Progettazione

"By Contract"

- **Documentazione di progetto**
  - Contratti per i metodi
- **Correttezza dell'implementazione**
  - invarianti per le classi
- **Verifica**
  - asserzioni
- **Eccezioni**

Progettazione *By Contract*

- **Contratti per i metodi**
  - formalizzazione del comportamento di ciascun metodo per controllare le interazioni con i clienti
- **Invarianti di classe**
  - Vincoli di buona formazione degli oggetti della classe, necessari per garantire la correttezza dei metodi
- **Asserzioni**
  - Controllo dinamico di contratti ed invarianti

Contratti per i metodi

- **La dichiarazione di un metodo specifica solo il tipo**
- **Un contratto specifica il comportamento;**
  - quale servizio il metodo implementa, sotto quali condizioni
- **Specifiche informali e/o incomplete**
  - Ambiguità di interpretazione
  - Contraddizione con altri contratti
- **Specifiche formali**
  - Precisione e univocità di interpretazione
  - Controlli a run-time su violazioni del contratto e/o errori di uso
  - Facilità di ragionamento su correttezza e uso dei metodi

Continua…

Contratti per i metodi

- **Esistono linguaggi che permettono la specifica completa del comportamento di componenti**
- **Nostro obiettivo**
  - Specifica formale di parte del comportamento
  - Scopo: controllare gli aspetti rilevanti del comportamento e dell'implementaizione
  - Verifica della specifica mediante asserzioni

Continua…

Contratti per i metodi

Precondizioni:

- condizioni che devono essere soddifatte al momento della chiamata
- vincoli per il *chiamante*, che deve assicurare che le condizioni siano vere

Postcondizioni:

- condizioni che devono essere soddifatte al momento del ritorno dalla chiamata
- vincoli per il *chiamato*, che deve dare garanzie al chiamante

Continua…

pre/post condizioni

- **Documentate con commenti in stile Javadoc**
  - due tag speciali

```
/**
    @pre precondizione
    @post postcondizione
*/
 public void metodo( ... ) { ... }
```

- **Non supportate da javadoc**
  - altri strumenti: JML, XjavaDoc, iContract, …

Continua…

pre/post condizioni

- **Espressioni booleane**
  - specificate in sintassi Java
- **Estensioni sintattiche utili alla specifica**
  - **`@result:`** il valore restituito da un metodo
  - ***`expr`*`@pre:`** il valore dell'espressione prima dell'esecuzione del metodo
  - **`@nochange:`** il metodo non modifica this
  - **`@forall:`*`dom`* `@`*`expr`*, **`@exists:`*`dom`* `@`*`expr`** espressioni quantificate (su un certo dominio)
  - **`==>, <=>:`** implicazione ed equivalenza logica

Continua…

pre/post condizioni

- **Specificate da una o più clausole `@pre` e `@post`**
  - in presenza di più clausole la condizione è definita dalla congiunzione di tutte le clausole
- **Specifiche non sempre esaustive**
  - la specifica delle pre/post condizioni integrata dalla documentazione standard
  - quando il commento iniziale esprime in modo preciso la semantica evitiamo di ripeterci nella specifica
- **Condizioni espresse sempre in termini dell'interfaccia pubblica**

Continua…

Esempio: MySequence<T>

```
interface MySequence<T> {

    // metodi accessors
    int size();
    boolean isEmpty();
    T element(int i);
    T head();
    . . .
    // metodi mutators
    void insert(T item, int i)
    void insertHead(T item);
    T remove(int i);
    T removeHead();
    . . .
}
```

size()

```
/**
 * Restituisce la dimensione della sequenza
 * @result = numero di elementi nella sequenza
 * @pre true
 * @post @nochange
 */
 public int size();
```

- **La precondizione `true` è sempre soddisfatta**
  - un metodo con questa precondizione può sempre essere invocato
- **La postcondizione indica solo l'assenza di side effects (il commento completa la specifica)**

isEmpty()

- **pre/post condizioni sono (devono essere) espresse in termini dell'interfaccia pubblica**
  - non avrebbe senso esporre nel contratto elementi dell'implementazione

```
/**
 * Restituisce true sse la sequenza è vuota
 *
 * @pre true
 * @result <=> size() == 0
 * @post @nochange
 */
 public boolean isEmpty();
```

element()

- **Precondizioni specificano vincoli**
  - sul valore degli argomenti di chiamata
  - sullo stato dell'oggetto al momento della chiamata

```
/**
 * Restituisce l'elemento in posizione i
 *
 * @pre 0 <= i && i < size()
 * @post @nochange
 */
 public T element(int i);
```

head()

- **Precondizioni specificano vincoli**
  - sul valore degli argomenti di chiamata
  - sullo stato dell'oggetto al momento della chiamata

```
/**
 * Restituisce il primo elemento
 *
 * @pre !isEmpty()
 * @result == element(0)
 * @post @nochange
 */
 public T head();
```

insert()

