Chiamate per valore e riferimento
Gestione delle eccezioni
File e flussi

Chiamate per valore e riferimento

- Un nuovo metodo **transfer** per la classe **BankAccount**:

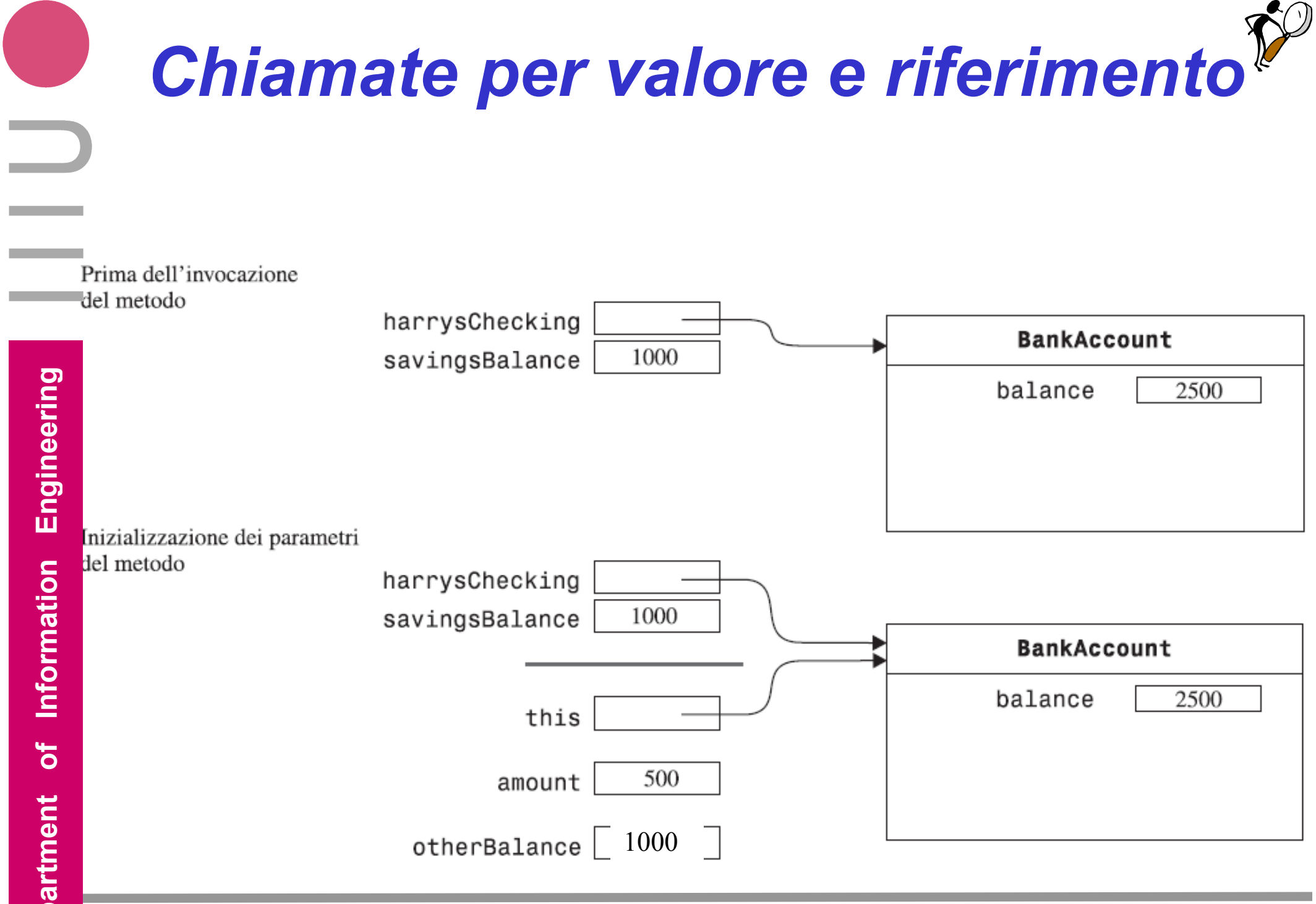
```
void transfer(double amount, double otherBalance)
{
   balance = balance - amount;
   otherBalance = otherBalance + amount;
}
```

- Proviamo ad usarlo:

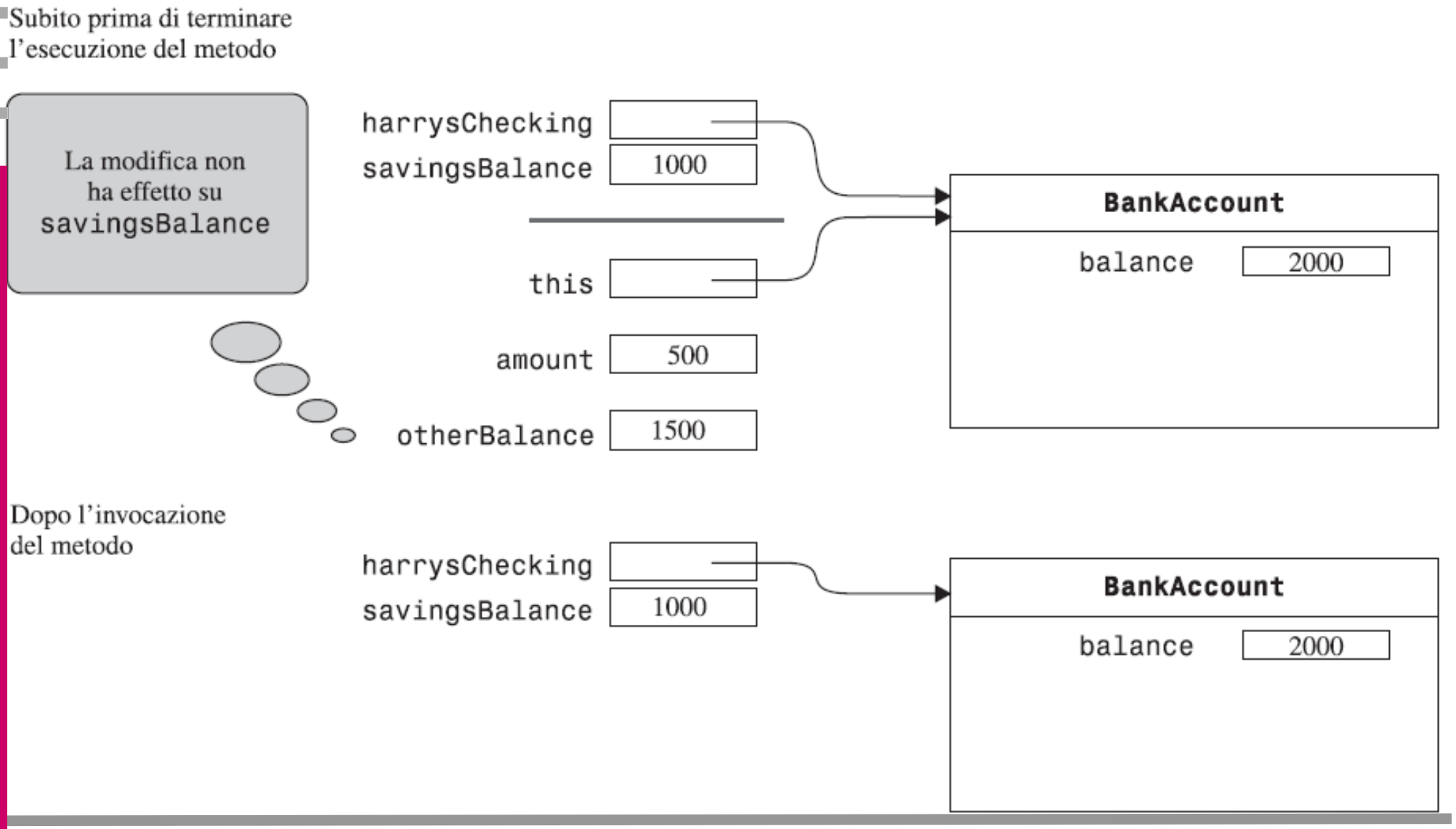
```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance);
System.out.println(savingsBalance);
```

