Programmazione Orientata agli Oggetti

Programmazione ad Eventi

GUI e JavaFX

Sommario

- Introduzione alla Programmazione ad Eventi
- Legami con la PC e la programmazione di GUI
- Source / Listener / Handler
- JavaFX
  - Scene Graph
  - Eventi
  - Application Thread
- Background Thread
- Fenomeni di Interferenza
- Approfondimenti su JavaFX
- Un modello concettuale
- Conclusioni

Programmazione Concorrente: Ulteriori Motivazioni

- Principale motivazioni della *Programmazione Concorrente* (PC) sinora:
  - ✔ Ottenere speed-up sfruttando le moderne architetture multi-core
- Lo studio delle principali nozioni di PC ha importanti motivazioni sin da prima dell'*era multi-core*:
  - ✔ tutte le applicazioni dotate di una interfaccia grafica (*Graphical User Interface - GUI)* sono *intrinsecamente* concorrenti
    - ✔ l'utente interagisce liberamente con la GUI in qualsiasi momento, non solo quando il thread che esegue il **`main()`** è «pronto» all'input

✔ N.B. Cercare una soluzione sequenziale, ad un problema intrisecamente concorrente, può solo causare grandi problemi...

Programmazione Concorrente & GUI

- Alcune nozioni di base di PC sono quindi *necessarie* per programmare applicazioni dotate di una GUI
- Ne deriva un problema molto sentito (nell'industria):
  - ✔ La PC è difficile, non adatta a programmatori poco esperti
  - ✔ Le GUI sono diffusissime
    - praticamente tutte(!) le applicazioni «mobile» ne fanno uso
    - molte di quelle desktop (gli IDE come Eclipse per primi)
- Come rendere possibile la programmazione di GUI *anche* da parte di programmatori poco esperti?
- Le librerie per sviluppo di GUI «impongono» un modello di programmazione per nascondere gli aspetti più sofisticati e legati alla natura concorrente…
  - ✔ (per quanto possibile! >>)

La Programmazione ad Eventi (1)

- Questo modello di programmazione spesso prende il nome di *Programmazione ad Eventi*

- I programmi si dicono *ad eventi* perché l'esecuzione viene modellata come risposta a «stimoli» dall'«esterno»
  - chiamati «eventi»
  - *asincroni* rispetto all'esecuzione, ovvero:
    - non è possibile prevedere in quale momento arrivino rispetto all'esecuzione del thread *Main* (quello che esegue **`main()`**)
    - e quest'ultimo, non può passare la propria esistenza ad aspettarli, se ha anche altro da fare…

La Programmazione ad Eventi (2)

- Si discosta sensibilmente dalla programmazione sequenziale
  - Perché, praticamente, gli eventi NON vengono gestiti dall'usuale thread *Main*
  - Sono gestiti da un *unico thread aggiuntivo* dedicato allo scopo
  - ✔ E' il più semplice modello che rispecchia la natura, comunque concorrente, del problema
- Viene infatti adottato da praticamente tutte le librerie a supporto dello sviluppo di applicazioni grafiche
  - ✔ Il nome del thread cambia di libreria in libreria, ma è sempre uno solo
- Ci limitiamo alla programmazione ad eventi con riferimento allo sviluppo di applicazioni grafiche
  - ✔ la *Programmazione ad Eventi* ha conosciuto importanti applicazioni anche in contesti molto diversi ed alieni alle GUI

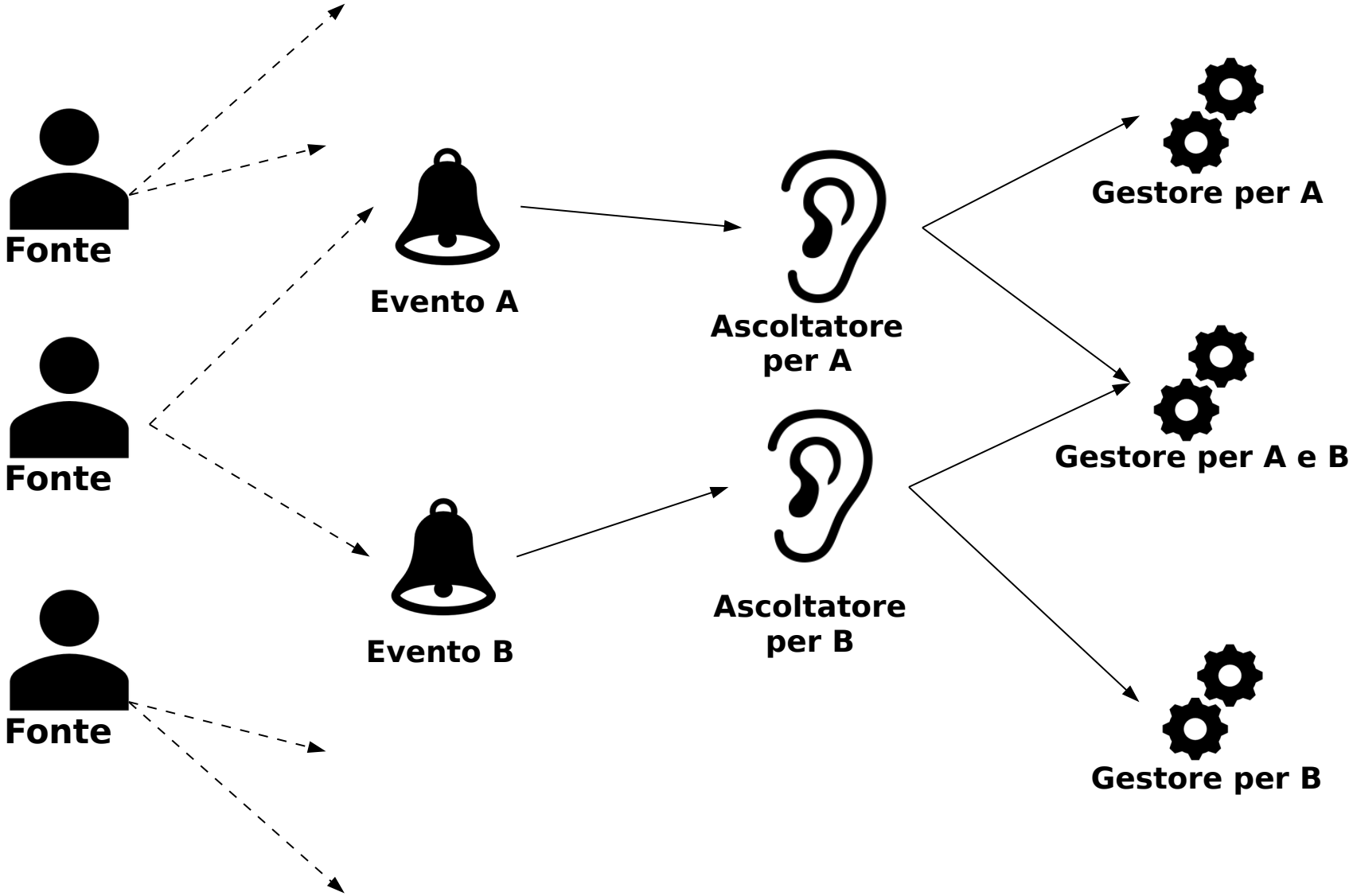
Eventi

- Un’applicazione dotata di GUI «genera» eventi quando ad es. l’utente esegue operazioni interattive come:
  - «Click del mouse»
  - «Movimento del mouse»
  - «Pressione di un tasto sulla tastiera»
  - «Selezione di un elemento grafico»

  ... e molti molti altri ancora...

