

---

## **Le eccezioni**

**Leggere cap. 13 di Programmazione di base e avanzata  
con Java**

# Che cosa sono le eccezioni

---

- In un programma si possono generare situazioni critiche che provocano errori
- Non è però accettabile, soprattutto in applicazioni complesse, che un qualunque errore possa bloccare e far terminare in modo anomalo un programma
- Le situazioni di errore devono essere gestite
- **Le eccezioni sono lo strumento messo a disposizione da Java per gestire in modo ordinato le situazioni anomale**

## Esempio

---

- Scriviamo un semplice programma che converte in numero una stringa passata sulla riga dei comandi:

```
public class EsempioEccezione
{
    public static void main(String args[])
    {
        int a = 0;
        String s = args[0];
        a = Integer.parseInt(s);
    }
}
```

- Se la stringa passata non contiene un numero ci troviamo di fronte ad una situazione critica.
- **Il programma termina con un errore!**

## Prevenire è meglio che curare...

---

- La soluzione “classica” consiste nel cercare di prevenire la situazione di errore:

```
public class EsempioEccezione
{
    public static boolean isNumeric(String s)
    {
        boolean ok = true;
        for(int i=0; i<s.length(); i++)
            ok = ok && (Character.isDigit(s.charAt(i)));
        return ok;
    }
    public static void main(String args[])
    {
        int a = 0;
        String s = args[0];
        if (isNumeric(s))
            a = Integer.parseInt(s);
    }
}
```

## **...o forse no**

---

- In situazioni semplici un approccio di questo tipo può funzionare
- Ma in generale non è una soluzione efficace
- Infatti:
  - In situazioni complesse i possibili errori sono molti e non si riesce ad individuarli e prevenirli tutti
  - I test sono spesso complessi da realizzare
  - I test devono essere eseguiti anche quando tutto va bene e questo può creare problemi di prestazioni
- Sarebbe quindi preferibile poter gestire gli errori solo quando si verificano

## Gestire un'eccezione

---

- La soluzione corretta in Java è quella di inserire le istruzioni “a rischio” in un blocco controllato:

```
public class EsempioEccezione
{
    public static void main(String args[])
    {
        int a = 0;
        String s = args[0];
        try
        {
            a = Integer.parseInt(s);
        }
        catch (Exception ex)
        {
            a = 0;
        }
        System.out.println(a);
    }
}
```

## Blocchi try/catch

---


- Un blocco controllato è costituito da una clausola **try** e da una o più clausole **catch**:

```
try
{
    /* operazione critica */
}
catch (Exception ex)
{
    /* gestione dell'eccezione */
}
```


- Nel blocco **try** inseriamo le istruzioni che possono generare situazioni di errore
- Se tutto va bene il blocco try viene eseguito e si passa all'istruzione successiva al blocco catch
- Se si verifica un'eccezione l'esecuzione del blocco try termina e si passa al blocco **catch** dove si può intervenire per gestire correttamente l'anomalia

# Flusso delle eccezioni

- Riprendiamo il nostro esempio e vediamo cosa accade nei due casi:



```
s = "123";  
try  
{  
    a =  
        Integer.parseInt(s);  
}  
catch (Exception ex)  
{  
    a = 0;  
}  
a = a + 2;  
  
// Il valore di a è 125
```



```
s = "xyz";  
try  
{  
    a =  
        Integer.parseInt(s);  
}  
catch (Exception ex)  
{  
    a = 0;  
}  
a = a + 2;  
  
// Il valore di a è 2
```



## Ancora sul flusso

---

- 💣 **Attenzione:** se in un blocco try abbiamo più istruzioni quando si verifica un'eccezione le istruzioni successive non vengono eseguite:

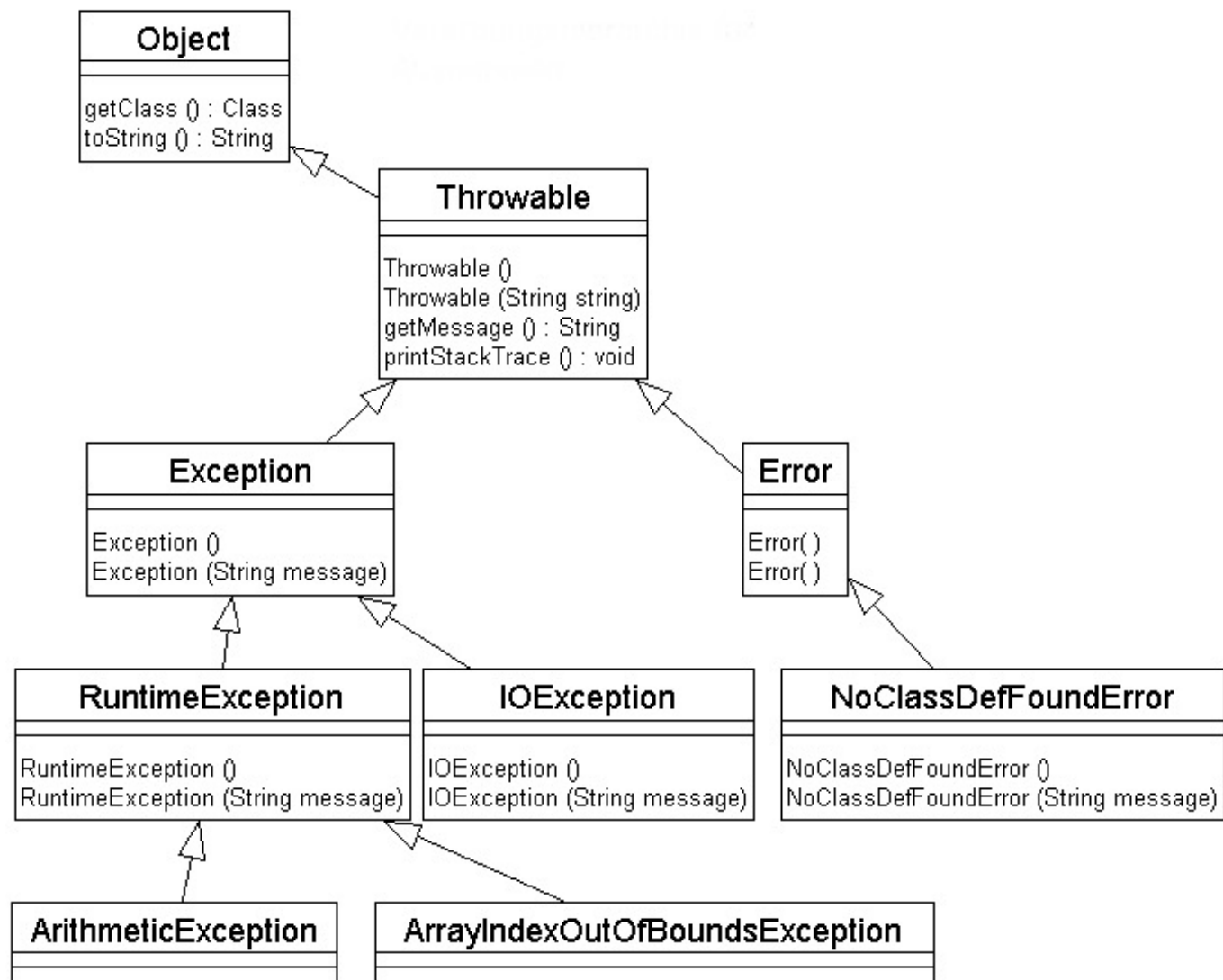
```
s = "xyz";  
try  
{  
    a = Integer.parseInt(s);  
    a = a + 5; // Non viene eseguita!  
}  
catch (Exception ex)  
{  
    a = 0;  
}  
// dopo il catch il flusso riprende qui sotto  
a = a + 2 // a vale 2 e non 7!
```