- Non funziona!
  - All'inizio dell'esecuzione del metodo la variabile parametro **otherBalance** ha il valore di **savingsBalance**
  - La modifica del valore di otherBalance non ha effetto su savingsBalance perché **sono variabili diverse**
  - Alla fine dell'esecuzione **otherBalance non esiste più**

Chiamate per valore e riferimento



Chiamate per valore e riferimento



Chiamate per valore e riferimento

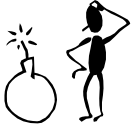
- A volte si dice **impropriamente** che in Java *i numeri sono passati per valore e gli oggetti per riferimento*
- In realtà in Java il passaggio è **sempre "per valore"**
  - il valore del parametro effettivo è assegnato al parametro formale, ma
  - passando per valore una variabile oggetto, si passa una **copia del riferimento** in essa contenuto
  - l'effetto "pratico" di questo passaggio è la possibilità di **modificare lo stato dell'oggetto stesso**, come avviene con il passaggio "per riferimento"
- Altri linguaggi (come C++) consentono il passaggio dei parametri **"per riferimento"**, rendendo possibile la modifica dei parametri effettivi

Department of Information Engineering

Materiale di complemento (Capitoli 3/8)

Errori tipici con i costruttori

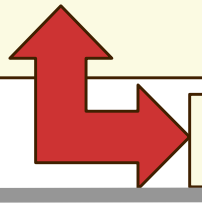
Invocare metodi senza oggetto



```
public class Program
{  public static void main(String[] args)
   {  BankAccount account = new BankAccount();
      deposit(500); // ERRORE
   }
}
```

- Il compilatore non sa a quale oggetto applicare il metodo **deposit**, che non è statico e quindi richiede sempre un riferimento a un oggetto da usare come parametro implicito
- Invece è possibile invocare un metodo non statico senza riferimento a un oggetto, quando lo si invoca da un altro metodo non statico della stessa classe

```
public void withdraw(double amount)
{  balance = getBalance() - amount;
}
```



```
this.getBalance()
```

Tentare di ricostruire *un oggetto*

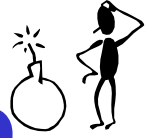
- A volte viene la tentazione di ***invocare un costruttore su un oggetto già costruito*** con l'obiettivo di ***riportarlo alle condizioni iniziali***

```
BankAccount account = new BankAccount();
account.deposit(500);
account.BankAccount(); // NON FUNZIONA!
```

```
cannot resolve symbol
symbol   : method BankAccount ( )
location : class BankAccount
```

- Il messaggio d'errore è un po' strano… il compilatore cerca un ***metodo*** **BankAccount**, non un ***costruttore***, e naturalmente non lo trova!

Tentare di ricostruire *un oggetto*



- Un costruttore crea un ***nuovo oggetto*** con prefissate condizioni iniziali
  - un costruttore può essere invocato ***soltanto*** con l'operatore **new**
- La soluzione è semplice
  - ***assegnare un nuovo oggetto alla variabile oggetto che contiene l'oggetto che si vuole "ricostruire"***

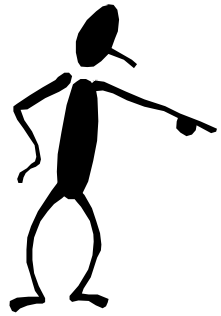
```
BankAccount account = new BankAccount();
account.deposit(500);
account = new BankAccount();
```

Pacchetti

I pacchetti di classi (package)

- Tutte le classi della libreria standard sono raccolte in ***pacchetti*** (***package***) e sono organizzate per argomento e/o per finalità
  - *Esempio*: la classe **Rectangle** appartiene al pacchetto **java.awt** (Abstract Window Toolkit)
- Per ***usare*** una classe di una libreria, bisogna ***importarla*** nel programma, usando l'enunciato
  - **import *nomePacchetto.NomeClasse***
- Le classi **System** e **String** appartengono al pacchetto **java.lang**
  - il pacchetto **java.lang** viene ***importato automaticamente***

Importare classi da un pacchetto



- Sintassi: `import nomePacchetto.NomeClasse;`
  - Scopo: importare una classe da un pacchetto, per poterla utilizzare in un programma
- Sintassi: `import nomePacchetto.*;`
  - Scopo: importare tutte le classi di un pacchetto, per poterle utilizzare in un programma
- Nota: le classi del pacchetto **java.lang** non hanno bisogno di essere importate
- Attenzione: non si possono importare ***più pacchetti*** con un solo enunciato

```
import java.*.*; // ERRORE
```

Stili per l'importazione di classi

- Un enunciato **import** per ogni classe importata

```
import java.math.BigInteger;
import java.math.BigDecimal;
```

- Un enunciato **import** per ***tutte le classi di un pacchetto***
  - non è un errore importare classi che non si usano!

```
import java.math.*;
```

- Se si usano più enunciati di questo tipo
  - non è chiaro il pacchetto di appartenenza delle classi
  - Ci possono essere classi omonime in pacchetti diversi

```
import java.util.*;
import javax.swing.*;
```

  - La classe **Timer**, appartiene a **entrambi i pacchetti**
    - Se si definisce una variabile di tipo **Timer** il compilatore segnala un errore di "**riferimento ambiguo**"

Stili per l'importazione di classi

- È possibile non usare per nulla gli enunciati **import**, e indicare sempre il ***nome completo*** delle classi utilizzate nel codice

```
java.math.BigInteger a =
    new java.math.BigInteger("123456789");
```

- Questo stile è ***assai poco usato***, perché è molto noioso, aumenta la probabilità di errori di battitura e aumenta la lunghezza delle linee del codice (diminuendo così la leggibilità del programma)

La gestione delle eccezioni (capitolo 11)

Argomenti inattesi (precondizoni)

- Spesso un metodo richiede che i suoi argomenti
  - siano di un tipo ben definito
    - questo viene garantito dal compilatore
  - abbiano un valore che rispetti certi ***vincoli***, per esempio sia un numero positivo
    - in questo caso il compilatore non aiuta...
- Come deve reagire il metodo se riceve un parametro che non rispetta i ***requisiti richiesti*** (chiamati ***precondizioni***)?