- Ad ogni evento è possibile associare:
  - La *Sorgente* o *Fonte*: chi genera l’evento (*Source*)
  - Chi è interessato ad intercettare l’evento (*Listener*)
  - Il *Gestore* dell’evento (*Handler*) che lo prende in carico una volta intercettato

Source – Listener - Handler



JavaFX: Una Libreria per la Progettazione di GUI Moderne

- *JavaFX* è una libreria (Java) che consente lo sviluppo di *Rich Client Applications*
  - GUI costruite componendo elementi grafici predefiniti fornite dalla libreria stessa o da sue estensioni
  - Widget

- L'API di JavaFX direttamente integrata in JDK6+
  - Portabilità su diversi sistemi operativi (anche mobile!)
    - Non più da Java 11+! *https://openjfx.io/openjfx-docs/*
      - Per avere cicli di rilascio disaccoppiati Java / JavaFX
    - Sempre possibile ottenere separatamente la libreria (>>)

JavaFX ed Obiettivi Formativi

- JavaFX: Terzo tentativo della serie (!) dopo
  - *AWT (1995)*
  - *Swing (1996)*
- Forse l'unico veramente riuscito
  - ✔ Ma fuori tempo massimo!
  - Ormai molte GUI sono su web...
- *Disclaimer:* l'uso che facciamo di JavaFX è strumentale al nostro vero obiettivo (formativo) che consiste nell'introduzione alla *Programmazione ad Eventi*
  - Discuteremo anche la tecnologia, ma *solo* per quello che serve allo scopo
  - ✔ La discussione in *ampiezza* dei dettagli della *tecnologia* JavaFX non è tra i nostri obiettivi formativi
- ✔ N.B.: Il modello di programmazione che presenteremo è comunque sottostante molte altre piattaforme, incluso tutte quelle più diffuse, per la progettazione di applicazioni grafiche (>>)

Avviare/Terminare un'Applicazione JavaFX

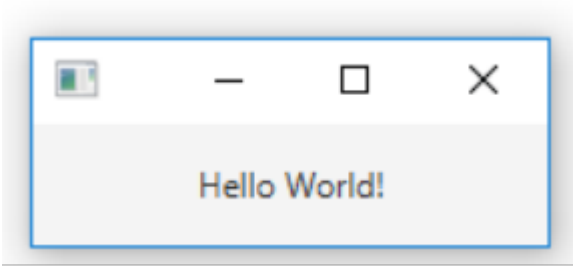
- La classe principale di una applicazione con GUI JavaFx deve estendere la classe astratta **`javafx.application.Application`**

```
public abstract class Application {
    public abstract void start(Stage palcoscenico) throws Exception;
}
```

- Il metodo **`main()`** di solito finisce per contenere solo la chiamata al metodo statico **`Application.launch()`**
- Il metodo **`launch()`** si occuperà, tra le altre cose, anche di chiamare il metodo **`Application.start()`**
  - Quest'ultimo si occupa del reale avvio dell'applicazione costruendo lo **`Stage`** (ovvero, il "palcoscenico") della GUI
- Una volta avviata, l'applicazione permane in attesa di eventi
- Terminerà quando:
  - viene chiamato il metodo **`Platform.exit()`**, oppure:
  - vengono chiuse tutte le finestre dell'applicazione

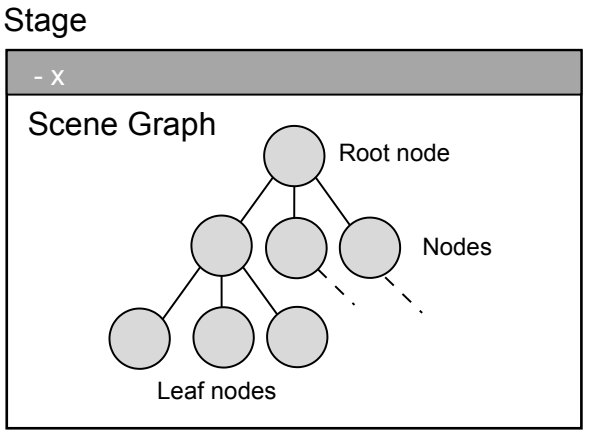
Esempio Applicazione JavaFX

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.stage.Stage;

public class HelloWorldGUI extends Application {

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

    @Override
    public void start(Stage palcoscenico) throws Exception {
        final Label etichetta = new Label("Hello World!");
        etichetta.setAlignment(Pos.CENTER);

        final Scene scena = new Scene(etichetta);

        palcoscenico.setScene(scena);
        palcoscenico.show(); // apre il “sipario”
    }
}
```



Elementi in «Scene Graph»

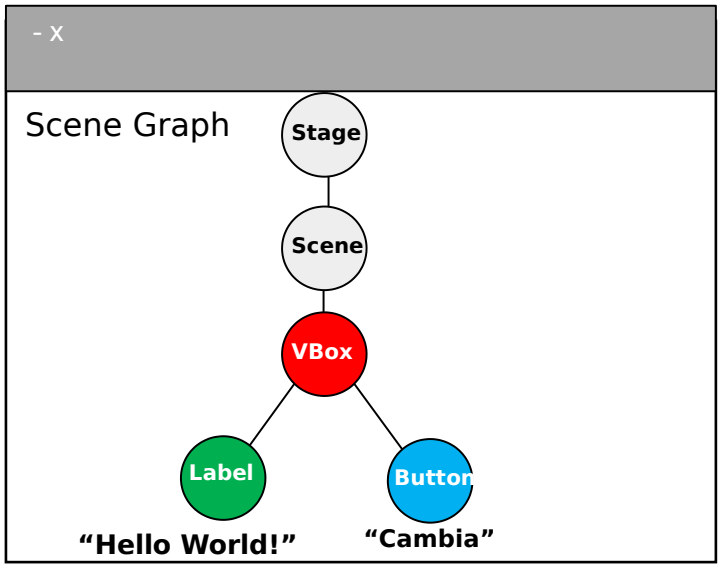
- Sinora «scena» composta di una sola **`Label`**
- Applicazioni più verosimili contano molti elementi
  - ➢ questi devono essere disposti ed organizzati, anche graficamente, per occupare posizioni prevedibili ed opportune nella GUI mostrata agli utenti
- Serve un modo per disporre gli elementi della scena, lo «Scene Graph»
  - Elementi *foglia*:
    - Possiedono una controparte visuale diretta
      - √ Ad es. inseriamo un pulsante **`Button`**
  - Elementi di *composizione*:
    - Non possiedono una controparte visuale diretta
    - Servono ad organizzare altri elementi (che figurano come figli dello stesso)
      - √ Ad es. **`VBox`**
      - Permette di incolonnare nodi uno sotto l'altro

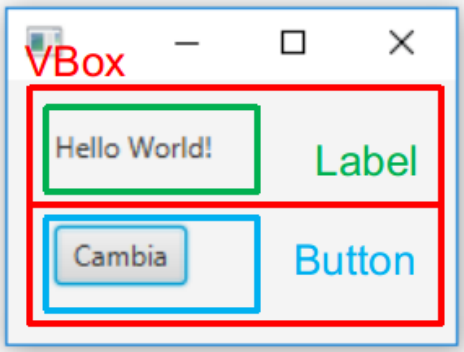


Organizzazione Gerarchia degli Elementi Grafici di una GUI

- Gli elementi di una GUI sono organizzati in un albero
  - Radicato in **`Stage`** padre di **`Scene`**
  - Esempio:
    - **`Scene`** padre di un oggetto di tipo **`VBox`** con due figli
    - Una **`Label`**: Usata per mostrare testi (“Hello World!”)
    - Un **`Button`**: Usato per creare pulsanti (“Cambia”)
- Quando si crea una nuova scena è necessario sempre specificare il nodo radice

```
Scene scena = new Scene(vbox);
```



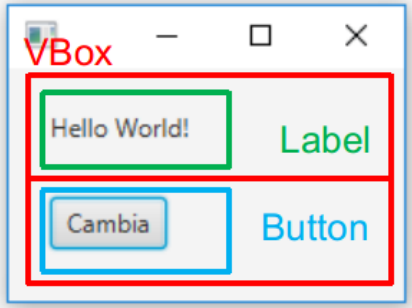


JavaFX GUI: Esempio (1)

```
@Override
public void start(Stage palcoscenico) throws Exception {
    final VBox incolonnati = new VBox(5.);
    incolonnati.setPadding(new Insets(10.));
    final Label etichetta = new Label("Hello World!");
    etichetta.setAlignment(Pos.CENTER);
    final Button pulsante = new Button("Cambia");


    incolonnati.getChildren().add(etichetta);
    incolonnati.getChildren().add(pulsante);
    final Scene scena = new Scene(incolonnati);
    palcoscenico.setScene(scena);
    palcoscenico.show();
}
```

Impostazioni "visuali"

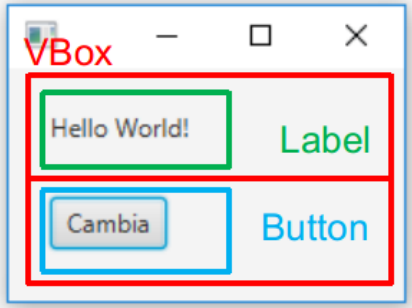
- **`VBox`** contiene un'etichetta e un pulsante
  - **`getChildren()/add()`** per aggiungere figli
- Alcuni parametri necessari per raffinare alcuni dettagli visuali
  - sembrano meno importanti, ma hanno effetti decisivi sull'usabilità della GUI



JavaFX GUI: Esempio (2)

