

Лекция 7.b

Вътрешни и анонимни класове Част II

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

- 7b.1 Наследственост при вътрешни класове
- 7b.2 Closure и Callback
- 7b.3 Създаване и изпълнение на потребителски дефинирани събития
- 7b.4 Анонимни вътрешни класове
- 7b.5 Вътрешни класове и обработка на събития
- 7b.6 Приложение на Система за управление на събития
- 7b.7 Текстови полета и примери за обработка на събитие ActionEvent
- 7b.8 Общи типове събития и съответните им интерфейс-и
- 7b.9 Схематично представяне на модела за обработка на събития в Java

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***

7b.1 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)

```
// 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, not
не и външния клас

```
// example B shows how inner classes behave in inheritance
// An inner class cannot be overridden
// 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()
```

```
// 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 произведен 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() презаредената версия на f() се използва.

Повторното извикване на Egg2.Yolk() е на базовия клас конструктор в BigEgg2.Yolk. Презаредената 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()
```

7b.2 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

// 1. Create interface for service method descriptions

```
public interface CoffeeMachineService {
    CoffeeCup prepareCoffee( double price);
}

public class CoffeeCup { // Sample output
}
```

// 2. Create a class where inner classes describe service implementors

```
public class CoffeeServiceProviders {
    // Note: All inner classes implement the service ddescription interface
    private class LavatzaProvider implements CoffeeMachineService{

        public CoffeeCup prepareCoffee(double price) {
            System.out.println("Lavazza coffee served.");
            return new CoffeeCup();
        }
    }
}
```

// Note: A method in the outer class creates an object of the inner class

```
public CoffeeMachineService makeLavazza(){
    return new LavatzaProvider();
}

private class SpetemaProvider implements CoffeeMachineService{
    public CoffeeCup prepareCoffee(double price) {
        System.out.println("Spetema coffee served.");
        return new CoffeeCup();
    }
}

public CoffeeMachineService makeSpetema(){
    return new SpetemaProvider();
}
}
```

// 3. Create the callback

```
public class CoffeeDrinker {
    private CoffeeMachineService callback;
    // ...plus other data members
    public CoffeeDrinker(CoffeeMachineService callback) {
        this.callback = callback;
    }

    public CoffeeMachineService getCallback() {
        return callback;
    }

    public void setCallback(CoffeeMachineService callback) {
        this.callback = callback;
    }

    // Option 1. Use callback as data member
    public void drinkCoffee(double price){
        if (callback != null) {
            CoffeeCup cup = callback.prepareCoffee(price);
        }
    }

    // Option 2 . Pass callback as parameter
    public void drinkCoffee(double price, CoffeeMachineService callback ){
        if (callback != null) {
            CoffeeCup cup = callback.prepareCoffee(price);
        }
    }
}
```

//4. Run the Callback application

```
public class CoffeeConsumer {  
    public static void main(String[] args) {  
        CoffeeServiceProviders providers = new CoffeeServiceProviders();  
  
        CoffeeMachineService lavazzaProvider = providers.makeLavazza();// Lavazza provider selected  
        CoffeeMachineService spetemaProvider = providers.makeSpetema();// Spetema provider selected  
        CoffeeDrinker drinker = new CoffeeDrinker(lavazzaProvider);  
  
        drinker.drinkCoffee(1.00); // Drnk Lavazza for 1 lv  
        drinker.setCallback(spetemaProvider);  
        drinker.drinkCoffee(1.00); // Drnk Spetema for 1 lv  
  
        drinker.drinkCoffee(1.00, lavazzaProvider);// Drnk Lavazza for 1 lv  
    }  
}
```

C:\Users\echrk\.jdk\openjdk-15\bin\java.exe

Lavazza coffee served.

Spetema coffee served.

Lavazza coffee served.

7b.3 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

7b.3 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

```
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;
    }
}
```

```

public class ActionEventSource // event source
{
    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;
    }
    //...
    public void addActionListener(ActionListener al){
        ie = al;
    }
    public void doWork ()
    {
        // Check the predicate, which is set elsewhere.
        if (onAction )
        {
            // 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`

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


7b.4 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 произведен 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 произведен 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:
        return new Wrapping(x)
        {
            private String label = dest;
            public int value()
            {
                return super.value() * count;
            }
        }; // Semicolon required
    }
    public static void main(String[] args)
    {
        Parcel7 p = new Parcel7();
        Wrapping w = p.wrap(10, "Captured string");
    }
}
public class Wrapping
{
    private int i;
    public Wrapping(int x) { i = x; }
    public int value() { return i; }
}

```

Анонимен клас с конструктор-
 предаваме аргумент на базовия
 конструктор, 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 рефериран 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** !)

7b.4 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", 107b.395F);
        System.out.println(d.readlabel());
    }
}
```

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";
    }
}
```

7b.5 Вътрешни класове и обработка на събития

Application framework е class или съвкупност 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 приложения *important* е в програмиране на graphical user interface (GUI), която изцяло управлявана от събития

7b.5 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

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

7b.5 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**.

7b.5 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 да дефинира **Event** обектите.

EventSet примерно е контейнер за 100 **Events**.

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

```
// 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;
    }
}
```

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

7b.5 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 да се 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

7b.5 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.”*

На практика изразяваме различни действия със създаване на различни производни класове на `class Event`.

7b.5 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.

7b.6 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.

7b.6 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 термотат, 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 наследява от `class Event` и предефиниране на метод `action()`.

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

```
//: 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";
        }
    }
}
```

Начало на
класът за
управление
на събития в
парник

Пример за
дефиниране
на събитие
за включване
на светлина

```
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";
    }
}
```

```
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";
    }
}
```

```
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";
    }
}
```

```
// 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";
    }
}
```



```
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";
    }
}
```

Пример за
генериране
"пакет" от
всички
събития

```
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

3.1.6 Създаване на JavaFX приложение

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.

3.1.6 Създаване на JavaFX приложение

Графичните компоненти на JavaFX изграждат йерархична дървовидна структура, която се описва с FXML.

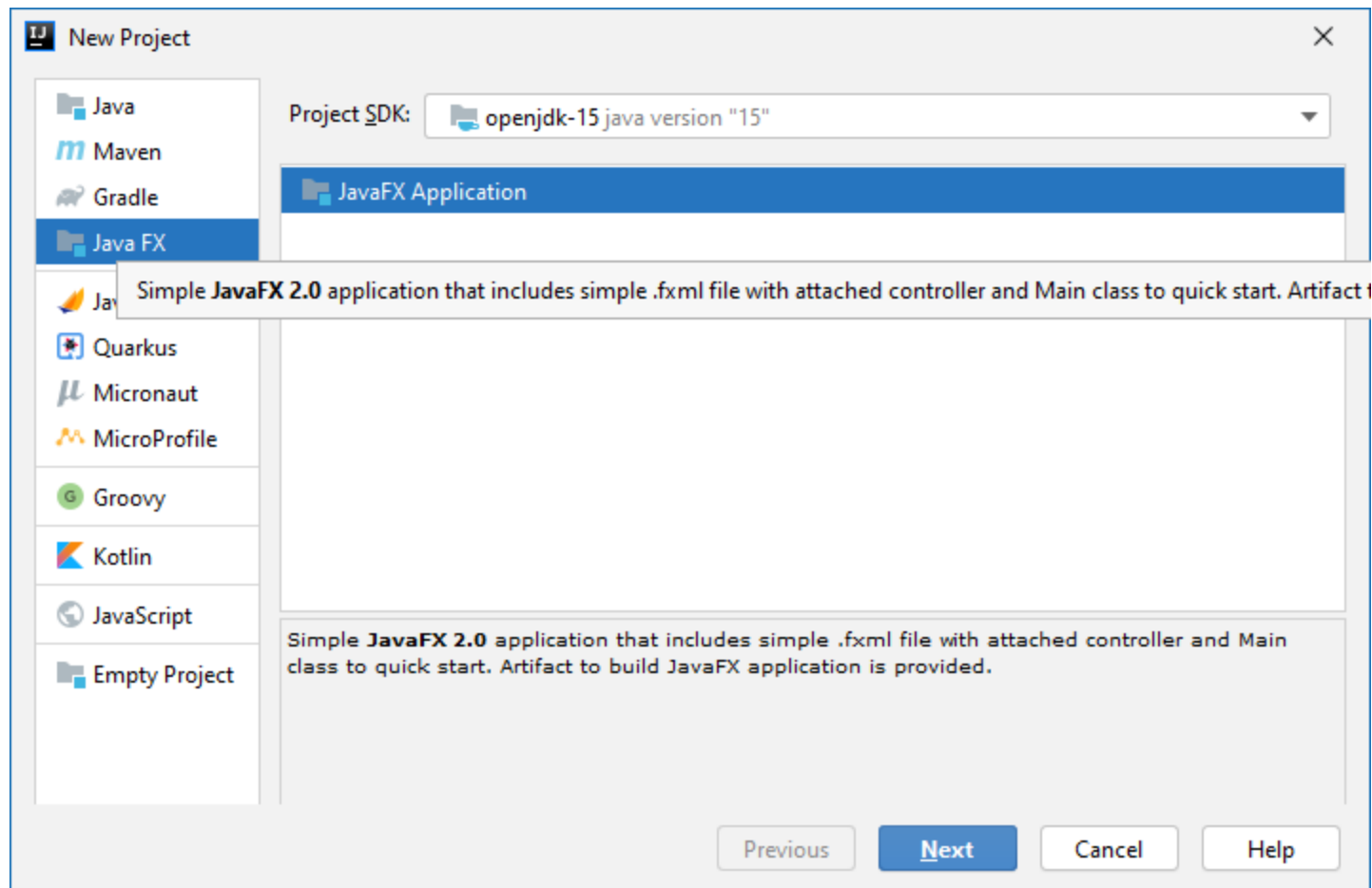
Графичният редактор SceneBuilder генерира FXML представянето на Сцена с тези компоненти заедно с описанието на настройките за свойствата и методите за обработка на събитията. По този начин се генерира файл с FXML описание на Сцената и Контролер с Java код за обработка на събитията в графичния контекст.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

