

# La gestione delle eccezioni

Marco Patrignani

[mailto: marco.patrignani@unitn.it](mailto:marco.patrignani@unitn.it)

(basato sulle slides di Picco, Ronchetti, Marchese)

# Il problema

- Spesso, si tende a scrivere programmi assumendo che tutto vada a buon fine
- In pratica, vi sono spesso situazioni *impreviste* che devono essere gestite dal programma
  - Esempi: disco pieno, connessione di rete non disponibile
- La mancata gestione di tali situazioni solitamente determina la terminazione improvvisa del programma

# La gestione degli errori

- Tradizionalmente, gli errori vengono gestiti:
  - Terminando il programma nel metodo in cui si verifica l'errore
    - Spesso, una scelta drastica ...
    - ... che peraltro spetta al chiamante, e non al chiamato
  - Usando valori di ritorno convenzionali per segnalare errori al chiamante
    - Causano confusione se gli errori sono dello stesso tipo del valore di ritorno
    - In ogni caso impossibile per i costruttori
    - Non forniscono informazione riguardo alla natura dell'errore
  - Usando una funzione di gestione degli errori
    - Centralizza la gestione degli errori, che invece spetterebbe al chiamante
    - Diminuisce la leggibilità del programma

# Le eccezioni

- Un'**eccezione** è un evento imprevisto, anomalo e indesiderato che si verifica durante l'esecuzione di un programma
- I linguaggi di programmazione moderni dedicano costrutti appositi alla gestione delle eccezioni
  - permettono di superare i limiti della gestione tradizionale degli errori
- La gestione delle eccezioni presenta i seguenti vantaggi:
  - separare la gestione degli errori dal codice applicativo
  - consentire una propagazione controllata degli errori
  - raggruppare o differenziare gli errori

# Separare la gestione degli errori dal codice applicativo

```
leggiFile {
    apri file;
    determina dimensione file;
    alloca memoria;
    trasferisci file in memoria;
    chiudi file;
}

leggiFile {
    try {
        apri file;
        determina dimensione file;
        alloca memoria;
        trasferisci file in memoria;
        chiudi file;
    } catch (aperturaFileFallita) {
        ...
    } catch (determinaDimFallito) {
        ...
    } catch (allocazioneMemFallita) {
        ...
    } catch (letturaFallita) {
        ...
    } catch (chiusuraFileFallita) {
        ...
    }
}
```

Codice senza gestione errori

```
tipoCodiceErr leggiFile {
    inizializza codiceErr = 0;
    apri file;
    if (fileAperto) {
        determina dimensione file;
        if (ottenutaLunghezza) {
            alloca memoria;
            if (ottenutaMemoria) {
                trasferisci file in memoria;
                if (trasferimentoFallito) {
                    codiceErr = -1;
                } else {
                    codiceErr = -2;
                } else {
                    codiceErr = -3;
                }
                chiudi file;
                if (fileNonChiuso && codiceErr == 0) {
                    codiceErr = -4;
                } else {
                    codiceErr = codiceErr and -4;
                }
            } else {
                codiceErr = -6;
            }
        }
    }
    return codiceErr;
}
```

Codice con gestione errori tradizionale

Codice con gestione eccezioni

# Propagare gli errori

```
metodo1 {  
    chiama metodo2;  
}  
metodo2 {  
    chiama metodo3;  
}  
metodo3 {  
    chiama leggiFile;  
}
```

Codice  
senza  
gestione  
errori

```
metodo1 {  
    try {  
        chiama metodo2;  
    } catch(exception) {  
        gestisciErrore;  
    }  
    metodo2 throws exception {  
        chiama metodo3;  
    }  
    metodo3 throws exception {  
        chiama leggiFile;  
    }  
}
```

