Лекция 11

Inner and anonymous classes

Основни теми

Въведение

Интерфейсите в Java позволяват да се реализира множествено наследяване

Може да се разглеждат като “чист” abstract **class. Съдържат:**

- **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.**

Въведение

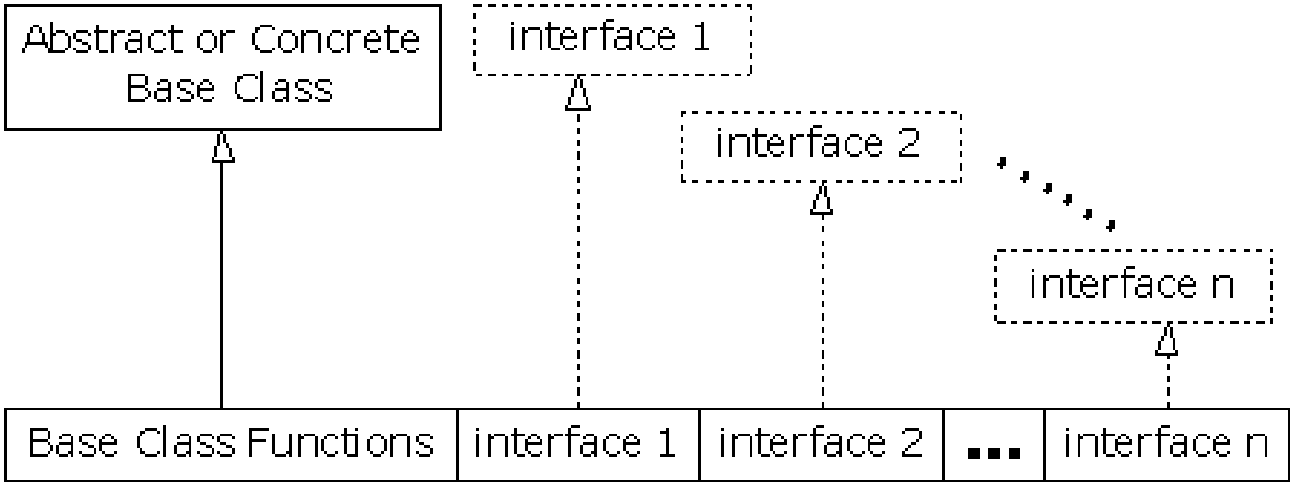
След имплементирането на **interface**, тази имплементация става конкретен клас, който може да се разширява според релацията "*обект A* ***е*** *обект B*"

При имплементацията ***implement*** на **interface**, методите на **interface** трябва да се ***public*** **(те са *public по декларация в* interface)**

Пропускането на ***public*** води до предефиниране на тези методи с пакетен "***приятелски***" достъп и с това се намалява нивото на достъп при наследственост, което е забранено.

Този синтаксис не позволява имплементиране на метод на интерфейс освен като ***public*** **метод.**

Въведение



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() {}
}
```

Резюме

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

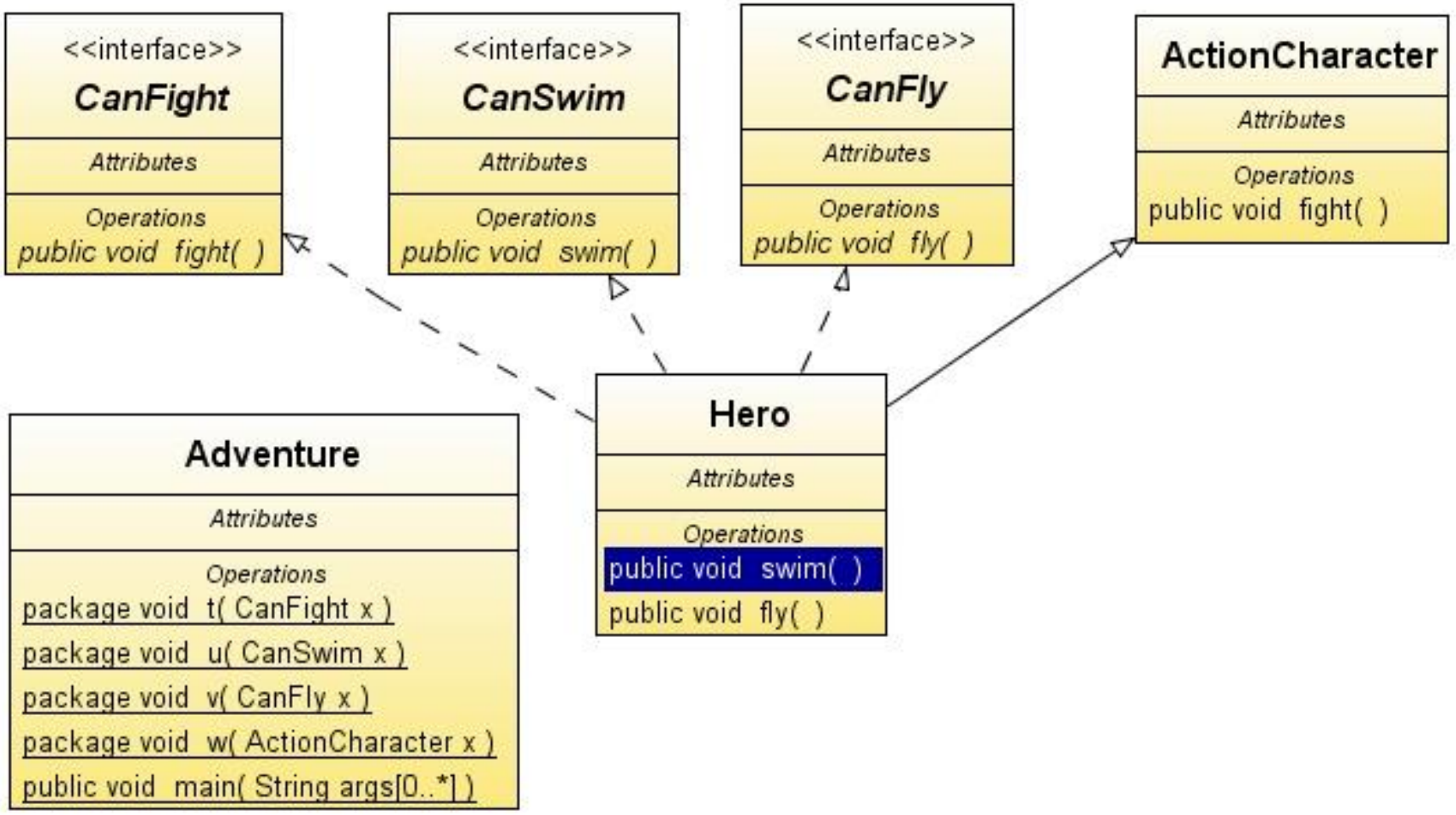
Резюме

```
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



Въведение

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 наследява от базовия клас `ActionCharacter` и това е достатъчно да се създават обекти от `class Hero`.

Въведение

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`***.

При този синтаксис не е възможно да имплементираме по различен начин един и същ метод, деклариран в различни интерфейси.

11.1 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**.

11.1 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**.

11.1 Вътрешни класове

Синтаксис в Java за 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.

Резюме

```
// 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();
    }
    private 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);
        }
    }
```

Пример:

Създаване на обекти от вътрешен клас

Достъп на вътрешен клас до данни и методи от външния клас

Само вътрешни класове могат да бъдат static. . Използват се за групиране на класове.

Само вътрешни класове могат да бъдат private.

11.1 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**

11.1 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 различени 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”)**!

11.1 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.

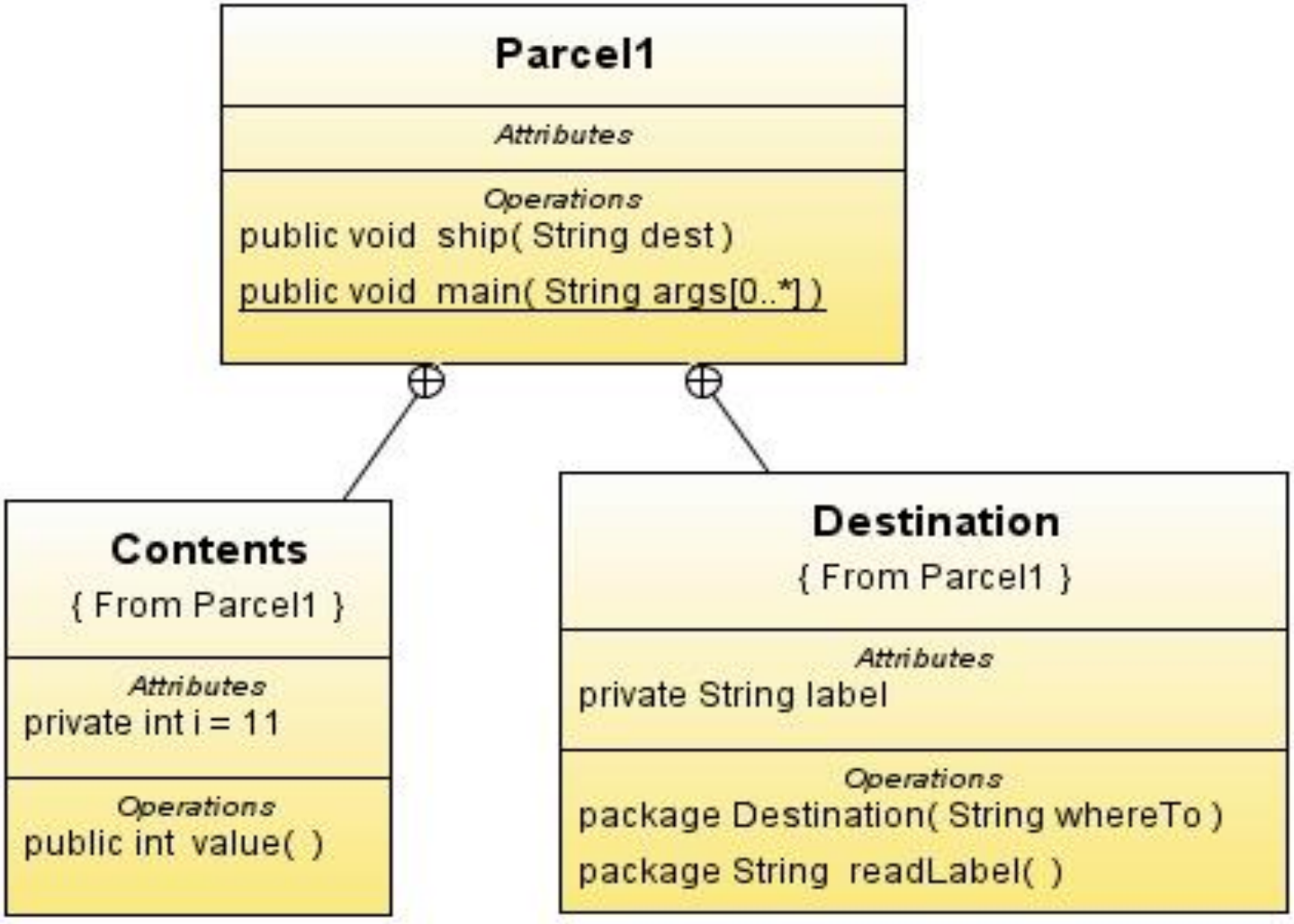
Резюме

```
// 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*

UML diagram – class Parcel1



11.2 Външен клас и методи за връзка с вътрешни класове

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

11.2 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();
  }
```