В приложения файл

SetupJFXNonModularProjectWithJDK13IntelliJ.v2.pdf

на сайта на текущата седмица са дадени подробни инструкции за създаване и конфигуриране на JavaFX приложение с IntelliJ.

3.1.6 Създаване на JavaFX приложение



3.1.6 Създаване на JavaFX приложение

The screenshot shows the IntelliJ IDEA IDE. On the left, the 'New Project' dialog is open, with 'Project name' set to 'JFXdemo' and 'Project location' set to 'E:\Temp\JFXdemo'. The 'Project' tab is selected, showing the project structure: 'JFXdemo' (E:\Temp\JFXdemo) contains '.idea', 'src', and 'sample'. The 'sample' directory contains 'Controller', 'Main', and 'sample.fxml'. The 'Main' class is selected. The 'Main.java' file is open in the editor, showing the following code:

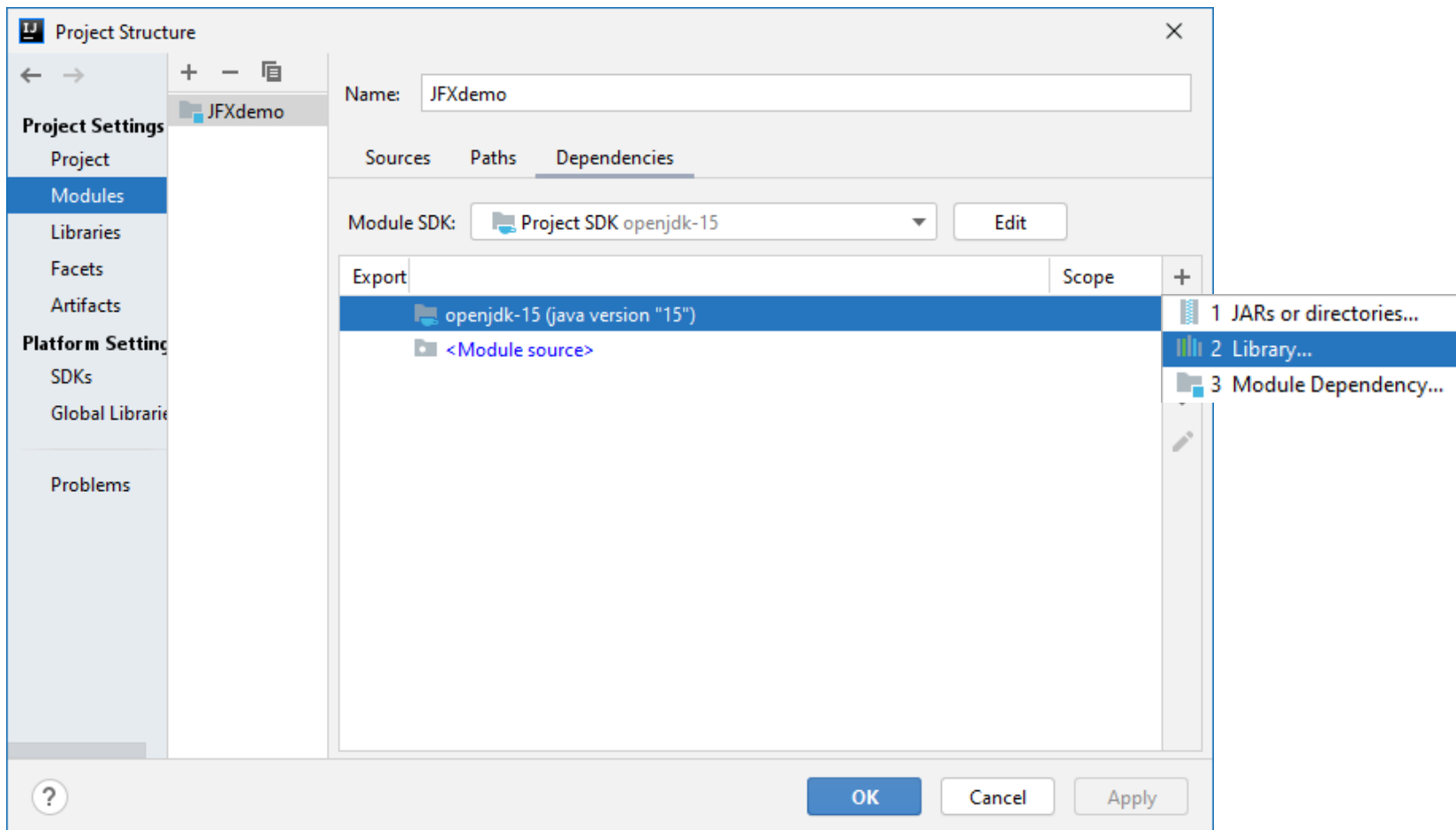
```
8  
9 public class Main extends Application {  
10  
11     @Override  
12     public void start(Stage primaryStage) throws Exception{  
13         Parent root = FXMLLoader.load(getClass().getResource("sample.fxml"));  
14         primaryStage.setTitle("Hello World");  
15         primaryStage.setScene(new Scene(root, 300, 275));  
16         primaryStage.show();  
17     }  
18  
19 }
```

A red arrow points from the text box below to the 'Main' class in the code editor.