```
@Override
public void start(Stage palcoscenico) throws Exception {
     final VBox incolonnati = new VBox(5.);
     incolonnati.setPadding(new Insets(10.));
     final Label etichetta = new Label("Hello World!");
     etichetta.setAlignment(Pos.CENTER);
     final Button pulsante = new Button("Cambia");
     final GestoreClick gestore = new GestoreClick(etichetta);
     pulsante.addEventHandler(MouseEvent.MOUSE_CLICKED, gestore);
     incolonnati.getChildren().add(etichetta);
     incolonnati.getChildren().add(pulsante);
     final Scene scena = new Scene(incolonnati);
     palcoscenico.setScene(scena);
     palcoscenico.show();
}
```

- Al click sul pulsante viene generato un evento `MouseEvent.MOUSE_CLICKED`
- L'evento viene gestito da un *handler* istanza della classe `GestoreClick`
  - quando usato cambia il testo della **`Label`**



Esempio di `EventHandler`: `GestoreClick`

```
public class GestoreClick implements EventHandler<MouseEvent> {

    private Label daModificare;

    public GestoreClick(Label etichetta) {
        this.daModificare = etichetta;

    }

    @Override
    public void handle(MouseEvent evento) {
        if (evento.getButton().equals(MouseButton.PRIMARY))
            this.daModificare.setText("Cambiato!");
    }
}
```

- Quando si verifica l'evento «*pulsante premuto*» viene attivato il corrispondente handler **`GestoreClick`**
  - Invocandone il metodo **`handle()`**
  - Nell'es. si occupa di cambiare il testo della **`Label daModificare`**
    - Verso la quale possiede un rif. ricevuto tramite costruttore
    - **`MouseButton.PRIMARY`** è il tasto sinistro del mouse

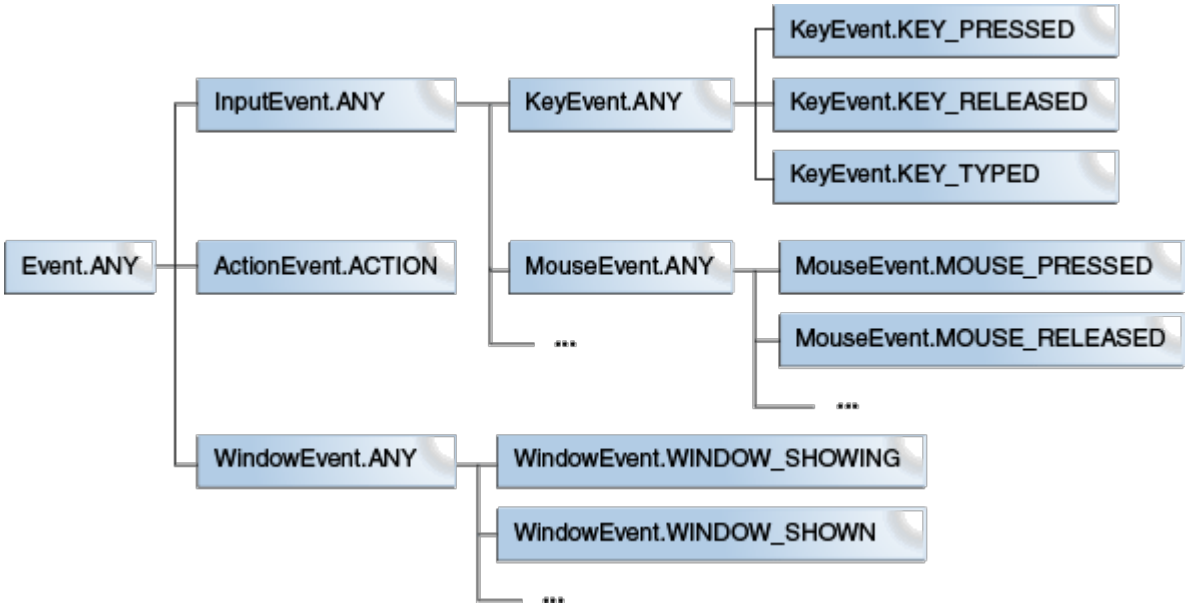
Definizione di un Gestore di Eventi

- Le azioni da eseguire quando si verifica un *evento* sono specificate mediante un gestore
- Interface **`EventHandler<T>`** a singolo metodo astratto **`handle()`**:

```
public interface EventHandler<T extends Event>
                    extends EventListener {
    void handle(T event);
}
```

  - **`handle()`** invocato nel momento in cui l'evento si verifica
  - Il parametro contiene informazioni sull'evento
  - Può *modificare* la scena se dispone dei riferimenti ai suoi *nodi*

Tipi e Gerarchie di Eventi

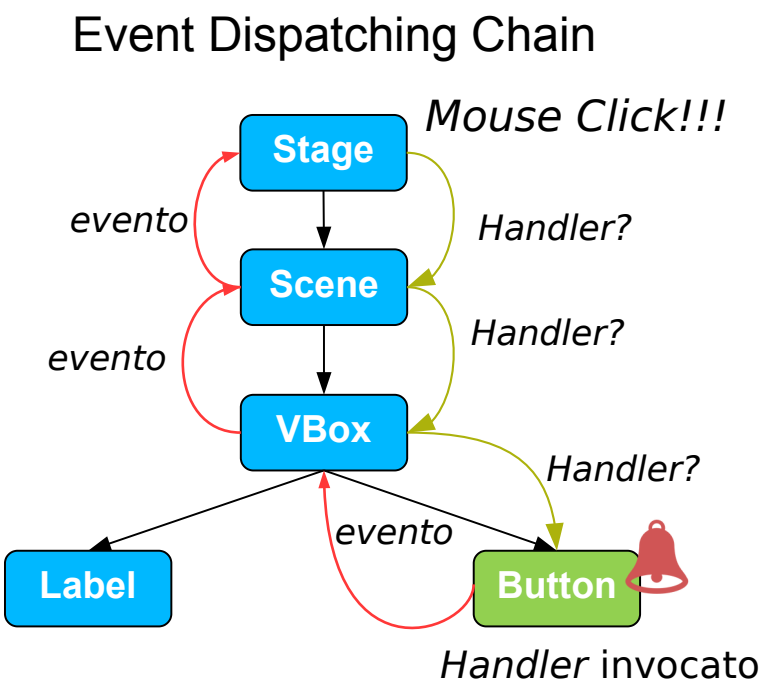


- Eventi associati ad una gerarchia di tipi
- La radice è **`Event.ANY`**
  - ✔ (cattura *ogni* tipo di evento)
- Verso le foglie eventi via via più specifici

- Esiste una miriade di possibili eventi da gestire
- Gli eventi sono organizzati gerarchicamente su un doppio «binario»
  - Eventi modellati da oggetti sottotipo della classe **`javafx.event.Event`**
  - «Tipo di evento»: **`javafx.event.EventType<T extends Event>`**
  - «Event target»: destinatari degli eventi **`javafx.event.EventTarget`**
- I gestori registrati presso gli elementi della GUI possono identificare e catturare specifici tipi di eventi
  - *click del mouse*
  - *pressione di un tasto (qualsiasi;* oppure: *uno in particolare)*

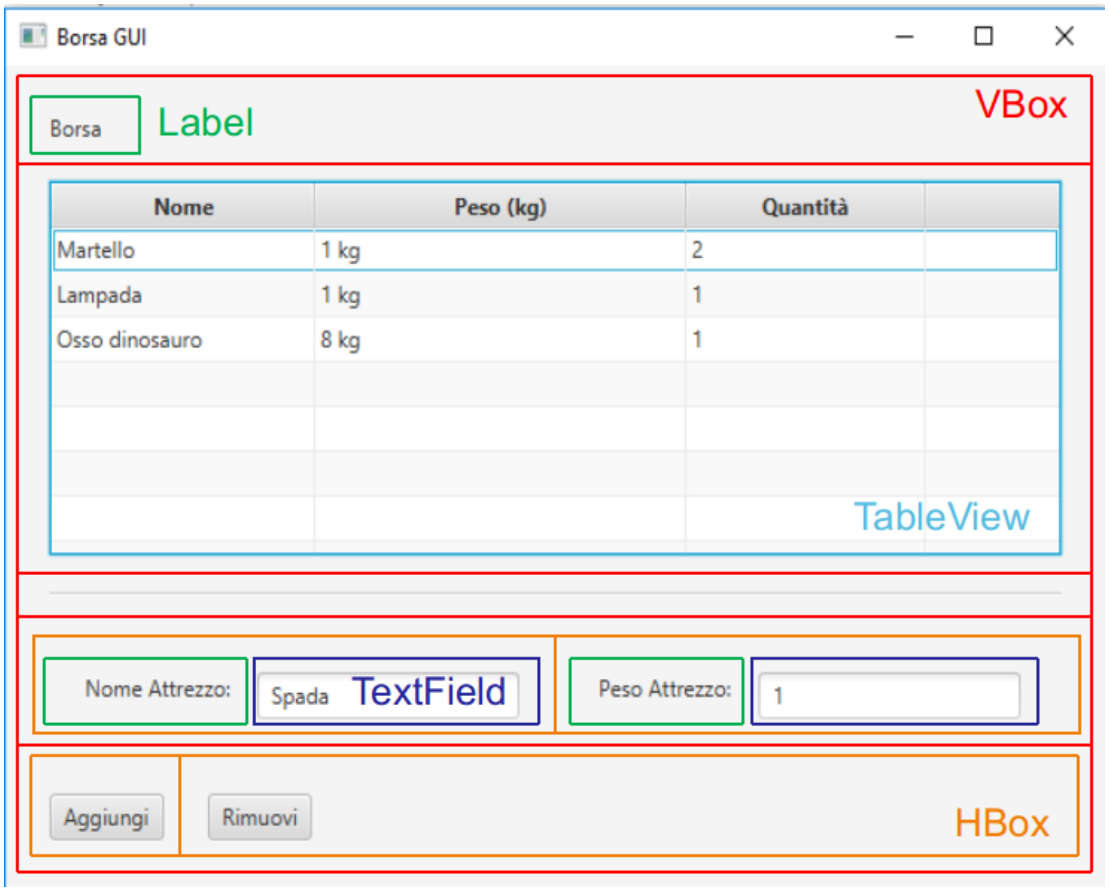
Event Dispatching Chain

- Uno *Stage* rappresenta l'intera finestra dell'applicazione
  - Comprensiva di barra del titolo



- Gli eventi sono propagati lungo la cosidetta *Event Dispatching Chain* della *Scene Graph* secondo un ordinamento *top-down* che confluisce nel nodo *target* (ad es. un pulsante) su cui si è generato l'evento stesso
- Gli handler degli elementi coinvolti, se presenti e registrati per un evento di tipo compatibile con quello in corso di distribuzione, sono chiamati secondo un ordinamento *bottom-up*

JavaFX: un Mondo di Possibilità

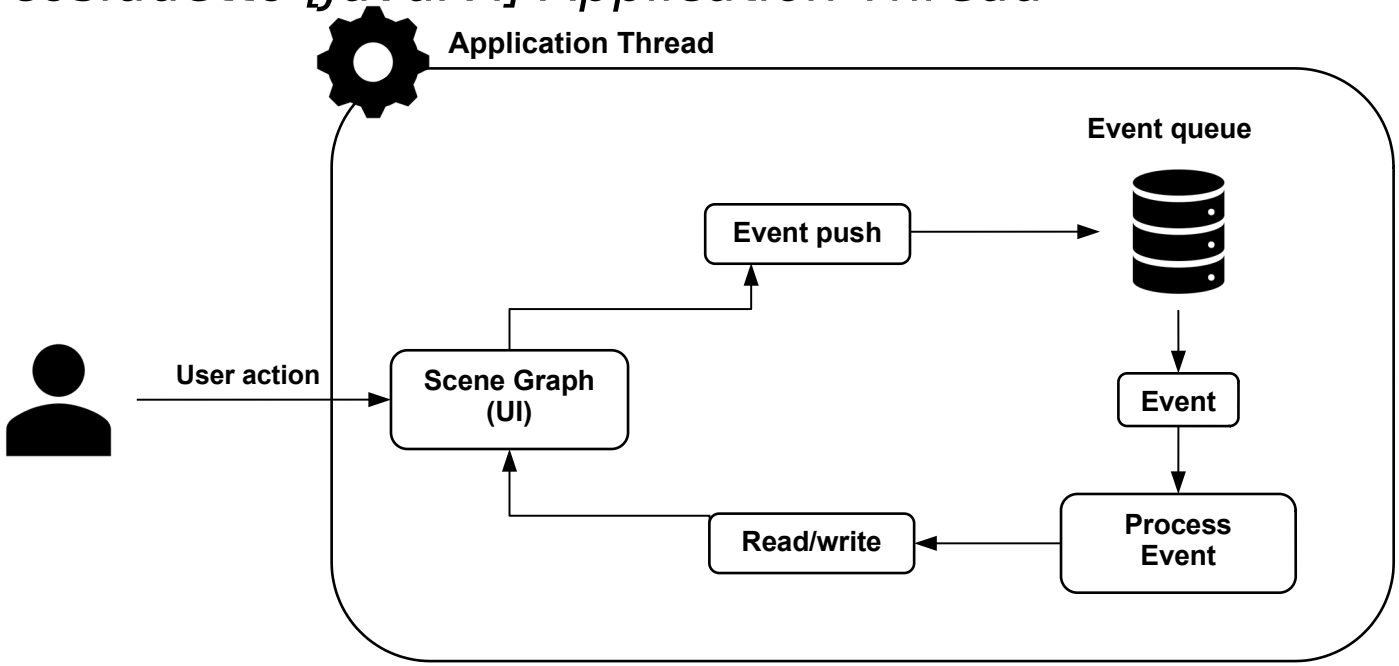


- JavaFX offre una moltitudine di componenti differenti
- La gestione di una borsa nello studio di caso...
- Alcuni tipi elementi già incontrati come:
  - **`Vbox`**
  - **`Label`**
  - **`Button`**
- Ma anche altri
  - **`HBox`** versione orizzontale di **`VBox`**
  - **`TableView`** che mostra dati in forma tabulare
  - **`TextField`** permette di inserire un campo di testo

... e molti altri ancora

Modello di Concorrenza di JavaFX: *Application Thread*

- Quando un'applicazione JavaFX viene lanciata questa crea il cosiddetto *[JavaFX] Application Thread*



- Gli eventi generati sono catturati e messi in un coda
  - La *Event Queue*
- L'*Application Thread* è adibito alla gestione della GUI:
  - preleva dalla coda e processa **serialmente** tutti gli eventi
  - legge e/o modifica lo stato degli elementi in scena

Confinamento per Thread in JavaFX

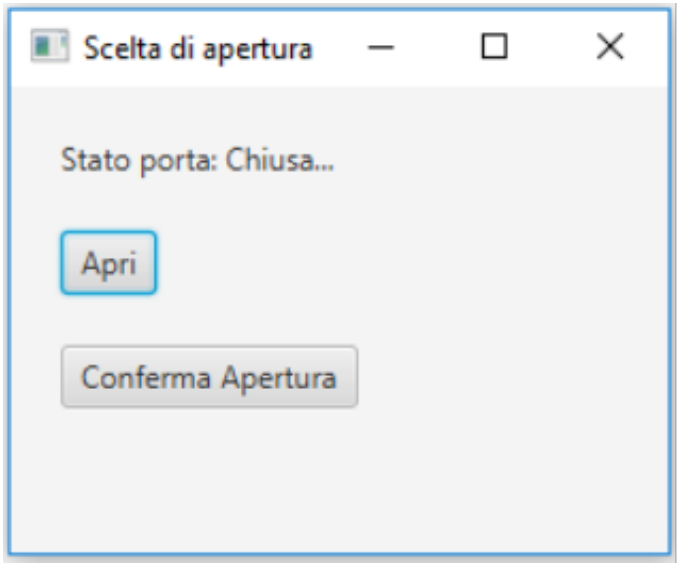
- Per applicazioni JavaFX l'*Application Thread*
  - esegue il metodo **`start()`**
    - crea **`Scene`** e **`Stage`**
  - quindi viene adibito al processamento continuo e seriale degli eventi
- Questo modello di programmazione concorrente si basa sulla cosidetta *tecnica di confinamento per thread*:
  - Esiste un unico thread autorizzato a cambiare lo stato della GUI
    - possiede delega *esclusiva* agli aggiornamenti della GUI
- Perché tutta questa attenzione ai thread usati?

✔ Diversi thread, senza precauzioni, *NON* posso aggiornare e leggere contemporaneamente le stesse aree di memoria

- Altrimenti, vedremo dopo: *Pericolo di Interferenza* (>>)

Application Thread ed Eventi

- L'*Application Thread* processa gli eventi *sequenzialmente*:
  - se, volutamente, includiamo un evento che impiega molto tempo per essere gestito, si chiarisce il suo ruolo
  - non può passare a processare gli eventi successivi sino a quando questo, «pesante», non viene «evaso»
- Una buona scusa per avere un evento con tempo di processamento lungo:
  - Una "porta temporizzata"
- Si sblocca solo con una "sequenza di apertura" che prevede due pulsanti
  - *Uno chiede l'apertura* (inizio sequenza)
  - *L'altro la conferma* (fine sequenza)
- Solo dopo aver premuto il pulsante "Apri" si abilita il pulsante "Conferma Apertura"
  - *deve essere premuto entro un breve intervallo di tempo dall'azionamento del primo pulsante*

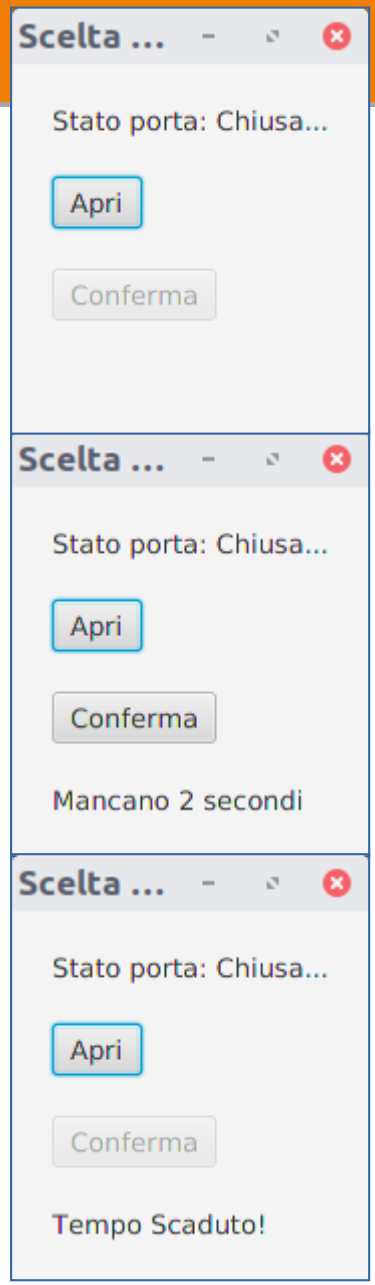


Un Evento Lungo Due Secondi

- Una **`Label`** riporta lo stato della porta come

  “Chiusa” | “Aperta”

- Un primo **`Button`** permette l’inizio della sequenza
- Un secondo **`Button`** è inizialmente disattivato
  - Viene attivato premendo il primo pulsante “Apri”
  - Accetta poi una conferma purché arrivi entro 2 sec.
  - Allo scadere dei 2 sec. viene ridisattivato
  - Se premuto quando attivo
- Un’ulteriore **`Label`** per il *timing*
  - viene aggiornata allo scorrere del tempo



SceltaTemporizzataGUI