11.2 Външен клас и методи за връзка с вътрешни класове

Important

If the class is declared `public`, then the default constructor is implicitly given the access modifier `public`; if the class is declared `private`, then the default constructor is implicitly given the `access` modifier `private`; otherwise, the default constructor has the `default` access implied by no access modifier

(Oracle, The Java® Language Specification, Java SE 14 Edition

Chapter 6. Names)

Резюме

```
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( ).

11.3 Вътрешни класове скриване на имплементацията

Вътрешните 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 по същество е същият, както **преобразуване нагоре** до ***базов*** клас.

Това е така, понеже вътрешният 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 програмистът получава, е **референция до *базов клас*** или до ***interface***.

Резюме

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

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

Contents.java и *Destination.java* дефинират интерфейси, предоставяни на клиент програмиста

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

```
// 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();
  }
}
```

Резюме

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*.**

Обобщение

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

PContents е **private** , и само, ***Parcel3*** може да има достъп до него.

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

PDestination е **protected**, но само ***Parcel3***, и класовете от пакета на ***Parcel3*** (понеже **protected** дава package access—т.е., **protected** е също "package" достъп), и производните на ***Parcel3*** 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

11.3.1 Вътрешни класове в методи

Типичната употреба на вътрешни класове води до код лесно разбираем и използва.

Вътрешен клас може да се **създава вътре в метод .**

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.

Резюме

```
// 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

11.3.2 Вътрешни класове в блок от код

Допълнителна възможност е да се **създава вътрешен клас вътре в блок от код**.

Резюме

```
// 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
```

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

11.4 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 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 д аизползват този интерфейс за създаване на пораждащ код

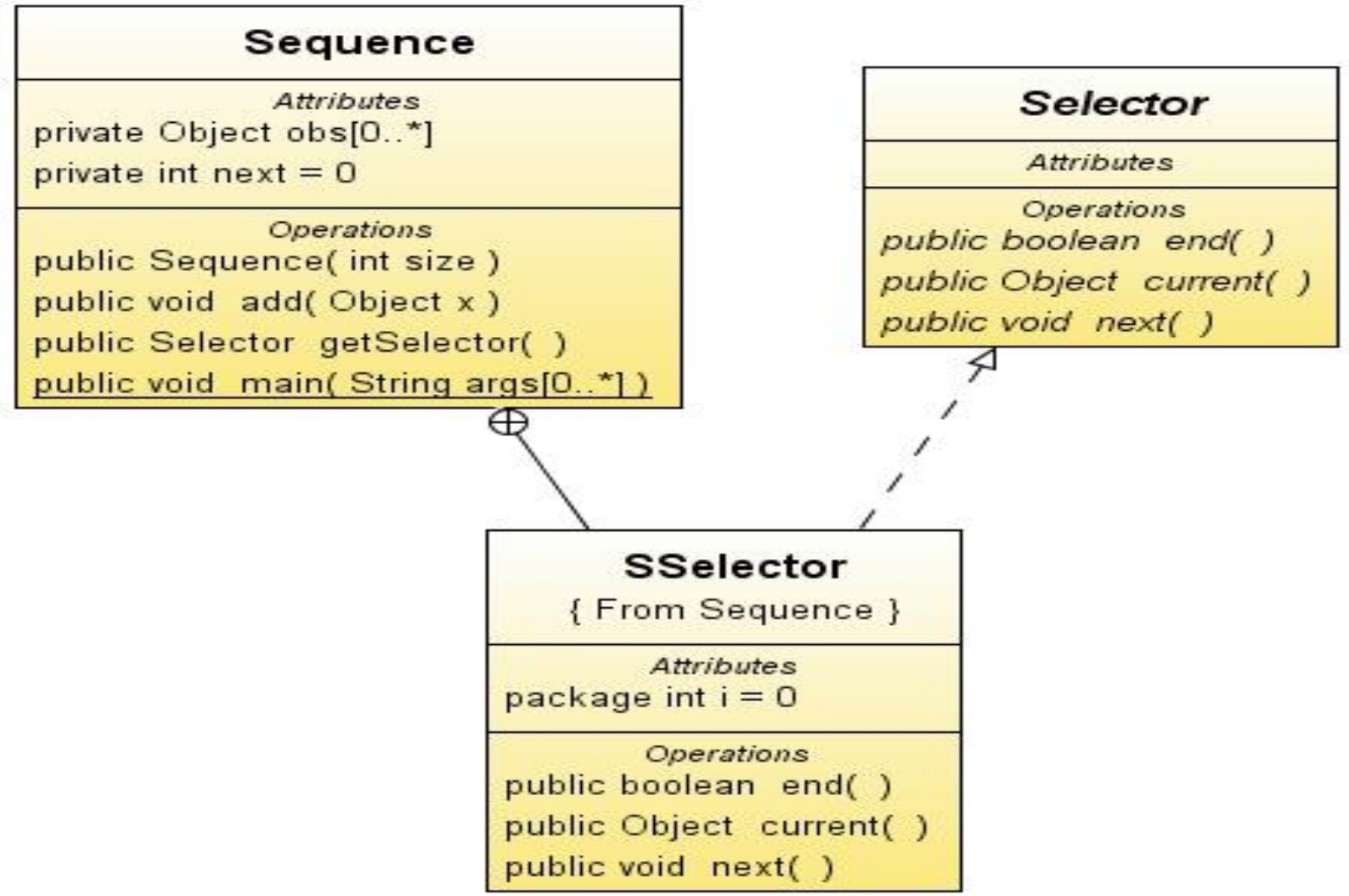
Резюме

```
// 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.

UML diagram- Selector



11.5 Наследственост при вътрешни класове-примери

Конструкторът на вътрешен клас трябва да има референция към името на външния клас. Затова, **при онаследяване от единствен вътрешен клас**, тази референция трябва да е явно записана в производния клас (example A)

Вътрешните класове са **изцяло отделни същности**, всяко със собствена област от имена от това на външния клас (example B)

Явно наследяване от вътрешен клас (example C)

Резюме

```
// example A shows how to inherit
// an inner class only.

class WithInner {
  class Inner {} // inner class to inherit
}
public class InheritInner  extends
WithInner.Inner {
  // extending the inner class
  //! InheritInner() {}
  // Won't compile
  InheritInner( WithInner wi) {
    wi.super(); // reference to the outer
class required!!
  }
  public static void main(String[] args) {
    WithInner wi = new WithInner();
    InheritInner ii = new InheritInner(wi);
  }
 }
}
```

Използвайте синтаксис `enclosingClassReference.super();`

в конструктора на производния клас.

Забележете:
`InheritInner` онаследява само вътрешния клас, но не и външния клас

Резюме

```
// example A shows how to inherit
// an inner class only.

class WithInner {
  static class Inner {} // inner class to
inherit
}
public class InheritInner  extends
WithInner.Inner {
  // extending the static inner class

  // compiles
  InheritInner( ) {
     super(); // reference to the outer
class is not required!!
  }
  public static void main(String[] args) {
    WithInner wi = new WithInner();
    InheritInner ii = new InheritInner();
  }
 }
}
```

Използвайте синтаксис `enclosingClassReference.super();`

в конструктора на производния клас.

Забележете: `InheritInner` онаследява само вътрешния клас, но не и външния клас

```
// example B shows how inner classes behave in inheritance
// An inner class cannot be overriden
// like a method.

class Egg {
  private Yolk y;
  protected class Yolk {
    public Yolk() {
      System.out.println("Egg.Yolk() ");
    }
  }

  public Egg() {
    System.out.println("New Egg() ");
    y = new Yolk();
  }
}

public class BigEgg extends Egg {
  public class Yolk {
  // cannot override inner class Yolk in class Egg
    public Yolk() {
      System.out.println("BigEgg.Yolk() ");
    }
  }
  public static void main(String[] args) {
    new BigEgg();
  } // What is the output? Why?
}
```

Конструкторът по подразбиране **BigEgg()** Се генерира от компилаторът, и това извиква конструкторът по подразбиране на базовия клас **Egg()** което от своя страна извиква конструкторът по подразбиране **Egg.Yolk()**

а не конструкторът по подразбиране на **BigEgg.Yolk()**

The output is:

```
New Egg()
Egg.Yolk()
```

Резюме

```
// example C shows explicit inheritance of inner classes
// Inheritance from the outer class and its inner classes
class Egg2 { // base class
  protected class Yolk { // inner class in the base class
    public Yolk() {
      System.out.println("Egg2.Yolk()");
    }
    public void f() {
      System.out.println("Egg2.Yolk.f()");
    }
  }
  private Yolk y = new Yolk();
  public Egg2() {
    System.out.println("New Egg2()");
  }
  public void insertYolk(Yolk yy) { y = yy; }
  public void g() { y.f(); }
}
public class BigEgg2 extends Egg2 {// derived outer class
  public class Yolk extends Egg2.Yolk{
                                    //derived inner class
    public Yolk() {
      System.out.println("BigEgg2.Yolk()");
    }
    public void f() {
      System.out.println("BigEgg2.Yolk.f()");
    }
  }
  public BigEgg2() { insertYolk(new Yolk()); }

  public static void main(String[] args) {
    Egg2 e2 = new BigEgg2();// upcast to Egg2
    e2.g();// call the overridden version of f()
  }
}
```

BigEgg2.Yolk е производен на Egg2.Yolk и презарежда методите му.
Методът insertYolk( ) позволява BigEgg2 да преобразува един от обектите си до своите Yolk обекти в референцията y на Egg2.
Така, когато g( ) извиква y.f( ) презареденатa версия на f( ) се използва.
Повторното извикване на Egg2.Yolk( ) е на базовия клас конструктор в BigEgg2.Yolk. Презареденатa версия на f( ) се използва при извикване на g( ).

The output is:

```
// super()
Egg2.Yolk()
New Egg2()
// insertYolk(new Yolk());
Egg2.Yolk()
BigEgg2.Yolk()
// e2.g()
BigEgg2.Yolk.f()
```

11.6 Closures и Callbacks

closure е многократно изикван обект, който може да съхранява информацията в обхвата в който е създаден

Вътрешният клас и **пример** за **closure**, понеже не само има достъп до всеки член на външния клас, но и поддържа референция до този клас с разрешение да манипулира членовете му

При ***callback***, на даден обект дава на някой друг обект **частична информация**, която му **позволява да се обади обратно** на първия клас в последващ момент

Резюме