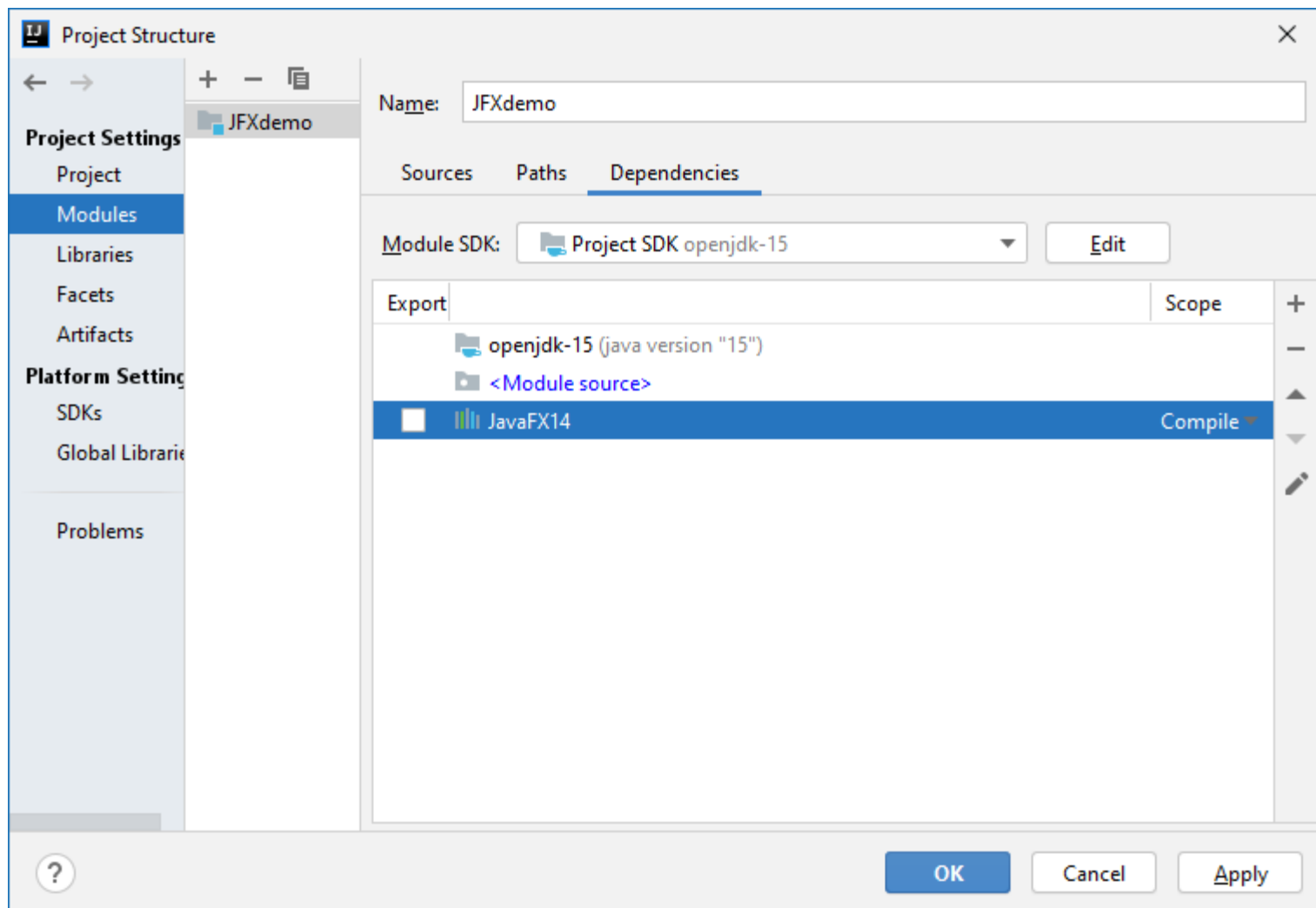
Нужно е да се конфигурират пакетите на JavaFX