```
@Override
public void start(Stage palcoscenico) throws Exception {
  final VBox verticale = new VBox(20.);
  verticale.setPadding(new Insets(20.));

  final Label stato = new Label("Stato porta: Chiusa...");
  final Label timer = new Label();
  final Button apri = new Button("Apri");
  final Button conferma = new Button("Conferma");
  conferma.setDisable(true);

  GestoreRichiesta gestoreRichiesta = new GestoreRichiesta(conferma,stato);
  GestoreConferma gestoreConferma = new GestoreConferma(stato);

  apri.addEventHandler(MouseEvent.MOUSE_CLICKED, gestoreRichiesta);
  conferma.addEventHandler(MouseEvent.MOUSE_CLICKED, gestoreConferma);

  verticale.getChildren().addAll(stato, apri, conferma, timer);

  final Scene scena = new Scene(verticale);
  palcoscenico.setScene(scena);
  palcoscenico.setTitle("Scelta Temporizzata");
  palcoscenico.show();
}
```

GestoreRichiesta

- **`this.conferma.setDisable(boolean)`**
  attiva/disattiva il pulsante di conferma ricevuto già nel costruttore

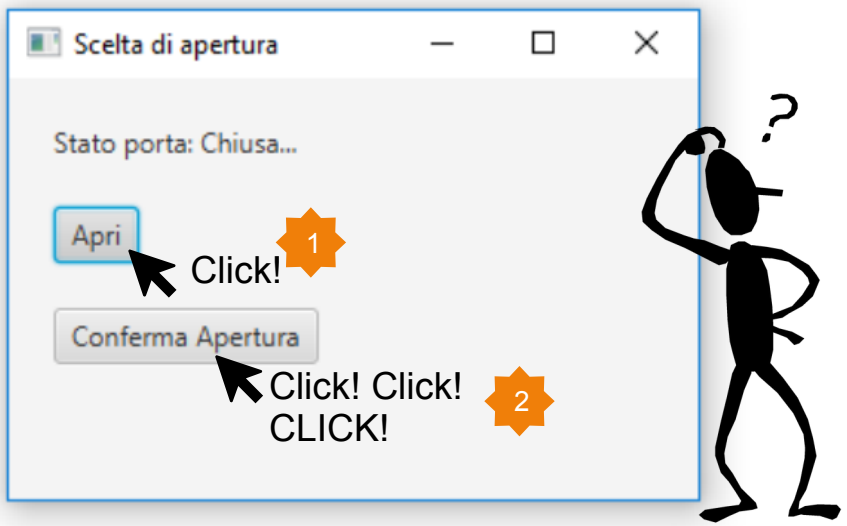
```
public class GestoreRichiesta implements EventHandler<MouseEvent> {
    private Button conferma;
    private Label timer;
    public GestoreRichiesta(Button pulsante, Label label) {
        this.conferma = pulsante;
        this.timer = label;
    }
    @Override
    public void handle(MouseEvent event) {
        if (event.getButton() == MouseButton.PRIMARY) {
            try {
                this.conferma.setDisable(false); // attiva conferma
                timer.setText("Mancano 2 secondi");
                Thread.sleep(1000); // pausa di un secondo
                timer.setText("Manca 1 secondo");
                Thread.sleep(1000);
                timer.setText("Tempo Scaduto!");
                this.conferma.setDisable(true); // disattiva conferma
            } catch (InterruptedException e) { }
    }}
```

GestoreConferma

```
public class GestoreConferma
        implements EventHandler<MouseEvent> {

    private Label stato;

    public GestoreConferma(Label daCambiare) {
        this.stato = daCambiare;

    }

    public void handle(MouseEvent evento) {
        if (event.getButton() == MouseButton.PRIMARY)
            this.stato.setText("Stato porta: Aperta!");

