всеки *inner class* може да **наследява независимо** от външния клас. Вътрешният клас не е ограничен от наследствеността на външния клас

Вътрешните класове позволяват “**многократно прилагане на наследственост.**” Това позволява един клас **да реализира наследственост от повече от един не-интерфейс базов клас.**

7a.9 Вътрешни класове

Създаване на обект от вътрешен клас:

- **навсякъде освен в non-static метод** на външен клас, **типът** на такъв обект се задава като

OuterClassName.InnerClassName.

- За създаване на обект от вътрешен клас е нужно да има първо създаден обект от външния му клас

Забележка: Създаването на обект от нестатичен вътрешен клас става с референция към външния клас и ключовата дума `new`, разделени с точка (*виж следващия слайд*)

- Вътрешният клас има пълен достъп до всички данни и методи от външния клас, дори те да са дефинирани като *private*

Забележка: Статичен вътрешен клас има достъп само до статични данни и методи от външния клас.

```
// Creating objects from inner classes.
```

```
public class NestedClass {  
    private String name = "instance name";  
    private static String staticName = "static name";  
  
    public static void main(String args[]) {  
        NestedClass nt = new NestedClass();  
        // create object from inner classes  
        NestedClass.NestedOne nco = nt.new NestedOne();  
        NestedClass.NestedTwo nct =  
            new NestedClass.NestedTwo();  
    }  
    class NestedOne { // a non- static inner class  
        NestedOne() { // gets full access to outer class  
            System.out.println(name);  
            System.out.println(staticName);  
        }  
    }  
    static class NestedTwo { // a static inner class  
        NestedTwo() { // gets full access to outer class  
            System.out.println(staticName);  
        }  
    }  
}
```

Резюме

Пример:

Създаване на
обекти от
вътрешен
клас

Достъп на
вътрешен
клас до
данни и
методи от
външния
клас

Е. Кръстев,
OOP Java, FMI,
2014.

7a.9 Вътрешни класове

- Използваме

OuterClassName.this.member

или (при релация на наследственост)

OuterClassName.super.member

за достъп до член на външния клас

Например, във вътрешния клас

OuterClassName.this.toString()

изпълнява *toString()* от външния клас за **разлика** от *this.toString()*

КОЕТО изпълнява *toString()* от вътрешния клас

7a.9 Вътрешни класове

Допълнителни свойства:

- Вътрешните класове могат да имат **множество инстанции**, всяка от които с независимо поведение и данни от **обекта на външния клас**.
- В един единствен външен клас може да има **няколко независими вътрешни класа**, всеки от които да **реализира един и същи interface** или **онаследява един и същи клас по различни начини**. (*Callback приложения*)
- **Моментът на създаване** на обект от вътрешен клас **не е обвързан** със създаване на обект от външния клас.
- Няма объркване с релацията “**is-a**” по отношение на **вътрешния клас**; той е отделна същност която е съставна част на външния клас-”**външният клас ИМА вътрешен клас**” и **допълнително**, **вътрешен клас ИМА независимо поведение на наследственост (“is-a”)!**

7a.9 Вътрешни класове

Вътрешният клас има пълен достъп до всички данни и методи от външния клас, дори те да са дефинирани като *private*

Забележка: Статичен вътрешен клас има достъп само до статични данни и методи от външния клас.