Buttons: Previous, Finish, Cancel, Help

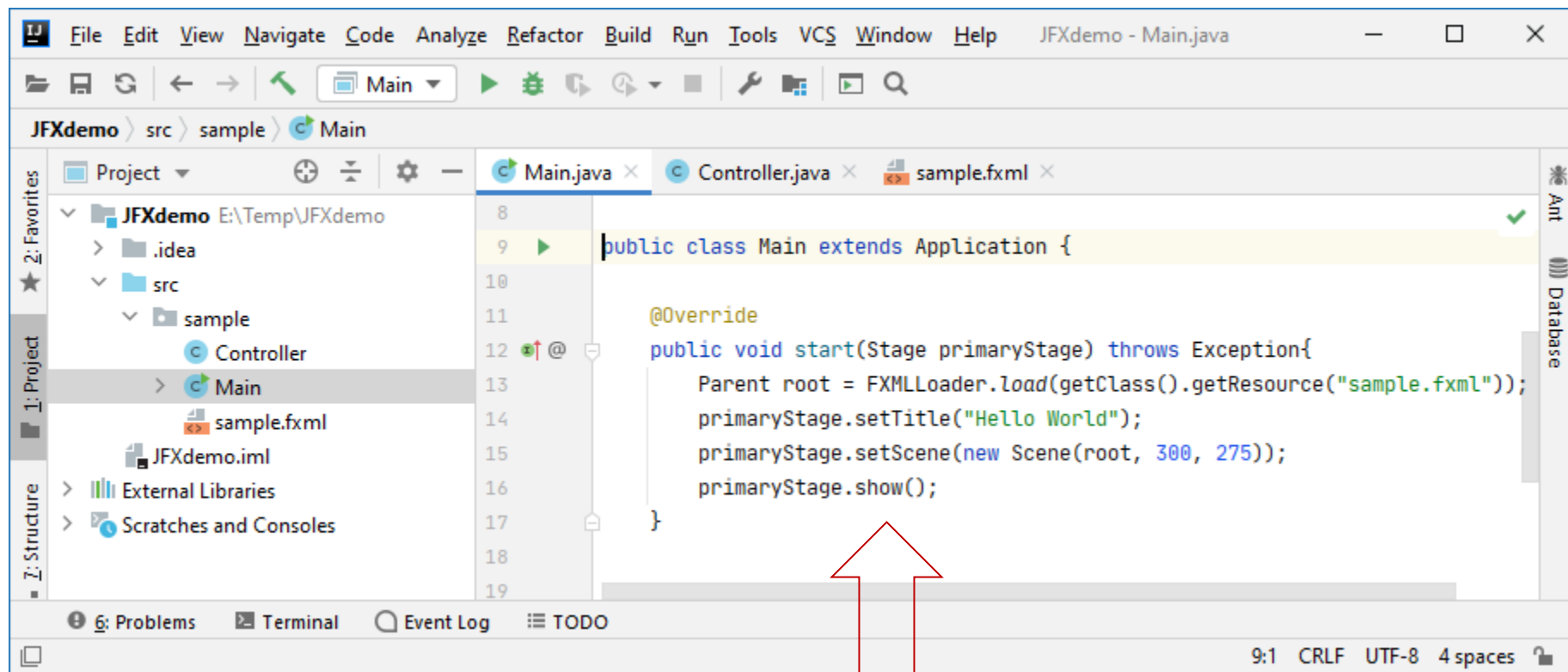
3.1.6 Създаване на JavaFX приложение



3.1.6 Създаване на JavaFX приложение

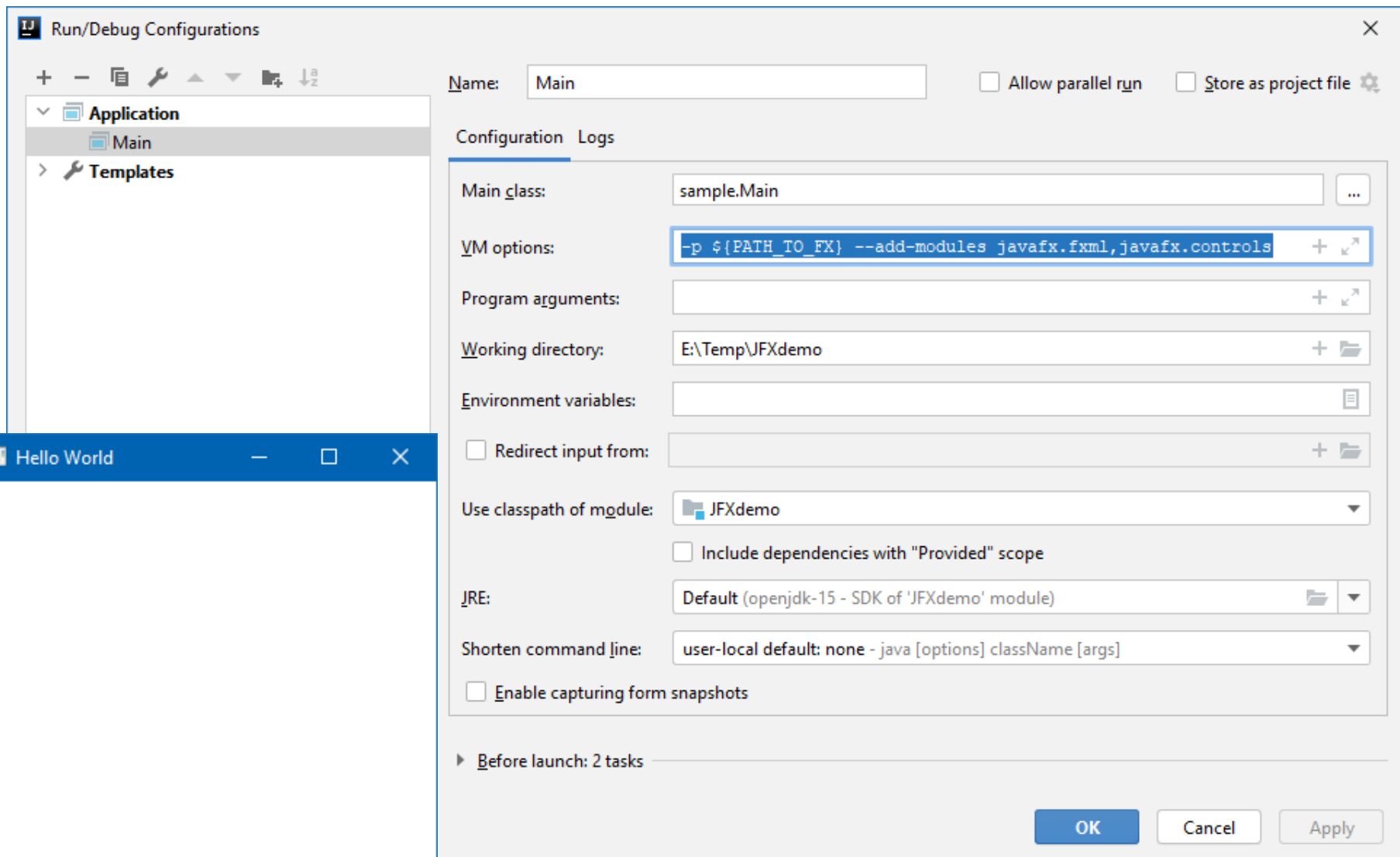


3.1.6 Създаване на JavaFX приложение

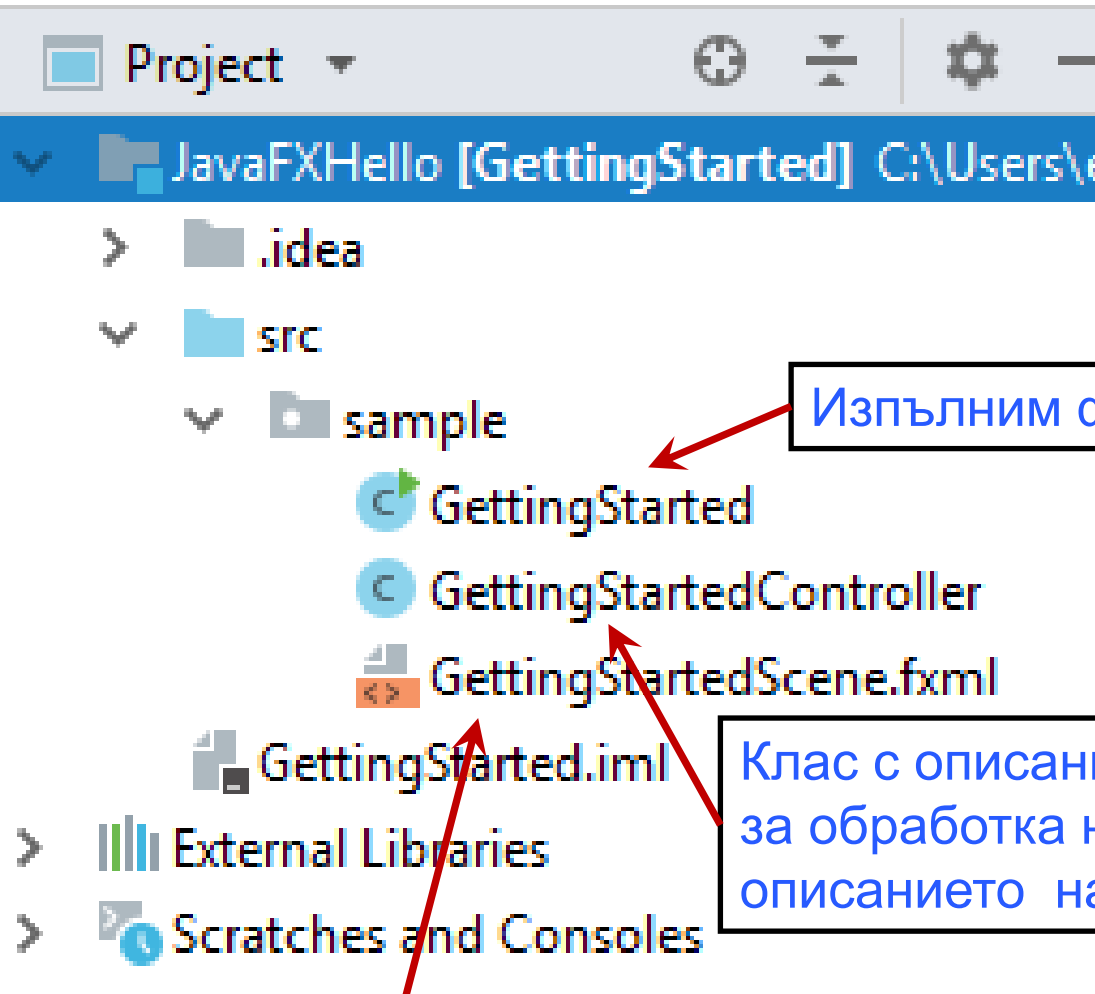


Пакетите на JavaFX са
конфигурирани правилно

3.1.6 Създаване на JavaFX приложение



3.1.6 Създаване на JavaFX приложение



The screenshot shows the IntelliJ IDEA project structure for a JavaFX application. The project is named "JavaFXHello [GettingStarted]" and is located at "C:\Users\...". The project structure is as follows:

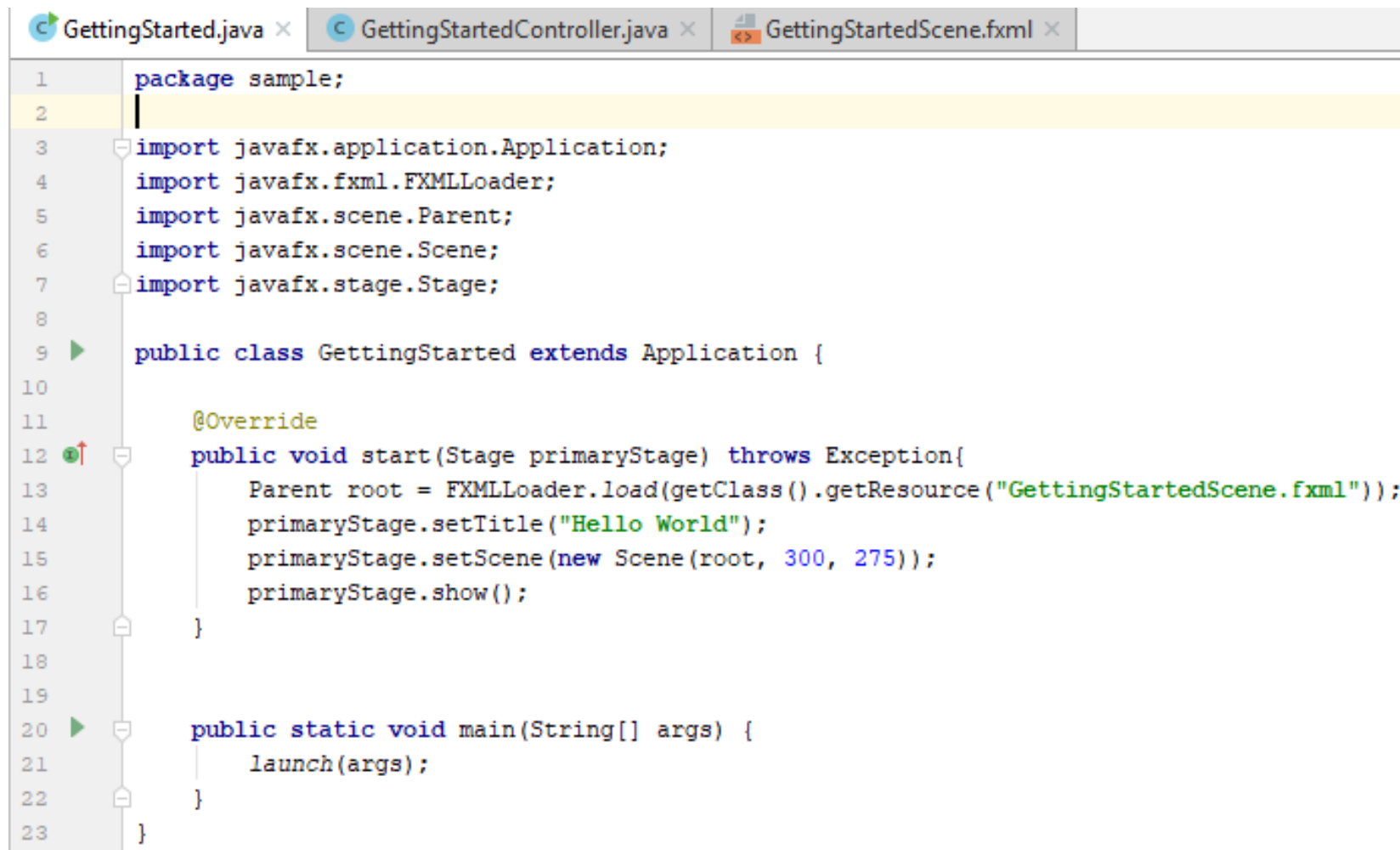
- Project
 - .idea
 - src
 - sample
 - GettingStarted (Java class)
 - GettingStartedController (Java class)
 - GettingStartedScene.fxml (FXML file)
 - GettingStarted.iml (Module file)
 - External Libraries
 - Scratches and Consoles