```
// Using inner classes for callbacks
// 1. Describe the service(services)
interface Incrementable {
  void increment(); // Service description
}
```

Callee1 очевидно е най-простото решение за реализиране на interface

Резюме

```
// Using inner classes for callbacks
// 2. Define implementations of the services
// in inner classes
// Hide their implementation

class Callee {
  private int i = 0;
  private void incr() {
    i++;
    System.out.println(i);
  }
  private class Closure1  implements Incrementable
  {
    public void increment() {  }
  }
  public Incrementable getCallbackReference1() {
    return new Closure();
    //returns a reference to the inner class
  }
  private class Closure2  implements Incrementable
  {
    public void increment() { incr(); }
  }
  public Incrementable getCallbackReference2() {
    return new Closure();
    //returns a reference to the inner class
  }
}
```

Callee имплементира интерфейса във вътрешни класове. Всеки от тях представлява различна имплементация на услугите, описани в интерфейса Incrementable

Заберлежете, че всичко в Callee е *private* освен getCallbackReference1( ) getCallbackReference2( )

Който получи референция към Incrementable може да извика логически издържания метод increment( ) и нищо друго

Резюме

```
// Using inner classes for callbacks
// 3. Define the callback implementation
class Caller {
  private Incrementable callbackReference;
  public Caller(Incrementable cbh) {
    callbackReference = cbh;
  }
  void goOne() {
    callbackReference.increment();
  }
  void goTwo(Incrementable callback) {
    callback.increment();
  }
}
// 4. Test the callback implementation
public class CallbackTest {
  public static void main(String[] args) {
    Callee c1 = new Callee();
    Caller caller1 = new
                     Caller(c1.getCallbackReference1());
    caller1.goOne();
    caller1.goTwo(c1.getCallbackReference1());
    caller1.goTwo(c1.getCallbackReference2());
  }
}
```

class Caller взима за аргумент референция Incrementable в конструктора си (макар че, референция към callback обекта може да стане в произволен друг момент) и впоследствие използва тази референция за "call back" обратно извикване на обекта от вътрешен клас на class Callee

11.7 Създаване и изпълнение на потребителски дефинирани събития

Event functionality is provided by **three interrelated elements**:

1. a class that provides event data,
2. an event interface,
3. the class that raises the event.
4. The class that consumes the event

11.7.1 Създаване и изпълнение на потребителски дефинирани събития

Assume event name is `Action`:

1. A class that provides event data

`ActionEventArgs` or `ActionEvent`

2. An event interface

`ActionEventHandler` or `ActionListener`

3. The class that raises the event.

`ActionEventSource` or any other custom defined name

3. The class that consumes the event

`ActionEventConsumer` or any other custom defined name

Резюме

```
public interface ActionListener { // defines action event
// This is just a regular method so it can return something
// or take arguments if you like.
    public void actionPerformed (ActionEvent args) ;
}

public class ActionEvent {
// This is a class that defines the event arguments
// takes arguments as you like.
    private MyArg arg; // some arguments

    public ActionEvent (MyArg arg) {
       setMyArg(arg);
    }

    public MyArg getMyArg() {
     return arg; // return a copy of this.arg
     }

    private void setMyArg(MyArg value) {
     // validate and set arg;
     this.arg = value;
    }

}
```

Резюме

```
public class ActionEventSource  // event source (callback)
{
    private ActionListener ie;
    // ActionListener may be also public available!
    private boolean onAction;
    public EventSource(ActionListener event)
    {
    // Save the event object for later use.
    ie = event;
    // Nothing to report yet.
    onAction = false;
    }
    //... Register event handler
    public void addActionListener(ActionListener al){
      ie = al;
    }
    public void doWork ()
    {
    // Check the predicate, which is set elsewhere.
    if (onAction ) // execute callback
        {
        // Signal the even by invoking the interface's method.
        // Create MyArg object and pass it to the event handler
        if (ie != null) // event is handled!!!
          ie.actionPerformed( new ActionEvent(new MyArg()));
          // the event is fired!
        }
    //...
    }
    // ...
}
```

Ако ActionListener ie е public, **то** ie **може да се инициализира директно в** CallMe **класовете на** ActionListener **обект, който е инстанция на вътрешен клас в клас** CallMe

Резюме

```
//class that handles the event, ActionListener implementor
public class CallMe implements ActionListener
{
    private ActionEventSource en;
    // component that fires the event

    public CallMe ()
    {
    // hook the event handler to the event source.
      en = new ActionEventSource(new OnAction());

    }
    // Implement the ActionListener in inner class
    private class OnAction implements ActionListener
    {

      // Define the actual handler for the event.
       public void actionPerformed (EventArgs args)
      {
       //  Something really interesting must have occurred!
       // get args
       // and
       // Do something...
      }
    }
    //...
}
```

11.8 Анонимни вътрешни класове

Конструкция за скриване на имена и организация на код

Използва се за описване на събития изискващи малко код за описанието им

```
//: c08:Parcel6.java
// A method that returns an anonymous inner class.
public class Parcel6
 {
    public Contents cont()
    {
      return new Contents()
       {
         private int i = 11;
         public int value()
           {
              return i;
           }
       };
   // Semicolon required in this case
   }
   public static void main(String[] args)
     {
       Parcel6 p = new Parcel6();
       Contents c = p.cont();
     }
 }

// Contents is created using a default constructor
// This is required for the above to compile
interface Contents {
   int value();
}
/* or is required for the above to compile
class Contents {
  int value(){ return 0;}
}
*/
```

Метод *cont( )* комбинира връщане на стойност и дефиниране на клас, който описва връщаната стойност!

Допълнително, класът е анонимен *anonymous*– няма име.

По друг начин казано: "*създай обект от безименен клас който е производен на Contents.*"

Това е *съкратен запис* на:

```
class MyContents implements Contents
{
  private int i = 11;
  public int value()
   {
    return i;
   }
 }
 return new MyContents();
```

```
//: c08:Parcel6.java
// A method that returns an anonymous inner class.
public class Parcel6
 {
    public Contents cont()
    {
      class MyContents implements Contents
      {
         private int i = 11;
         public int value() { return i; }

       }

     return new MyContents();
    }
   // Semicolon required in this case
   }
   public static void main(String[] args)
     {
       Parcel6 p = new Parcel6();
       Contents c = p.cont();
     }
 }

// Contents is created using a default constructor
// This is required for the above to compile
interface Contents {
   int value();
}
/* or is required for the above to compile
class Contents {
  int value(){ return 0;}
}
*/
```

Метод *cont( )* комбинира връщане на стойност и дефиниране на клас, който описва връщаната стойност!

Допълнително, класът е анонимен *anonymous*– няма име.

По друг начин казано: "*създай обект от безименен клас който е производен на Contents.*"

Това е *съкратен запис* на:

```
class MyContents implements Contents
{
  private int i = 11;
  public int value()
   {
    return i;
   }
 }
 return new MyContents();
```

```
//: c08:Parcel7.java Using an argument
// The base class needs a constructor
// with an argument.
public class Parcel7
 {
  private int count = 2;
  public Wrapping wrap(int x,
                       final String dest)
   {
    // Base constructor call, x is NOT final!
    return new Wrapping(x)
      { // directly access local var dest
        private String label = dest;
        public int value()
         {// directly access datamember count
           return super.value() * count;
         }
    }; // Semicolon required
  }
  public static void main(String[] args)
  {
   Parcel7 p = new Parcel7();
   Wrapping w = p.wrap(10);
  }
 }
 public class Wrapping
 {
  private int i;
  public Wrapping(int x) { i = x; }
  public int value() { return i; }
}
```

Резюме

Анонимен клас с конструктор- предаваме аргумент на базовия коструктор, представен тук като x в *new Wrapping(x)*.

Анонимният клас не може да има конструктор където да използвате както обикновено *super( )*

Ако искате да използвате обект в анонимен вътрешен клас, където обектът е реферирана с локална променлива, дефинирана извън този клас, то компилаторът изисква тази променлива да.

(примерно, dest е final в списъка от аргументи на метода дефиниращ анонимния клас!)

Анонимният клас има пълен достъп до данните на външния клас (пример с *count* !)

11.8 Анонимни вътрешни класове

Effectively final local variable.

Anonymous classes can access **instance** and **`static`** variable of the outer class. This is called ***variable capture***. Instance and **`static`** variables may be **used and changed without restriction** in the body **of an anonymous class**

The use of **local variables**, however, is more **restricted**: **capture of local variables is not allowed unless they are *effectively final* *in* *JDK 8***, i.e. once a local variable is captured inside an anonymous class or inner class, its initial value cannot be changed even though it is not declared as **`final`**. Missing to declare an **effectively `final`** variable as **`final`** would not cause a compilation failure

```
public class CaptureTest {
    private static int count = 5;// captured inside the anonymous class
    private String str = "Captured string";
    public void method(String dest) {//dest is effectively final
        int localVar = 5;             //localVar is effectively final
        Wrapping wr = new Wrapping(localVar) {// localVar is captured here
            private String label = dest;      // dest is captured here
            public int value() {
                //dest = "";//    // not allowed, dest variable is captured!!
                return super.value() * count;
            }
            public String toString(){
                return str;
            }
        }; // Semicolon required
        // localVar = 7;// not allowed, localVar variable is captured!!
        count  = 8; // allowed
        str    = "New captured string";//allowed
    }
}
class Wrapping {
    private int i;
    public Wrapping(int x) {
        i = x;
    }
    public int value() {
        return i;
    }
}
```

```
//: c08:Parcel9.java
// Using "instance initialization" to perform
// construction on an anonymous inner class.

public class Parcel9 {
  public Destination dest(final String dest,
                          final float price)
  {
   return new Destination()
   {
      private int cost;
      // Instance initialization for each object:
      {
        cost = Math.round(price);
        if(cost > 100)
           System.out.println("Over budget!");
      }
      private String label = dest;

      public String readLabel() { return label; }
   };
  }

  public static void main(String[] args)
  {
    Parcel9 p = new Parcel9();
    Destination d = p.dest("Tanzania", 1011.395F);
    System.out.println(d.readlabel());
  }
}
```

Резюме

За инициализация на данни в анонимен клас се използва блок от код, вместо конструктор за общо ползване.

Тук

```
public interface Destination {
 String readLabel();
}
```

Какво се изпълнява и защо, ако

```
public class Destination {
 String readLabel()
    {
    return "SOS";
    }
}
```

11.8 Анонимни вътрешни класове пример

```
/**
 * Superclass can be class or interface
 * Notice how call to superclass constructor is done, if parameters are needed.
 * param1, param2, etc are parameters to the superclass constructor,
 * ala super(param1, param2, etc)
 */
SuperClass instance = new SuperClass(param1, param2, etc) {
    FieldType field;  // private field for internal use