Codice con  
gestione  
eccezioni

```
metodo1 {  
    tipoCodiceErr errore;  
    errore = chiama metodo2;  
    if (errore) gestisciErrore;  
    else procedi;  
}  
tipoCodiceErr metodo2 {  
    tipoCodiceErr errore;  
    errore = chiama metodo3;  
    if (errore) gestisciErrore;  
    else procedi;  
}  
tipoCodiceErr metodo3 {  
    tipoCodiceErr errore;  
    errore = chiama leggiFile;  
    if (errore) return errore;  
    else procedi;  
}
```

Codice con  
gestione  
errori  
tradizionale

# Le eccezioni in Java

- Un’eccezione è rappresentata da un oggetto
  - Contiene informazioni circa la causa dell’eccezione
- Un’eccezione viene «sollevata» esplicitamente dal programmatore mediante il costrutto **throw**, seguito dall’oggetto che rappresenta l’eccezione
  - Le eccezioni possono anche essere generate automaticamente dal runtime system di Java
- Se un metodo solleva eccezioni, queste devono obbligatoriamente essere dichiarate
  - Ciò viene fatto nell’interfaccia del metodo, mediante il costrutto **throws**
- Esempio:

```
public Object pop() throws EmptyStackException {  
    if (!isEmpty()) return stack[--ptr];  
    else throw new EmptyStackException();  
}
```

In questo caso, l’eccezione viene sollevata dal programmatore in seguito al mancato rispetto del “contratto” del metodo

# Come delimitare e gestire l'eccezione

- Il segmento di codice che può sollevare l'eccezione viene delimitato da un blocco **try**
- Esso è sempre seguito da uno o più blocchi **catch**, che contengono il codice di gestione dell'eccezione
  - I blocchi **catch** vengono chiamati *exception handlers* (gestori di eccezione)
  - *Handlers* diversi sono necessari se il codice in questione può sollevare eccezioni diverse
- Esempio:

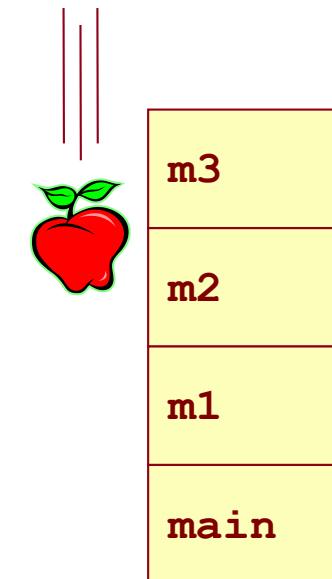
```
...
Stack s = new Stack();
try {
    Object o = s.pop();
} catch(EmptyStackException e) {
    System.out.println("Stack vuoto!");
    System.exit(1);
}
```

# La propagazione delle eccezioni

- L'esecuzione di un'istruzione **throw** solleva un'eccezione, e dà inizio al seguente processo:
  - viene terminata l'esecuzione del blocco di codice che contiene l'istruzione **throw**
  - l'eccezione argomento della **throw** viene propagata lungo la catena dinamica delle chiamate, verificando di volta in volta se esiste nel chiamante un blocco **try/catch** con un gestore appropriato per l'eccezione:
    - se sì, l'esecuzione riprende dal codice contenuto nella **catch** così determinata (si dice che il gestore ha “catturato” l'eccezione);
    - altrimenti, la propagazione continua al chiamante del chiamante
  - (*Solo per le eccezioni run-time o errori, vedi dopo*)  
Se non viene trovato nessun blocco **try/catch** compatibile, l'esecuzione termina

# Un esempio

```
public class Test {  
    public static void main(String[] args) {  
        try {  
            m1();  
        } catch (TestException e) { ... }  
    }  
    void m1() throws TestException { m2(); }  
    void m2() throws TestException { m3(); }  
    void m3() throws TestException {  
        ...  
        throw new TestException();  
    }  
}
```