Argomenti inattesi (precondizioni)

- Ci sono quattro modi per reagire ad argomenti inattesi
  - ***non fare niente***
    - il metodo termina la propria esecuzione senza segnalare errori. In questo caso la documentazione del metodo deve indicare chiaramente le precondizioni. La responsabilità di ottemperare alle precondizioni è del metodo chiamante.
    - Eseguire solo se le precondizioni sono soddisfatte:

```
if (precondizioni) {... corpo del metodo ...}
```

      - questo però si può fare solo per metodi con valore di ritorno **void**, altrimenti che cosa restituisce il metodo?
      - Se restituisce un valore casuale senza segnalare un errore, chi ha invocato il metodo probabilmente andrà incontro a un errore logico

Argomenti inattesi (precondizioni)

- ***Terminare il programma*** con **System.exit(1)**
  - questo è possibile soltanto in programmi non professionali
  - La terminazione di un programma attraverso il metodo statico **System.exit()** non fornisce un meccanismo standard per informare dell'avvenuto

    Invece, **System.exit(0)** termina l'esecuzione segnalando che **tutto è andato bene**
- ***Usare una asserzione***
  - Il meccanismo delle "asserzioni" si usa solo per i programmi in fase di sviluppo e collaudo.
  - Noi **non** lo esaminiamo
- ***Lanciare un'eccezione***

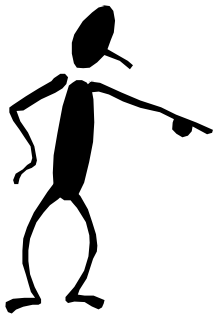
Lanciare una eccezione

- Il meccanismo generale di segnalazione di errori in Java consiste nel "lanciare" (***throw***) un'***eccezione***
  - si parla anche di ***sollevare*** o ***generare*** un'eccezione
- Lanciare un'eccezione in risposta a un parametro che non rispetta una precondizione è la soluzione più corretta in ambito professionale
  - la libreria standard mette a disposizione l'eccezione:

    IllegalArgumentException

```
public void deposit(double amount)
{   if (amount <= 0)
       throw new IllegalArgumentException();
    balance = balance + amount;
}
```

Enunciato throw



- Sintassi:

```
throw oggettoEccezione;
```

- Scopo: lanciare un'eccezione
- Nota:
  - di solito l'***oggettoEccezione*** viene creato con **new *ClasseEccezione*( )**
  - Non è necessario memorizzare il riferimento in una variabile oggetto

Le eccezioni in Java

- Quando un metodo ***lancia*** un'eccezione
  - l'esecuzione del metodo viene **interrotta**
  - l'eccezione viene "**propagata**" al metodo chiamante, la cui esecuzione viene a sua volta interrotta
  - l'eccezione viene via via propagata fino al metodo **main**, la cui interruzione provoca **l'arresto anormale** del programma con la segnalazione dell'eccezione che è stata la causa di tale terminazione prematura
- Il lancio di un'eccezione è quindi un modo per terminare un programma in caso di errore
  - ***non sempre, però, gli errori sono così gravi...***

Gestire le eccezioni

- Problema tipico: convertire in formato numerico una stringa introdotta dall'utente
  - se la stringa non contiene un numero valido, viene generata un'eccezione **NumberFormatException**
  - vogliamo **intercettare** e **gestire** l'eccezione, segnalando l'errore all'utente e chiedendo di inserire un nuovo dato, anziché **terminare** prematuramente il programma
- L'invocazione del metodo che può generare l'eccezione deve essere racchiuso in un **blocco try**

```
try {
    ...
    n = Integer.parseInt(line);
    ...
}
```

Gestire le eccezioni

- Il blocco try è seguito da una **clausola catch**
  - definita in modo simile a **un metodo** che riceve **un solo parametro**, del ***tipo dell'eccezione*** che si vuole gestire
  - Bisogna sapere di che **tipo** è l'eccezione generata dal metodo, nel nostro caso **NumberFormatException**

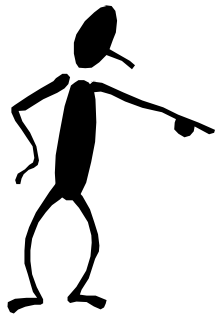
```
catch (NumberFormatException e)
{   System.out.println("messaggio");
}
```

- Nel blocco **catch** si trova il codice che deve essere eseguito **nel caso in cui si verifichi l'eccezione**
  - l'esecuzione del blocco **try** viene interrotta nel punto in cui si verifica l'eccezione e non viene più ripresa

Esempio: gestire NumberFormatException

```
import java.util.Scanner;
public class IntEater
{   public static void main(String[] args)
    {
      System.out.println("Passami un numero int!");
      Scanner in = new Scanner(System.in);
      int n = 0;
      boolean done = false;
      while (!done)
      {  try{  String line = in.nextLine();
               n = Integer.parseInt(line);
               // l'assegnazione seguente e` eseguita
               // solo se NON viene lanciata l'eccezione
               done = true;  }
         catch(NumberFormatException e)
             { System.out.println("No, voglio un int");
               // all'uscita del blocco catch done e` false
             }
       }
   System.out.println("Grazie, " + n + " e` un numero int!");
    }
}
```

Enunciati try/catch

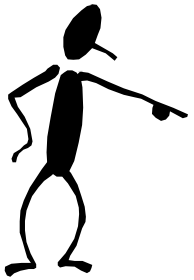


- Sintassi:

```
try
{   enunciatiCheForseGeneranoUnaEccezione
}
catch (ClasseEccezione oggettoEccezione)
{   enunciatiEseguitiInCasoDiEccezione
}
```

- Scopo: eseguire enunciati che possono generare una eccezione
  - ***se si verifica l'eccezione*** di tipo ***ClasseEccezione***, eseguire gli enunciati contenuti nella ***clausola*** **catch**
  - ***altrimenti***, ignorare la ***clausola*** **catch**

Enunciati try/catch



- Se gli enunciati nel blocco try possono generare più eccezioni di diverso tipo, è necessario prevedere un blocco catch per ogni tipo di eccezione

```
try
{ enunciatiCheForseGeneranoPiu'Eccezioni
}
catch (ClasseEccezione1 oggettoEccezione1)
{ enunciatiEseguitiInCasoDiEccezione1
}
catch (ClasseEccezione2 oggettoEccezione2)
{ enunciatiEseguitiInCasoDiEccezione2
}
```

Perché usare eccezioni?