    // "Initializer block" = anonymous constructor
    {
        // runs after the superclass constructor
        // No need to pass in parameters -- why?
     }
    /**
     * Overriden method of superclass/interface provides public behavior
     */
    public ReturnType overridenMethod(Type1 t1, Type2 t2) {
        // desired code
    }
    /**
     * private method for internal use.
     */
    Stuff internalMethod(){
       // code
    }
};
```


11.8 Анонимни вътрешни класове пример

Initializer block

Блокът от код се изпълнява след изпълнението на конструктора на базовия клас, когато анонимния клас наследява клас. Този блок се използва за инициализиране на данните на анонимния клас.

Заключване на локални променливи:

За заобикаляне на заключването ползвайте "***one-element-array***":

Декларирайте локалната променлива като масив с един елемент
```
MyType[] localVar = {initialValue};
```

В анонимния клас работете с елемента на масива, а не със самия масив т.е. работете с `localVar[0]`.

Тогава е правилно да пишете
```
localVar[0] = newValue;
```

Понеже променяте съдържанието на , а не самата
`localVar` променлива.

11.9 Вътрешни класове и обработка на събития

Application framework е class или съвъкупност от класове създадени за решаване определена приложна задача.

Application framework се използва като се наследи от един или повече от класове и се предефинират методите на наследените класове.

Control framework е частен случай на *Application framework* в случай на задача изискваща реагиране на събития и се нарича **система, управлявана от събития**.

Едно от най- важните й приложения important е в програмиране на graphical user interface (GUI), която изцяло управлявана от събития

11.9 Вътрешни класове и обработка на събития

Пример: Система за управление чиято **задача е да изпълнява събития**, които са готови “ready” за изпълнение.

Под “ready” ще говорим в смисъл на **готовност за изпълнение по отношение на системното време.**

Примерът дава **общ скелет за изграждане на такъв тип система** за управление.

Използва се **abstract** class вместо **interface** за описване на произволно управляващо събитие поради нужда изпълнението да се синхронизира със системното време

Резюме

```
//: c08:controller:Event.java
// The common methods for any control event.
package c08.controller;

abstract public class Event {
  private long evtTime;
  public Event(long eventTime) {
    evtTime = eventTime;
  }
  public boolean ready() {
    return System.currentTimeMillis() >= evtTime;
  }
  abstract public void action();
  abstract public String description();
}
```

11.9 Вътрешни класове и обработка на събития

Конструкторът на **class Event** капсулира времето на създаване на събитието с оглед създаване на определена наредба в изпълнението на събития **Event**

Методът **ready( )** казва дали е време да се изпълни събитието. Допълнително, **ready( )** би могло да се предефинира в производен клас на **Event** и да се използва друг критерий за готовност

Методът **action( )** е методът който се изпълнява след като обектът **Event** е **ready( )**, и **description( )** дава текстово описание на обектът **Event**.

11.9 Вътрешни класове и обработка на събития

Следващите класове съдържат същинската част на система за управление на събития. Първият клас **EventSet** е помощен – целта му е на дефинира **Event** обектите.

EventSet примерно е контейнер за 100 **Event**s.

index се използва да следи за свободна памет за следващо събитие,

next реферира следващото събитие **Event** в списъка с метода **getNext( )**,

Обектите **Event** се премахват от списъка чрез **removeCurrent( )**) след като се изпълнят и **getNext( )** може да открие “дупки” в списъка при преминаване към следващо събитие

Резюме

```
// Along with Event, the generic: Controller.java
// framework for all control systems:
package c08.controller;
// This is just a way to hold Event objects.
class EventSet {
  private Event[] events = new Event[100];
  private int index = 0;
  private int next = 0;
  public void add(Event e) {
    if(index >= events.length)
      return; // (In real life, throw exception)
    events[index++] = e;
  }
  public Event getNext() {
    boolean looped = false;
    int start = next;
    do {
      next = (next + 1) % events.length;
      // See if it has looped to the beginning:
      if(start == next) looped = true;
      // If it loops past start, the list
      // is empty:
      if((next == (start + 1) % events.length)
         && looped)
        return null;
    } while(events[next] == null);
    return events[next];
  }
  public void removeCurrent() {
    events[next] = null;
  }
}
```

Резюме

```
public class Controller {
  private EventSet es = new EventSet();
  public void addEvent(Event c) { es.add(c); }
  public void run() {
    Event e;
    while((e = es.getNext()) != null) {
      if(e.ready()) {
        e.action();
        System.out.println(e.description());
        es.removeCurrent();
      }
    }
  }
}
```

11.9 Вътрешни класове и обработка на събития

- Забележете **`removeCurrent( )`** инициализира референцията към събитието като **`null`**. Така ресурсите заемани от това събитие се освобождават.
- **`Controller`** осъществява истинската работа по управление на събитията
- Използва контейнера **EventSet** за съхраняване на обектите **`Event`** като метод **`addEvent( )`** позволява д асе добавят нови събития към списъка
- Най- важен е методът **`run( )`, който обхожда списъка в EventSet** търсейки за обект **`Event`** готов **`ready( )`** да се изпълни. Когато обект **`Event`** е **`ready( )`**, се изпълнява **`action( )`** , отпечатва се **`description( ),`** и обектът **`Event`** се премахва от списъка

11.9 Вътрешни класове и обработка на събития

- **Най- важното дотук е абстрактната дефиниция за това какво един обект `Event` прави.**
- **Основен принцип в моделирането “*отделяне на нещата, които са неизменни от тези, които се променят*.”**
- **<u>На практика</u> изразяваме различни действия със създаване на различни производни класове на `class Event`.**

11.9 Вътрешни класове и обработка на събития

Това е ролята на вътрешните класове – те позволяват:

- **Да се създаде изцяло нова реализация на система за управление в един единствен клас като се капсулира в нея всичко необходимо за нейната реализация**
- **Използваме вътрешните класове за изразяване на различните видове действия на `action()` за решаване на задачата. В допълнение използваме `private` вътрешни класове, чиято имплементация остава скрита.**
- **Вътрешните класове осигуряват достъп до членовете на външния клас и кодът остава логически компактен.**

11.10 Приложение на Система за управление на събития

Това е ролята на вътрешните класове – те позволяват:

- **Да се създаде изцяло нова реализация на система за управление в един единствен клас като се капсулира в нея всичко необходимо за нейната реализация**
- **Използваме вътрешните класове за изразяване на различните видове действия на action( ) за решаване на задачата. В допълнение използваме private вътрешни класове, чиято имплементация остава скрита.**
- **Вътрешните класове осигуряват достъп до членовете на външния клас и кодът остава логически компактен.**

11.10 Приложение на Система за управление на събития

Да разгледаме частно приложение на система за управление на събития в Парник , където действията са: включване на светлина, вода, включване на термостат, включване на звуков сигнал, и рестартиране на системата.

Вътрешните класове реализират различна версия на един и същ базов клас, `Event`, като това става в един единствен клас. Всяко едно събитие се дефинира във вътрешен клас, който онаследява от `class Event` и предефиниране на метод `action()`.

Както в общия случай на приложна система class `GreenhouseControls` е производен на клас `Controller`

Начало на класът за управление на събития в парник

Пример за дефиниране на събитие за включване на светлина

```
//: c08:GreenhouseControls.java
// This produces a specific application of the
// control system, all in a single class. Inner
// classes allow you to encapsulate different
// functionality for each type of event.
import c08.controller.*;

public class GreenhouseControls
                    extends Controller
{
  private boolean light = false;
  private boolean water = false;
  private String thermostat = "Day";

  private class LightOn extends Event {
    public LightOn(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here to
      // physically turn on the light.
      light = true;
    }
    public String description() {
      return "Light is on";
    }
  }
```

Резюме

```
private class LightOff extends Event {
  public LightOff(long eventTime) {
    super(eventTime);
  }
  public void action() {
    // Put hardware control code here to
    // physically turn off the light.
    light = false;
  }
  public String description() {
    return "Light is off";
  }
}
private class WaterOn extends Event {
  public WaterOn(long eventTime) {
    super(eventTime);
  }
  public void action() {
    // Put hardware control code here
    water = true;
  }
  public String description() {
    return "Greenhouse water is on";
  }
}
```

Резюме

```
private class LightOff extends Event {
  public LightOff(long eventTime) {
    super(eventTime);
  }
  public void action() {
    // Put hardware control code here to
    // physically turn off the light.
    light = false;
  }
  public String description() {
    return "Light is off";
  }
}
private class WaterOn extends Event {
  public WaterOn(long eventTime) {
    super(eventTime);
  }
  public void action() {
    // Put hardware control code here
    water = true;
  }
  public String description() {
    return "Greenhouse water is on";
  }
}
```

Резюме

```
private class WaterOff extends Event {
   public WaterOff(long eventTime) {
     super(eventTime);
   }
   public void action() {
     // Put hardware control code here
     water = false;
   }
   public String description() {
     return "Greenhouse water is off";
   }
 }
 private class ThermostatNight extends Event {
   public ThermostatNight(long eventTime) {
     super(eventTime);
   }
   public void action() {
     // Put hardware control code here
     thermostat = "Night";
   }
   public String description() {
     return "Thermostat on night setting";
   }
 }
 private class ThermostatDay extends Event {
   public ThermostatDay(long eventTime) {
     super(eventTime);
   }
   public void action() {
     // Put hardware control code here
     thermostat = "Day";
   }
   public String description() {
     return "Thermostat on day setting";
   }
 }
```

Резюме

```
// An example of an action() that inserts a
// new one of itself into the event list:
private int rings;

private class Bell extends Event {
  public Bell(long eventTime) {
    super(eventTime);
  }
  public void action() {
    // Ring every 2 seconds, 'rings' times:
    System.out.println("Bing!");
    if(--rings > 0)
      addEvent(new Bell(
        System.currentTimeMillis() + 2000));
  }
  public String description() {
    return "Ring bell";
  }
}
```

Резюме

Пример за генериране "пакет" от всички събития

```
private class Restart extends Event {
  public Restart(long eventTime) {
    super(eventTime);
  }
  public void action() {
    long tm = System.currentTimeMillis();
    // Instead of hard-wiring, you could parse
    // configuration information from a text
    // file here:
    rings = 5;
    addEvent(new ThermostatNight(tm));
    addEvent(new LightOn(tm + 1000));
    addEvent(new LightOff(tm + 2000));
    addEvent(new WaterOn(tm + 3000));
    addEvent(new WaterOff(tm + 8000));
    addEvent(new Bell(tm + 9000));
    addEvent(new ThermostatDay(tm + 10000));
    // Can even add a Restart object!
    addEvent(new Restart(tm + 20000));
  }
  public String description() {
    return "Restarting system";
  }
}
```