# Gestire l'errore o propagarlo?

- Java richiede che ogni metodo debba:
  - gestire le eccezioni che possono essere sollevate all'interno del proprio corpo, definendo un blocco **try** seguito da uno o più blocchi **catch**;  
**oppure**
  - propagare tali eccezioni, dichiarando ciò mediante una clausola **throws** nell'interfaccia del metodo
- Dal secondo requisito, è evidente come la dichiarazione delle eccezioni sia parte integrante dell'interfaccia di ogni metodo
  - Ciò è necessario affinché gli utilizzatori del metodo siano consapevoli delle eccezioni che possono essere sollevate al suo interno

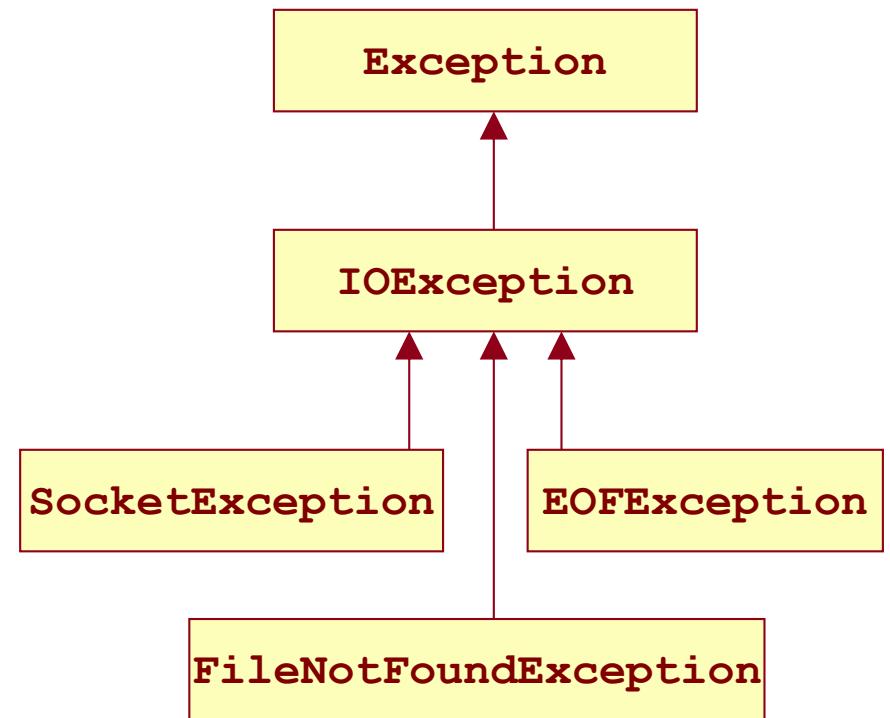
# La clausola **finally**

- Opzionalmente, alla fine di un blocco **try/catch** può apparire una clausola **finally**
- Essa viene eseguita in ogni caso e cioè:
  - quando si verifica un'eccezione e viene eseguito uno dei blocchi **catch**, ...
  - ... oppure no (e quindi viene eseguito tutto il codice nel blocco **try**)
- Spesso usata per operazioni di “pulizia” finale durante I/O
- Evitano duplicazione di codice, che viceversa dovrebbe essere ripetuto nel blocco **try** e in ogni blocco **catch**
- Esempio:

```
try {  
    FileInputStream f = new FileInputStream(filename);  
    ... usa f ...  
} catch(IOException ioe) {  
    ... gestisci l'errore di I/O ...  
} finally {  
    f.close();  
}
```

# Eccezioni ed ereditarietà

- Le eccezioni possono essere raggruppate in categorie
  - es., eccezioni relative all'I/O, al display, ecc.
- Poiché le eccezioni sono rappresentate come oggetti, l'ereditarietà fornisce un meccanismo naturale per rappresentare tale raggruppamento
- Inoltre, l'ereditarietà consente di creare di eccezioni definite dal programmatore, come sottoclassi di **Exception**