- **Idea generale**: per gestire un errore di esecuzione ci sono due compiti da svolgere
  - **Individuazione**
  - **Ripristino**
- **Problema**: il punto in cui viene individuato l'errore di solito non coincide con il punto di ripristino
  - Esempio: il metodo **parseInt** fallisce (ovvero la stringa in esame non può essere convertita in intero)
  - Il metodo **parseInt** non sa come gestire questo fallimento: **dipende dal contesto!**
- **Soluzione**: il meccanismo di lancio/cattura di eccezioni
  - Consente di gestire un errore di esecuzione **in un punto diverso** rispetto a dove questo si è generato
  - **Obbliga** a gestire l'errore, altrimenti l'esecuzione termina

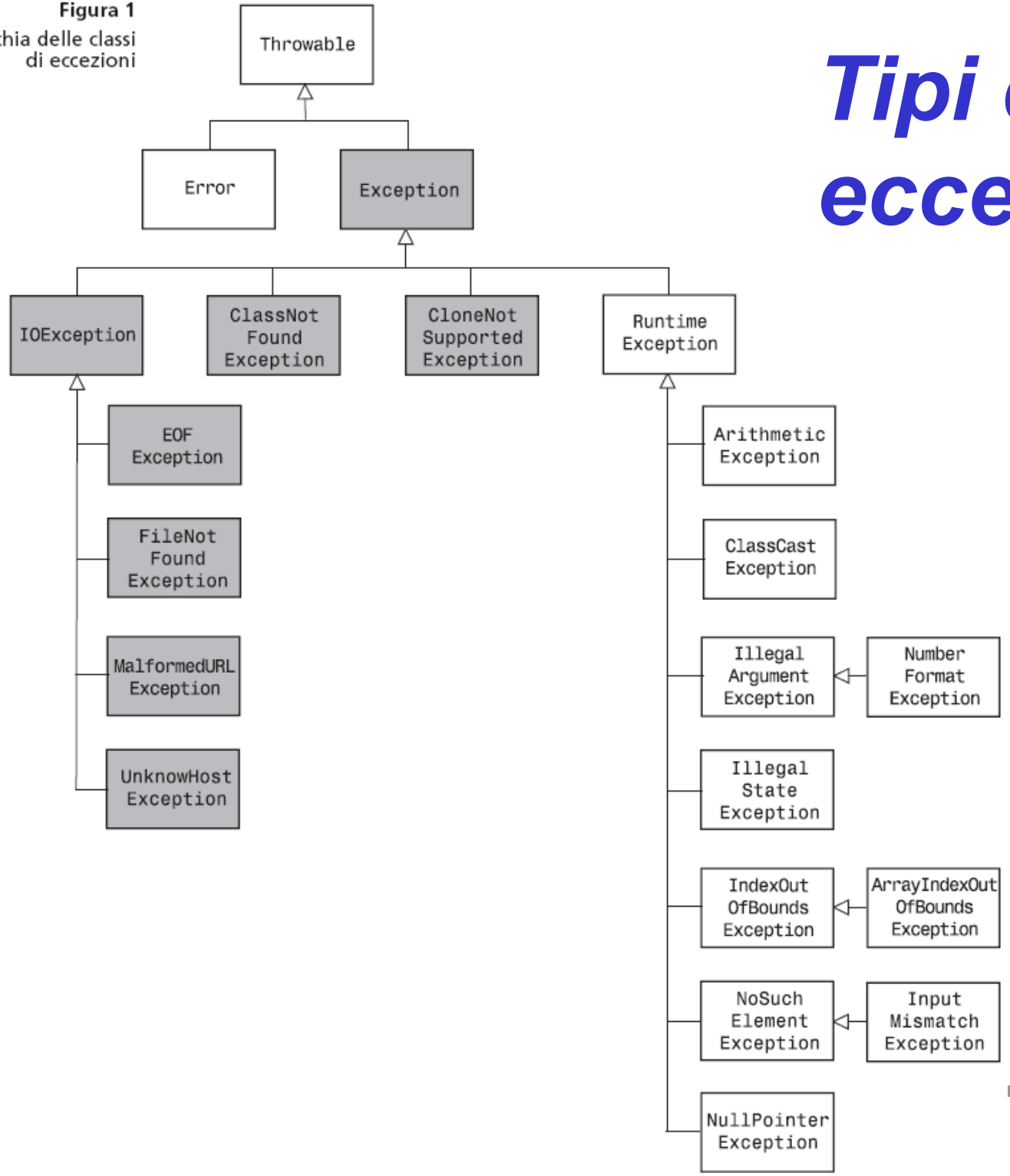
Come gestire le eccezioni?

“Lanciare presto, catturare tardi”

- Quando un metodo **incontra un problema** che non è in grado di gestire al meglio
  - Conviene lanciare un’eccezione piuttosto che mettere in atto una soluzione imprecisa/incompleta
  - **“Lanciare presto”**
- Quando un metodo **riceve un’eccezione** lanciata da un altro metodo
  - Conviene catturarla (catch) **solo** se si è effettivamente in grado di risolvere il problema in maniera competente
  - Altrimenti è meglio lasciare **propagare** l’eccezione (eventualmente fino al metodo **main** in esecuzione)
  - **“Catturare tardi”**