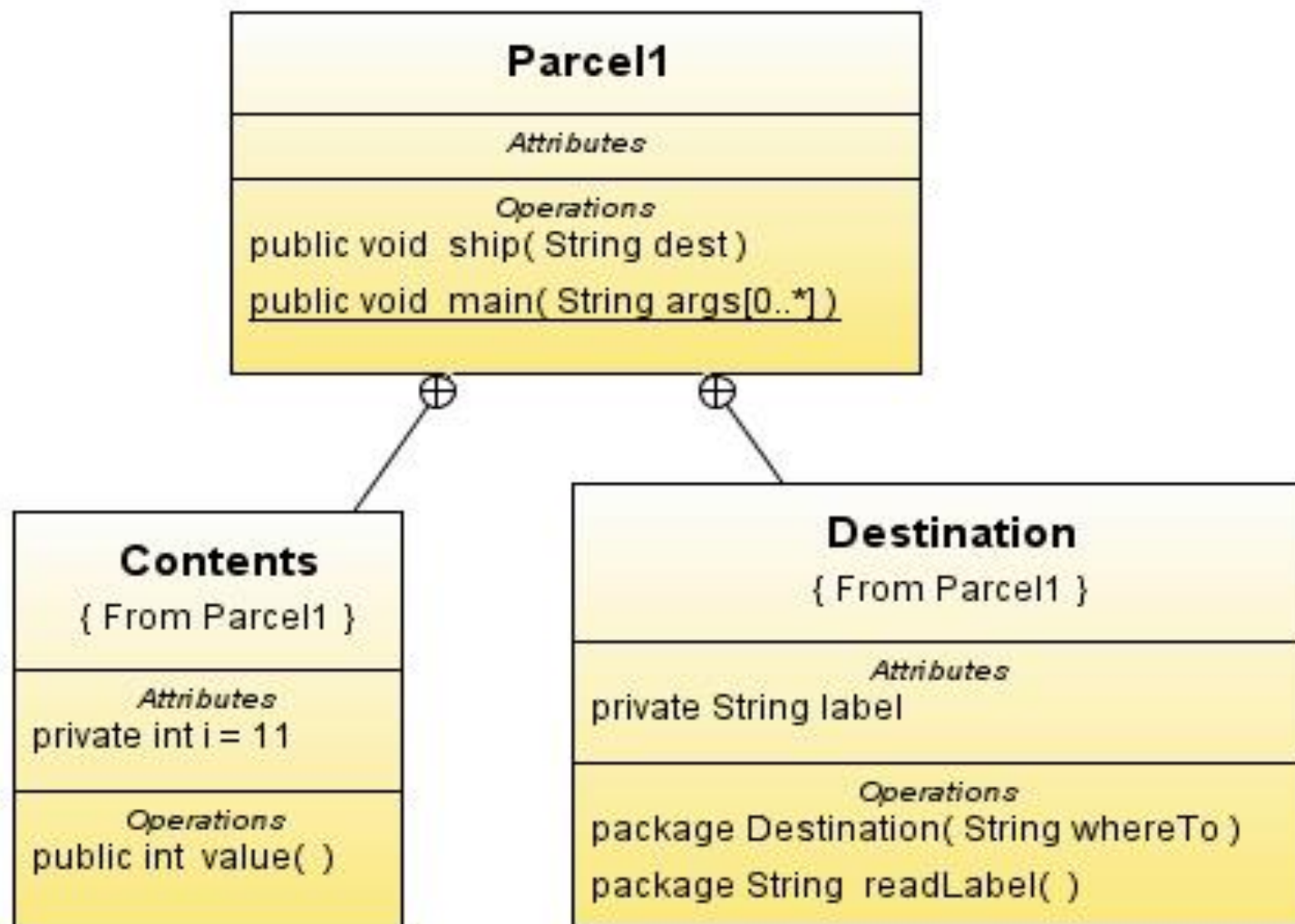
```
// Creating inner classes.
// by placing the class definition inside a
surrounding class
public class Parcel1 {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) {
            label = whereTo;
        }
        String readLabel() { return label; }
    }
    // Using inner classes looks just like
    // using any other class, within Parcel1:
    public void ship(String dest) {
        Contents c = new Contents();
        Destination d = new Destination(dest);
        System.out.println(d.readLabel());
    }
    public static void main(String[] args) {
        Parcel1 p = new Parcel1();
        p.ship("Tanzania");
    }
}
```

Вътрешен клас
използван в метод,
ship(), изглежда като
всички други
класове.

На практика
единствената разлика
е, че имената на
вътрешните класове
са вложени в клас
difference is that **the**
names are nested
within *Parcel1*

Е. Кръстев, OOP
Java, 2014.

UML diagram – class Parcel1



7а.10 Външен клас и методи за връзка с вътрешни класове

- Най- често, *външния class* трябва да има метод който връща референция към вътрешния class

7a.10 Външен клас и методи за връзка с ВЪТРЕШНИ КЛАСОВЕ

```
// Returning a reference to an inner class.
public class Parcel2 {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) {
            label = whereTo;
        }
        String readLabel() { return label; }
    }
    public Destination to(String s) {
        // return a reference to the inner class
        return new Destination(s);
    }
    public Contents cont() {
        // return a reference to the inner class
        return new Contents();
    }
}
```

```

public void ship(String dest) {
    Contents c = cont();
    Destination d = to(dest);

    System.out.println(d.readLabel());
}

public static void main
(String[] args) {
    Parcel2 p = new Parcel2();
    p.ship("Tanzania");
    Parcel2 q = new Parcel2();

    // Defining references to
    inner classes:

    Parcel2.Contents c = q.cont();

    Parcel2.Destination d =
q.to("Borneo");
}
}

```

При нужда да се направи **object** от вътрешен клас навсякъде **освен в non-static метод** на външен клас, **типът на такъв обект се задава като**

OuterClassName.InnerClassName

Вижте примера в метода
main().

