Politecnico
di Milano

I packages

Package

- Scopo: organizzare il codice
- Clausola **import** per accedere alle classi di un package:
  - Utilizzo della classe **Vector** che si trova nel package java.util
    - **java.util.Vector mioVettore = new java.util.Vector();**
  - oppure:
    - **import java.util.Vector;**
    - **Vector mioVettore = new Vector();**
  - Accedere a tutte le classi del package (import multiplo):
    - **import java.util.*;**

Definizione di un package

- Definire come prima istruzione di una unità di compilazione l'appartenenza al package:

```
package mialibreria;
public class MiaClasse {
…
}

package polimi.mialibreria;
public class MiaClasse {
…
}
```

- Classi in package diversi possono avere lo stesso nome
  - Se vengono utilizzate nella stessa unità di compilazione è necessario utilizzare il loro nome completo (comprensivo del nome del package)

Memorizzazione dei package su file system ed esecuzione

- Le classi appartenenti ad un package vengono memorizzate in una directory che ha il nome del package
  - C:\Documents and Settings\dinitto\My Documents\home\didattica\SE_TLC0203\materiale didattico\Esempi\miaLibreria
  - C:\Documents and Settings\dinitto\My Documents\home\didattica\SE_TLC0203\materiale didattico\Esempi\polimi\miaLibreria
- In fase di esecuzione è necessario specificare il nome del package
  - java miaLibreria.MiaClasse
  - java polimi.miaLibreria.MiaClasse

Java API Packages

- package java.applet
- package java.awt - Abstract Window Toolkit
- package java.beans
- package **java.io** - gestione stream input output
- package **java.lang** - tipi di dati, thread, eccez.
- package java.math
- package java.net - socket e URL
- package java.rmi - Remote Method Invocation
- package java.security
- package java.sql
- package **java.util** - hashtable, vector, string tokenizer
- java.io e java.lang sono package "sempre osservabili": sono automaticamente importati in ogni unità di compilazione che scriviamo

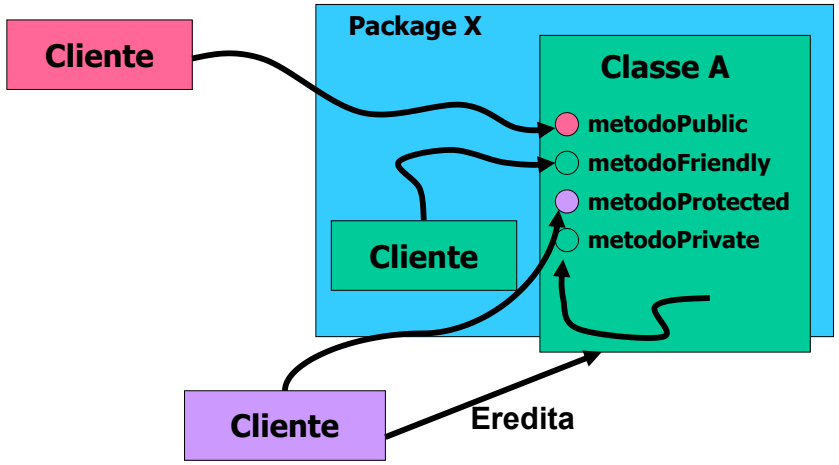
Convenzioni nei nomi dei package

- I nomi dei package devono essere univoci
- Le convenzioni sui nomi dei package hanno lo scopo di garantirne l'univocità
- Si suggerisce di formare il nome del package a partire dal domain name di appartenenza.
- Esempi:
  - Chi appartiene al dominio `sun.com` userà come prefisso `com.sun`.
  - Chi appartiene al dominio `polimi.it` userà come prefisso `it.polimi`.
  - La rimanente parte del nome verrà costruita utilizzando convenzioni di nomi interne (per esempio, si utilizzerà il nome del progetto, oppure della divisione/dipartimento, …).
  - `com.sun.pippo` e `it.polimi.pippo` vengono riconosciuti come package diversi.
  - `com.sun.pippo.Pluto` e `it.polimi.pippo.Pluto` sono classi distinte.