Tipi di eccezioni

Figura 1
chia delle classi
di eccezioni

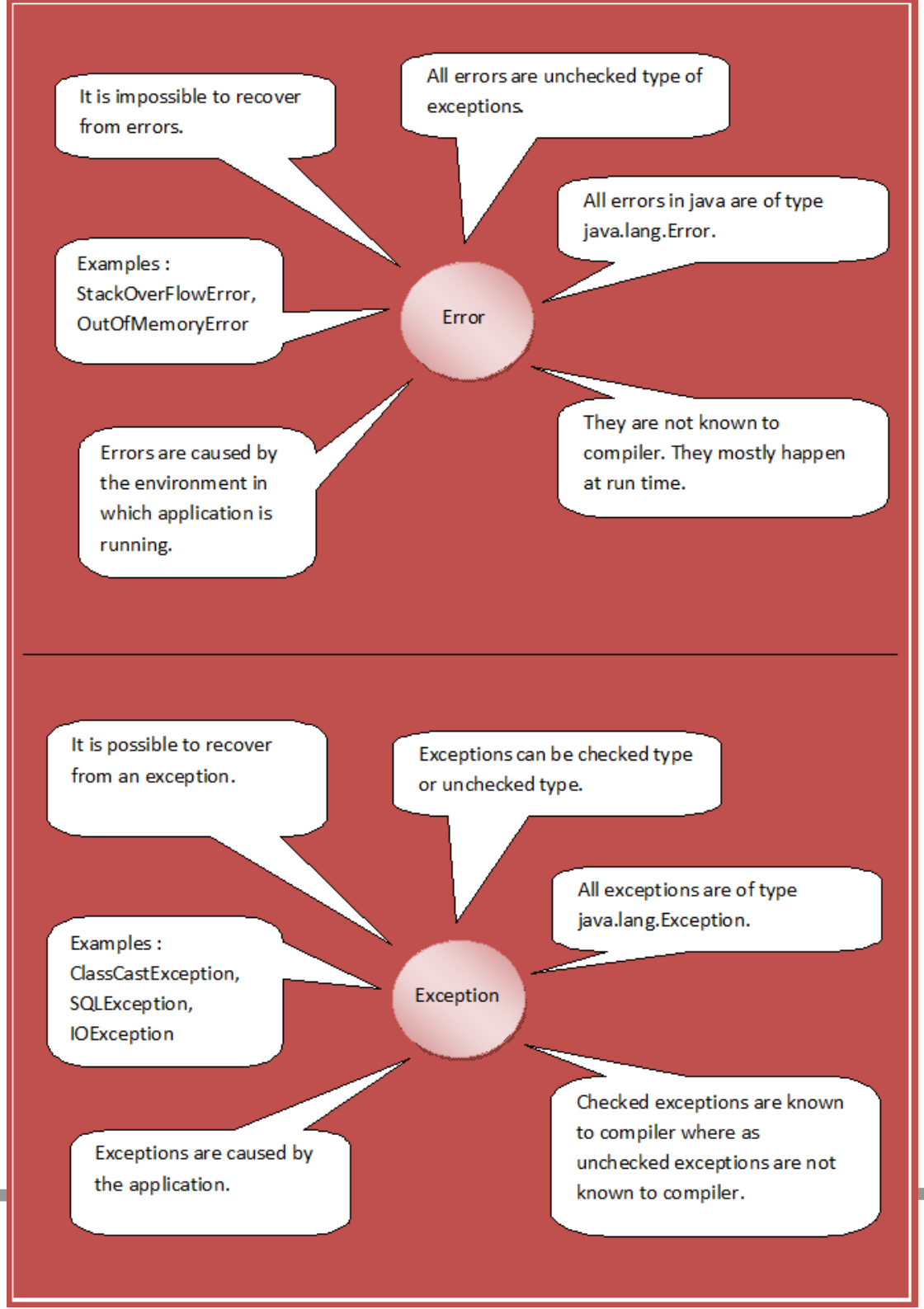


Diversi tipi di eccezioni

- In Java esistono diversi tipi di eccezioni (cioè diverse classi di cui le eccezioni sono esemplari)
  - eccezioni di tipo **Error**
  - eccezioni di tipo **Exception**
    - un sottoinsieme sono di tipo **RuntimeException**
- Eccezioni di tipo **Error** e di tipo **RuntimeException** sono **eccezioni non controllate**
  - La loro gestione è ***facoltativa***, ovvero lasciata alla scelta del programmatore
  - se non vengono gestite e vengono lanciate, provocano la terminazione del programma
- Le altre sono **eccezioni controllate**
  - la loro gestione è ***obbligatoria***

Eccezioni controllate e non

- Le eccezioni **controllate**
  - Descrivono problemi che possono verificarsi prima o poi, indipendentemente dalla bravura del programmatore
    - Per questo motivo le eccezioni di tipo **IOException** sono controllate
    - Se si invoca un metodo che può lanciare un'eccezione controllata, è **obbligatorio** gestirla con try/catch
    - Altrimenti viene segnalato un **errore in compilazione**
- Le eccezioni **non controllate**
  - Descrivono problemi dovuti a errori del programmatore e che quindi non dovrebbero verificarsi (in teoria…)
    - Per questo motivo le eccezioni di tipo **RuntimeException** sono non controllate
    - Non è obbligatorio catturarle tramite try/catch


It is impossible to recover from errors.
All errors are unchecked type of exceptions.
All errors in java are of type java.lang.Error.
Examples : StackOverFlowError, OutOfMemoryError
Error
They are not known to compiler. They mostly happen at run time.
Errors are caused by the environment in which application is running.
It is possible to recover from an exception.
Exceptions can be checked type or unchecked type.
All exceptions are of type java.lang.Exception.
Examples : ClassCastException, SQLException, IOException
Exception
Checked exceptions are known to compiler where as unchecked exceptions are not known to compiler.
Exceptions are caused by the application.

Eccezioni controllate

Gestire le eccezioni di Input/Output

- **Scanner** può essere usata anche per leggere file (lo vedremo...); in questo caso il parametro esplicito è un oggetto di classe **FileReader**
  - Se non trova il file, il costruttore **FileReader()** lancia l'**eccezione controllata java.io.FileNotFoundException**

```
import java.io.FileNotFoundException;    // QUESTO CODICE
import java.io.FileReader;               // NON COMPILA!
import java.util.Scanner;
public class MyInputReader
{
    public static void main(String[] args)
    {   String filename = "filename.txt";
        FileReader reader = new FileReader(filename);
        Scanner in = new Scanner(reader);
    }
}
```

MyInputReader.java:9:
unreported exception java.io.FileNotFoundException;
must be caught or declared to be thrown

Gestire le eccezioni di IO

- Perché le eccezioni di IO sono controllate?
  - Perchè sono associate a problemi che si possono verificare **indipendentemente** dalla bravura del programmatore
- Quando si leggono dati da un **flusso di input**
  - se l'input proviene da un file su disco, il disco può essere **difettoso**
  - se l'input proviene da una connessione di rete, la connessione può **interrompersi** durante la lettura
  - se l'input proviene dalla tastiera, l'utente può inserire informazioni **non corrette** (per esempio può **non** inserire input)

Gestire le eccezioni di IO

- Ci sono **varie strategie** per gestire eccezioni di IO
- Il modo più semplice: **terminazione** del programma
  - **intercettiamo** (**catch**) l'eccezione quando si verifica, usando un **blocco try** con una **clausola catch**, e terminiamo l'esecuzione del programma con una segnalazione d'errore

```
try {
    String name = buffer.readLine();
}
catch (IOException e)
{   System.out.println(e);
    System.exit(1);
}
```