# Specificare più gestori

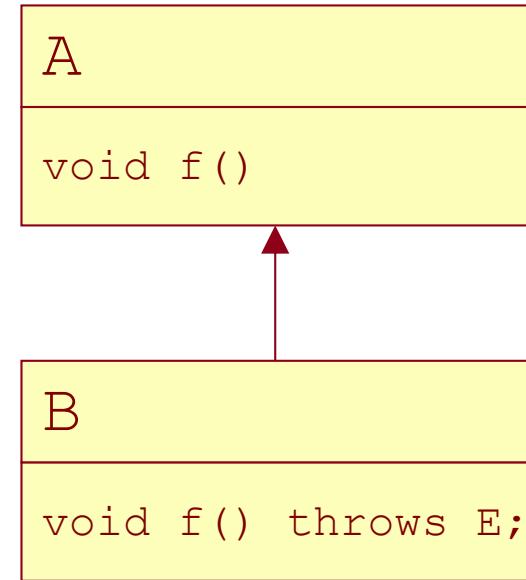
- Un blocco **try/catch** può contenere più di un gestore di eccezioni (cioè più di un blocco **catch**)
- L'ereditarietà applicata alle eccezioni fa sì che sia possibile che vi siano più gestori compatibili con l'eccezione sollevata
- Esempio:

```
try {  
    FileOutputStream f = new FileOutputStream("f.dat");  
} catch (FileNotFoundException fnfe) { ... }  
} catch (IOException ioe) { ... }  
} catch (SecurityException se) { ... }  
} catch (Exception e) { ... }
```

- Viene eseguito il primo blocco di **catch** compatibile con l'eccezione sollevata
  - Le sottoclassi vanno dichiarate prima delle rispettive superclassi (altrimenti non verrebbero mai selezionate)

# Eccezioni e polimorfismo

- Un metodo  $m$  definito in una sottoclasse  $D$  che ridefinisce un metodo  $m$  della superclasse  $C$  può sollevare soltanto le eccezioni  $E_1, E_2, \dots, E_n$  dichiarate nella firma di  $m$  in  $C$ , oppure sottoclassi di  $E_1, E_2, \dots, E_n$
- In altre parole: la specifica delle eccezioni può solo “restringersi” procedendo verso le sottoclassi, ma non “allargarsi”
  - al contrario delle normali regole di ridefinizione
- Vincolo necessario per preservare la semantica del polimorfismo
  - Non vale per i costruttori

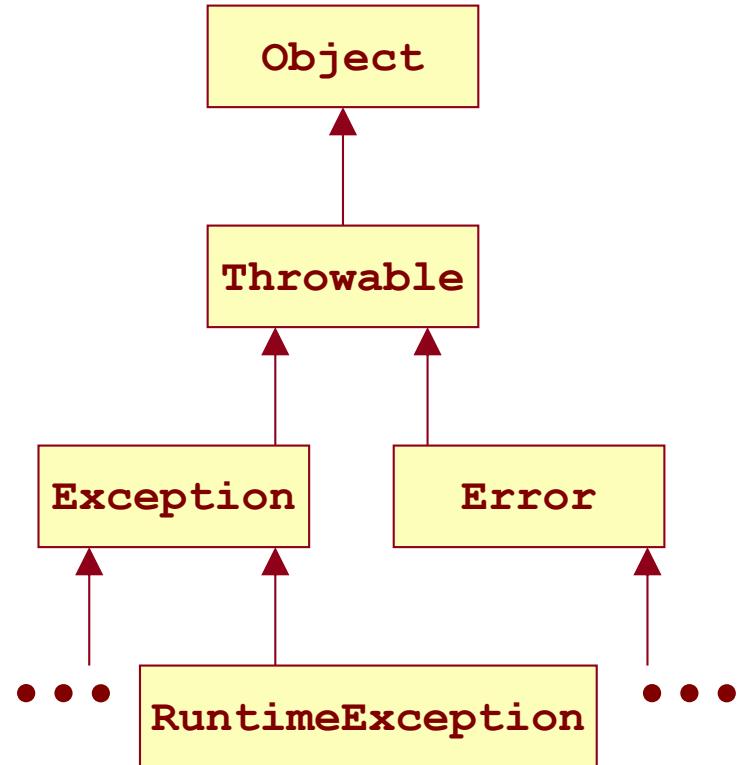


```
A a = new B();
a.f();
```

Può sollevare un'eccezione anche se non dichiarata in A!!!