# Che cos'è un'eccezione?

---

- Una eccezione è un oggetto
- E' un'istanza di **java.lang.Throwable** o di una sua sottoclasse.
- Le due sottoclassi più comuni sono **java.lang.Exception** e **java.lang.Error**
- La parola “eccezione” è spesso riferita a entrambe ma c'è una differenza:
  - Un **Error** indica un grave problema di sistema, normalmente irrecoverabile: quindi non deve essere gestito
  - Una **Exception** indica invece una situazione recuperabile: dovrebbe essere gestita
- Nei casi di nostro interesse abbiamo quindi a che fare con istanze di sottoclassi di **java.lang.Exception**

# Gerarchia delle eccezioni



## Eccezioni come oggetti - 1

---

- Riprendiamo il nostro esempio: quando si verifica un'eccezione nel metodo `parseInt` viene creata un'istanza di una sottoclasse di `Exception` (in questo caso **`NumberFormatException`**)
- Questa istanza viene passata al blocco **`catch`**  
**`catch (Exception ex)`**  

```
{  
    a = 0;  
}
```
- La variabile **`ex`** (il nome non è fisso) è quindi un riferimento all'istanza di `NumberFormatException`
- Essendo di tipo `Exception`, in virtù del **`subtyping`**, la variabile **`ex`** può puntare ad istanze di una qualunque sottoclasse di `Exception`

## Eccezioni e metodi

---

- Dal momento che un'eccezione è un oggetto possiamo invocare su di essa i metodi definiti dalla classe a cui appartiene.
- In particolare tutte le eccezioni implementano il metodo **getMessage()** (è definito nella classe base `Throwable`)
- **getMessage()** fornisce una descrizione dell'eccezione
- Potremmo quindi scrivere:

```
catch (Exception ex)
{
    System.out.println(ex.getMessage());
}
```
- Le sottoclassi possono poi definire metodi specifici che forniscono ulteriori informazioni.

## Una gestione più accurata

---

- La gerarchia delle eccezioni e la possibilità di avere più blocchi catch consente di differenziare la gestione delle eccezioni.

```
try
{
    a = Integer.parseInt(s) ;
}
catch (NumberFormatException e)
{
    a = 0 ;
}
catch (Exception e)
{
    System.out.println(e.getMessage()) ; System.exit(1) ;
}
```

- In questo modo gestiamo in maniera completa l'eccezione specifica che ci interessa, e in maniera generica le altre

## Catch multipli

---

- I blocchi catch vengono gestiti in cascata: se un'eccezione non è del tipo specificato si passa a quello successivo
- Mettendo in fondo un blocco che ha Exception come tipo si catturano tutte le eccezioni (però non gli errori di sistema)
- Quindi:
  - Se si verifica un'eccezione di tipo `NumberFormatException` viene eseguito il primo blocco che recupera la situazione attribuendo un valore di default ad a
  - In tutti gli altri casi di eccezione viene eseguito il secondo blocco catch: si mostra a video un messaggio e si esce dal programma

## Rilanciare le eccezioni

---

- Java prevede un meccanismo per garantire una gestione corretta delle eccezioni.
- Un metodo in cui si può verificare un'eccezione è obbligato a fare una delle seguenti due cose:
  - **Gestire l'eccezione**, con un costrutto try/catch
  - **Rilanciarla** esplicitamente all'esterno del metodo, delegandone in pratica la gestione ad altri
- Se si sceglie questa seconda strada, il metodo deve indicare quale eccezione può “uscire” da esso, con la clausola **throws**. **Se non lo fa il compilatore dà un errore**
- **Catch or Specify Requirement**
- Si crea quindi una **catena di responsabilità** nella gestione delle situazioni critiche: ad ogni livello possiamo quindi decidere se l'azione correttiva può essere eseguita o se dobbiamo rimandarla più in alto



## Catch or Specify Requirement

---

- Il codice che non rispetta il Catch or Specify Requirement non compila
- Non tutte le eccezioni sono soggette al Catch or Specify Requirement.
- Tre tipi di eccezioni
- **Checked exceptions**: condizioni eccezionali che un'applicazione ben scritta dovrebbe prevedere e recuperare
- Le Checked exceptions sono soggette al Catch or Specify Requirement.
- Tutte le eccezioni sono checked exceptions, eccetto che quelle indicate da Error, RuntimeException, e le loro sottoclassi.