    }
}
```

- **`GestoreConferma`** viene interpellato quando l'utente conferma l'apertura tramite il secondo pulsante
  - cambia il testo della **`Label`**

Un Evento Troppo Lungo!?



Effettuando questa sequenza:

(a) Click su “Apri”

(b) Click su “Conferma Apertura”

Qualcosa non va: il pulsante non può essere premuto!

(c) Dopo 2 secondi, la GUI torna a funzionare

è comunque possibile azionare solo il pulsante “Apri”

✔ Perché il pulsante “Conferma Apertura” non può essere azionato?

Gestione Seriale degli Eventi

- JavaFX prevede un solo thread che si occupa di
  - Aggiornare la grafica
  - Gestire gli eventi *serialmente*
    - Eventi processati uno alla volta, ciascuno solo dopo aver completato l'elaborazione dell'evento precedente
    - Il tempo di gestione di ciascun evento finisce per avere (anche se solo indirettamente) un importante effetto sul tempo di risposta a tutti gli eventi successivi
- L'*Application Thread* rimane impossibilitato ad elaborare gli eventi che invece di norma è invece chiamato a gestire quanto più tempestivamente possibile per non perdere reattività

  ✔ La perdita di reattività ha effetti facilmente osservabili:
    - Impossibile interagire con l'UI

Background Thread

- Bisogna pertanto liberare l'*Application Thread* dai compiti troppo gravosi
  - ✔ Altrimenti nessuno rimane a gestire i nuovi eventi in arrivo
- Si usa un meccanismo di delega verso uno o più **ulteriori** *Background Thread*
  - ✔ si parla anche di *worker (*o *helper) thread*
  - N.B. passiamo da 2 thread (per la nostra applicazione con UI: *Main* & *Application Thread*) a 2+ thread
- Il loro ruolo è quello di «aiutare/lavorare» per l'*Application Thread* «liberandolo» dei compiti più gravosi

SistemaSicurezzaPorta

```
public class SistemaSicurezzaPorta extends Thread {

     private Button conferma; // di conferma apertura porta
     private Label timer;

     public SistemaSicurezzaPorta(Button pulsante, Label timer) {
          this.conferma = pulsante;
          this.timer = timer;
     }

     @Override
     public void run() {
          try {
               this.conferma.setDisable(false);
               this.timer.setText("Mancano 2 secondi");
               Thread.sleep(1000);
               this.timer.setText("Manca 1 secondo");
               Thread.sleep(1000);
               this.timer.setText("Tempo Scaduto!");
               this.conferma.setDisable(true);
          } catch (InterruptedException e) { }
     } }
```

✔ Questa classe estende **`java.lang.Thread`**

- Aspetta 2 secondi prima di disattivare il pulsante di conferma
  - Aggiorna anche il timer ogni secondo

GestoreRichiesta

```
@Override
public void handle(MouseEvent event) {
  if (event.getButton() == MouseButton.PRIMARY) {
    final Thread t = new SistemaSicurezzaPorta(this.pulsante);
    t.start();
  }
}
```

- La classe **`GestoreRichiesta`** va modificata specificando la nuova logica per attivare il background thread
- In questo modo l'*Application Thread* non viene più *bloccato* nell'attesa
  - libero di gestire i nuovi eventi, incluso quelli relativi al pulsante di conferma
- Quindi ….>>….

Una Seconda Esecuzione

….Cliccando sul pulsante “Apri” si ottiene però un’eccezione

- JavaFX NON permette a thread diversi dall’*Application Thread* di modificare gli elementi della scena
  - Si forza il rispetto dell’«*esclusiva*» con un controllo a tempo dinamico sul thread che tenta l’aggiornamento
    - ✔ controllo implementato usando il metodo statico **`Thread.currentThread()`**
  - L’eccezione viene sollevata quando andiamo a modificare il valore della **`Label`** che rappresenta il nostro conto alla rovescia

```
Exception in thread "Thread-3" java.lang.IllegalStateException: Not on FX application thread; currentThread = Thread-3
      at com.sun.javafx.tk.Toolkit.checkFxUserThread(Toolkit.java:279)
      at com.sun.javafx.tk.quantum.QuantumToolkit.checkFxUserThread(QuantumToolkit.java:423)
      at javafx.scene.Parent$2.onProposedChange(Parent.java:367)
      at com.sun.javafx.collections.VetoableListDecorator.setAll(VetoableListDecorator.java:113)
      at com.sun.javafx.collections.VetoableListDecorator.setAll(VetoableListDecorator.java:108)
      at com.sun.javafx.scene.control.skin.LabeledSkinBase.updateChildren(LabeledSkinBase.java:575)
      at com.sun.javafx.scene.control.skin.LabeledSkinBase.handleControlPropertyChanged(LabeledSkinBase.java:204)
      at com.sun.javafx.scene.control.skin.LabelSkin.handleControlPropertyChanged(LabelSkin.java:49)
      at com.sun.javafx.scene.control.skin.BehaviorSkinBase.lambda$registerChangeListener$61(BehaviorSkinBase.java:197)
      at
com.sun.javafx.scene.control.MultiplePropertyChangeListenerHandler$1.changed(MultiplePropertyChangeListenerHandler.java:55)
      at javafx.beans.value.WeakChangeListener.changed(WeakChangeListener.java:89)
      at com.sun.javafx.binding.ExpressionHelper$SingleChange.fireValueChangedEvent(ExpressionHelper.java:182)
      at com.sun.javafx.binding.ExpressionHelper.fireValueChangedEvent(ExpressionHelper.java:81)
      at javafx.beans.property.StringPropertyBase.fireValueChangedEvent(StringPropertyBase.java:103)
      at javafx.beans.property.StringPropertyBase.markInvalid(StringPropertyBase.java:110)
      at javafx.beans.property.StringPropertyBase.set(StringPropertyBase.java:144)
      at javafx.beans.property.StringPropertyBase.set(StringPropertyBase.java:49)
      at javafx.beans.property.StringProperty.setValue(StringProperty.java:65)
      at javafx.scene.control.Labeled.setText(Labeled.java:145)
      at SistemaSicurezzaPorta.run(SistemaSicurezzaPorta.java:20)
```

Interferenza

- Perché tutta questa attenzione ai thread usati?
  - Diversi thread, senza precauzioni, *NON* posso aggiornare e leggere contemporaneamente le stesse aree di memoria
- L'esecuzione di due o più thread che accedono (anche in scrittura) ad una stessa area di memoria concorrentemente può produrre stati inconsistenti degli oggetti interessati con effetti
  - Imprevedibili
  - Irriproducibili
- Banalizzando: l'interferenza si ha ogni qualvolta un *f.d.e.* legge ciò che altri *f.d.e.* scrivono ancora prima che abbiano finito di scriverlo
  - Serio e concreto rischio di leggere oggetti in stati transitoriamente *inconsistenti*

Esempio di Interferenza (1)

```java
package it.interferenza;

class Persona {

	private String nome;

	private Coppia<Persona> coppia;

	public Persona(String nome) {
		this.nome = nome;
	}

	public String getNome() {
		return this.nome;
	}

	public void setCoppia(Coppia<Persona> coppia) {
		this.coppia = coppia;
	}

	public boolean isSposata() {
		return (this.coppia != null );
	}

}
```

```java
package it.interferenza;


public class Coppia<T> {

    private T primo;
    private T secondo;

    public Coppia(T primo, T secondo) {
        this.primo = primo;
        this.secondo = secondo;
    }

    public T getPrimo() {
        return this.primo;
    }

    public T getSecondo() {
        return this.secondo;
    }

}
```

Esempio di Interferenza (2)

```
package it.interferenza;
import java.util.*;

public class Cerimoniere {
  final static int N = 100;
  static public void main(String[] args) throws Exception {
    new Cerimoniere().main();
  }
  private Random random;
  private void main() throws InterruptedException {
    this.random = new Random();
    final List<Persona> uomini = creaNpersone(N);
    final List<Persona> donne = creaNpersone(N);
    final List<Integer> indiciU = numeriCasualiDa0AdNmeno1(N);
    final List<Integer> indiciD = numeriCasualiDa0AdNmeno1(N);
    … … … // (>> codice a seguire)
  }
  private List<Integer> numeriCasualiDa0AdNmeno1(int n) {
    final List<Integer> indici = new ArrayList<>(n);
    for(int i=0; i<n; i++)
      indici.add(i);
    Collections.shuffle(indici); // permuta casualmente
    return indici;
  }
  private List<Persona> creaNpersone(int n) {
    final List<Persona> persone = new ArrayList<Persona>();
    for(int i=0; i<n; i++)
      persone.add(new Persona("-"+i+"-"));
    return persone;
  }…
}
```

Esempio di Interferenza (3)