```
 public static void main(String[] args) {
    GreenhouseControls gc =
      new GreenhouseControls();
    long tm = System.currentTimeMillis();
    gc.addEvent(gc.new Restart(tm));
    gc.run();
  }
}
```

Резюме

Край на класът за управление на събития в парник

Стартиране на системата за управление на събитията

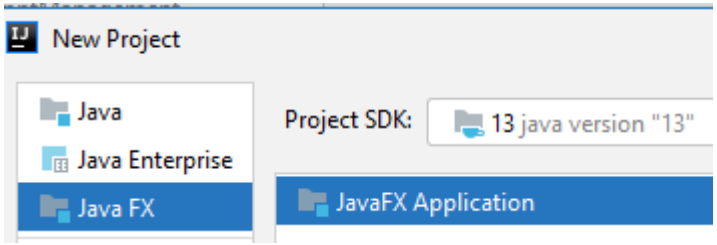
11.11 Обработка на събития в JavaFX

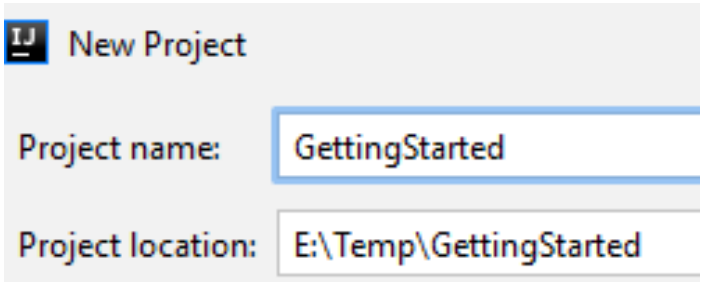
Представените досега програми се изпълняват в текстов режим (Java Console Applications). Това са програми, при които потребителят взаимодейства с програмата посредством въвеждане на текст или комбинация от управляващи клавиши от клавиатурата.

На практика са разпространени програмни средства, които позволяват изграждане на интерактивен графичен интерфейс с богато съдържание. В Java технологиите JavaFX е стандартно средство за изграждане на такъв потребителски интерфейс. Изграждането на сложни приложения с JavaFX се извършва с графични визуални редактори като SceneBuilder. Те в голяма степен автоматизират генерирането на сорс код като използват анотации (@FXML) за писане на стандартни програмни конструкции и по- специално при обработката на събития.

11.11 Обработка на събития в JavaFX

Графичните компоненти на JavaFX изграждат йерархична дървовидна структура, която се описва с FXML. Графичният редактор SceneBuilder генерира FXML представянето на Сцена с тези компоненти заедно с описанието на настройките за свойствата и методите за обработка на събитията. По този начин се генерира файл с FXML описание на Сцената и Контролер с Java код за обработка на събитията в графичния контекст. IntelliJ улеснява създаването на такъв тип проекти посредством шаблона New Project ->JavaFX Application.





```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class GettingStarted extends Application {

    @Override
    public void start(Stage primaryStage) {
        Button btn = new Button();
        btn.setText("Say 'Hello World'");
        btn.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });

        StackPane root = new StackPane();
        root.getChildren().add(btn);

        Scene scene = new Scene(root, 300, 250);

        primaryStage.setTitle("Hello World!");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

Import classes from the **`javafx`** package

JavaFX applications **extend** the **`Application`** class

Setup GUI content (nodes)

Define a root Node (always a Pane)

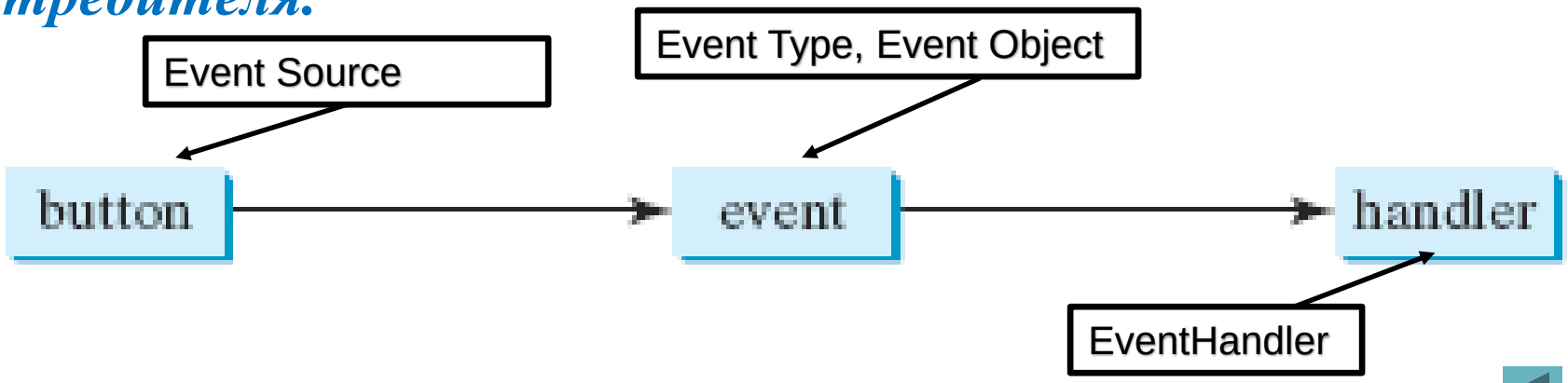
Add content to the JavaFX parent (root) Pane

Create the Scene

Set the title and the Scene of the Stage. Finally, shoe the Stage (the Application window)

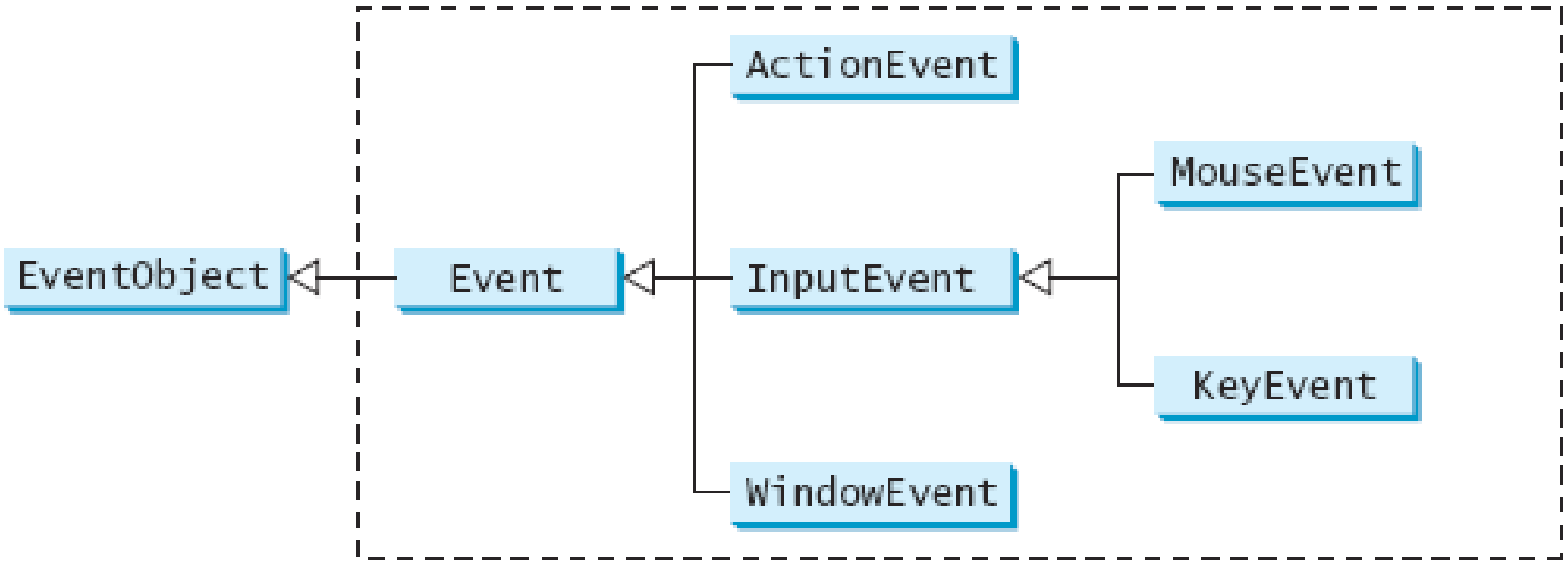
11.11 Обработка на събитие от тип Action

В графичния интерфейс на JavaFX всяко събитие има стандартно наименование (`Action`, `Mouse`, `Key` и пр.). Контролата, която реагира на въздействието на потребителя се нарича Източник на събитието. В резултат от това въздействие, Източникът на събитието създава съответен за това събитие Обект на събитието. Обектът на събитието капсулира в себе си данни, които са предмет на обработка от Метод за обработка на събитие (*event handler*). Този метод *определя последователността от действия, които трябва да се извършат в отговор на действието на потребителя.*



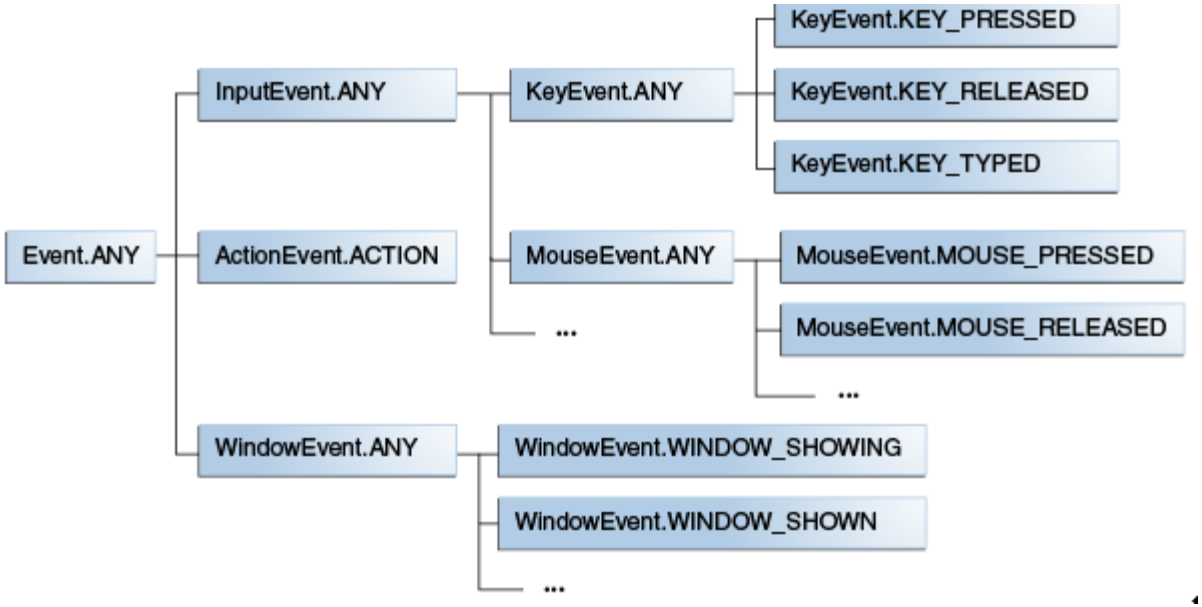
11.11 Обработка на събитие от тип Action