Package unnamed

- Un'unità di compilazione che non ha una dichiarazione di package è parte di un package anonimo (unnamed)
- Le implementazioni della piattaforma Java devono supportare almeno un package anonimo
- L'implementazione decide quali unità di compilazione appartengono ad uno stesso package anonimo
- Tipicamente, si associa un package anonimo alla directory corrente
  - Tutte le unità di compilazione che si trovano in quella directory appartengono allo stesso package anonimo

Specificatori di accesso



Politecnico
di Milano

La gestione delle eccezioni

Operazioni parziali

- Molte procedure sono parziali, cioè hanno un comportamento specificato solo per un sottoinsieme del dominio degli argomenti
- Per esempio

```
/** REQUIRES: n > 0
    EFFECTS: restituisce il fattoriale di n */
public static int fact (int n)
```

- La clausola REQUIRES restringe il dominio
- Ciò rende l'operazione poco sicura:
  - Che succede se i parametri non rispettano il vincolo?
  - Esempio: la procedura per n<0 calcola un valore scorretto che poi è usato da altre parti del programma: l'errore si propaga a tutto il programma, rovinando i risultati, i dati memorizzati, ecc.

Procedure "parziali" e "robustezza"

- Le procedure parziali compromettono la "robustezza" dei programmi
  - un programma è "robusto" se, anche in presenza di errori o situazioni impreviste, ha un comportamento ragionevole (o, per lo meno, ben definito)
  - per le procedure parziali il comportamento al di fuori delle precondizioni è semplicemente non definito dalla specifica
  - se una procedura non è definita per alcuni valori (in quanto "inattesi"), si ottengono errori run-time o, peggio, comportamenti imprevedibili quando tali valori sono passati come paramentri
- Per ottenere programmi robusti, le procedure devono essere "totali"!!!

Gestione di errori e situazioni eccezionali

- Una procedura deve poter segnalare l'impossibilità di produrre un risultato significativo o la propria terminazione scorretta
- Gestione tradizionale a fronte di errori e situazioni eccezionali: la procedura può
  - terminare il programma
  - restituire un valore convenzionale che rappresenti l'errore
  - restituire un valore corretto e portare l'oggetto o l'intero programma in uno stato "scorretto" (es. usare un attributo ERROR)
  - richiamare una funzione predefinita per la gestione degli errori

Problemi (i/ii)

- Terminare il programma
  - è spesso una soluzione troppo drastica
  - a rigore è una scelta che spetta al chiamante e non al chiamato
- Uso di valori di ritorno convenzionali
  - può non essere fattibile
    - perché la procedura non ha un valore di ritorno
    - o perché qualsiasi valore di ritorno è ammissibile
  - in generale dà poche informazioni riguardo l'errore incontrato
  - condiziona il chiamante
    - non posso scrivere espressioni del tipo z = x + fact(y)
    - devo scrivere: int r = fact(y); if (r>0) z = x + r; else ...

Problemi (ii/ii)

- Portare il programma in uno stato scorretto
  - la procedura chiamante potrebbe non accorgersi che il programma è stato portato in uno stato "scorretto"
- Uso di una funzione predefinita per la gestione degli errori
  - diminuisce la leggibilità del programma
  - centralizza la gestione degli errori (che spetterebbe al chiamante)

Soluzione: gestione esplicita delle eccezioni

- Una procedura può terminare normalmente (con un risultato valido) o sollevare un'eccezione
- Le eccezioni vengono segnalate al chiamante che può gestirle
- Le eccezioni hanno un tipo e dei dati associati che danno indicazioni sul problema incontrato
- Le eccezioni possono essere definite dall'utente (personalizzazione)

Uso delle Eccezioni in Java

Interfacce delle procedure

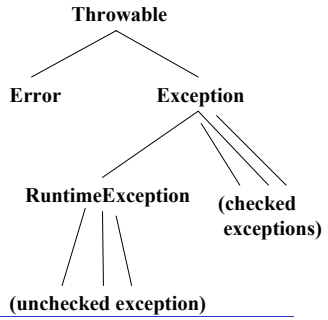
- Il fatto che una procedura possa terminare sollevando un'eccezione è dichiarato nella sua interfaccia
- In Java questo è possibile per mezzo della clausola "throws"

```
public static int fact (int n) throws NonPositiveException

public static int search (int[] a, int x)
    throws NullPointerException, NotFoundException
```