- **Postcondizioni specificano vincoli**
  - sul valore calcolato dal metodo
  - sullo stato dell'oggetto al termine della chiamata

```
/**
 * Inserisce un nuovo elemento alla posizione i
 *
 * @pre item != null && i >= 0 && i < size()
 * @post size() = size()@pre + 1
 * @post @forall k : [0..size()-1] @
 *       (k < i ==> element(k)@pre == element(k)) &&
 *       (k == i ==> item == element(k)) &&
 *       (k > i ==> element(k-1)@pre == element(k)
 */
 public void insert(T item, int i);
```

insertHead()

```
/**
 * Inserisce un elemento sulla testa della sequenza
 *
 * @pre item != null
 * @post size() = size()@pre + 1
 * @post item == element(0))
 * @post @forall k : [1..size()-1]
 *                  @ element(k-1)@pre == element(k)
 */
 public void insertHead(T item);
```

remove()

```
/**
 * Rimuove l'elemento alla posizione i
 *
 * @pre size() > 0
 * @pre i >= 0 && i < size()
 * @result == element(i)@pre
 * @post size() = size()@pre - 1
 * @post @forall k : [0..size()-1] @
 *       (k < i ==> element(k)@pre == element(k)) &&
 *       (k >= i ==> element(k+1)@pre == element(k))
 */
 public T remove(int i);
```

removeHead()

```
/**
 * Rimuove l'elemento in testa e lo restituisce
 *
 * @pre size() > 0
 * @post @result == element(0)@pre
 * @post size() = size()@pre - 1
 * @post @forall k : [0..size()-1]
 *                  @ element(k+1)@pre == element(k)
 */
 public T removeHead();
```

Domanda

- **Come completereste la specifica del metodo `deposit()` nella gerarchia `BankAccount`?**

```
/**
 * Deposita un importo sul conto
 *
 * @pre
 * @post
 */

 public void deposit(double amount) { . . . }
```

Risposta

```
/**
 * Deposita un importo sul conto
 *
 * @pre amount > 0
 * @post getBalance() = getBalance()@pre + amount
 */

 public void deposit(double amount) { . . . }
```

Domanda

- **Vi sembra corretta la seguente specifica del metodo `withdraw()`?**

```
/**
 * Preleva un importo sul conto
 *
 * @pre amount > 0
 * @post balance = balance@pre - amount
 */

 public void withdraw(double amount) { . . . }
```

Risposta

- No, le pre/post condizioni devono essere espresse sempre in termini dell'interfaccia pubblica mentre `balance` è una variabile privata della classe
- la versione corretta è espressa utilizzando il metodo `getBalance()` così come visto per il metodo `deposit()`.

Contratti, sottotipi e *method overriding*

- **A proposito di ereditarietà …**
  - la ridefinizione di un metodo della superclasse nella sottoclasse deve rispettare il tipo del metodo nella superclasse
- **Compatibilità di tipi necessaria per la correttezza del principio di sostituibilità:**
  - istanze di sottotipo possono essere assegnate a variabili di un supertipo
- **Necessaria, non sufficiente**
  - il comportamento del metodo nella sottoclasse deve essere compatibile con il metodo della superclasse

Continua…

Contratti, sottotipi e *method overriding*

- **Design by contract**
  - ciascun metodo di una classe deve rispettare il contratto del corrispondente metodo nella sua superclasse e/o nelle interfacce implementate dalla classe

Continua…

Contratti, sottotipi e *method overriding*

```
class B extends A { . . .   }
```

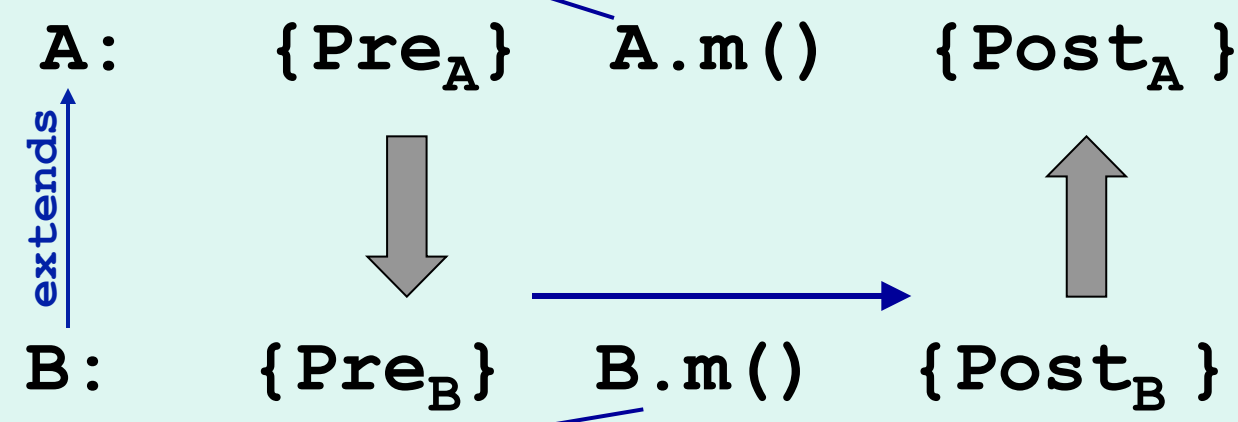
- **precondizione**
  - la precondizione di ciascun metodo di **`B`** deve essere implicata dalla precondizione del metodo corrispondente di **`A`**
- **postcondizione**
  - la postcondizione di ciascun metodo di **`B`** deve implicare la postcondizione del metodo corrispondente di **`A`**

Continua…

Contratti, sottotipi e *method overriding*

L'esecuzione di **B.m()** garantisce $\mathbf{Post_B}$ e dunque anche $\mathbf{Post_A}$ come atteso dal chiamante

selezione statica



dispatch dinamico

Se il chiamante garantisce $\mathbf{Pre_A}$, anche $\mathbf{Pre_B}$ è verificata come richiesto da **B.m()**