Класовете за обработка на събития в JavaFX се намират в пакета **javafx.event.** Тук е показана йерархията на наследственост на класовете, представящи Обект на събитието. **ActionEvent** е обект на събитието, създадено при обработка на събитие от тип **ACTION**



11.11 Обработка на събитие от тип Action

Тук е показана йерархията на наследственост на класовете, представящи **типовете събития** в JavaFX. Забелязваме, че всеки тип събитие е свързано със съответния му **Обект на събитие**. Например, събитието от тип **ACTION** е свързано със клас на **Обект на събитие** **ActionEvent.**



11.11.2 Обработка на събитие от тип Action с вътрешни класове

Вложен (*вътрешен*) клас е клас, чиято дефиниция се съдържа изцяло в (вътре) дефиницията на друг клас

При обработка на събития вътрешен клас обикновено се използва да "пакетира" метод(ите) за обработка на събитието

За всяко събитие има определен `interface`, който определя *`методите`*, с които може да се обработва това Обекта на това събитие. При обработка на събитието `ACTION` се имплементира интерфейса `EventHandler<ActionEvent>`

Обект от вътрешния клас, който реализира дадения интерфейс, трябва да се регистрира към съответната компонента, за да може тази компонента да реагира на събитието, по начина по който са реализирани методите на интерфейса

11.11.2 Обработка на събитие от тип Action с вътрешни класове

Пример за обработка на събитие ACTION

class `javafx.scene.control.Control`

- **Базов клас на class** `TextField`
- **Базов клас на class** `PasswordField`
  - **Добавя** `echo` **символ (звезда) за скриване на текста при печатането му в текстовото поле**
- **Позволява на потребителя да въвежда текст, когато компонентата получи фокус.**
- **Поражда събитието** `ActionEvent`**, което се обработва с метод** `public void handle(ActionEvent event)`
- **За еднообразие, всеки клиентски клас, който обработва** `ActionEvent` **трябва да реализира метод** `handle()` **на** `interface EventHandler<ActionEvent>`

```
1  import javafx.application.Application;
2  import javafx.event.ActionEvent;
3  import javafx.event.EventHandler;
4  import javafx.geometry.Insets;
5  import javafx.geometry.Pos;
6  import javafx.scene.Scene;
7  import javafx.scene.control.Alert;
8  import javafx.scene.control.PasswordField;
9  import javafx.scene.control.TextField;
10 import javafx.scene.layout.FlowPane;
11 import javafx.stage.Stage;
12 /**
13  * TextFieldsScene.java
14  * Event handling with inner class
15  */
16 public class TextFieldsScene extends Application {
17
18     private TextField txtInputField;
19     private TextField txtInputFieldWithPrompt;
20     private TextField txtUneditableTextField;
21     private PasswordField txtPasswordField;
22     private Alert messageBox;
23     @Override
24     public void start(Stage primaryStage) {
25         FlowPane pane = new FlowPane(14,14);
26         pane.setAlignment(Pos.CENTER);
27         messageBox = new Alert(Alert.AlertType.INFORMATION);
```

Реализира интерфейс за обработка на Обект на събитие `ActionEvent`

Създава базов връх `FlowPane` от тип Панел с хоризонтално и вертикално отстояние 14

Създава диалогов прозорец от тип `INFORMATION`

```
28
29          txtInputField = new TextField();
30          txtInputFieldWithPrompt = new TextField("Enter text here ...");
31          txtUneditableTextField = new TextField("Uneditable Textfield ...");
32          txtUneditableTextField.setEditable(false);// disable editing
33          txtPasswordField = new PasswordField();
34          // event handling with inner classes
35          ActionHandlerClass actionHandler = new ActionHandlerClass();
36          txtInputField.setOnAction(actionHandler);
37          txtInputFieldWithPrompt.setOnAction(actionHandler);
38          txtUneditableTextField.setOnAction(actionHandler);
39          txtPasswordField.setOnAction(actionHandler);
40
41          pane.getChildren().addAll(txtInputField, txtInputFieldWithPrompt,
42                  txtUneditableTextField, txtPasswordField);
43
44          Scene scene = new Scene(pane, 400, 150);
45
46          primaryStage.setTitle("ActionEvent handling");
47          primaryStage.setScene(scene);
48          primaryStage.show();
49      }
```

Създава обект **TextField**

Създава обект **PasswordField**

Използва свойство на събитие **Action** за регистриране на **ActionEventHandler**

Създава обект **TextField** без да позволява редактиране

Добавя възлите към базовия **Панел** (**FlowPane**)

Реализира обработка на **ActionEvent** в обект на **вътрешния** клас **ActionEventHandler**

```
50      // private inner class
51      private class ActionHandlerClass implements EventHandler<ActionEvent> {
52          public void handle(ActionEvent event) {
53              String string = ""; // declare string t
54
55              // user pressed Enter in txtInputField
56              if (event.getSource() == txtInputField) {
57                  string = String.format("Input Field: %s",
58                                         txtInputField.getText());
59
60              } // user pressed Enter in txtInputFieldWithPrompt
61              else if (event.getSource() == txtInputFieldWithPrompt) {
62                  string = String.format("InputField With Prompt: %s",
63                                         txtInputFieldWithPrompt.getText());
64
65              } // user pressed Enter in txtUneditableTextField
66              else if (event.getSource() == txtUneditableTextField) {
67                  string = String.format("Uneditable TextField: %s",
68                                         txtUneditableTextField.getText());
69
70              } // user pressed Enter in txtPasswordField
71              else if (event.getSource() == txtPasswordField) {
72                  string = String.format("Password Field: %s",
73                                         txtPasswordField.getText());
74              }
```

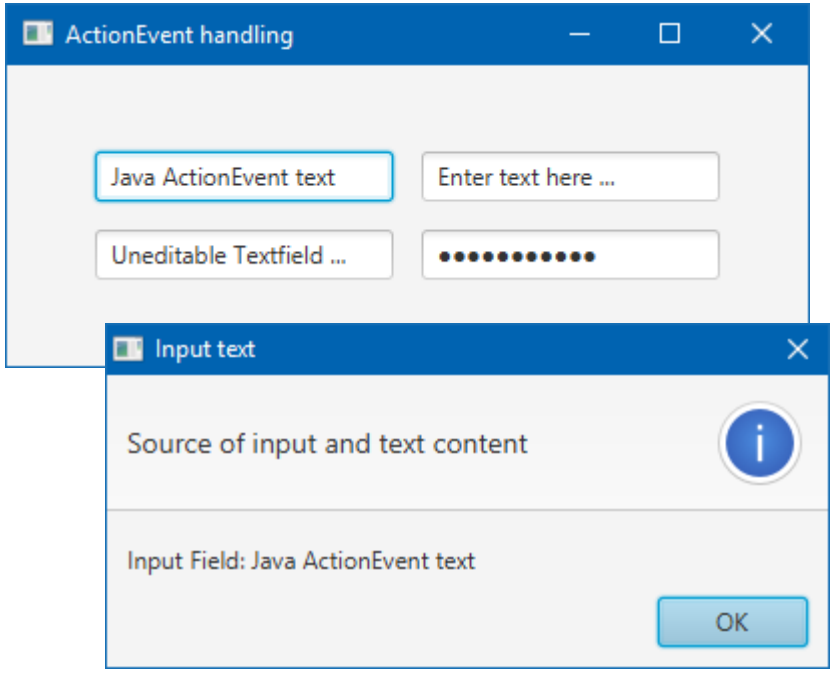
Проверява дали **txtInputField** е **Източникът на събитието**

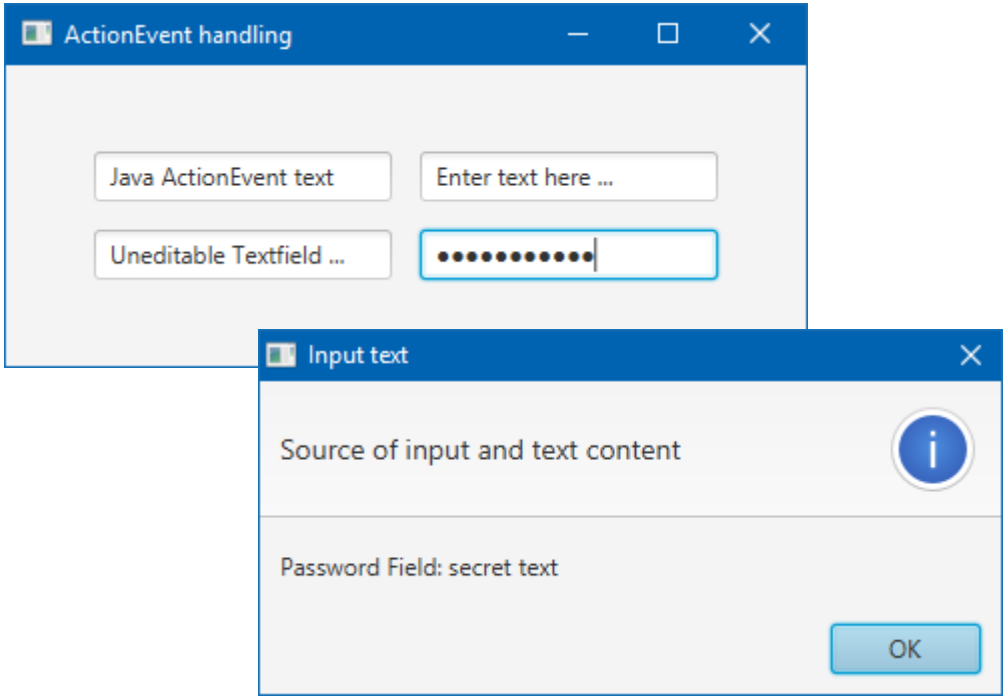
Дефинира метод **handle()** за обработка на Обект на събитие **ActionEvent**

Прочита текст въведен в тестовополе

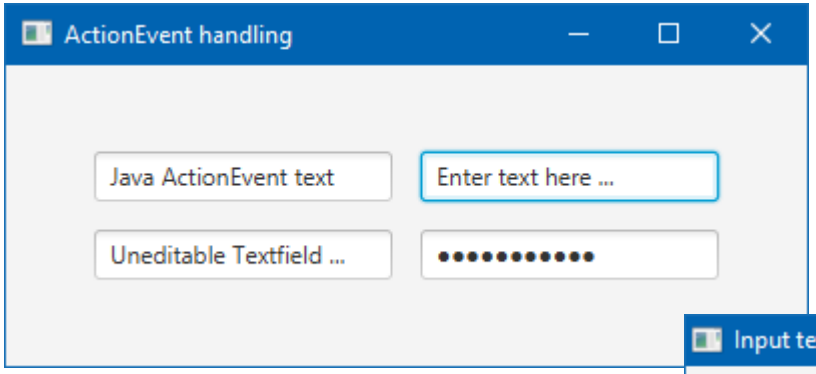
```
75              // display content of the curently active textfield
76              messageBox.setTitle("Input text");
77              messageBox.setHeaderText("Source of input and text content");
78              messageBox.setContentText(string);
79              messageBox.showAndWait();
80          }
81      }
82
83      public static void main(String[] args) {
84          launch(args);
85      }
86  }
```