Tipi di eccezioni

- Un'eccezione è un sottotipo del tipo **Throwable**
- Esistono due tipi di eccezioni:
  - eccezioni checked
    - sottotipo di **Exception**
  - eccezioni unchecked
    - sottotipo di **RuntimeException**



Sollevare le eccezioni

- Per sollevare esplicitamente un'eccezione, si usa il comando **throw** seguito da un oggetto del tipo dell'eccezione
- Semantica (informale) del comando throw
  - termina l'esecuzione del blocco di codice che lo contiene, generando un'eccezione del tipo specificato

```
public static int fact (int n) throws NonPositiveException{
        if (n<0) throw new NonPositiveException();
        else
          if (n==0 || n==1) return 1;
          else return (n * fact(n-1));
}
```


Gestire le eccezioni

- Un'eccezione può essere catturata e gestita attraverso il costrutto:

```
try {<blocco>}
catch(ClasseEccezione e) {<codice di gestione>}
```

```
try {
      x = x + fact(y);
}
catch (NonPositiveException e) {
      /* codice per gestire l'eccezione.
      Qui è possibile usare l'oggetto e */
}
```

- Più clausole catch possono seguire lo stesso blocco try
- Un ramo *catch(Ex e)* può gestire un'eccezione di tipo *T* se *T* è di tipo *Ex* o *T* è un sottotipo di *Ex*

Clausole catch

```
try {
    //Codice che potrebbe generare un'eccezione
} catch(Type1 id1) {
    //Gestisce le eccezioni di Type1
} catch(Type2 id2) {
    //Gestisce le eccezioni di Type2
} catch(Type3 id3) {
    //Gestisce le eccezioni di Type3
}
```

- Ciascuna clausola catch è un piccolo metodo che prende esattamente un parametro (l'eccezione sollevata).
- Se un'eccezione è sollevata nel blocco try, il sistema di gestione delle eccezioni esegue il primo blocco catch con un argomento dello stesso tipo o di un supertipo dell'eccezione sollevata.

Esempio: I/O

Lettura da Input in Java richiede la gestione eccezioni

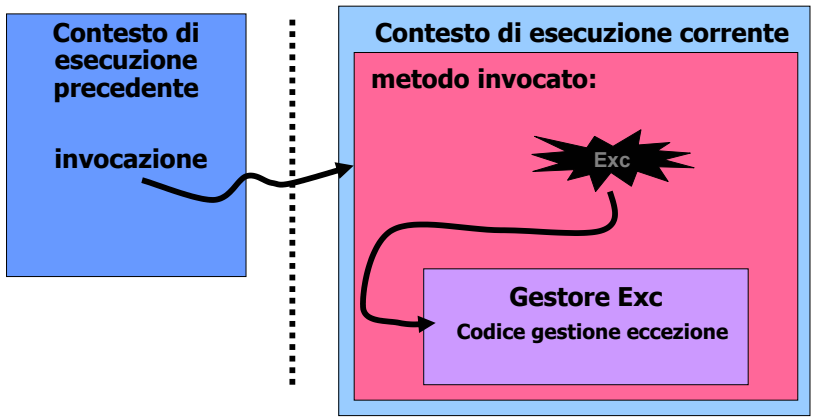
```
public static void main(String[] args) {
 //def. stream di ingresso
 BufferedReader stdin = new
     BufferedReader(new InputStreamReader(System.in));
 try{
     System.out.print("Enter a line:");
     System.out.println(stdin.readLine());
 } catch(IOException e) {
     System.out.println("IO Exception");
 }
}
```

- la readLine() può generare eccezione ("IOException")
- se si verifica eccezione facendo readLine(), si salta al catch. Se eccezione è del tipo `IOException` si esegue il codice, altrimenti il programma termina. Eseguito il catch si prosegue con l'istruzione successiva al catch (non c'e' ripristino).