# Throwable e sottoclassi

- Gli oggetti che possono essere “sollevati” da una **throw** devono avere come tipo una sottoclasse di **Throwable**
- Tra tali sottoclassi, in Java si distingue fra **Error** ed **Exception**:
  - Le sottoclassi di **Error** rappresentano errori “gravi”, che non possono essere gestiti dai programmi; è inusuale che un programma sollevi un **Error**
    - Es. **VirtualMachineError**, **LinkageError**
  - Le sottoclassi di **Exception** rappresentano le comuni eccezioni, di cui abbiamo trattato finora
    - Es. **IOException**



# RuntimException

- Fra le sottoclassi di **Exception**, le sottoclassi di **RuntimException** definiscono una famiglia di eccezioni cosiddette runtime, con regole proprie
- Tali classi rappresentano eccezioni che avvengono nell'interprete Java (*virtual machine*) ma che non sono così gravi da essere considerati **Error**
  - Un esempio classico è **NullPointerException**, che si verifica quando un metodo cerca di accedere a un membro di un oggetto attraverso un riferimento il cui valore è **null** (es. **obj = null; obj.m();**)
- Tali eccezioni si possono verificare praticamente ovunque: forzare il requisito di gestire oppure dichiarare tali eccezioni porterebbe alla stesura di codice illeggibile
- Per questo motivo, non è necessario gestire o dichiarare questo tipo di eccezioni (anche se è possibile farlo, quando necessario e/o ragionevole)
  - Non è consigliabile definire tutte le eccezioni come **RuntimException!**

# Fornire informazioni circa l'eccezione

- La classe che descrive un'eccezione può possedere attributi e metodi che vengono usati per fornire informazioni aggiuntive al chiamante
- Esempio:

```
public class DataIllegale extends Exception {  
    int giorno, mese, anno;  
    DataIllegale(int g, int m, int a) {  
        giorno = g; mese = m; anno = a;  
    }  
}  
public class Data {  
    private int giorno, mese, anno;  
    private boolean corretta(int g,int m,int a) { ... }  
    public Data(int g, int m, int a) throws DataIllegale {  
        if(!corretta(g,m,a)) throw new DataIllegale(g,m,a);  
        giorno = g; mese = m; anno = a;  
    }  
}
```

# Le eccezioni e il debugging

- La gestione delle eccezioni è utile in fase di debugging, in quanto dà informazioni sul verificarsi di un'anomalia
  - In seguito al verificarsi di un'eccezione viene stampata una *stack trace*, che contiene l'indicazione delle procedure attive nella catena dinamica, e dei numeri di linea in cui l'eccezione è stata propagata
  - Esempio:

```
Exception in thread "main" java.lang.NullPointerException
        at TestException.m3 (TestException.java:8)
        at TestException.m2 (TestException.java:7)
        at TestException.m1 (TestException.java:6)
        at TestException.main (TestException.java:3)
```
- Per questo motivo è consigliabile non inserire *mai* un'istruzione di questo tipo:

```
try {  
    ...  
} catch (Exception e) {}
```

NO!!!