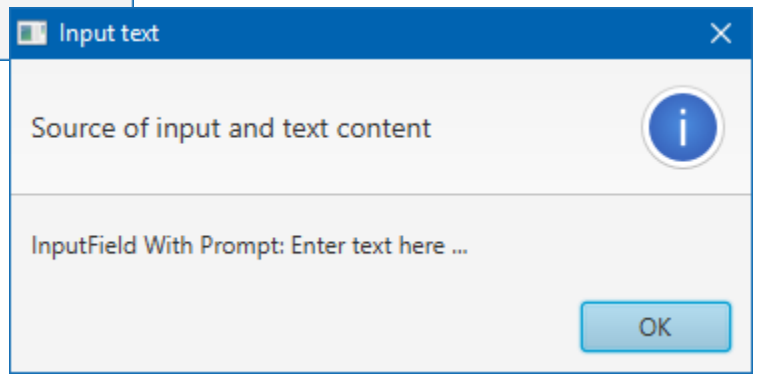
Извежда съобщение с въведения текст в диалогов прозорец

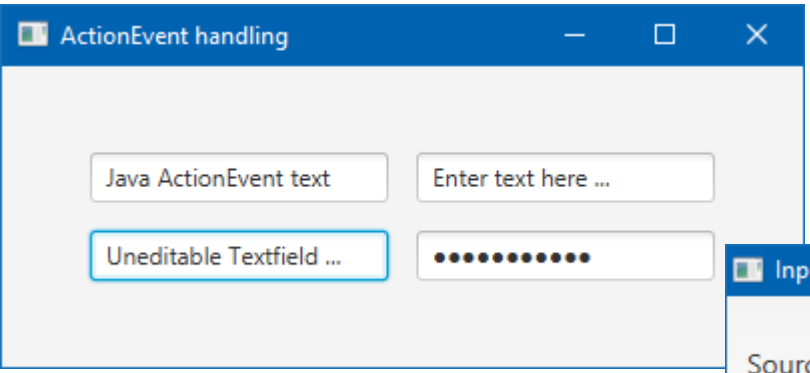


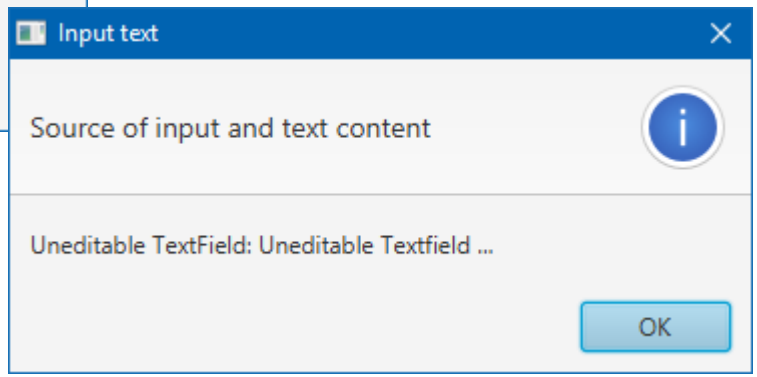


Outline


ActionEvent handling
Java ActionEvent text
Enter text here ...
Uneditable Textfield ...
••••••••••••


Input text
Source of input and text content
InputField With Prompt: Enter text here ...
OK


ActionEvent handling
Java ActionEvent text
Enter text here ...
Uneditable Textfield ...
••••••••••••


Input text
Source of input and text content
Uneditable TextField: Uneditable Textfield ...
OK

11.11.2 Обработка на събитие от тип Action с вътрешни класове

Анонимният клас е *вътрешен*, чиято дефиниция се съдържа изцяло в (вътре) дефиницията на метод, който връща обект от типа на интерфейса, имплементиран във вътрешния клас друг клас. Както и всеки вътрешен клас, анонимният клас има пълен достъп до данните и методите във външния клас. Това позволява, обработката на събитието да се дефинира в метод на външния клас, който се извиква от метода за обработка на събитието в анонимния клас.

По този начин създадения обект на анонимния клас "пакетира" метод(ите) за обработка на събитието, които са дефинирани във външния клас.

```
1  import javafx.application.Application;
2  import javafx.event.ActionEvent;
3  import javafx.event.EventHandler;
4  import javafx.geometry.Insets;
5  import javafx.geometry.Pos;
6  import javafx.scene.Scene;
7  import javafx.scene.control.Alert;
8  import javafx.scene.control.PasswordField;
9  import javafx.scene.control.TextField;
10 import javafx.scene.layout.FlowPane;
11 import javafx.stage.Stage;
12 /**
13  * TextFieldsScene.java
14  * Event handling with inner class
15  */
16 public class TextFieldsScene extends Application {
17
18     private TextField txtInputField;
19     private TextField txtInputFieldWithPrompt;
20     private TextField txtUneditableTextField;
21     private PasswordField txtPasswordField;
22     private Alert messageBox;
23     @Override
24     public void start(Stage primaryStage) {
25         FlowPane pane = new FlowPane(14,14);
26         pane.setAlignment(Pos.CENTER);
27         messageBox = new Alert(Alert.AlertType.INFORMATION);
```

Реализира интерфейс за обработка на Обект на събитие `ActionEvent`

Създава базов връх `FlowPane` от тип Панел с хоризонтално и вертикално отстояние 14

Създава диалогов прозорец от тип `INFORMATION`

```
28
29          txtInputField = new TextField();
30          txtInputFieldWithPrompt = new TextField("Enter text here ...");
31          txtUneditableTextField = new TextField("Uneditable Textfield ...");
32          txtUneditableTextField.setEditable(false);// disable editing
33          txtPasswordField = new PasswordField();
34          // ebent handling with anonymous classes
35          txtInputField.setOnAction(new EventHandler<ActionEvent>() {
36              public void handle(ActionEvent event) {
37                                      onAction(event);
38              }
39          });
40          txtInputFieldWithPrompt.setOnAction(new EventHandler<ActionEvent>() {
41              public void handle(ActionEvent event) {
42                                      onAction(event);
43              }
44          });
45          txtUneditableTextField.setOnAction(new EventHandler<ActionEvent>() {
46              public void handle(ActionEvent event) {
47                                      onAction(event);
48              }
49          });
50          txtPasswordField.setOnAction(new EventHandler<ActionEvent>() {
51              public void handle(ActionEvent event) {
52                                      onAction(event);
53              }
54          });
```

Анонимният клас **имплементира** EventHandler<ActionEvent>. Обектът на Събитието ActionEvent **се обработва в метода handle()**

Методът **onAction(event) е дефиниран във** външния клас

```
55
56          pane.getChildren().addAll(txtInputField, txtInputFieldWithPrompt,
57                                    txtUneditableTextField, txtPasswordField);
58
59          Scene scene = new Scene(pane, 400, 150);
60
61          primaryStage.setTitle("ActionEvent handling");
62          primaryStage.setScene(scene);
63          primaryStage.show();
64      }
65
66      public void onAction(ActionEvent event) {
67          String string = ""; // declare string to display
68
69          // user pressed Enter in txtInputField
70          if (event.getSource() == txtInputField) {
71              string = String.format("Input Field: %s", txtInputField.getText());
72
73          } // user pressed Enter in txtInputFieldWithPrompt
74          else if (event.getSource() == txtInputFieldWithPrompt) {
75              string = String.format("Input Field With Prompt: %s",
76                                     txtInputFieldWithPrompt.getText());
77
78          } // user pressed Enter in txtUneditableTextField
79          else if (event.getSource() == txtUneditableTextField) {
80              string = String.format("Uneditable TextField: %s",
81                                     txtUneditableTextField.getText());
82
```

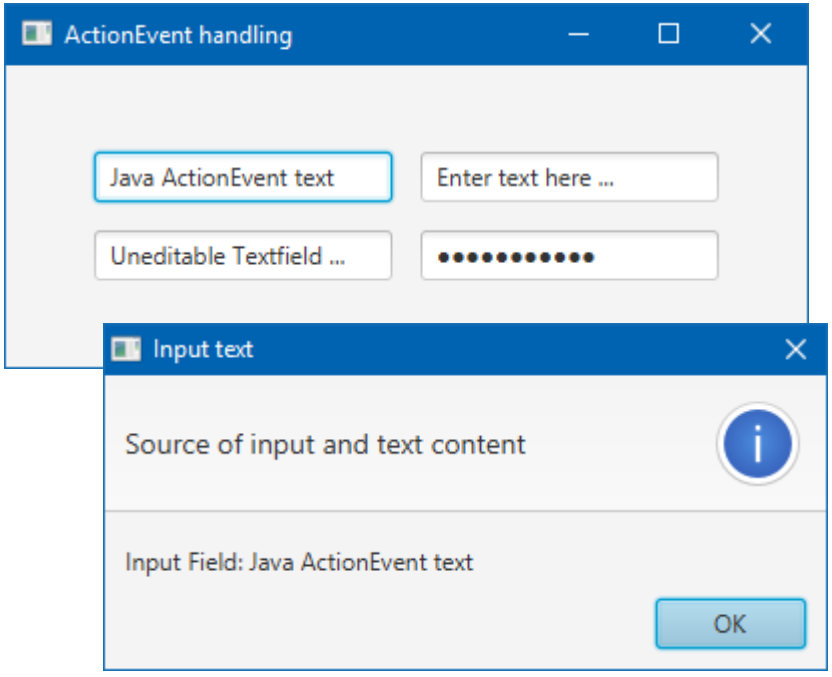
Метод onAction() във външния клас

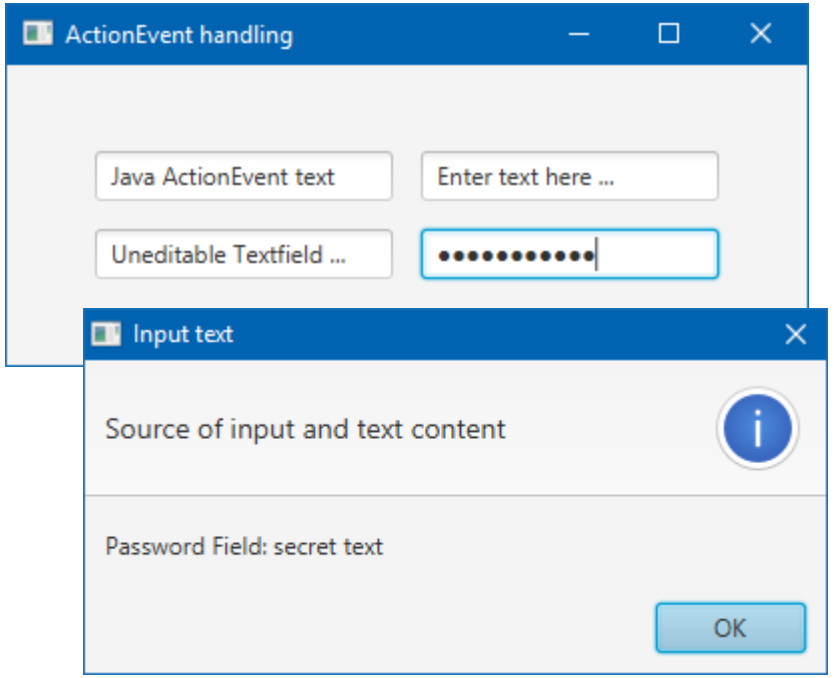
Обработка на Обекта на събитието ACTION

```
83          } // user pressed Enter in txtPasswordField
84          else if (event.getSource() == txtPasswordField) {
85              string = String.format("Password Field: %s",
86                                     txtPasswordField.getText());
87          }
88          // display content of the curently active textfield
89          messageBox.setTitle("Input text");
90          messageBox.setHeaderText("Source of input and text content");
91          messageBox.setContentText(string);
92          messageBox.showAndWait();
93      }
94
95      public static void main(String[] args) {
96          launch(args);
97      }
98  }
```