Annotations and arrows point to specific files:

- A red arrow points from the text "Изпълним файл на JavaFX приложението" to the `GettingStarted` class.
- A red arrow points from the text "Клас с описание на възли (Node) и методи за обработка на събития, съпътстващ с описанието на Сцената" to the `GettingStartedController` class.
- A red arrow points from the text "FXML файл с описание на Сцената. Създава се със SceneBuilder" to the `GettingStartedScene.fxml` file.

FXXML файл с описание на Сцената. Създава се със SceneBuilder

Начално състояние на изпълнимия файл



The screenshot shows an IDE with three tabs: GettingStarted.java, GettingStartedController.java, and GettingStartedScene.fxml. The GettingStarted.java tab is active, displaying the following code:

```
1 package sample;
2
3 import javafx.application.Application;
4 import javafx.fxml.FXMLLoader;
5 import javafx.scene.Parent;
6 import javafx.scene.Scene;
7 import javafx.stage.Stage;
8
9 public class GettingStarted extends Application {
10
11     @Override
12     public void start(Stage primaryStage) throws Exception{
13         Parent root = FXMLLoader.load(getClass().getResource("GettingStartedScene.fxml"));
14         primaryStage.setTitle("Hello World");
15         primaryStage.setScene(new Scene(root, 300, 275));
16         primaryStage.show();
17     }
18
19
20     public static void main(String[] args) {
21         launch(args);
22     }
23 }
```

Създаване на JavaFX приложение

```
GettingStarted.java x GettingStartedController.java x GettingStartedScene.fxml x
1 package sample;
2 import javafx.application.Application;
3 import javafx.event.ActionEvent;
4 import javafx.event.EventHandler;
5 import javafx.scene.Scene;
6 import javafx.scene.control.Button;
7 import javafx.scene.layout.StackPane;
8 import javafx.stage.Stage;
9 public class GettingStarted extends Application {
10     @Override
11     public void start(Stage primaryStage) throws Exception{
12         // Задава панел за управление на наредбата на графичните компоненти
13         StackPane root = new StackPane();
14         // Създаваме Node обекти
15         Button btn = new Button();
16         btn.setText("Say 'Hello World'");
17         btn.setOnAction(new EventHandler<ActionEvent>() {
18             @Override
19             public void handle(ActionEvent event) {
20                 System.out.println( "Hello World!");
21             }
22         });
23         // Добавяме възлите към базовия панел root
24         root.getChildren().add(btn);
25         // Създаваме Сцена с базов панел root
26         Scene scene = new Scene(root, 300, 250);
27         // Parent root = FXMLLoader.load(getClass().getResource("GettingStartedScene.fxml"));
28         primaryStage.setTitle("Hello World"); // задава заглавие на прозореца
29         primaryStage.setScene(new Scene(root, 300, 275));
30         primaryStage.show();
31     }
32     public static void main(String[] args) {
33         launch(args);
34     }
35 }
```

```
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;
```

Import classes from the
javafx package

```
public class GettingStarted extends Application {
```

JavaFX applications **extend** the
Application class

```
@Override
```

```
public void start(Stage primaryStage) {
```

Setup GUI content (nodes)

```
    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!");
        }
    });
```

Define a root Node (always a
Pane)

```
    StackPane root = new StackPane();
```

```
    root.getChildren().add(btn);
```

Add content to the JavaFX
parent (root) Pane

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

Create the Scene

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

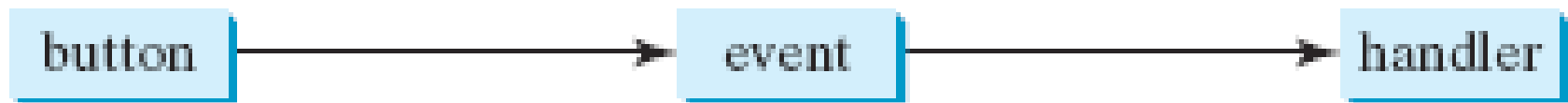
Set the title and the Scene of the
Stage. Finally, shoe the Stage
(the Application window)

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



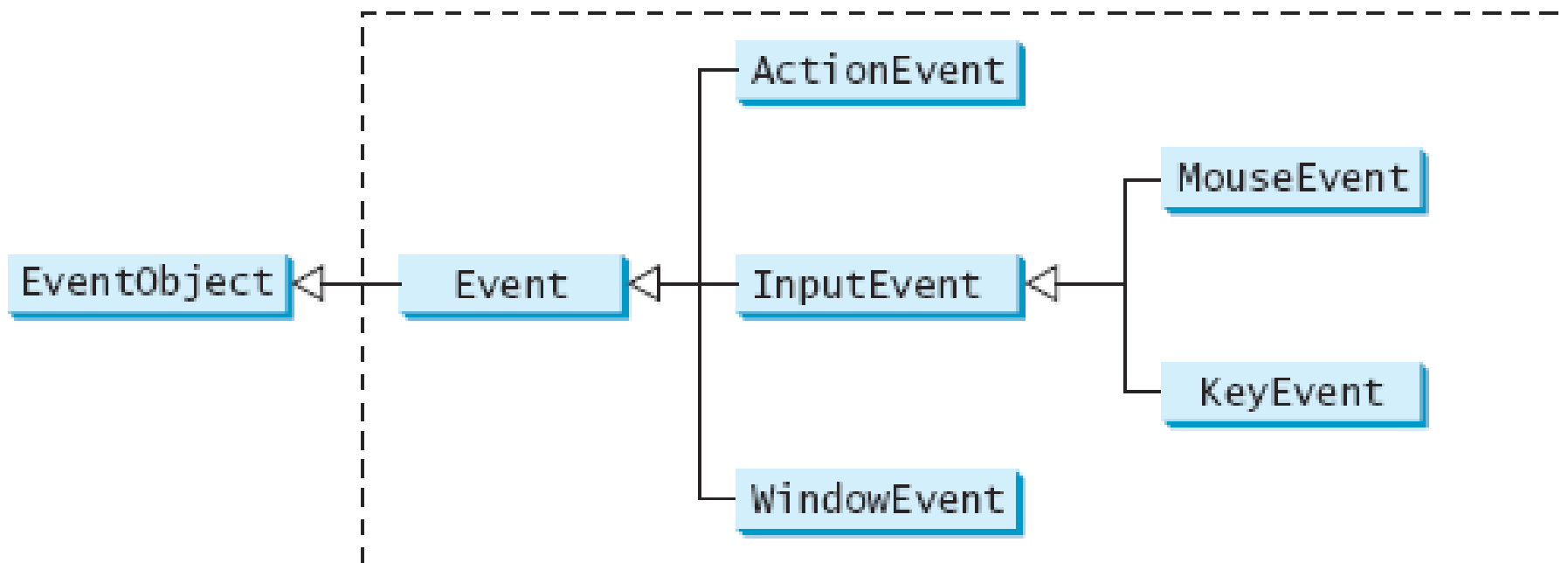
3.1.7 Обработка на събитие от тип Action