## Catch or Specify Requirement

---

- Secondo tipo di eccezioni: **errors**. Queste sono condizioni eccezionali che sono esterne all'applicazione e che l'applicazione non può prevedere e recuperare.
- Gli errors non sono soggetti al Catch or Specify Requirement.
- Gli errors sono quelli indicate da Error e dalle sue sottoclassi.
- Terzo tipo di eccezioni: **runtime exceptions**. Queste sono condizioni eccezionali che sono interne all'applicazione e che l'applicazione non può prevedere o recuperare.
- Di solito indicano bug di programmazione, come errori nella logica o uso improprio di una API

## Catch or Specify Requirement

---

- Le Runtime exceptions **non sono soggette** al Catch or Specify Requirement.
- Le Runtime exceptions sono quelle indicate da RuntimeException e dalle sue sottoclassi
- Gli Errors e le runtime exceptions sono collettivamente chiamate **unchecked exceptions**.

# Gestione o rilancio

## Gestione

```
public class EsempioEcc2
{
    public static void
        main(String args[])
    {
        int a = 0;
        String s = args[0];
        try
        {
            a = Integer.parseInt(s);
        }
        catch (Exception e)
        { a = 0; }
    }
}
```

## Rilancio

```
public class EsempioEcc1
{
    public static void
        main(String args[])
        throws
            NumberFormatException
    {
        int a = 0;
        String s = args[0];
        a = Integer.parseInt(s);
    }
}
```

## Lanciare eccezioni - 1

---

- Anche nei metodi scritti da noi possiamo generare eccezioni per segnalare situazioni anomale
- Definiamo per esempio una classe che consente di convertire stringhe in numeri solo per numeri <1000

```
public class Thousand
{
    public static int parseInt(String s)
        throws NumberFormatException
    {
        int a = Integer.parseInt(s);
        if (a >= 1000)
        {
            NumberFormatException e = new NumberFormatException();
            throw e;
        }
    }
}
```

## Lanciare eccezioni - 2

---

- Quindi:
    - Prima **si crea l'oggetto eccezione** da lanciare, come istanza di una sottoclasse di Exception
    - Poi lo si lancia con l'istruzione **throw**
  - Il metodo deve inoltre dichiarare che può mandare all'esterno un'eccezione `NumberFormatException`, che può essere generata da `Integer.parseInt()` oppure dal metodo stesso
- 💣 **Attenzione:** non bisogna confondere la clausola **throws** con l'istruzione **throw**:
- **throw** genera (si dice anche **solleva**) un'eccezione
  - **throws** dichiara che un metodo rilancia all'esterno un'eccezione

## Definizione di eccezioni

---

- Nell'esempio precedente abbiamo utilizzato un tipo di eccezione predefinito (`NumberFormatException`)
- Possiamo però definire un'eccezione specifica per il nostro scopo.
- Per far questo è sufficiente definire una sottoclasse di **Exception**:

```
public class NumberTooBigException extends Exception
{
    public NumberTooBigException() { super(); }
    public NumberTooBigException(String s) { super(s); }
}
```

- Dobbiamo definire i due costruttori standard:
  - Quello di default
  - Quello con un parametro stringa (il messaggio)

## Esempio con eccezione definita

---

- Il nostro esempio diventa quindi:

```
public class Thousand
{
    public static int parseInt(String s)
        throws NumberFormatException,
               NumberTooBigException
    {
        int a = Integer.parseInt(s);
        if (a >= 1000)
        {
            NumberTooBigException
                e = new NumberTooBigException();
            throw e;
        }
    }
}
```

- Dobbiamo dichiarare che il metodo può emettere due tipi di eccezioni: quella di `Integer.parseInt()` e la nostra



## La clausola finally

---

- L'istruzione try prevede una clausola finally opzionale:

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

- Il blocco finally deve essere messo sempre alla fine
- Le istruzioni del blocco finally vengono eseguite comunque:
  - In assenza di eccezioni il blocco finally viene eseguito subito dopo il blocco try
  - Se si verificano eccezioni viene eseguito prima l'eventuale blocco catch e poi il blocco finally
- E' possibile utilizzare finally senza che siano presenti blocchi catch.