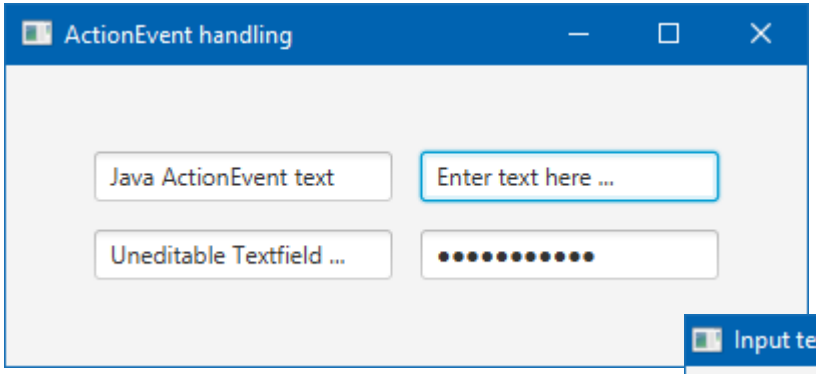
Обработка на **Обекта на събитието** `ACTION`

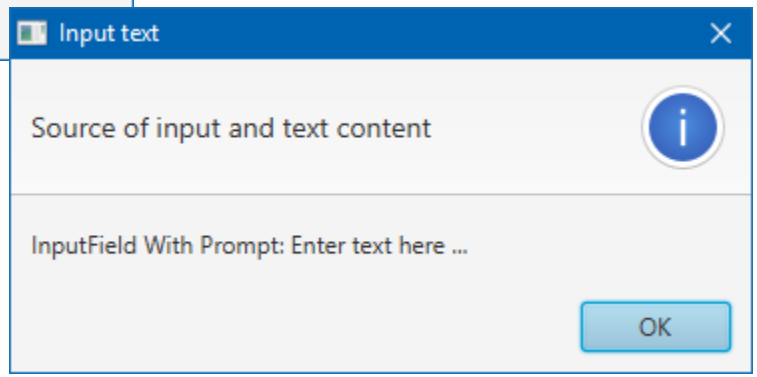
Край на метода `onAction()` **във** външния клас

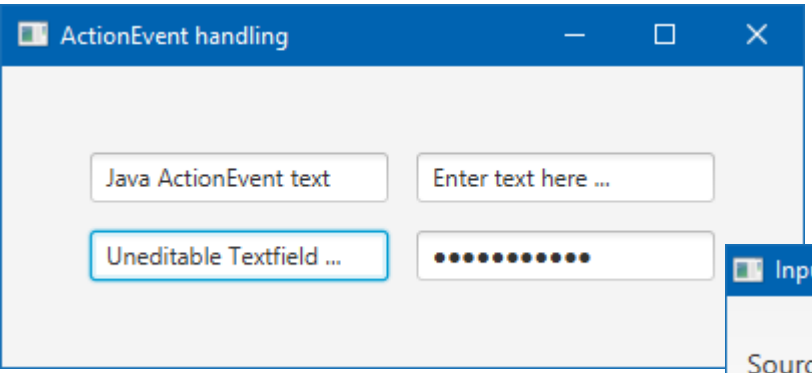


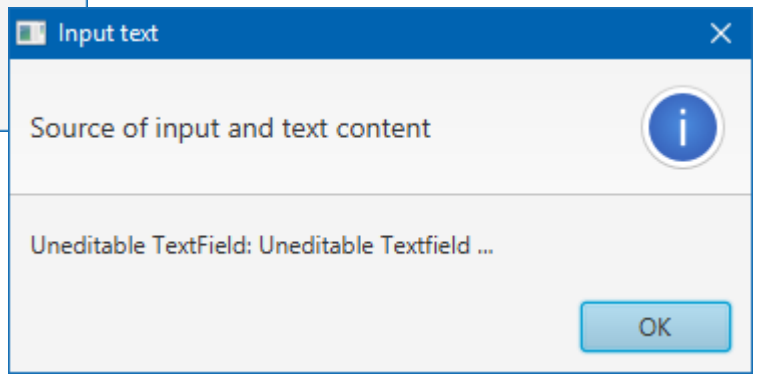


Outline


ActionEvent handling
Java ActionEvent text
Enter text here ...
Uneditable Textfield ...
••••••••••••


Input text
Source of input and text content
InputField With Prompt: Enter text here ...
OK


ActionEvent handling
Java ActionEvent text
Enter text here ...
Uneditable Textfield ...
••••••••••••


Input text
Source of input and text content
Uneditable TextField: Uneditable Textfield ...
OK

11.11.2 Регистриране на метода за обработка на събитието

Регистрирането се извършва като се

- **Извиква метод**

`setOn`*`Event-Type`*`( EventHandler<`*`Object-of-Event`*`>)`

- **Например, за събитието ACTION**
- `setOnAction ( EventHandler<ActionEvent>)`

- `EventHandler<ActionEvent>` **обектът започва да "*слуша*" за събития от типа на ACTION**
- ***Пропускането да се регистрира обработчика на събитието, не позволява на приложението да реагира на това събитие***

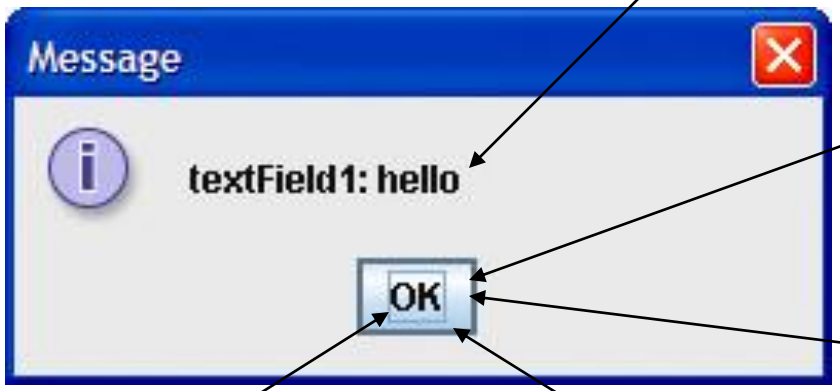
11.11.3 Използване на метода handle()

Приема за аргумент обект от породеното събитие (*ActionEvent*)

- **този обект се предава на метода handle() от Източника на събитието (Event source)**
- **Източникът на събитието е компонентата, от която е породила събитието (създала е обектът от клас *ActionEvent*)**
- **Референция към Източника на събитието се съхранява в Обекта на събитието и може да се получи с метода getSource()**
- **Когато източникът е текстово поле TextField, текстът в него се прочита и променя съответно с getText() и setText(String)**
- **Когато източникът е текстово поле PasswordField, текстът в него се прочита и променя съответно с getText() и setText(String)**

11.13 Обработване на събитие в JavaFX

5. Компонентата подава обекта на събитието (ActionEvent) като аргумент на метода (handle()) от обект за обработка на събитието (EventHandler) и изпълнява този метод



4. По кода на сигнала компонентата открива регистрирания към нея обект за обработка на събитието (handler)

3. По кода на сигнала се компонентата създава обект на събитие (ActionEvent)

1. При натискане на бутона мишката изпраща сигнал на Операционната система

2. ОС изпраща съобщение с кода на сигнала до компонентата, която "*слуша*" за такъв сигнал

11.13.1 Регистриране на събитията

Всеки обект от клас производен на `Node` поддържа списък с референции към обекти от тип

`EventHandler<Object-of-Event>`

регистрирани с тази компонента

Този списък е в съответствие с уникалния код на сигнала, подаван от ОС при възникване на събитието

- ✓ **Например, при изпълнение на**

```
textField.setOnAction( EventHandlerObject );
```

- ✓ ***`EventHandlerObject`*** **се добавя към списъка с обекти на възела** ***`textField1`***
- ✓ **Обектът** ***`textField`*** **ще приема сигнали от ОС (чрез JVM) при възникване на** ***`ActionEvent`***

Задачи

Задача 1

Напишете ***class A*** , който **има** само **конструктор** за общо ползване (няма конструктор по подразбиране). **Напишете** ***class B***, който има метод ***getA()***, връщащ референция до ***class A*** – нека връщаният обектът се създава като **използвате анонимен клас**, наследяващ от ***class A.*** **Напишете** приложение на ***Java*** за тестване на метод ***getA().***

Задача 2

В задача 1 добавете ***public void tryMe(String txt)*** метод към ***class A*** и го предефинирайте (***override***), в анонимния клас дефиниран в метода ***getA()*** на ***class B.*** **Напишете** приложение на ***Java*** за тестване на метод ***tryMe(String txt)***

Задача 3

Напишете class A , който има private данна и **private** метод. **Напишете** вътрешен клас с метод, който **променя данната** на външния клас и **изпълнява метода** на външния клас. **Напишете** втори метод във външния клас, който **създава** обект от вътрешния клас, **извиква** метода на вътрешния клас и **проверява** как се е променила данната на външния клас. **Напишете** приложение на **Java** за тестване и обяснете резултата.

Задачи

Задача 4

Напишете ***UML диаграма за системата за управление на събития*** **class** **`GreenhouseControls`**. Напишете в този контролер вътрешни класове за дефиниране на събитие **`turnFansOn`** и **`turnFansOff`** (включване и изключване на вентилатора)