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.*



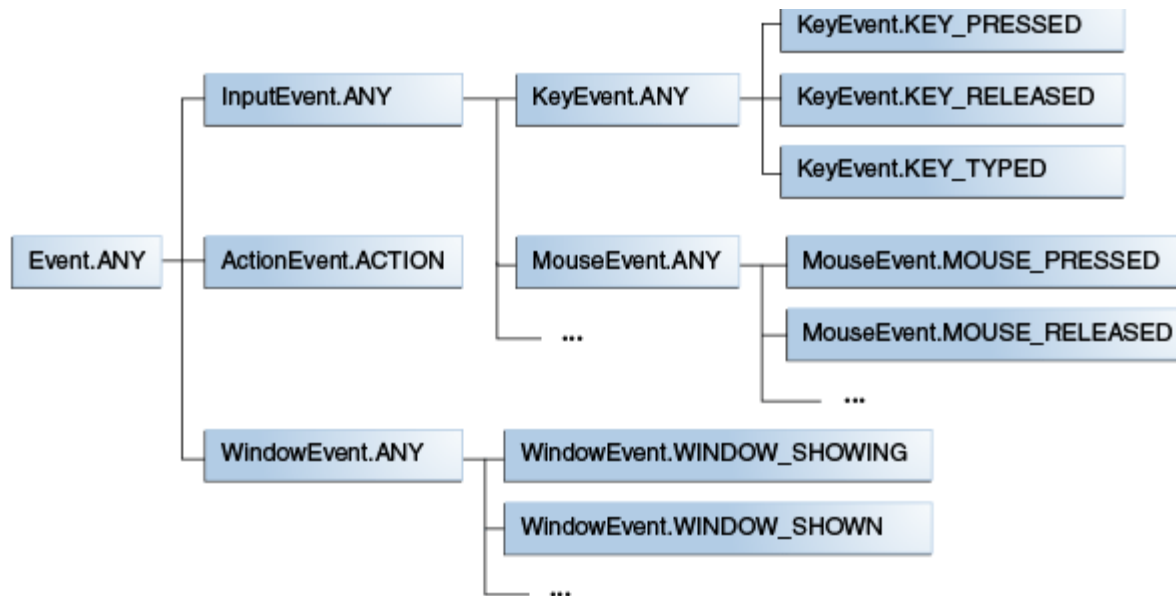
3.1.7 Обработка на събитие от тип Action

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**



3.1.7 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**.



3.1.7 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 прилага 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**

3.1.7 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>`

```

1import javafx.application.Application;
2import javafx.event.ActionEvent;
3import javafx.event.EventHandler;
4import javafx.geometry.Insets;
5import javafx.geometry.Pos;
6import javafx.scene.Scene;
7import javafx.scene.control.Alert;
8import javafx.scene.control.PasswordField;
9import javafx.scene.control.TextField;
10import javafx.scene.layout.FlowPane;
11import javafx.stage.Stage;
12/**
13 * TextFieldsScene.java
14 * Event handling with inner class
15 */
16public 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**

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
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

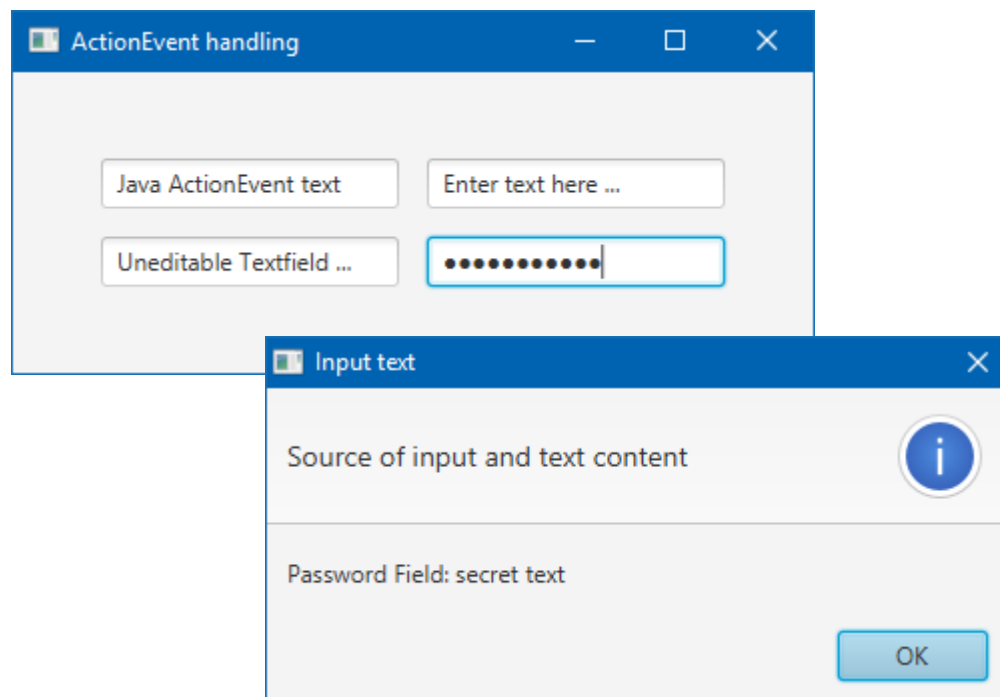
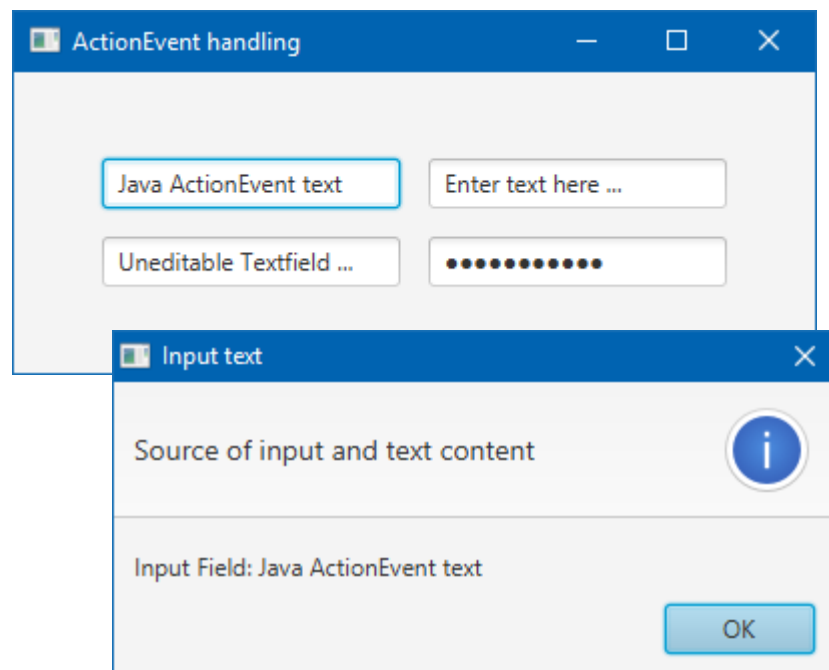
Прочита текст въведен в тестовополе