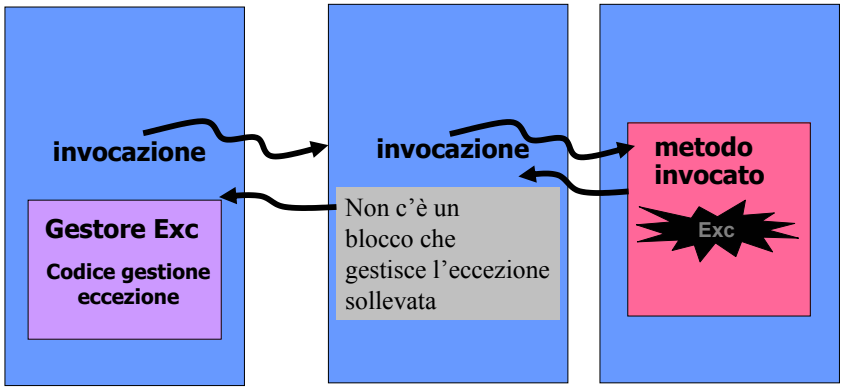
Propagazione delle eccezioni

- Se invocando una procedura si verifica un'eccezione:
  - si termina l'esecuzione del blocco di codice in cui si è verificata l'eccezione e...
  - se il blocco di codice corrente è un blocco try/catch, si passa il controllo al primo dei rami catch in grado di gestire l'eccezione, altrimenti...
  - si risalgono eventuali blocchi di codice più esterni fino a trovare un blocco try/catch che contenga un ramo catch che sia in grado di gestire l'eccezione, altrimenti...
  - l'eccezione viene propagata nel contesto del chiamante
  - la propagazione continua fino a che
    - si trova un blocco try/catch che gestisce l'eccezione
    - il programma termina
- Quando un'eccezione viene gestita da un blocco try catch, successivamente l'esecuzione del programma continua dal comando successivo al blocco stesso

Gestione delle eccezioni nel contesto corrente



Propagare le eccezioni



Uso delle Eccezioni in Java

Eccezioni checked e unchecked

- Eccezioni checked
  - devono essere dichiarate dalle procedure che possono sollevarle (altrimenti si ha un errore a compile-time)
  - quando una procedura *P1* invoca un'altra procedura *P2* che può sollevare un'eccezione di tipo *Ex* (checked), una delle due seguenti affermazioni deve essere vera:
    - l'invocazione di *P2* in *P1* avviene internamente ad un blocco try/catch che gestisce eccezioni di tipo *Ex* (quindi, *P1* gestisce l'eventuale eccezione)
    - il tipo *Ex* (o un suo sopra-tipo) fa parte delle eccezioni dichiarate nella clausola *throws* della procedura *P1* (quindi, *P1* propaga l'eventuale eccezione)
- Eccezioni unchecked
  - possono propagarsi senza essere dichiarate in nessuna signature di procedura e senza essere gestite da nessun blocco try/catch

Esempi di eccezioni unchecked

- ArithmeticException: segnala situazioni quali la divisione per zero
- ArrayStoreException: segnala il caso in cui si tenta di inserire in un array un valore non compatibile con il tipo degli elementi dell'array
- IndexOutOfBoundsException: un indice di un array, string, vector, … oppure un subrange si trova al di fuori del range di valori atteso
- ClassCastException: segnala il caso in cui è stato fatto un cast non compatibile con il tipo di oggetto in questione
- NullPointerException: segnala il caso in cui si è tentato di utilizzare un riferimento nullo al posto di un riferimento ad oggetto

Esempio con eccezioni unchecked

```
public static float frazione(int numeratore int denominatore) {
  if (denominatore==0)
      throw new ArithmeticException("Denominatore nullo");
  else return numeratore/denominatore;
}
```

- Se `denominatore` e' nullo, viene generata un'eccezione (il programma torna al chiamante fino a trovare un blocco `try/catch` che contenga un ramo `catch` che "soddisfi" l'eccezione, ecc.)