Continua…

Domanda

- **Delle due seguenti versioni di `withdraw()`, quale deve stare nella superclasse e quale nella sottoclasse secondo il principio di compatibilità?**

```
/**
 * @pre amount > 0
 * @post getBalance() = getBalance()@pre - amount
 */
 public void withdraw(double amount) { . . . }
```

```
/**
 * @pre amount > 0 && getBalance() >= amount
 * @post getBalance() = getBalance()@pre - amount
 */
 public void withdraw(double amount) { . . . }
```

Risposta

- **nella sottoclasse**

```
/**
 * @pre amount > 0
 * @post getBalance() = getBalance()@pre - amount
 */
 public void withdraw(double amount) { . . . }
```

- **nella superclasse**

```
/**
 * @pre amount > 0 && getBalance() >= amount
 * @post getBalance() = getBalance()@pre - amount
 */
 public void withdraw(double amount) { . . . }
```

Invarianti di classe

- **Lo stato di un oggetto si dice**
  - *transiente* durante l'esecuzione di un metodo invocato sull'oggetto
  - *stabile* se l'oggetto è stato inizializzato e nessuno dei metodi della classe è in esecuzione sull'oggetto stesso
- **Invarianti di classe**
  - condizione verificata su tutti gli oggetti della classe che si trovano in stato stabile

Continua…

Invariante di classe per `MySequence<T>`

- **Supponiamo che la rappresentazione della sequenza sia mediante una lista doppia con puntatori alla testa e coda della lista**

```
// classe interna
private static class Node<T> {
   T element;
   Node<T> next, prev;
}
// testa e coda della sequenza
private Node<T> head, tail;
// contatore degli elementi in sequenza
private int count;
```

Continua…

Invariante di classe per `MySequence<T>`

- **Vincoli di consistenza sulla rappresentazione della sequenza**
  - se la sequenza è vuota **`tail = head = null`**;
  - se la sequenza è non vuota, **`head`** punta al primo nodo, **`tail`** all'ultimo
  - **`count`** = numero di elementi in sequenza
  - per ogni nodo intermedio il **`next`** del precedente e il **`prev`** del seguente puntano al nodo stesso
  - il **`prev`** del primo nodo e il **`next`** dell'ultimo sono **`null`**

Continua…

Invariante di classe per `MySequence<T>`

```
protected boolean _invariant() {
      int n = 0;
      for (Node<T> p = head; p != null; p = p.next) {
            n++;
            if (p.prev != null) {
                  if (p.prev.next != p) return false;
            } else { // p è il primo
              if (head != p) return false;
            }
            if (p.next != null) {
                  if (p.next.prev != p) return false;
            } else { // p è l'ultimo
              if (tail != p) return false;
            }
       }
      if (n == 0 && (tail != null || head != null))
               return false;
      return n == count;
}
```

Continua…

Invariante di classe per `MySequence<T>`

- **L'invariante può (deve) essere documentato nella implementazione della classe**
- **utilizziamo una tag specifica**

```
public class LinkedSequence<T> implements Sequence<T>
{
    /**
     * @invariant _invariant()
     */
    protected boolean _invariant() { . . . }
   . . .
}
```

Invarianti di classe

- **A differenza delle pre/post condizioni, l'invariante esprime vincoli sulla rappresentazione interna delle classi**
- **Utilizzato per giudicare la correttezza dell'implementazione**

Continua…

Invarianti di classe

- **Devono essere garantiti dai costruttori**
  - l'invariante deve valere dopo l'inizializzazione
  - tutti i costruttori pubblici di una classe devono avere l'invariante di classe come post condizione
- **Devono essere preservati dai metodi pubblici**
  - l'invariante può essere assunto come precondizione del corpo di ciascun metodo pubblico,
  - deve essere trattato come una postcondizione da soddisfare al termine dell'esecuzione

Invarianti e correttezza dei metodi

Specifica astratta data in termini del contratto di m() in A

$$\{Pre_A\} \quad A.m() \quad \{Post_A\}$$

$$\{Pre_A \text{ AND } Inv_A\} \; Body_{A.m()} \{Post_A \text{ AND } Inv_A\}$$

Specifica concreta, verificata nell'implementazione
Inv_A è l'invariante di classe

Domanda

- **L'invariante della classe `BankAccount` è specificato in modo completo?**

```
class BankAccount {
   /** @pre import > 0
    *  @post getBalance() = getBalance()@pre - import */
    public void withdraw(double import) { balance =- import;}

   /** @pre import > 0
    *  @post getBalance() = getBalance()@pre + import */
    public void deposit(double import) { balance =+ import; }

   /** @pre true
    *  @post @nochange  */
    public double getBalance() { return balance; }

   /** saldo e fido associati al conto
    *  @invariant balance >= credit  */
    private double balance, credit;

 }
```

Risposta

- **Nop, non definisce la relazione tra campi ed i corrispondenti metodi *get*. Correggiamo…**

```
class BankAccount {
   /** @pre import > 0
    *  @post getBalance() = getBalance()@pre - import */
    public void withdraw(double import) { balance =- import;}

   /** @pre import > 0
    *  @post getBalance() = getBalance()@pre + import */
    public void deposit(double import) { balance =+ import; }

   /** @pre true
    *  @post @nochange  */
    public double getBalance() { return balance; }

   /** saldo e fido associati al conto
    *  @invariant balance >= credit
           && balance == getBalance() && credit == getCredit() */
    private double balance, credit;
 }
```