7a.11 Върхешни класове и преобразуване нагоре- приложение

- Върхешните класове наистина са от полза при *преобразуване нагоре до базов клас*, и особено- до интерфейс *interface*. (*Пример* ?)
- Ефектът от *получаване на референция до interface* от обект който го реализира е по същество същият като *преобразуване нагоре* до *базов* клас.
- Това е така понеже *върхешния клас* – *реализиращ interface*- може да е напълно *невидим и недостижим за всеки клиент*, и това е удобно за реализиране на *скриване на реализацията на даден интерфейс*.
Всичко, което клиент програмиста получава е *референция до базов клас* или до *interface*.


```
//: separately defined interfaces
public interface Destination {
    String readLabel();
}
```

```
//: separately defined interfaces
public interface Contents {
    int value();
}
```

Тези интерфейси са напълно достъпни до клиент програмиста
(interface members are public)

Сега може да скриете реализацията на тези методи във вътрешни класове, както е показано в примера, чрез дефиниране на вътрешните класове inner classes private или protected и връщане на референция до тези вътрешни класове

Contents.java

и

Destination.java

дефинират
интерфейси,
предоставяни на
клиент програмиста

Е. Кръстев, OOP
Java, 2014.

При получаване на *референция* до **базов** class или *interface*, е възможно **даже да се скрие истинския тип на референцията**

За тестване на тази техника се изпълнява не *Parcel3*, а:

java Test

защото е необходимо, *main()* метода да е в **in a** **отделен class** за да се демонстрира недостъпност до *private* вътрешния клас *PContents*.

Е. Кръстев, OOP
Java, 2014.

```
// Returning a reference to an inner class.

public class Parcel3 {
    private class PContents implements Contents {
        private int i = 11;
        public int value() { return i; }
    }
    protected class PDestination implements
Destination {
        private String label;
        private PDestination(String whereTo) {
            label = whereTo;
        }
        public String readLabel() { return label; }
    }
    public Destination dest(String s) {
        return new PDestination(s);
    }
    public Contents cont() {
        return new PContents();
    }
}

class Test {
    public static void main(String[] args) {
        Parcel3 p = new Parcel3();
        Contents c = p.cont();
        Destination d = p.dest("Tanzania");
        // Illegal -- can't access private class:
        //! Parcel3.PContents pc = p.new PContents();
    }
}
```

Обобщение

Забележете: методите на външния клас връщат референции към интерфейсите имплементирани във вътрешните класове

PContents е **private**, и само, **Parcel13** може да има достъп до него.

Забележете: За разлика от външните класове, **вътрешните класове** могат да са **private**!

PDestination е **protected**, но само **Parcel13**, и класовете от пакета на **Parcel13** (понеже **protected** дава package access—т.е., **protected** е също “package” достъп), и производните на **Parcel13** имат достъп до **PDestination**.

Следователно, **клиент програмистът има ограничен достъп до вътрешните класове и методите, които те реализират.**

Използване на **private** вътрешен class дава възможност на class проектантът **напълно да забрани зависимости от типа при програмиране** а също и да **скрият напълно подробности за реализацията на даден метод**

7a.12 Вътрешни класове в методи – други начини за скриване на имплементацията

- Типичната употреба на вътрешни класове води до код лесно разбираем и използва.
- Допълнителна възможност е да се **създава вътрешен клас вътре в метод или произволен обхват**.
- Причини за това:
 - Логиката с вътрешни класове е да **имплементират някакъв interface** така че да може да върнете референция към този **интерфейс**.
 - Решавате сложен проблем и искате да създадете клас за решението му, **но не искате този клас да е общо достъпен**.

```
// This example shows the creation of an entire class
// within the scope of a method .
```

```
public class Parcel4 {
    public Destination dest(String s) {
        //inner class in a method
        class PDestination implements Destination {
            private String label;
            private PDestination(String whereTo) {
                label = whereTo;
            }
            public String readLabel(){ return label; }
        }
        return new PDestination(s);
    }
    public static void main(String[] args) {
        Parcel4 p = new Parcel4();
        Destination d = p.dest("Tanzania");
    }
}
// PDestination cannot be accessed outside of
dest( ).
// Upcasting occurs in the return statement—
// nothing comes out of dest( ) except
// a reference to Destination,
// the base class.
```

Това е **удобна** конструкция за малки по обем вътрешни класове.