```

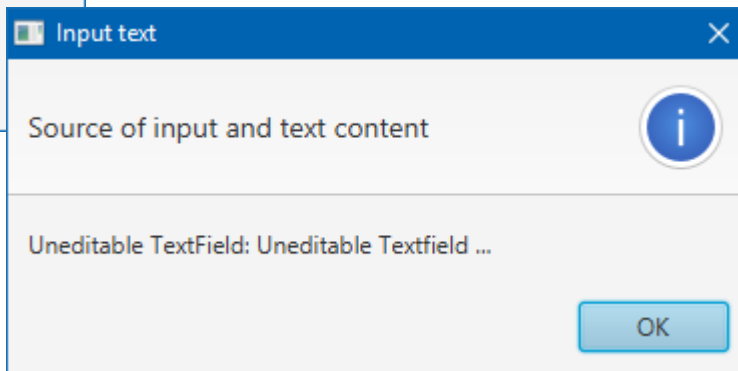
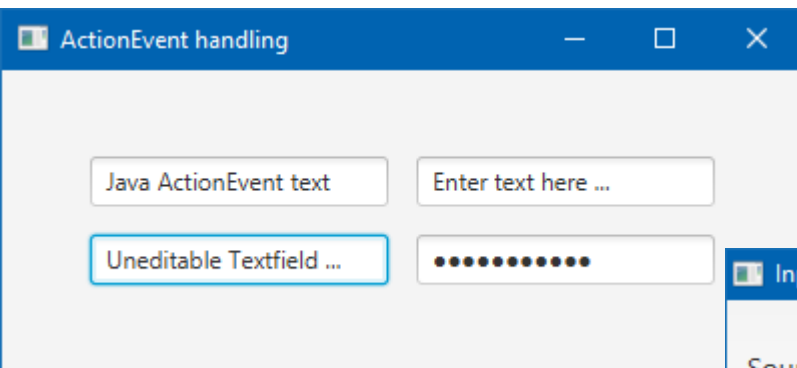
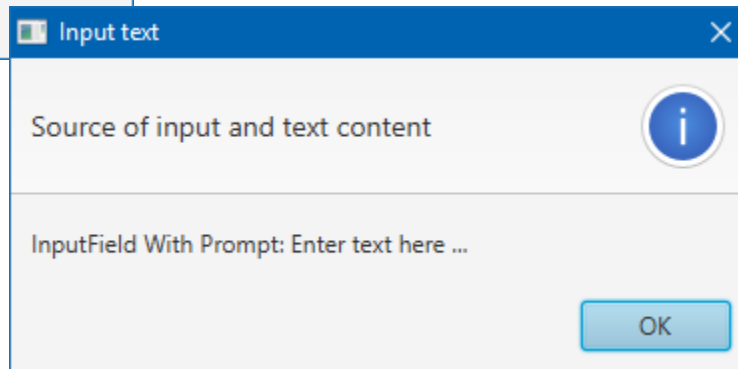
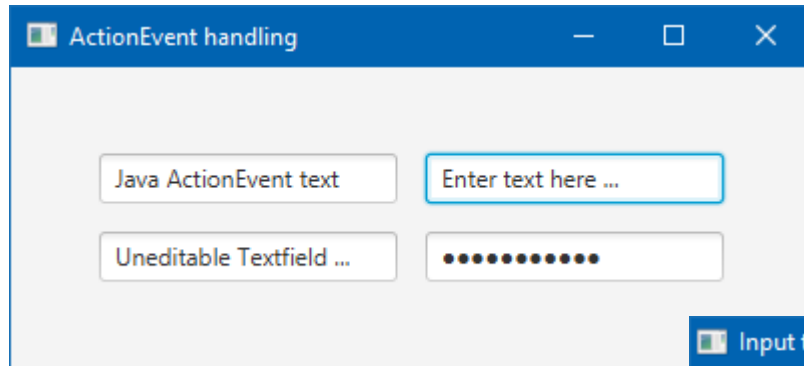
75 // display content of the curently active textfield
76 messageBox.setTitle("Input text");
77 messageBox.setHeaderText("Source of input and text content");
78 messageBox.setContentText(string);
79 messageBox.showAndWait();
80 }
81 }
82
83 public static void main(String[] args) {
84     launch(args);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

```

Извежда съобщение с
въведения текст в диалогов
прозорец



Outline



3.1.8 Обработка на събитие от тип Action с анонимен клас

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.

```

1import javafx.application.Application;
2import javafx.event.ActionEvent;
3import javafx.event.EventHandler;
4import javafx.geometry.Insets;
5import javafx.geometry.Pos;
6import javafx.scene.Scene;
7import javafx.scene.control.Alert;
8import javafx.scene.control.PasswordField;
9import javafx.scene.control.TextField;
10import javafx.scene.layout.FlowPane;
11import javafx.stage.Stage;
12/**
13 * TextFieldsScene.java
14 * Event handling with inner class
15 */
16public 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 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	<code>pane.getChildren().addAll(txtInputField, txtInputFieldWithPrompt,</code>	
57	<code>txtUneditableTextField, txtPasswordField);</code>	
58		
59	<code>Scene scene = new Scene(pane, 400, 150);</code>	
60		
61	<code>primaryStage.setTitle("ActionEvent handling");</code>	
62	<code>primaryStage.setScene(scene);</code>	
63	<code>primaryStage.show();</code>	Метод <code>onAction()</code> ВЪВ ВЪНШНИЯ КЛАС
64	<code>}</code>	
65		Обработка на Обекта на събитието <code>ACTION</code>
66	<code>public void onAction(ActionEvent event) {</code>	
67	<code>String string = ""; // declare string to display</code>	
68		
69	<code>// user pressed Enter in txtInputField</code>	
70	<code>if (event.getSource() == txtInputField) {</code>	
71	<code>string = String.format("Input Field: %s", txtInputField.getText());</code>	
72		
73	<code>} // user pressed Enter in txtInputFieldWithPrompt</code>	
74	<code>else if (event.getSource() == txtInputFieldWithPrompt) {</code>	
75	<code>string = String.format("Input Field With Prompt: %s",</code>	
76	<code>txtInputFieldWithPrompt.getText());</code>	
77		
78	<code>} // user pressed Enter in txtUneditableTextField</code>	
79	<code>else if (event.getSource() == txtUneditableTextField) {</code>	
80	<code>string = String.format("Uneditable TextField: %s",</code>	
81	<code>txtUneditableTextField.getText());</code>	
82		