Gestire eccezioni di IO

```
catch (IOException e)
{ System.out.println(e);
  System.exit(1);
}
```

```
System.out.println(e);
```

- Se viene generata l'eccezione, il programma stampa sull'output standard la descrizione testuale standard dell'eccezione

```
System.exit(1);
```

- Poi il programma **termina l'esecuzione** segnalando che qualcosa **non ha funzionato correttamente**

Invece, **System.exit(0)** termina l'esecuzione segnalando che **tutto è andato bene**

- Si possono, ovviamente, adottare strategie di gestione più **elaborate**
  - Per esempio consentire l'immissione di un nuovo nome di file se il file non viene trovato

Propagare eccezioni di IO: `throws`

- Come per le eccezioni non controllate, un metodo può non gestire un'eccezione controllata e **propagare** l'eccezione
  - Il metodo termina la propria esecuzione
  - E lascia la gestione al metodo chiamante
- **Attenzione**: Per dichiarare che un metodo propaga un'eccezione controllata, si contrassegna il metodo con il marcatore **throws**

```
//questo metodo read non gestisce le FileNotFoundException e lo
//deve dichiarare tramite il marcatore throws
public void read(String filename)  throws FileNotFoundException
{
  FileReader reader = new FileReader(filename);
  Scanner in = new Scanner(reader);
  ...
}
```

Propagare eccezioni di IO da main

- Se non si vuole gestire un'eccezione controllata, si può dichiarare che il metodo **main la propaga**
  - Questo è in assoluto **il modo più semplice** (anche se **non il migliore**) di scrivere un main in cui si possono generare eccezioni di IO

```
import java.io.FileNotFoundException;    // QUESTO CODICE
import java.io.FileReader;               // COMPILA!
import java.util.Scanner;
public class MyInputReader
{
    public static void main(String[] args) throws IOException
    {   String filename = "filename.txt";
        FileReader reader = new FileReader(filename);
        Scanner in = new Scanner(reader);
    }
}
```

Le eccezioni di Scanner

- **Scanner** è una **classe di utilità**. **Non** appartiene al pacchetto IO e non lancia eccezioni di IO!
- Le principali eccezioni di **Scanner** (Unchecked)**:**
  - **NoSuchElementException** lanciata per es. da **next** e **nextLine** se l'input non è disponibile
  - **InputMismatchException** lanciata da **nextInt nextDouble**, ecc., se l'input non corrisponde al formato richiesto
  - **IllegalStateException** lanciata da molti metodi se invocati quando lo Scanner è “chiuso” (ovvero dopo un'invocazione del metodo **close**)
- **Suggerimento**: quando possibile usiamo solo i metodi **next** e **nextLine** di **Scanner**, se necessario convertiamo stringhe in numeri usando **Double.parseDouble** e **Integer.parseInt**

Department of Information Engineering

Materiale di complemento (capitolo 11 – eccezioni di IO)

Gestire le eccezioni di input

- Abbiamo detto che un flusso va sempre **chiuso** (usando il metodo **close**) per rilasciare le risorse
  - E se l'esecuzione viene interrotta da una eccezione prima dell'invocazione del metodo close?
  - Si verifica una situazione di **potenziale instabilità**
- Si può usare la clausola **finally**
  - Il corpo di una clausola **finally** viene eseguito comunque, indipendentemente dal fatto che un'eccezione sia stata generata oppure no

Gestire eccezioni di input

- Esempio: uso di **try/finally**

```
FileReader reader = new FileReader(filename);
Scanner in = new Scanner(reader);
readData(in);
reader.close(); // può darsi che non si arrivi mai qui
```

```
FileReader reader = new FileReader(filename);
try
   {
     Scanner in = new Scanner(reader);
     readData(in);
   }
finally
   { reader.close(); }  // questa istruzione viene eseguita
                        // comunque prima di passare
                        // l'eccezione al suo gestore
```

Progettare nuovi tipi di eccezioni

- Ora sappiamo
  - programmare nei nostri metodi il lancio di eccezioni appartenenti alla libreria standard (**throw new** …)
  - gestire eccezioni (**try/catch**)
  - lasciare al chiamante la gestione di eccezioni sollevate nei nostri metodi (**throws** nell'intestazione del metodo)
- Non sappiamo ancora **creare eccezioni** solo nostre
  - Per esempio, se vogliamo lanciare l'eccezione **LaNostraEccezioneException** come facciamo?
- Per ora **saltiamo questo argomento**
  - Lo riprenderemo quando avremo studiato il concetto di **ereditarietà** in Java

Department of Information Engineering

File e flussi (capitolo 11)

Leggere/scrivere file di testo

Gestione di file in Java

- Finora abbiamo visto programmi Java che interagiscono con l'utente soltanto tramite i **flussi di ingresso e di uscita standard**
- È possibile leggere e scrivere file **all'interno** di un programma Java?
- Ci interessa soprattutto affrontare il problema della ***gestione di file di testo*** (file contenenti caratteri)
  - esistono anche i ***file binari***, che contengono semplicemente configurazioni di bit che rappresentano qualsiasi tipo di dati
- La gestione dei file avviene interagendo con il sistema operativo mediante classi del pacchetto **java.io** della libreria standard

Leggere un file di testo

- Il modo più semplice:
  - Creare un oggetto "**lettore di file**" (**FileReader**);
  - Creare un oggetto **Scanner**, che già conosciamo
  - **Collegare** l'oggetto **Scanner** al lettore di file invece che all'input standard

```
FileReader reader = new FileReader("input.txt");
Scanner in = new Scanner(reader);
```

- In questo modo possiamo usare i consueti metodi di **Scanner** (**next**, **nextLine**, ecc.) per leggere i dati contenuti nel file

La classe FileReader

- Prima di leggere caratteri da un file (esistente) occorre **aprire il file in lettura**
  - questa operazione si traduce in Java nella creazione di un oggetto di tipo **FileReader**

```
FileReader reader = new FileReader("file.txt");
```

  - il costruttore necessita del nome del file sotto forma di **stringa**: “file.txt”
- ***Attenzione***: se il file non esiste, viene lanciata l’eccezione **FileNotFoundException** a gestione obbligatoria

Leggere file con Scanner

- Si possono leggere file usando i metodi di **FileReader**
  - Noi non lo facciamo
  - È più comodo "**avvolgere**" l'oggetto **FileReader** in un oggetto di tipo **Scanner**

```
FileReader reader = new FileReader("file.txt");
Scanner in = new Scanner(reader);
while(in.hasNextLine())
{   String line = in.nextLine();
    //... elaborazione della stringa
} // il costruttore FileReader lancia IOException, da gestire!!
```