Забележете, че интерфейс Destination трябва да е дефиниран, за да може да компилирате приложението

Е. Кръстев, OOP
Java, 2014.

Пример за **създаване на вътрешен клас в програмен блок** дефиниран с две фигурни скоби (отваряща и затваряща)

Ефектът е същият като при вътрешен клас дефиниран в метод

Е. Кръстев, OOP
Java, 2014.

```
// example shows how you can nest an inner class
// within any arbitrary scope.

public class Parcel5 {
    private void internalTracking(boolean b) {
        if(b) {
            class TrackingSlip {
                private String id;
                TrackingSlip(String s) {
                    id = s;
                }
                String getSlip() { return id; }
            }
            TrackingSlip ts = new TrackingSlip("slip");
            String s = ts.getSlip();
        }
        // Can't use it here! Out of scope:
        //! TrackingSlip ts = new TrackingSlip("x");
    }
    public void track() { internalTracking(true); }
    public static void main(String[] args) {
        Parcel5 p = new Parcel5();
        p.track();
    }
}

// class TrackingSlip is nested inside the scope of an if
statement
// It's not available outside the scope in which it is
defined
```

7a.13 Вътрешният клас е връзка с външния клас

- Вътрешните класове не служат само за скриване на типове и по- добра организация на кода
- При създаване на вътрешен клас, един **обект от този вътрешен клас има връзка с външния клас** който го е създал, и също **има пълен достъп до всички членове на външния клас без ограничения!**
- Така, **вътрешни класове имат право за достъп дори до `private` данните на външния клас**
- Един вътрешен клас има достъп до методите и данните на външния клас все едно те са негови данни и методи

```
// example shows how you can have an inner class
// links to the outer class
// Selector Holds a sequence of Objects.
```

```
interface Selector {
    boolean end();
    Object current();
    void next();
}

public class Sequence {
    private Object[] obs;
    private int next = 0;
    public Sequence(int size) {
        obs = new Object[size];
    }
    public void add(Object x) {
        if(next < obs.length) {
            obs[next] = x;
            next++;
        }
    }
    // the inner class definition
    // follows next
```

class **Sequence** има масив от **Object** елементи и вътрешен клас, който реализира методи за обработка на този масив.

Методът **add()** добавя нов **Object** в края на sequence (ако има място).

Нека **interface Selector** декларира методи за достъп до елементите на обекти от **Sequence** (например, проверява дали сте в края **end()**, да работите с текущия **current()** **Object**, или да преминете към следващия **next()** **Object** елемент на обект от **Sequence**). Понеже **Selector** е **interface**, той може да се реализира по различен начин от вътрешни класове, и много методи, могат да използват този интерфейс за създаване на пораждащ код