```
  private void main() throws InterruptedException {
    … // (<< codice slide precedente)
      // quanti uomini ne escono sposati, quante donne?
      for(int i=0; i<15; i++) {
        System.out.println("Sposati al passo i: "
                           + "Uomini=" + contaSposati(uomini)
                           + " Donne=" + contaSposati(donne));
        Thread.sleep(1000);
      }
    }
  }

  private int contaSposati(List<Persona> persone) {
    int count = 0;
    for(Persona persona : persone) {
      if (persona.isSposata()) count++;
    }
    return count;
  }

}
```

Esempio di Interferenza (3)

```
private void main() throws InterruptedException {
  …
  // celebrazione di massa, un thread per coppia
  for(int i=0; i<N; i++) {
    new Thread(new Runnable() {
      @Override public void run() { cerimonia(uomini,indiciU,donne,indiciD); }
    }).start();
  }
  …
}

private void cerimonia(List<Persona> uomini, List<Integer> uomoAcaso,
                       List<Persona> donne, List<Integer> donnaAcaso) {
  try {
    final Persona sposo = uomini.get(uomoAcaso.remove(0));
    final Persona sposa = donne.get(donnaAcaso.remove(0));
    final Coppia<Persona> coppia = new Coppia<Persona>(sposo,sposa);
    Thread.sleep(this.random.nextInt(10000));

    sposo.setCoppia(coppia);
    Thread.sleep(this.random.nextInt(1000));
    sposa.setCoppia(coppia);
  } catch (InterruptedException e) { e.printStackTrace(); }
}
```

Esempio di Interferenza (4)

```
 0) Uomini Sposati=1 Donne Sposate=0
 1) Uomini Sposati=19 Donne Sposate=12
 2) Uomini Sposati=23 Donne Sposate=20
 3) Uomini Sposati=38 Donne Sposate=26
 4) Uomini Sposati=48 Donne Sposate=44
 5) Uomini Sposati=56 Donne Sposate=52
 6) Uomini Sposati=68 Donne Sposate=63
 7) Uomini Sposati=74 Donne Sposate=70
 8) Uomini Sposati=86 Donne Sposate=82
 9) Uomini Sposati=93 Donne Sposate=92
10) Uomini Sposati=100 Donne Sposate=95
11) Uomini Sposati=100 Donne Sposate=100
12) Uomini Sposati=100 Donne Sposate=100
...
```

- ✔ Fuori metafora, errori legati ai fenomeni di interferenza dovuti ad una programmazione errata di questo tipo, spesso causano lo stallo dell'intera applicazione (come con le *Swing*)
- ✔ Meglio invece fallire ancor prima, appena possibile, come *JavaFX*
- ✔ Solleva un eccezione a tempo dinamico al primo tentativo di aggiornamento dello stato della GUI da un thread diverso dall'unico autorizzato, appunto lo *JavaFX Application Thread*

Contro l'Interferenza

- Come capire e risolvere sino al dettaglio questo problema?
  - Con un corso intero sulla PC (>>>>)
- Come NON avere questo problema?
  - Seguendo le soluzioni appositamente preconfezionate dai progettisti di toolkit grafici per evitarli. Ad es.:
    - ✔ Avendo un solo scrittore, appunto il *JavaFX Application Thread* adibito all'aggiornamento, in esclusiva, della GUI
- Ed è ciò che praticamente *tutti* i toolkit grafici fanno…
  - Non solo JavaFX
  - Anche la precedente libreria per lo sviluppo GUI (Swing)

  e molte altre ancora (>>)

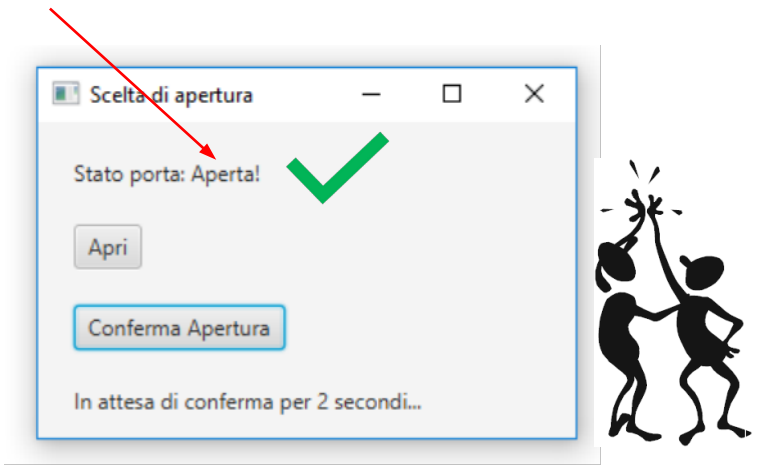
- Quindi? bisogna «affidare» le scritture all'*Application Thread* effettuando una sorta di «delega», perché possiede un'«esclusiva»

Platform.runLater (1)