Domanda

- **Ora la specifica è corretta?**

```
class BankAccount {
   /** @pre import > 0
    *  @post getBalance() = getBalance()@pre - import */
    public void withdraw(double import) { balance =- import;}

   /** @pre import > 0
    *  @post getBalance() = getBalance()@pre + import */
    public void deposit(double import) { balance =+ import; }

   /** @pre true
    *  @post @nochange  */
    public double getBalance() { return balance; }

   /** saldo e fido associati al conto
    *  @invariant balance >= credit
           && balance == getBalance() && credit == getCredit() */
    private double balance, credit;
 }
```

Risposta

- Nop, la postcondizione di `widthdraw()` non implica l'invariante. Correggiamo…

```
class BankAccount {
   /** @pre import > 0 0 && getBalance() - amount >= getCredit()
    *  @post getBalance() = getBalance()@pre - amount */
    public void withdraw(double amount) { balance =- amount;}

   /** @pre import > 0
    *  @post getBalance() = getBalance()@pre + import */
    public void deposit(double amount) { balance =+ amount; }

   /** @pre true
    *  @post @nochange  */
    public double getBalance() { return balance; }

   /** saldo e fido associati al conto
    *  @invariant balance >= credit
           && balance == getBalance() && credit == getCredit() */
    private double balance, credit;
 }
```

Invarianti ed ereditarietà

- Siano date due classi `Circle` ed `Ellipse`
- E' corretto immaginare di stabilire la seguente relazione?
  - `Circle <: Ellipse`
- Dipende …

Cerchi ed Ellissi

```
class Ellipse {
   private double xradius;
   private double yradius;

   // @pre: xr >=0 && yr >=0
   public Ellipse(double xr, double yr)
   { xradius = xr; yradius = yr; }

   // @pre: dx >=0 && dx >=0
   public void squash(double dx, double dy)
   { xradius += dx/100 * xradius; yradius += dy/100 *yradius; }

   public double xradius() { return xradius; }

   public double yradius() { return yradius; }

   protected boolean _invariant()
   {
       return xradius() == xradius && yradius() == yradius;
   }
}
```

Cerchi ed Ellissi

```
class Circle extends Ellipse {

   // pre: radius >=0
   public Circle(double radius)
   { super(radius, radius); }

   public radius() { return xradius(); }

   protected boolean _invariant()
   {
      return super._invariant() && xradius() == yradius()
   }
 }
```

- Il problema è che, evidentemente, l'invariante inteso nella classe `Circle` non è tale, perchè non mantenuto dal metodo `squash()` ereditato da `Ellipse`

Cerchi ed Ellissi

- **Possiamo rettificare ridefinendo il metodo `squash()`?**

```
class Circle extends Ellipse {
        . . .
    // @pre: dx >=0 && dx == dy
    public void squash(double dx, double dy)
    {
          super.squash(dx, dx);
    }
      . . .
}
```

- **Uh: la precondizione nella superclasse non implica quella della sottoclasse . . .**

Cerchi ed Ellissi

- **E' corretto allora immaginare di stabilire la seguente relazione?**
  - `Ellipse <: Circle`
- **Uhm …**

Cerchi ed Ellissi

```
class Circle
{
   private double radius;
   // @pre: r >=0
   public Circle (double r) { radius = r; }
   public double radius()   { return radius; }
   protected boolean _invariant() { return radius() == radius; }
}

class Ellipse extends Circle
{
   // che senso ha qui, il metodo radius()?
   // in che modo potremmo ragionevolmente ridefinirlo?
   . . .
}
```

Cerchi ed Ellissi

- **Possiamo modificare la specifica dei metodi in modo da rendere correntta la relazione**
  - `Circle <: Ellipse`
- **Come? …**

Cerchi ed Ellissi

```
class Ellipse {
   private double xradius;
   private double yradius;

   // @pre: xr >=0 && yr >=0
   public Ellipse(double xr, double yr)
   { xradius = xr; yradius = yr; }

   // @pre: dx >=0 && dx >=0
   // @post: nochange
   public Ellipse squash(double dx, double dy)
   {
     return new Ellipse(xradius + dx/100 * xradius
                        yradius + dy/100 * yradius);
   }
}
```

- Ora il metodo `squash()` può essere ereditato da `Circle` senza conseguenze indesiderate

Invarianti e Contratti in JAVA

Asserzioni

- **Una asserzione è una affermazione che permette di testare le assunzioni riguardo determinati punti del programma**
- **Ogni asserzione contiene una espressione booleana che si assume essere verificata**
- **La verifica delle asserzioni permette di effettuare dinamicamente controlli sulla correttezza del codice**

Continua…

Asserzioni: sintassi

assert *Expression*;

Esempio:

```
assert  i >= 0 && i < size();
```

Scopo:

Verificare se una condizione è soddisfatta. Se le asserzioni sono abilitate e la condizione è falsa lancia un errore di asserzione. Altrimenti non ha effetto.

Continua…

Asserzioni: sintassi

assert *Expression1 : Expression2*;

Esempio:
assert i >= 0 && i < size() : “indice fuori range”

Scopo:
Come nel caso precedente, ma utilizza la seconda espressione per documentare l’errore.
Se le asserzioni sono abilitate e *Expression1* è falsa valuta *Expression2* e passa il risultato insieme all’errore di asserzione.
Altrimenti non ha effetto.