## Casi per finally

---

1. Il blocco try esegue fino alla fine e nessuna eccezione è lanciata. Il blocco finally viene eseguito dopo il blocco try
2. Una eccezione viene lanciata nel blocco try e raccolta in uno dei blocchi catch. Il blocco finally viene eseguito dopo il blocco catch
3. Una eccezione viene lanciata nel blocco try e nessun blocco catch la raccoglie. Il metodo termina e l'eccezione è passata al metodo chiamante. Prima che il metodo termini, viene eseguito il blocco finally
4. Prima che il blocco try arrivi alla conclusione, il controllo viene restituito al metodo chiamante. Prima di tornare al metodo chiamante, il codice nel blocco finally viene eseguito. Ricordare: il blocco finally viene comunque eseguito anche se c'è un return nel blocco try.

## Esempio completo: la classe Stack

---

- Esaminiamo un esempio completo costituito da una classe che definisce e gestisce correttamente le eccezioni.
- La classe si chiama **Stack** e implementa una “pila” (o catasta) di stringhe
- Uno stack è una struttura dati che funziona come una pila di fogli su cui è possibile aggiungere fogli in cima e toglierli nell’ordine inverso rispetto a quello in cui sono stati messi.
- In altre parole lo stack funziona con una logica **LIFO** (**L**ast **I**n **F**irst **O**ut): il primo foglio ad essere estratto è l’ultimo ad essere stato inserito.

## Stack – Specifiche

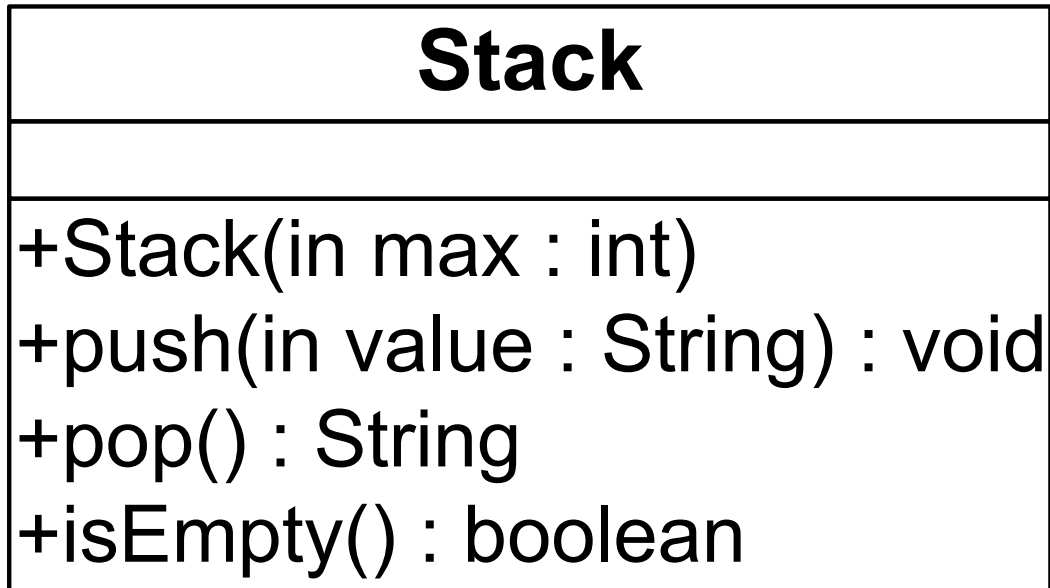
---

- Si chiede di realizzare una classe – denominata Stack – in grado di implementare una pila di stringhe
- Tale classe dovrà consentire di:
  - Stabilire all’atto della creazione il numero massimo di elementi che la pila può contenere (**costruttore**)
  - Mettere una stringa in cima alla pila (**push**)
  - Estrarre la stringa che si trova in cima alla pila (**pop**)
  - Sapere se la pila è vuota (**isEmpty**)
- La classe dovrà gestire – **utilizzando le eccezioni** - le situazioni anomale che si possono presentare.
- N.B. **push** e **pop** sono i nomi comunemente utilizzati per indicare le primitive di una struttura dati di questo tipo.