- Si può leggere **una riga alla volta** usando il metodo **nextLine** di **Scanner**
  - Quando il file finisce, il metodo **hasNextLine** di **Scanner** restituisce un valore false

Chiudere file in lettura

- Al termine della lettura del file (che non necessariamente deve procedere fino alla fine…) **occorre chiudere il file**

```
FileReader reader = new FileReader("file.txt");
Scanner in = new Scanner(reader);
...
reader.close();
```

- Il metodo **close()** lancia **IOException**, da gestire obbligatoriamente
- Se il file non viene chiuso non si ha un errore, ma una **potenziale situazione di instabilità** per il sistema operativo

Scrivere su un file di testo

- Il modo più semplice:
  - Creare un oggetto "scrittore" **PrintWriter**
  - **Collegare** l'oggetto **PrintWriter** a un file

```
PrintWriter out = new PrintWriter("output.txt");
```

  - In questo modo possiamo usare i metodi **print**, **println**, ecc. per scrivere i dati contenuti nel file
- ***Attenzione***: se *output.txt* non esiste viene **creato**. Ma se esiste già viene **svuotato** prima di essere scritto
  - E se vogliamo aggiungere in coda al file senza cancellare il testo gia' contenuto?

boolean append

```
FileWriter writer = new FileWriter("file.txt", true);
PrintWriter out = new PrintWriter(writer);
```

Scrivere su un file di testo

- ***Attenzione***: bisogna sempre **chiudere** un oggetto **PrintWriter** dopo avere terminato di usarlo

```
PrintWriter out = new PrintWriter("output.txt");
...
out.close();
```

- Anche questo metodo lancia **IOException**, da gestire **obbligatoriamente**
- Se non viene invocato non si ha un errore, ma è possibile che **la scrittura del file non venga ultimata** prima della terminazione del programma, lasciando il file **incompleto**

Rilevare la fine dell'input

- Molti problemi di elaborazione richiedono la **lettura** di una sequenza di dati in ingresso
  - Per esempio, calcolare la somma di numeri in virgola mobile, ogni numero inserito su una riga diversa
- Altrettanto spesso il programmatore **non sa** quanti saranno i dati forniti in ingresso
- **Problema**: leggere una sequenza di dati in ingresso **finché i dati non sono finiti**
  - In particolare quando leggiamo dati da un file
- **Possibili soluzioni**:
  - Terminazione esplicita. Abbiamo già visto che è possibile usare **caratteri sentinella** (per esempio "Q" per terminare un ciclo e mezzo)
  - Terminazione dell'input (per es. fine del file). **Scanner** possiede i metodi **hasNext**, **hasNextLine**, ecc., che restituiscono false se l'input è **terminato**

Rilevare la fine dell'input

- Usiamo il metodo predicativo **hasNextLine** come condizione di uscita dal ciclo di input

```
FileReader reader = new FileReader("input.txt");
Scanner in = new Scanner(reader);
// ma anche Scanner in = new Scanner(System.in);

while (in.hasNextLine())
{
   String line = in.nextLine();
        ... // elabora line
}
```

Esempio: LineNumberer.java

```
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;

public class LineNumberer
{
   public static void main(String[] args)
   {
      Scanner console = new Scanner(System.in);
      System.out.print("Input file: ");
      String inputFileName = console.nextLine();
      System.out.print("Output file: ");
      String outputFileName = console.nextLine();

      try
      {
                                                    //(continua)
```

Esempio: LineNumberer.java

```
            FileReader reader = new FileReader(inputFileName);
            Scanner in = new Scanner(reader);
            PrintWriter out = new PrintWriter(outputFileName);
            int lineNumber = 1;

            while (in.hasNextLine())
            {
               String line = in.nextLine();
               out.println("/* " + lineNumber + " */ " + line);
               lineNumber++;
            }
            out.close();
         }
         catch (IOException e) //gestione obbligatoria...
         {
            System.out.println("Error processing file:" + e);
         }
      }
}
```

È tutto chiaro? …

1. Cosa succede se al programma **LineNumberer** viene fornito lo stesso nome per file in ingresso e in uscita?
2. Cosa succede se al programma **LineNumberer** viene fornito un nome di file di ingresso inesistente?

Rilevare la fine dell'input da tastiera

- Dopo aver iniziato l'esecuzione di un programma, si introducono **da tastiera** i dati che si vogliono elaborare.
- Per terminare esplicitamente l'immissione, si può usare un **valore sentinella**
- Oppure, si può **comunicare al sistema operativo** che l'input da console è terminato, come se fossimo giunti alla fine di un file in lettura
  - in una finestra DOS/Windows bisogna digitare **Ctrl+Z**
  - in una ***shell*** di Unix bisogna digitare **Ctrl+D**
- Il flusso **System.in** di Java non leggerà questi caratteri speciali, ma riceverà dal sistema operativo la segnalazione che l'input è terminato, e **hasNextLine** restituirà un valore **false**

Esempio: calcolare la somma di numeri

```
import java.util.Scanner;

public class SumTester
{  public static void main(String[] args)
   {   Scanner in = new Scanner(System.in);
       double sum = 0;
       boolean done = false;
       while (!done)
       {
         String line;
         /* attenzione a questa condizione: stiamo usando la
            valutazione pigra e stiamo assegnando un nuovo
            valore a line */
          if (!in.hasNextLine() ||
               (line = in.nextLine()).equalsIgnoreCase("Q"))
             done = true;
          else
             sum = sum + Double.parseDouble(line);
       }
       System.out.println("Somma: " + sum);
   }
}
```

Ancora elaborazione dell'input

- **Scomposizione di stringhe in "token"**
- **redirezione, piping**
- **Il flusso di errore standard**

Scomposizione di stringhe

Scomposizione di stringhe

- Spesso è comodo o naturale per l'utente inserire **più dati per riga**
  - Esistono i metodi **next**, **nextInt**, **nextDouble** di **Scanner**
- **Strategia alternativa**: leggere un'intera riga con il metodo **nextLine**, di **Scanner**, poi estrarre le sottostringhe relative ai singoli dati che la compongono
  - non si può usare **substring**, perché in generale non sono note lunghezza e posizione di inizio dei singoli dati
- Una sottostringa delimitata da caratteri speciali (definiti **delimitatori**) si chiama **lessema** o **token**
  - I caratteri delimitatori **non** possono essere usati nei token
- **Scanner** considera come delimitatori di sottostringhe gli spazi, i caratteri di tabulazione e i caratteri di "a capo"

Scomposizione di stringhe

- Per scomporre una stringa in token usando **Scanner**, innanzitutto bisogna creare un oggetto della classe fornendo **la stringa** come parametro al costruttore

```
Scanner in = new Scanner(System.in);
String line = in.nextLine();
Scanner t = new Scanner(line);
```