Uso delle Eccezioni in Java

Definizione di nuove eccezioni

- Gli oggetti di una qualunque classe *T* definita dall'utente possono essere usati per sollevare e propagare eccezioni, a condizione che *T* sia definita come sotto-classe di Exception (o RuntimeException)
- Definizione

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

- uso: `throw new NewKindOfException("problema!!!")`
- gestione:

```
try{....}catch(NewKindOfException ecc){
        System.out.println(ecc); }
```

Esempio (1)

```
public class IllegalDataException extends Exception {
    public IllegalDataException() {super();}
    public IllegalDataException(String s) {super(s);}
}
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 IllegalDataException {
    if(!corretta(g,m,a)) throw new
       IllegalDataException(g+"/"+m+"/"+a);
    giorno=g; mese=m; anno=a;
  }
}
```

Esempio (2)

```
public class ProvaEcc {
  public static void main(String[] args) {
    int g,m,a;
    ... // leggi g, m, a
    try {d=new Data(g,m,a);}
    catch(IllegalDataException e) {
      System.out.println("Inserita una data illegale: “+e);
      System.exit(-1);
    }
  }
}
```

Uso delle Eccezioni in Java

Il ramo finally

- Un blocco try/catch può avere un ramo `finally` in aggiunta a uno o più rami `catch`
- Il ramo `finally` è comunque eseguito
  - sia che all'interno del blocco `try` non vengano sollevate eccezioni
  - sia che all'interno del ramo `try` vengano sollevate eccezioni. In tal caso il ramo `finally` viene eseguito dopo il ramo `catch` che gestisce l'eccezione

```
class Prova {
static void read(String fileName) {
    try{
          FileInputStream f=new FileInputStream(fileName);
          ... // use f
    } catch(Exception ex) {...}
  finally {f.close();}
}
}
```

Programmare con le eccezioni: reflecting

- La gestione dell'eccezione comporta la propagazione di una nuova eccezione (dello stesso tipo o di tipo diverso)

```
public static int min (int[] a)
  throws NullPointerException, EmptyException{
          int m;
          try { m=a[0]; }
          catch (IndexOutOfBoundsException e){
                  throw new EmptyException(“Arrays.min”);
          }
          for (int i; i<a.length; i++)
                  if (a[i] < m) m=a[i];
          return m;
  }
```

Programmare con le eccezioni: masking

- Dopo la gestione dell'eccezione, l'esecuzione continua seguendo il normale flusso del programma

```
public static boolean sorted (int[] a)
  throws NullPointerException {
        int prev;
        try { prev=a[0]; }
        catch (IndexOutOfBoundsException e){ return true; }
        for (int i=1; i<a.length; i++) {
              if (prev <= a[i]) prev=a[i];
              else return false;
        }
        return true;
  }
```

Progettare le eccezioni

- Sollevare eccezioni per:
  - gestire i casi in cui le precondizioni di una procedura non sono soddisfatte dal chiamante
  - gestire la codifica di informazioni particolari nei risultati delle procedure
- Le eccezioni unchecked dovrebbero essere evitate il più possibile. Il loro uso dovrebbe essere limitato ai casi in cui
  - c'è un modo conveniente e poco costoso di evitare l'eccezione (per gli array, le eccezioni di tipo OutOfBoundException possono essere evitate controllando in anticipo il valore dell'attributo *length* dell'array)
  - l'eccezione è usata solo in un contesto ristretto

Usare le eccezioni

- Dopo aver catturato le eccezioni:
  - Risolvere il problema e richiamare il metodo che ha generato l'eccezione
  - Sistemare le cose alla meglio e continuare l'esecuzione del programma senza tentare di richiamare il metodo
  - Calcolare un risultato alternativo a quello che il metodo avrebbe dovuto restituire
  - Fare tutto quello che è possibile nel contesto corrente e sollevare la stessa eccezione o un'altra eccezione verso il contesto chiamante
  - Terminare il programma

Consigli Utili

- aggiungere ai dati correlati con l'eccezione l'indicazione della procedura che l'ha sollevata (in modo da facilitare l'individuazione delle cause)

  public static int fact (int n) throws NotFoundException{
  ... throw new NotFoundException("fact"); ...}
- nel caso in cui la gestione di un'eccezione comporti un'ulteriore eccezione (reflecting), conservare le informazioni

  catch (NotFoundException e){
  throw new NewKindOfException("procedure.name" + e.toString()); }
- sebbene sia possibile scegliere liberamente i nomi delle nuove eccezioni definite, è buona convenzione farli terminare con la parola *Exception*

  NotFoundException piuttosto che NotFound
- è buona pratica prevedere un package contenente tutte le nuove eccezioni definite (migliora la struttura del progetto e facilità il riuso delle eccezioni)