## Stack – Diagramma UML

---

- Trasformiamo le specifiche in un diagramma UML in cui per il momento definiamo solo i metodi:



## Stack – Scelte implementative

---

- Ci servono essenzialmente due cose:
  - Una struttura dati per memorizzare le stringhe: la soluzione più adeguata è un array (**items[]**)
  - Un intero per memorizzare il numero di elementi effettivamente presenti (**count**)
- Il diagramma UML completo di attributi sarà quindi:

Stack
-count : int -items[] : String
+Stack(in max : int) +push(in value : String) : void +pop() : String +isEmpty() : boolean

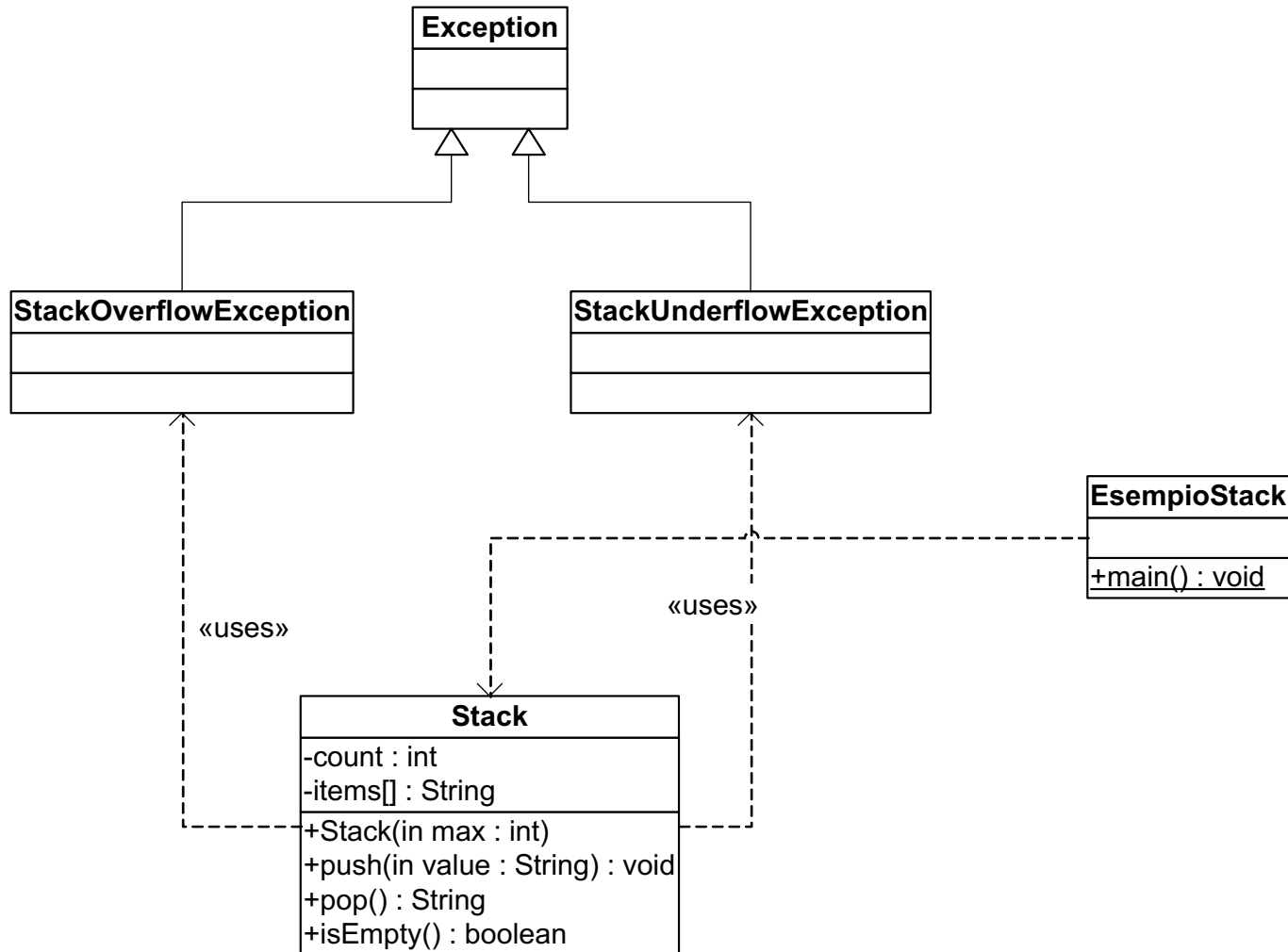
## Stack – Eccezioni

---

- Dal momento che il numero massimo degli elementi è fissato all'atto della creazione, le situazioni anomale che si possono verificare nell'uso della classe Stack sono due:
  - Tentativo di inserimento di un elemento quando lo stack è pieno (contiene il numero massimo di elementi possibili)
  - Tentativo di estrazione di un elemento quando lo stack è vuoto
- Ne consegue che avremo bisogno di definire ed emettere due eccezioni:
  - **StackOverflowException**
  - **StackUnderflowException**

# Stack – Diagramma delle classi completo

- Il diagramma UML completo è il seguente (EsemptoStack è la classe che ci serve per creare l'applicazione di esempio)





# Stack – Implementazione delle eccezioni

---

- Definiamo innanzitutto le due classi delle eccezioni

```
public class StackOverflowException extends Exception
{
    public StackOverflowException() { super();}
    public StackOverflowException(String s){super(s);}
}
```

```
public class StackUnderflowException extends Exception
{
    public StackUnderflowException() { super();}
    public StackUnderflowException(String s){super(s);}
}
```

# Stack – Implementazione di Stack/1

---

```
public class Stack
{
    private int count;
    private String[] items;