- Successive invocazioni del metodo **next** restituiscono successive sottostringhe, fin quando l'invocazione di **hasNext** restituisce **true**

```
while (t.hasNext())
{   String token = t.next();
    // elabora token
}
```

Esempio: contare token in un testo

```
import java.util.Scanner;

public class TokenCounter
{
  public static void main(String[] args)
  {
    Scanner in = new Scanner(System.in);
    int count = 0;
    while (in.hasNextLine())
    {
        String line = in.nextLine();
        Scanner t = new Scanner(line);
        while (t.hasNext())
        {
           t.next(); // non devo elaborare
           count++;
        }
    }
    System.out.println(count + " token");
  }
}
```

Riassunto: elaborare input in java

- Principali classi e metodi utilizzabili
  - **Scanner** e **nextLine()** per leggere intere righe di input
    - **FileReader** per gestire input da file
  - **Scanner** e **next** per scomporre le righe in parole
  - **Integer.parseInt()** e **Double.parseDouble()** per convertire stringhe in numeri
- Situazioni da gestire
  - **Terminazione di input**: metodo **hasNextLine()** di **Scanner**, e/o uso di caratteri sentinella (per es. Q)
  - **IOException** (da gestire **obbligatoriamente**) se lavoriamo con file
  - **NumberFormatException** (gestione opzionale) lanciate da **Integer.parseInt()** e **Double.parseDouble()**

Reindirizzamento di input e output, canalizzazioni ("pipes")

Reindirizzamento di input e output

- Usando i programmi scritti finora si inseriscono dei dati da tastiera, che al termine non vengono memorizzati
  - ***per elaborare una serie di stringhe bisogna inserirle tutte, ma non ne rimane traccia!***
- Una soluzione "logica" sarebbe che ***il programma leggesse le stringhe da un file***
  - questo si può fare con il **reindirizzamento dell'input standard**, consentito da quasi tutti i sistemi operativi

Reindirizzamento di input e output

```
//classe che ripete a pappagallo l'input inserito, 1 token a riga
import java.util.Scanner;
public class Pappagaller
{   public static void main(String[] args)
    {   Scanner in = new Scanner(System.in);
        while(in.hasNext())
           System.out.println(in.next());
    }
}
```


- Il reindirizzamento dell'input standard, sia in sistemi Unix che nei sistemi MS Windows, si indica con il carattere **<** seguito dal **nome del file da cui ricevere l'input**

```
java Pappagaller < testo.txt
```

  - Il file **testo.txt** viene ***collegato*** all'input standard
- Il programma non ha bisogno di alcuna istruzione particolare, semplicemente **System.in** non sarà più collegato alla tastiera ma al file specificato

Reindirizzamento di input e output

- A volte è comodo anche il reindirizzamento dell'**output**
  - Per esempio, quando il programma produce molte righe di output, che altrimenti scorrono velocemente sullo schermo senza poter essere lette

```
java Pappagaller > output.txt
```

- I due reindirizzamenti possono anche essere **combinati**

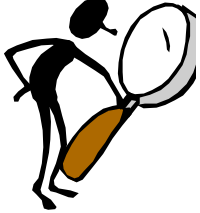
```
java Pappagaller < testo.txt > output.txt
```

Canalizzazioni ("pipes")

- Supponiamo di dovere ulteriormente elaborare l'output prodotto da un programma
- Per esempio, una elaborazione molto comune consiste nell'***ordinare*** le parole
  - questa elaborazione è così comune che molti sistemi operativi hanno un programma **sort**, che riceve da standard input un insieme di stringhe (una per riga) e le stampa a standard output ordinate lessicograficamente (una per riga)
  - Per ottenere le parole di **testo.txt** una per riga e ordinate, abbiamo bisogno di un ***file temporaneo*** (per es. **temp.txt**) che ***serve solo a memorizzare il risultato intermedio***, prodotto dal primo programma e utilizzato dal secondo

```
java Pappagaller < testo.txt > temp.txt
sort < temp.txt > testoOrdinato.txt
```

Canalizzazioni ("pipes")

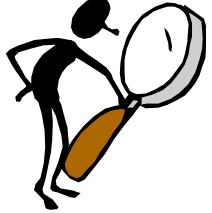


- Per ottenere le parole di **testo.txt** una per riga e ordinate, abbiamo bisogno di un ***file temporaneo***

```
java Pappagaller < testo.txt > temp.txt
sort < temp.txt > testoOrdinato.txt
```

- Il file temporaneo **temp.txt** ***serve soltanto per memorizzare il risultato intermedio***, prodotto dal primo programma e utilizzato dal secondo
- Questa situazione è talmente comune che quasi tutti i sistemi operativi offrono un'alternativa

Canalizzazioni ("pipes")



- Anziché utilizzare un file temporaneo per memorizzare l'output prodotto da un programma che deve servire da input per un altro programma, si può usare una ***canalizzazione*** ("***pipe***")

```
java Pappagaller < testo.txt | sort > testoOrdinato.txt
```

- La canalizzazione può anche ***prolungarsi***
  - Ad esempio, vogliamo sapere quante parole ci sono nel testo, senza contare le ripetizioni
  - Usiamo **sort** con l'opzione **-uf** (non ripete parole uguali, e non considera differenze tra maiuscole e minuscole)
  - Canalizziamo il risultato sul programma **wc** (word count)

```
java Pappagaller < testo.txt | sort -uf | wc
```

Il flusso di errore standard

Il flusso di errore standard

- Abbiamo visto che un programma Java ha sempre due flussi a esso collegati, forniti dal sistema operativo
  - il flusso di ingresso standard, **System.in**
  - il flusso di uscita standard, **System.out**
- In realtà esiste un altro flusso
  - Il flusso di **errore standard** (**standard error**), **System.err**
  - **System.err** è di tipo **PrintStream** come **System.out**
- La differenza tra **out** e **err** è solo convenzionale
  - si usa **System.out** per comunicare all'utente i risultati dell'elaborazione o qualunque messaggio previsto dal corretto e normale funzionamento del programma
  - si usa **System.err** per comunicare all'utente eventuali condizioni di errore (fatali o non) verificatesi durante il funzionamento del programma

Il flusso di errore standard

- In condizioni normali (cioè senza redirezione) lo **standard error** finisce sullo schermo insieme allo **standard output**

```
System.out.printl("Tutto ok");
System.err.println("Errore!");
```

- In genere il sistema operativo consente di effettuare la redirezione dello standard error in modo **indipendente** dallo standard output
  - in Windows è possibile ridirigere i due flussi verso due file distinti

```
C:\> java HelloTester > out.txt 2> err.txt
```

  - in Unix è (solitamente) possibile ridirigere i due flussi verso due file distinti (la sintassi dipende dalla shell)