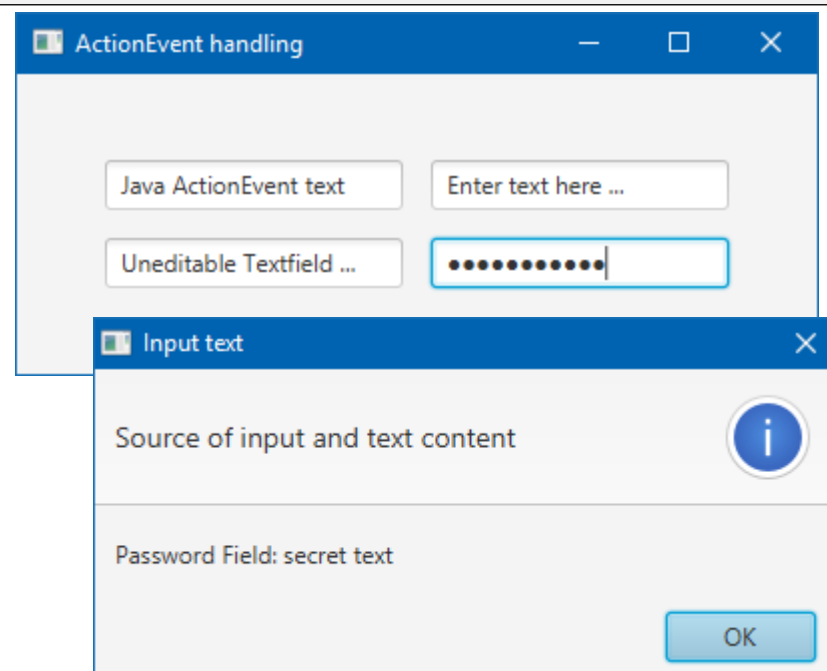
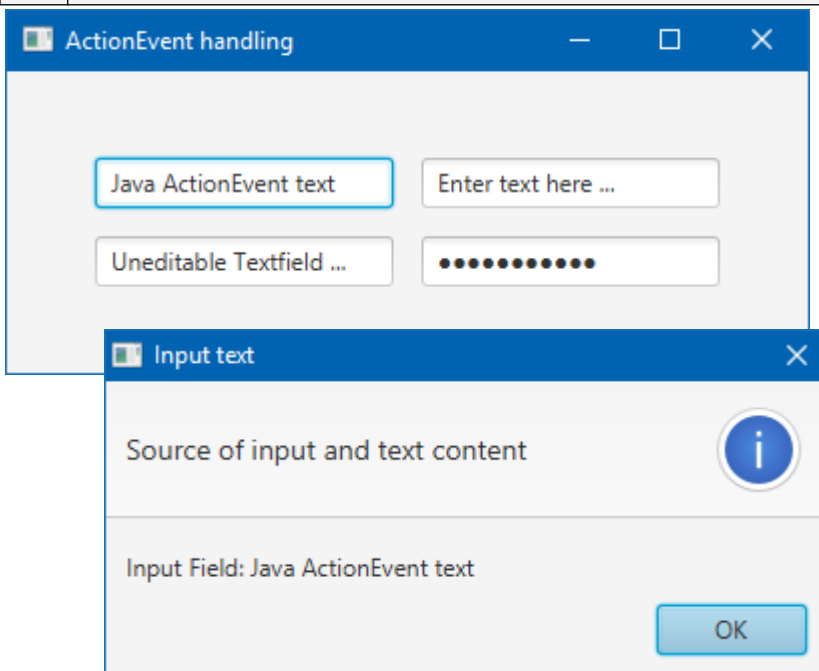
```

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  }

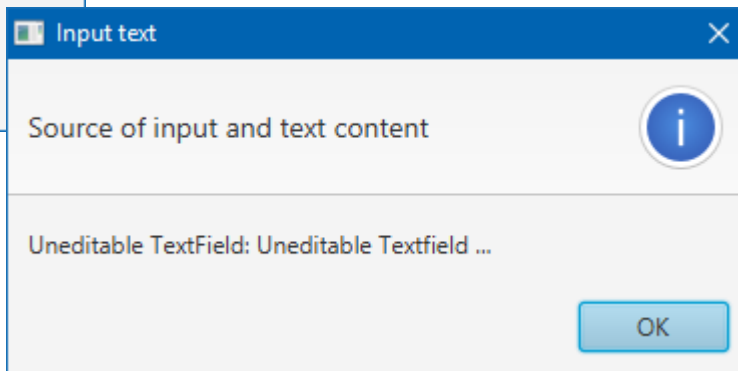
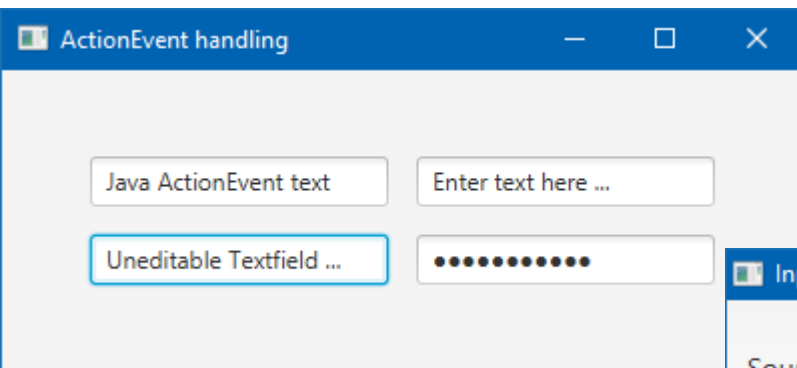
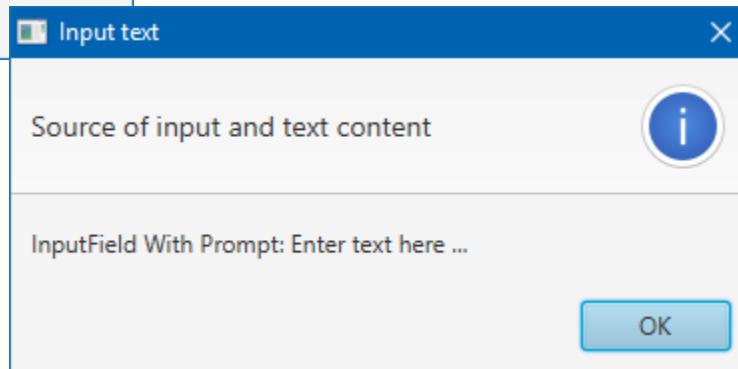
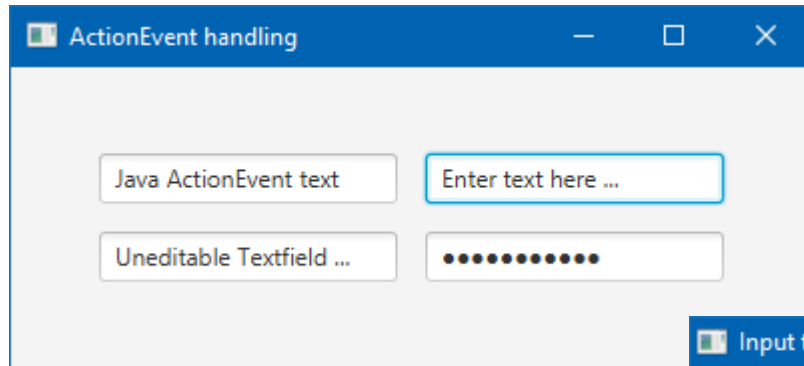
```

Обработка на Обекта на събитието **ACTION**

Край на метода **onAction()** във
ВЪНШНИЯ КЛАС



Outline



Регистриране на метода за обработка на събитието

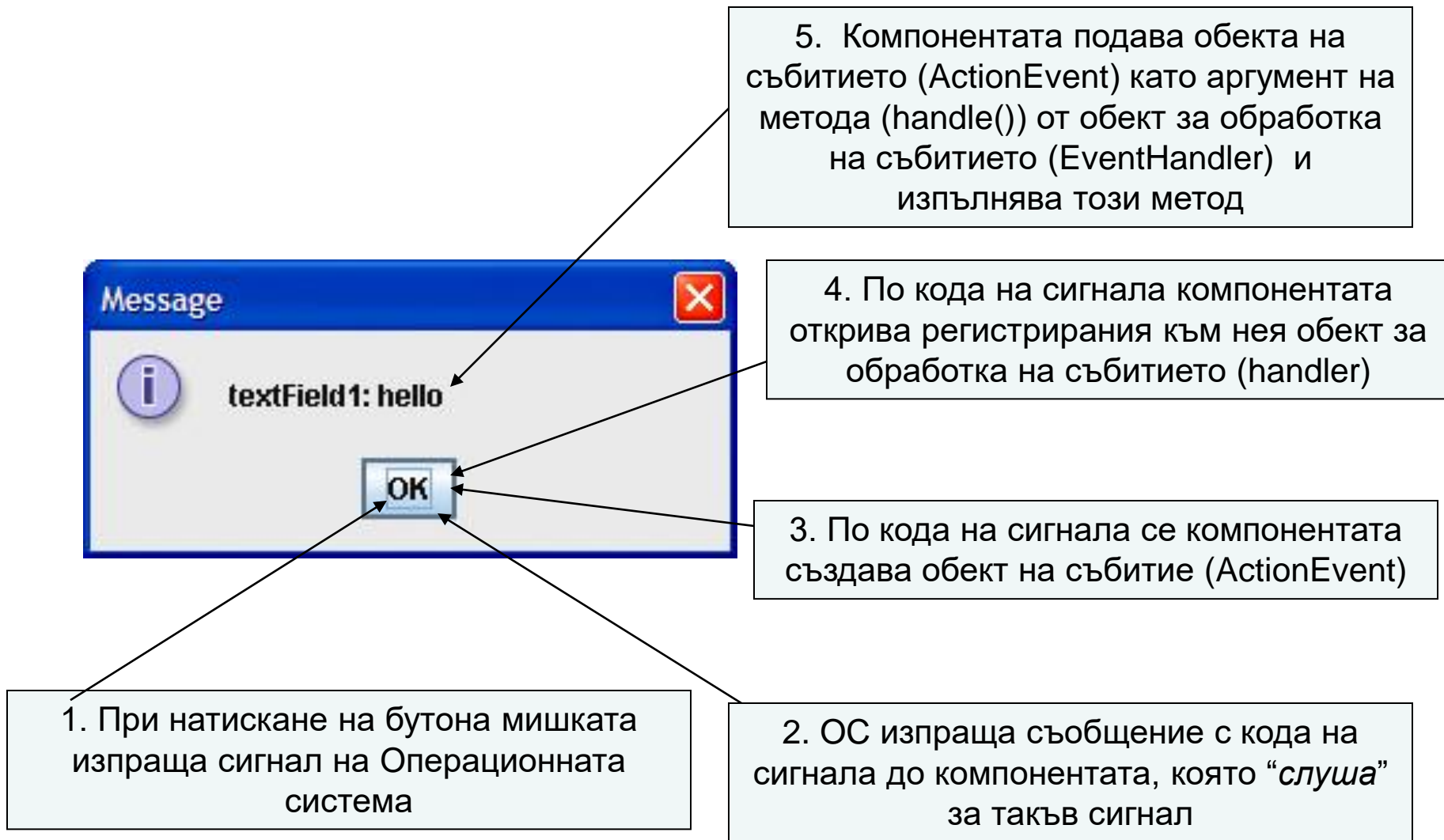
- Регистрирането се извършва като се
 - Извиква метод
`setOnEvent-Type(EventHandler<Object-of-Event>)`
 - Например, за събитието **ACTION**
 - `setOnAction (EventHandler<ActionEvent>)`
 - `EventHandler<ActionEvent>` обектът започва да “слуша” за събития от типа на ACTION
 - *Пропускането да се регистрира обработчика на събитието, не позволява на приложението да реагира на това събитие*

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 (създава 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)`

Обработване на събитие



Регистриране на събитията

Всеки обект от клас произведен 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)