    public Stack(int max)
    {
        count = 0;
        items = new String[max];
    }

    public boolean isEmpty()
    {
        return (count == 0);
    }

    ...
}
```

## Stack – Implementazione di Stack/2

---

```
...
    public void push(String value)
        throws StackOverflowException
    {
        try
        {
            items[count] = value; // possibile eccezione
            count++;
        }
        catch (ArrayIndexOutOfBoundsException ae)
        {
            StackOverflowException oe =
                new StackOverflowException();
            throw oe;
        }
    }
...

```

## Stack – Implementazione di Stack/3

---

```
...  
    public String pop()  
        throws StackUnderflowException  
    { String value;  
      try  
      {  
          count--; // se ==0 diventa -1!  
          value = items[count]; // possibile eccezione  
          return value;  
      }  
      catch (ArrayIndexOutOfBoundsException ae)  
      {  
          count = 0; // azione correttiva!  
          StackUnderflowException ue =  
              new StackUnderflowException();  
          throw ue;  
      }  
    }  
}
```

---

# Stack – Implementazione di EsempioStack

---

```
public class EsempioStack
{
    public static void main(String args[])
    {
        Stack sk = new Stack(100);
        try
        {
            sk.push("Pippo");
            sk.push("Pluto");
            System.out.println(sk.pop()); // ok: Pluto
            System.out.println(sk.pop()); // ok: Pippo
            System.out.println(sk.pop()); // underflow!
        }
        catch (StackOverflowException e)
        { System.out.println("Overflow!"); }
        catch (StackUnderflowException e)
        { System.out.println("Underflow!"); }
        catch (Exception e) // catturiamo comunque tutto
        {System.out.println("Errore strano:"+e.getMessage());}
    }
}
```