```
@Override
public void run() {
     try {
          Platform.runLater(new Runnable() {
               @Override
               public void run() { conferma.setDisable(false); }
          });
          Platform.runLater(new Runnable() {
               @Override
               public void run() { timer.setText("Mancano 2 secondi"); }
          });
          Thread.sleep(1000);
          Platform.runLater(new Runnable() {
               @Override
               public void run() { timer.setText("Manca 1 secondo"); }
          });
          Thread.sleep(1000);
          Platform.runLater(new Runnable() {
               @Override
               public void run() { conferma.setDisable(true); }
          });
          Platform.runLater(new Runnable() {
               @Override
               public void run() { timer.setText("Tempo Scaduto!"); }
          });
     } catch (InterruptedException e) {
          e.printStackTrace();
     }
}
```

`Platform.runLater` (2)

- L'*Application Thread* gestisce anche una *coda di* ***`Runnable`*** da eseguire per conto altrui
  - Invocando **`Platform.runLater(`*`Runnable runnable`*`)`** le operazioni specificate nel **`Runnable`** ricevuto vengono delegate all'esecuzione dell'*Application Thread* aggiungendole in fondo a questa coda
  - L'*Application Thread* le estrae ed esegue nell'ordine di ricezione
  - Gli aggiornamenti alla GUI sono pertanto eseguiti esclusivamente dall'*Application Thread* (anche se per conto altrui) senza eccezioni



- *N.B.* Dobbiamo usare **`Platform.runLater()`** anche in **`GestoreConferma`**
- Riusciamo finalmente ad *"Aprire"* la porta
  - Cliccando sul pulsante "Apri" non si creano più eccezioni e l'*Application Thread* non rimane bloccato in attesa della conferma

Concorrenza in JavaFX (I)

- Un programma con GUI (JavaFX) è multi-thread, con:
  - *1 Main Thread*
  - *1 JavaFX Application Thread*
  - *N+ Background Threads*
- Come possono questi thread comunicare?
- In particolare, come possono i *Background Thread* restituire i risultati delle loro elaborazioni all'*Application Thread* che pure li aveva creati per lo scopo
  - l'unico *legittimato* a fare modifiche allo stato della GUI eppure costretto, per mantenersi reattivo, a delegare ai *Background Thread* per tutti i lavori più "pesanti"
  - Sinora l'unica soluzione considerata è la delega tramite, ovvero il *Background Thread* anziché accedere alla GUI direttamente indica all'*Application Thread* un **`Runnable`** da eseguire invocando **`Platform.runLater(`*`Runnable`*`)`**

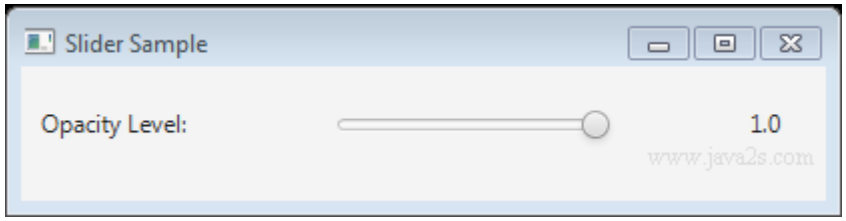
Concorrenza in JavaFX (II)

- Le librerie grafiche offrono molte altro, sempre cercando di "schermare" i programmatori da tutti i dettagli della comunicazione *inter-thread* che complica la intrinseca programmazione concorrente sottostante
- JavaFX mette a disposizione interi sotto-package (come **`javafx.concurrent`**) che facilitano la scrittura di codice concorrente adatto all'elaborazione di eventi nel contesto delle GUI
- Alcuni suggerimenti per interessanti approfondimenti
  - Supporto alla gestione di *Background Thread* di stato e progresso tracciabile e visualizzabile
    - ✓ Partire dal javadoc dell'interfaccia **`javafx.concurrent.Worker`**

      https://docs.oracle.com/javase/8/javafx/api/javafx/concurrent/Worker.html
    - ✓ Consente, ad esempio, di visualizzare la *progress bar*

Concorrenza in JavaFX (III)

- Molti *widget* (elementi grafici) di una GUI presentano proprietà (ovvero, variabili di istanza con valori) che risulta naturale poter voler interconnettere/legare ("binding") a proprietà di altri widget magari della stessa GUI
- Si pensi ad uno slider



- Si creano relazioni di dipendenza funzionale tra i valori delle properties

- *JavaFX Properties*
  - Cambiando il valore di una proprietà cambiano automaticamente tutte quelle che dipendono da questa
  - Anche se gli aggiornamenti sono fatti da thread diversi!
  - Senza preoccuparsi della gestione degli aggiornamenti concorrenti
- Gli aggiornamenti concorrenti sono gestiti internamente e resi trasparenti all'utilizzatore delle JavaFX Properties

  https://docs.oracle.com/javafx/2/binding/jfxpub-binding.htm

Conclusioni: Aspetti Tecnologici & Concetti

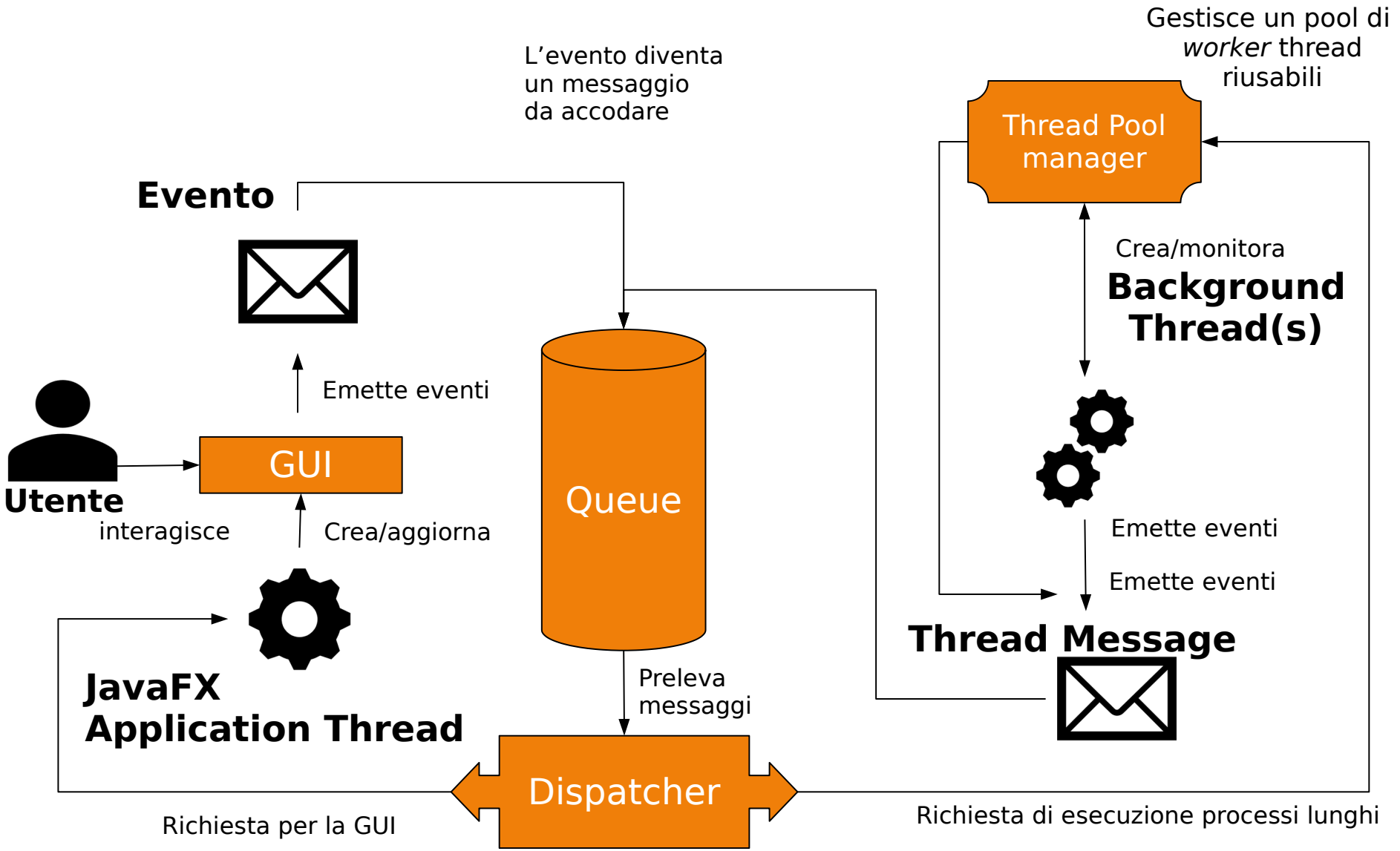
- Abbiamo visto le più importanti problematiche sottostanti l'utilizzo di librerie per la creazione di applicazioni con GUI
- *Non* approfondiremo ulteriormente gli aspetti tecnologici
  - ✔ Lasciati, nella sostanza, alla curiosità dello studente

- Così come *non* approfondiremo la ricchissima collezione di elementi grafici che *JavaFX* offre, sino a permettere la gestione efficace di video, modellazione 3D ecc. ecc.
  - ✔ Lasciati alle motivazioni dello studente

  http://tutorials.jenkov.com/javafx/index.html

- La tecnologia è «cangiante», al contrario gli aspetti logici e concettuali sottostanti finiscono per trovare sempre nuove «incarnazioni» così come risulta evidente nel contesto delle librerie a supporto della programmazione di GUI

Uno Schema Concettuale delle Librerie a Supporto dello Sviluppo di GUI



Stessi Concetti, Altri Nomi

Oggetto	Descrizione	Piattaforma			
		WinForm	JavaFX	Android	IoS
Message	User interaction, eventi S.O. e threads	User Actions / Threads	User Actions / Process Event	User Actions / Threads	User Actions / Threads
Queue	Coda di eventi	Message Queue	Queue	Message Queue	Main Queue, Highest Priority Queue, Lowest Priority Queue
Dispatcher	Preleva messaggi e li invia al gestore corretto	Message Pump	Process Event	Looper	Serial Dispatch Queue / Concurrent Dispatch Queue
GUI	Interfaccia utente	Main Form	The Scene Graph	Main Activity	View Scene
Thread Manager	Gestisce il thread pool, monitora e stoppa i thread in caso di anomalie. Emette eventi	Task Parallel Library, Async/Await pattern	Long-running Event Handler	ThreadPools, Executor	QueueOperation, NSOperation
Background Thread	Emette eventi per i long-running thread (Progresso, fine attività etc.)	BackgroudWorker	Progress Properties	AsyncTask	-

Importare JavaFX in un Progetto Eclipse

- Se sulla vostra macchina è già installato un JDK 8- allora JavaFX è già presente come libreria “esterna”
  - Basta verificare di avere il file **`javafxrt.jar`** in **`$JAVA_HOME/lib/ext/`**
  - Se il file è presente allora è necessario solo aggiungere la libreria
    - In Eclipse:
      1. Crea un nuovo progetto
      2. Tasto destro sul progetto poi su *Build Path>Add Libraries*
      3. Selezionare *User Library* e premere *Next*
      4. Selezionare *User Libraries* e premere *New*
      5. Digitare “JavaFX” e premere Invio
      6. Selezionando la libreria appena creata premere su *Add External Libraries*
      7. Selezionare il path **`$JAVA_HOME/lib/ext/javafxrt.jar`**
      8. Apply and close

- **NOTA:** Andando su *Build Path>Configure Build Path>Order And Export* assicurarsi che la libreria appena creata sia in cima e non sotto