Asserzioni *Howto's*

- **Compilazione**

  ```
  javac -source 1.5 <prog>.java
  ```

- **Esecuzione**
  - abilitazione/disabilitazione selettiva di eccezioni

    ```
    java –ea [:<package>|:<classe>] <prog>
         –da [:<package>|:<classe>] <prog>
    ```

  - abilitazione/disabilitazione di asserzioni di sistema

    ```
    java –esa
    java –dsa
    ```

Continua…

Asserzioni e *Unit Testing*

- **Le asserzioni sono molto efficaci per la verifica della corretta implementazione di una classe**
- **Derivate da:**
  - postcondizioni di metodi (pubblici e privati)
  - invarianti di classe
  - precondizioni di metodi privati

Asserzione di postcondizioni

```
/**
 * Restituisce il primo elemento della sequenza
 *
 * @pre !isEmpty()
 * @post @result == element(0)
 */
  public T head() {
        T result = (head != null ? head.item : null);
        assert result == element(0);
        return result;
 }
```

Asserzione di invarianti di classe

```
/**
 * Inserisce un nuovo elemento alla posizione i
 *
 * @pre . . .
 * @post size() = size()@pre + 1
 * @post . . .
 */
 public void insert(T item, int i) {
      assert _invariant();
      int size_pre = size();
       // ... codice di inserimento
      int size_post = size();
      assert size_post == size_pre + 1;
      assert _invariant();
}
```

Altri tipi di asserzioni

- **Per invarianti interni**
  - Dove tradizionalmente utilizzeremmo commenti …

```
if (i % 3 == 0) { . . . }
else if (i % 3 == 1) { . . . }
else { // a questo punto  i % 3 == 2
   . . .
}
```

  - … è più efficace utilizzare asserzioni

```
if (i % 3 == 0) { . . . }
else if (i % 3 == 1) { . . . }
else { assert i % 3 == 2
   . . .
}
```

Continua…

Altri tipi di asserzioni

- **Invarianti del flusso di controllo**
  - nei casi in cui vogliamo segnalare che un certo punto del programma non dovrebbe mai essere raggiunto;
  - possiamo asserire una costante sempre falsa

```
void m() {
  for (. . . ) {
     if (...)
     return;
  }
  assert false; // qui non ci dovremmo mai arrivare
}
```

Quando *non* usare asserzioni

- **Per verificare precondizioni di metodi pubblici**
  - essendo il metodo pubblico, non abbiamo controllo sul codice che invoca il metodo

```
/**
 * @pre !isEmpty()
 * @post @result == element(0)
 */
  public T head() {
        assert !isEmpty(); // brutta idea
        T result = (head != null ? head.item : null);
        assert result == element(0);
        return result;
 }
```

- **Meglio usare eccezioni in questi casi**

Continua…

Quando *non* usare asserzioni

- **Con espressioni che coinvolgono *side effects***
  - Esempio: vogliamo rimuovere tutti gli elementi **`null`** di una lista **`els`** e verificare che effettivamente la lista conteneva almeno un elemento **`null`**

```
// questa asserzione è scorretta _non_ usare
assert els.remove(null);
```

```
// questo è il modo giusto di costruire l'asserzione
boolean nullRemoved = els.remove(null);
assert nullRemoved;
```

Continua…

Defensive Programming

- **Pratica che mira ad evitare un utilizzo scorretto dei metodi di una classe**
- **Verifica delle precondizioni. Come?**
- **Dipende dalla situazione …**
- **Due situazioni:**
  - Abbiamo controllo sul chiamante (metodo privato)
    - asseriamo la precondizione
  - Evento fuori dal nostro controllo
    - Lanciamo una *eccezione*

Eccezioni

- **Eccezioni per codificare errori**
  - lancio di eccezioni
- **Eccezioni**
  - user defined
  - checked e unchecked
- **Gestione di eccezioni**
  - cattura di eccezioni

Gestione degli errori

- **Approccio tradizionale:**
  - codifica errori mediante codici
  - restituisci il codice dell'errore che si verifica
- **Problemi**
  - il chiamante dimentica di controllare i codici di errore: la notifica può sfuggire
  - il chiamante può non essere in grado di gestire l'errore per mancanza di informazione
  - l'errore deve essere propagato
  - il codice diventa illeggibile …

Continua…

Gestione degli errori

- **Propagazione degli errori:**
  - non è sufficiente gestire i casi di “successo”

```
x.doSomething()
```

  - al contrario: dobbiamo sempre strutturare il codice in funzione delle possibili situazioni di errore

```
if (!x.doSomething()) return false;
```

  - … faticoso e poco leggibile

Gestione di errori con eccezioni

- **Lancio**
  - Andare in errore corrisponde a lanciare (*throw*) una eccezione, che segnala il verificarsi dell'errore
- **Cattura**
  - Quando una eccezione viene lanciata, un altro pezzo di codice, detto *exception handler*, può catturare (*catch*) l'eccezione
- **Gestione**
  - trattamento dell'eccezione nel tentativo di ripristinare uno stato corretto per riprendere la computazione

Continua…

Lancio di eccezioni

- Le eccezioni sono oggetti, con tipi associati

```
IllegalArgumentException exn;
```

- Devono prima essere costruite

```
exn = new IllegalArgumentException("Bad Argument");
```

- Poi possono essere lanciate

```
throw exn;
```

Continua…

Lancio di eccezioni

- **Non è necessario separare le tre operazioni**

```
throw new IllegalArgumentException("Bad argument");
```

- **Lanciare una eccezione in un metodo causa l'interruzione immediata dell'esecuzione del metodo**
  - L'esecuzione prosegue con un gestore di eccezioni (se è stato programmato) o causa un errore

Lancio di eccezioni: sintassi

```
throw exceptionObject;
```

Esempio:

```
throw new IllegalArgumentException();
```

Scopo:

Lanciare una eccezione e trasferire il controllo ad un gestore per il corrispondente tipo di eccezione

Cattura di eccezioni

- **Catturiamo e trattiamo le eccezioni mediante la definizione di un *exception handler***
- **Costrutto `try/catch`**
  - blocco **`try`** include codice che può lanciare una eccezione (direttamente o invocando metodi che a loro volta lanciano eccezioni)
  - clausola **`catch`** contiene il gestore per un dato tipo di eccezione

Continua…

Cattura di eccezioni

```
try
{
   String filename = . . .;
   File f = new File(filename);
   Scanner in = new Scanner(f);
   String input = in.next();
   int value = Integer.parseInt(input);
   . . .
}
catch (IOException exception)
{
   exception.printStackTrace();
}
catch (NumberFormatException exception)
{
   System.out.println("Input was not a number");
}
```

Cattura di eccezioni

- **Flusso di esecuzione:**
  - Esegui il codice nel blocco **`try`**
  - Se non si verificano eccezioni, i gestori vengono ignorati le clausole **`catch`** vengono ignorate
  - Se si verifica una eccezione di uno dei tipi dichiarati, l'esecuzione riprende dal primo gestore compatibile con il tipo dell'eccezione sollevata
  - Se nessuno dei gestori è compatibile con il tipo dell'eccezione lanciata, l'eccezione viene rilanciata automanticamente
    - finchè non viene catturata da un altro gestore
    - oppure arriva al **`main`**

Continua…

Cattura di eccezioni

- **`catch (IOException e) block`**
  - **`e`** viene legata all'oggetto di tipo eccezione lanciato
  - può essere utilizzato in **`block`** per programmare la gestione dell'eccezione
  - esempio:

```
catch (IOException e)
   { e.printStackTrace() }
```

    restituisce lo stack di chiamate a partire dal metodo in cui l'eccezione è stata lanciata

Continua…

Cattura di eccezioni

- **I blocchi `try-catch` possono definire un qualunque numero di gestori purchè catturino tipi diversi di eccezioni**
- **Gestori esaminati in ordine testuale**
  - ordinare da sottotipo a supertipo

Continua…

Blocchi `try-catch`: sintassi

```
try
{
    statement
    statement
    . . .
}
catch (ExceptionClass exceptionObject)
{
    statement
    statement
    . . .
}
catch (ExceptionClass exceptionObject)
{
    statement
    statement
    . . .
}
. . .
```

Domanda

- **Quale è l'output di questo blocco?**

```
try
{
 // codice che lascia sfuggire NumberFormatException
}
catch (ArrayIndexOutOfBoundsException e)
{
   System.out.println("Io non centro");
}
catch (NumberFormatException e)
{
   System.out.println("La prendo io");
}
```

Continua…

Risposta

- **Gestori che non catturano l'eccezione lanciata sono ignorati**
  - output: **`"la prendo io"`**

Continua…

Domanda

- **Quale è l'output di questo blocco?**

```
try
{
  // codice che lascia sfuggire NumberFormatException
}
catch (ArrayIndexOutOfBoundsException e)
{
   System.out.println("Non è per me");
}
catch (ClassCastException e)
{
   System.out.println("Neance per me");
}
```

Continua…

Risposta

- **se nessun gestore è compatibile con il tipo dell'eccezione lanciata, questa**
  - output

```
Exception in thread "main"
 java.lang.NumberFormatException
```

Continua…

Domanda

- **Quale è l'output di questo blocco?**

```
try
{
  // codice che lascia sfuggire NumberFormatException
}
catch (ArrayIndexOutOfBoundsException e)
{
   System.out.println("Non per me");
}
catch (Exception e)
{
   System.out.println("Questa è per me");
}
```

Continua…

Risposta

- **compatibilita di tipi nei gestori basata su sottotipo**
  - output: **“questa è per me”**