Come minimo inserire  
**e.printStackTrace()**, che stampa sullo stream di errore l'intera catena delle chiamate con i numeri di linea dove si è verificata l'eccezione

in quanto essa di fatto “spegne” il meccanismo delle eccezioni e non dà modo al programmatore di accorgersi di un malfunzionamento

# Alcuni suggerimenti pratici

- Quando si definiscono le classi di un programma, è spesso utile lasciare la gestione delle eccezioni a chi *usa* tali classi, anziché gestirle internamente
  - Infatti, è spesso chi usa la classe che deve decidere come reagire a un evento anomalo, nel contesto specifico dell'applicazione
- Quando possibile, conviene sempre riusare un'eccezione già fornita dal linguaggio, se questa esiste
  - Per evitare la proliferazione di eccezioni, e per aumentare la comprensibilità e leggibilità del programma

# Gestire l'input utente

**Gian Pietro Picco**

Dipartimento di Ingegneria e Scienza dell'Informazione  
(DISI) University of Trento, Italy

[gianpietro.picco@unitn.it](mailto:gianpietro.picco@unitn.it)  
<http://disi.unitn.it/~picco>

# ... dalla console

```
import java.util.Scanner;  
public class UserInputConsole {  
    public static void main(String arg[]) {  
        Scanner scanner = new Scanner(System.in);  
        System.out.println("Nome: ");  
        String nome = scanner.next();  
        int eta = 0;  
        boolean ok = false;  
        System.out.println("Eta': ");  
        while (!ok) {  
            if (scanner.hasNextInt()) {  
                eta = scanner.nextInt();  
                ok = true;  
            } else {  
                scanner.next();  
                System.out.println("L'età deve essere un valore intero!");  
            }  
        }  
        scanner.close();  
        System.out.println("Mi chiamo " + nome + " e ho " + eta + " anni.");  
    }  
}
```

Nome:  
*Giovanni*

Eta':  
*100*

Mi chiamo Giovanni e ho 100 anni.

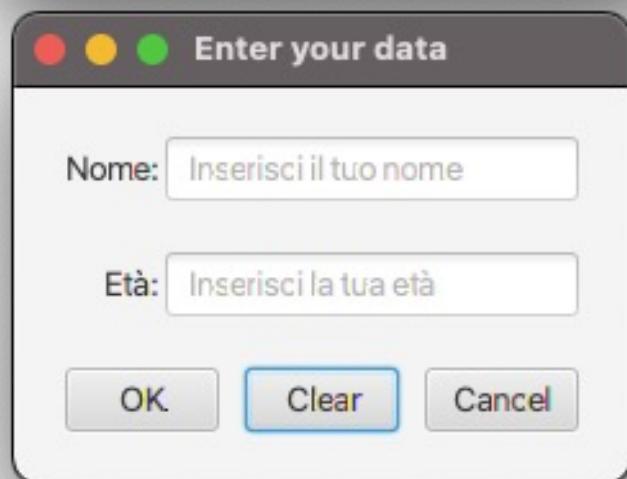
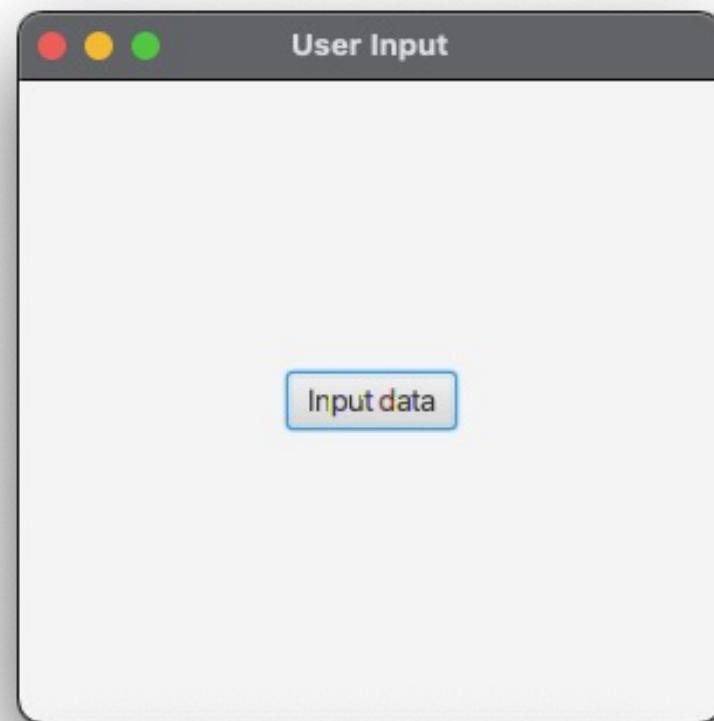
legge un «token» ...  
... dallo standard input

verifica che il token sia del tipo giusto...