Е. Кръстев, OOP
Java, 2014.


```
// example shows how you an inner class links to the
// outer class ... continued
private class SSelector implements Selector { // the
// inner class definition
    int i = 0;
    public boolean end() {
        return i == obs.length; // access outer class data
    }
    public Object current() {
        return obs[i];
    }
    public void next() {
        if(i < obs.length) i++;
    }
}

public Selector getSelector() {
    //returns inner class reference
    return new SSelector();
}

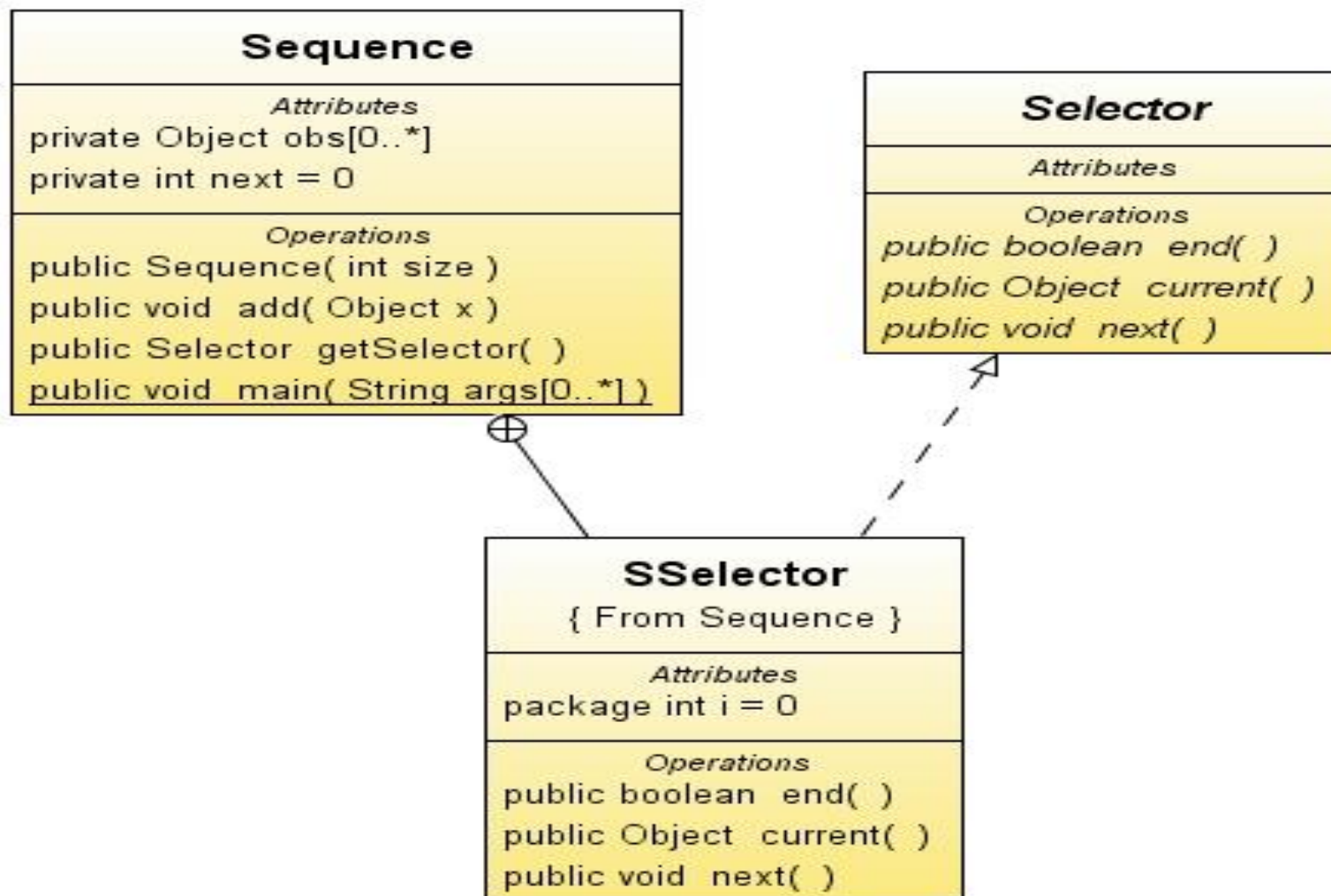
public static void main(String[] args) {
    Sequence s = new Sequence(10);
    for(int i = 0; i < 10; i++)
        s.add(Integer.toString(i));
    Selector sl = s.getSelector();
    // inner class object manages outer class members
    while(!sl.end()) {
        System.out.println(sl.current());
        sl.next();
    }
}
```

SSelector е private class реализиращ методите на **interface Selector** functionality

В **main()**, се създава **Selector** обект(с преобразуване нагоре на **SSelector** до **interface Selector**) и той се използва за добавяне на String object-и.

Е. Кръстев, OOP
Java, 2014.

UML diagram- Selector



Задачи

Задача 1

Напишете в даден *package interface A* с поне един метод .

Напишете и *class B* в друг package. Добавете *protected* вътрешен клас който реализира *interface A*. В трети package, наследете *class B* вътре в метод и върнете обект от *protected* вътрешния клас преобразувайки го до *interface A* при *return*

Тествайте така създадените класове

Задача 2

Напишете *SelectionSort* class и скрийте имплементацията във вътрешен клас. Вътрешният клас да реализира interface *Sortable* с метод

boolean greater(int i, int j)

Който връща резултатът от сравняване на i-тия и j-тия елемент на масива

Масивът от цели числа за сортиране да е данна на външния клас. Да има и метод за извеждане (*get*) на външния клас за проверка на сортирането