Domanda

- **Quale è l'output di questo blocco?**

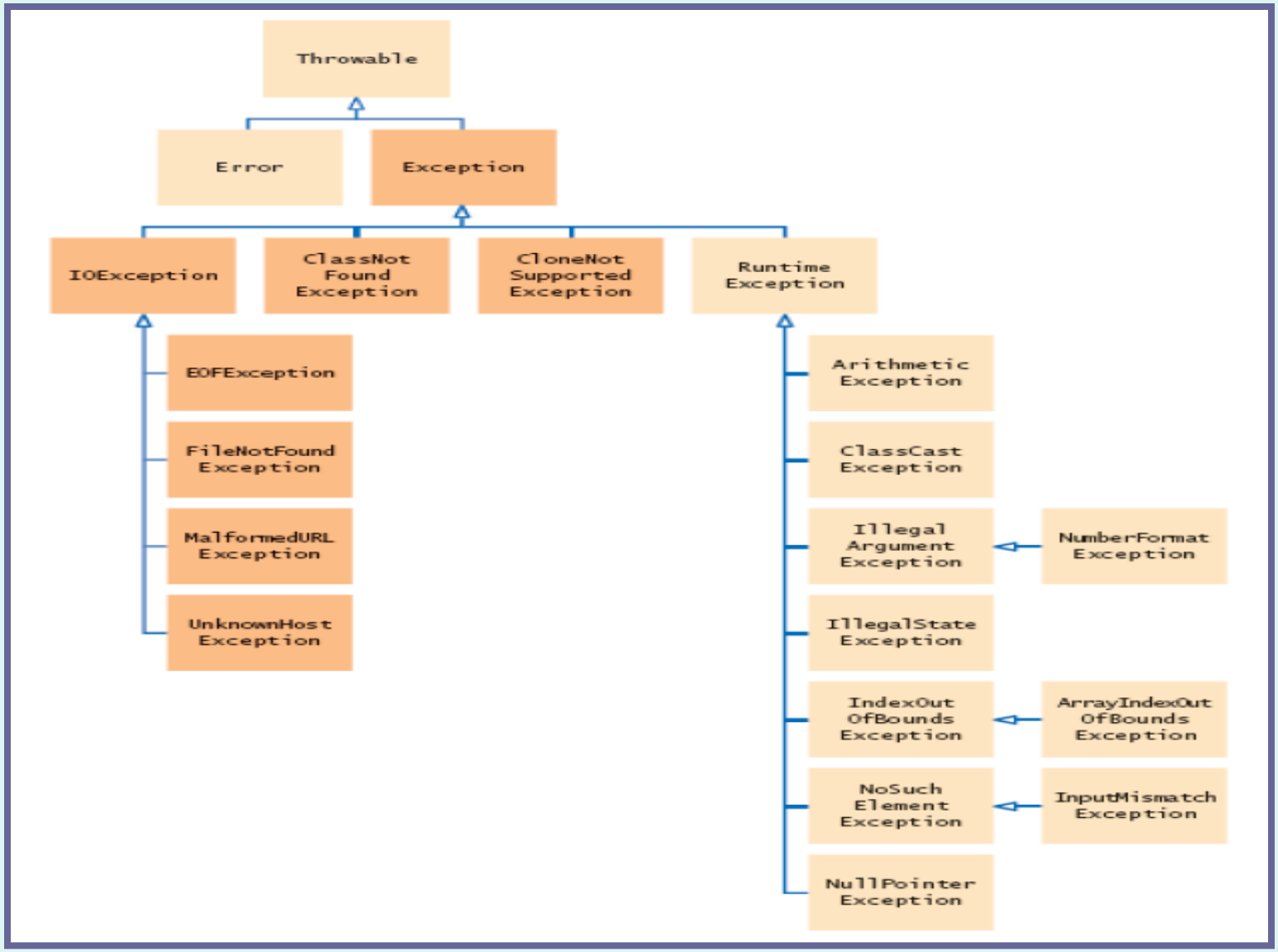
```
try
{
 // codice che lascia sfuggire NumberFormatException
}
catch (NumberFormatException e)
{
  throw new NumberFormatException()
}
catch (Exception e)
{
  System.out.println("Ooops!");
}
```

Risposta

- **gestori catturano solo eccezioni lanciate nel corrispondente blocco `try`:**
  - output

```
Exception in thread "main"
 java.lang.NumberFormatException
```

Tipi di eccezioni



Eccezioni *checked* e *unchecked*

- **Eccezioni *checked* (controllate)**
  - Derivate dalla classe **`Exception`**
  - Codificano situazioni anomale legate alla logica applicazione, o errori derivanti da condizioni esterne
- **Eccezioni *unchecked* (non controllate)**
  - Derivate dalle classi **`RuntimeException`** o **`Error`**
  - Codificano errori di programmazione
- **Predefinite e/o definite da utente**
- **Diverse solo per il compilatore**
  - i meccanismi computazionali sono identici

Eccezioni unchecked

- **`RuntimeException`: errori "locali"**

```
NumberFormatException
IllegalArgumentException
NullPointerException
```

- **`Error`: errori della JVM**

```
OutOfMemoryError
```

- **Non è necessario gestirle esplicitamente**
- **Possono essere definite da utente**
  - ma è una pratica sconsigliata …

Eccezioni checked

- **Devono essere gestite esplicitamente dal programma**
- **Due possibilità**
  - catturare l'eccezione
  - documentare che l'eccezione è uno dei possibili risultati dell'esecuzione del metodo

```
public void read(String filename)
                        throws FileNotFoundException
{
   File f = new File (filename);
   Scanner in = new Scanner(f);
   . . .
}
```

Continua…

Documentazione di eccezioni

- **Metodi devono dichiarare tutte le eccezioni checked che possono lanciare (e non catturano)**

```
public void read(String filename)
       throws IOException, ClassNotFoundException
```

- **Gerarchia dei tipi eccezione semplifica la documentazione**
  - Se un metodo può lanciare **`IOException`** e **`FileNotFoundException`** è sufficiente dichiarare **`IOException`** (il supertipo)

Metodi e clausole `throws`

```
accessSpecifier returnType
      methodName(parameterType parameterName, . . .)
            throws ExceptionClass, ExceptionClass, . . .
```

Esempio:

```
public void read(BufferedReader in) throws IOException
```

Scopo:

Indicare le eccezioni checked che un metodo può lanciare

Eccezioni checked e costruttori

- **I costruttori possono dichiarare eccezioni checked nella lista di throws**
  - ogni metodo che invoca il costruttore deve o catturare l'eccezione o dichiararla nella sua lista di throws.
- **Nota : se invochiamo un costruttore utilizzando `this(...)` o `super(...)` non abbiamo modo di catturare eccezioni (perchè le invocazioni esplicite devono essere la prima istruzione)**
- **Inizializzatori di campi possono invocare metodi solo se questi non dichiarano eccezioni**

Eccezioni checked e overriding

- **Ogni eccezione dichiarata dal metodo nella sottoclasse deve essere un sottotipo di almeno una delle eccezioni dichiarate dal metodo nella superclasse**
- **Quindi il metodo nella sottoclasse può dichiarare una lista di throws con**
  - meno eccezioni e/o
  - eccezioni che sono sottotipi delle eccezioni del metodo nella superclasse