... prima di leggerlo ...  
... altrimenti lo consuma e ripete la lettura

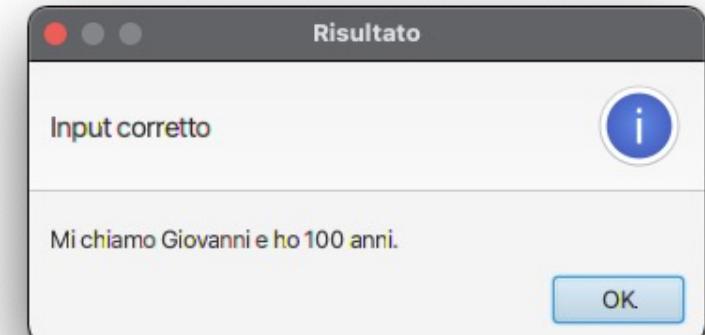
è buona pratica «chiudere»  
lo scanner quando non è più necessario

# ... da JavaFX



... consentendo  
l'inserimento  
dei dati ...

«Hello World» modificato:  
il bottone ora apre una finestra ...



... e relative  
verifiche di  
correttezza



# Costruire la finestra

```
// nel metodo start ...
```

```
Label lNome = new Label("Nome: ");
```

```
Label lEta = new Label("Età: ");
```

```
TextField tNome = new TextField();
```

```
TextField tEta = new TextField();
```

```
Button ok = new Button("OK");
```

```
Button clear = new Button("Clear");
```

```
Button cancel = new Button("Cancel");
```

```
ok.setMinWidth(BWIDTH);
```

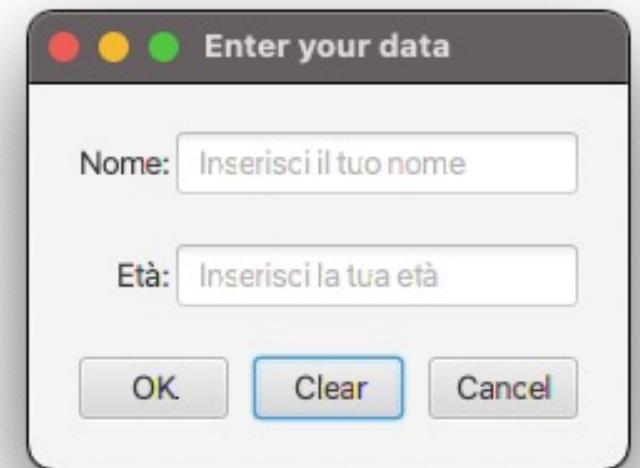
```
clear.setMinWidth(BWIDTH);
```

```
cancel.setMinWidth(BWIDTH);
```

```
tNome.setPromptText("Inserisci il tuo nome");
```

```
tEta.setPromptText("Inserisci la tua età");
```

```
...
```



**Label** non è modificabile  
(al contrario di **Text**)

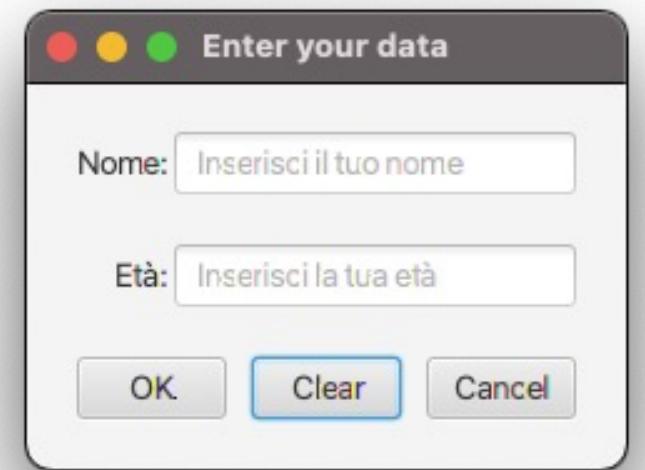
**TextField** raccoglie  
l'input dell'utente

è possibile specificare testo  
«prompt» visualizzato nel  
textfield quando è vuoto

# Layout della finestra

...

```
HBox hNome = new HBox(lNome,tNome);  
HBox hEta = new HBox(lEta,tEta);  
hNome.setAlignment(Pos.CENTER_RIGHT);  
hEta.setAlignment(Pos.CENTER_RIGHT);
```



```
TilePane buttons = new TilePane(ok,clear,cancel);  
buttons.setHgap(10);  
buttons.setPrefColumns(3);
```

```
VBox root2 = new VBox(hNome,hEta,buttons);  
root2.setSpacing(SPACING);  
root2.setPadding(new Insets(SPACING));
```

```
Scene scene2 = new Scene(root2);  
Stage stage2 = new Stage();  
stage2.setTitle("Enter your data");  
stage2.setScene(scene2);
```

...

Niente di nuovo...

# Gestire l'input

quando il bottone OK viene premuto

```
...
ok.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent actionEvent) {
        String nome = tNome.getText();
        String eta = tEta.getText();
        Alert alert;
        if (nome.isBlank() || eta.isBlank() || !isInteger(eta)) {
            alert = new Alert(Alert.AlertType.ERROR);
            alert.setTitle("Errore!");
            alert.setHeaderText("Input non corretto");
            if (nome.isBlank() || eta.isBlank())
                alert.setContentText("Non hai inserito tutti i dati.");
            else
                alert.setContentText("L'età non è un valore intero.");
        } else {
            alert = new Alert(Alert.AlertType.INFORMATION);
            alert.setTitle("Risultato");
            alert.setHeaderText("Input corretto");
            alert.setContentText("Mi chiamo " + nome +
                " e ho " + eta + " anni.");
        }
        alert.showAndWait();
    } });
...
... legge il contenuto dei text field
(vedi ultima slide)
... e se l'input non è corretto usa un oggetto
Alert, opportunamente configurato,
per mostrare un messaggio all'utente
```

# Gestire gli altri bottoni

...

```
clear.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent actionEvent) {  
        tNome.clear();  
        tEta.clear();  
    }});
```

«pulisce» i text field,  
rimuovendone il contenuto

```
cancel.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent actionEvent) {  
        stage2.close();  
    }});
```

chiude la finestra di input

```
Button button = new Button("Input data");  
button.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent actionEvent) {  
        stage2.show();  
    }});
```

nella finestra primaria,  
ora il bottone apre la finestra di input

```
StackPane root = new StackPane(button);  
primarystage.setTitle("User Input");  
primarystage.setScene(new Scene(root, 300, 275));  
primarystage.show();  
...
```

identico al template «Hello World»

# Stringa o intero?

- In Java non esiste un modo diretto per verificare se una stringa contiene un numero
  - dobbiamo costruirci un metodo apposito

```
private boolean isInteger(String s) {  
    for (char c : s.toCharArray())  
        if (!Character.isDigit(c))  
            return false;  
    return true;  
}
```

analizzando  
la stringa ...

```
private boolean isInteger(String s) {  
    try {  
        Integer.parseInt(s);  
        return true;  
    } catch (NumberFormatException nfe) {  
        return false;  
    }  
}
```

... oppure  
catturando  
un'eccezione  
runtime