Domanda

- **In quali casi le definizioni delle due classi seguenti compilano?**

```
class X
{
  public void m( ) throws E1, E2 { ....}
}

class SX extends X
{
  public void m( ) throws E1, E21, E22 { .... }
}
```

Risposta

- solo se `E21 <: E2` e `E22<:E2`, oppure
  `E21 <: E1` e `E22 <:E1`

La clausola `finally`

- **Permette di definire codice che viene eseguito sempre e comunque all'uscita dal blocco `try`**

```
reader = new File(filename);
Scanner in = new Scanner(reader);
  . . .
reader.close(); // non eseguito in caso di eccezioni
```

Continua…

La clausola `finally`

```
File f = new File (filename);
try
{
   Scanner in = new Scanner(f);
   . . .
}
finally
{
   f.close(); // la clausola finally viene eseguita comunque
              // prima che l'eventuale eccezione venga
              // passata al gestore
}
```

La clausola `finally`

- **Eseguita all'uscita dal blocco `try-catch` in tre possibili situazioni**
  - dopo l'ultimo comando del blocco `try`
  - dopo l'ultimo comando del gestore che ha catturato l'eccezione (se ne esiste uno)
  - quando l'eccezione non è stata catturata
- **`finally` deve essere l'ultimo (o l'unico) gestorre di un blocco `try-catch`**

Domande

```
FileReader reader = null;
try
{
    reader = new FileReader(filename);
    readData(reader);
}
finally
{
    reader.close();
}
```

- Perché dichiariamo la variabile `reader` fuori dal blocco?
- Quale è il flusso di esecuzione se `filename` non esiste?

Risposte

- Semplicemente per una questione di scope
- Il costruttore `FileReader` lancia una eccezione. Il controllo passa al codice nella clausola `finally` … ma `reader` è `null`, quindi `NullPointerException`

Eccezioni user-defined

- **Normalmente sono checked**

```
public class NotEnoughCreditException extends Exception
{
   public NotEnoughCreditException(String msg)
   {
      super(msg);
   }
}
```

Continua…

Eccezioni user-defined

- utili per validare le precondizioni dei metodi pubblici

```
public class BankAccount
{
   public void withdraw(double amount)
               throws NotEnoughCreditException
   {
      if (amount > balance)
         throw new NotEnoughCreditException("Amount
                                    exceeds balance");
      balance = balance - amount;
   }
   . . .
}
```

Gestori di eccezioni

- **Tre possibilità per strutturare la gestione**
  - propagazione automatica
  - riflessione
  - mascheramento

Gestione di eccezioni – propagazione

- **Metodo più semplice, ma meno efficace**
  - Nessun gestore cattura l'eccezione
  - se l'eccezione è checked dobbiamo dichiararla

Gestione di eccezioni – riflessione

- **Eccezione rilanciata dopo una gestione parziale**
  - possiamo rilanciare la stessa eccezione
  - lanciare una eccezione diversa, di diverso grado di astrazione

Continua…

Gestione di eccezioni – riflessione

- **Esempio: un metodo che cerca il minimo in un array**

```
public static int min(int[] a) throws EmptyException {
    int m;
    try { m = a[0]; }
    catch (IndexOutOfBoundException e)
    {
        throw new EmptyException();
    }
    // scorri l'array alla ricerca del minimo . . .
}
```

Gestione di eccezioni – mascheramento

- **Eccezione catturata e gestita localmente**
  - possiamo rilanciare la stessa eccezione
  - lanciare una eccezione diversa, di diverso grado di astrazione

Continua…

Gestione di eccezioni – mascheramento

- **Un metodo che controlla se un array è ordinato**

```
public static boolean sorted(int[] a) {
    int curr;
    try { curr = a[0];
     // scorri l'array e verifica . . .
    }
    catch (IndexOutOfBoundException e) { return true; }
}
```

Esempio

```
class ServerNotFoundException extends Exception
{
   public ServerNotFoundtException(String reason, int p)
   {
      super(reason); this.port = p;
   }
   public String reason()
   {
       return getMessage(); // ereditato da Throwable
   }
   public int port()
   {
       return port;
   }
   private int port;
 }
```

Continua…

Esempio

```
class HttpClient
{

  public void httpConnect(String server)
                      throws ServerNotFoundException
  {
     final int port = 80;
     int connected = open(server, port);
     if (connected == -1)
       throw new
         ServerNotFoundException("Could not connect", port);
  }
```

Continua…

Esempio

```
public void fetchURL(String URL, String proxy) {
   String server = getHost(URL);
   try {
      httpConnect(server);
   }
   catch (ServerNotFoundException e)
   {
     System.out.println("Server unavailable: trying proxy");
     try {
        httpConnect(proxy);
     }
     catch (ServerNotFoundtException e)
     {
        System.out.println("Failed " + e.port()+ e.reason());
        throw new Error("I give up");
     }
    }
    // accedi alla pagina sul server o sul proxy
 }
```

Domanda

- Assumiamo che il seguente blocco compili correttamente.

```
try
{   . . .
    throw new E1();
}
catch (E2 e)
{
    throw new E2();
}
catch (E1 e)
{
    . . .
}
```

- In quali caso lascia sfuggire una eccezione?

Risposta

In nessun caso

- Il blocco compila, quindi `E2 <: E1`.
- Quindi il primo `catch` non cattura l'eccezione, che viene invece catturata dal secondo blocco
- Quindi nessuna eccezione sfugge

Domanda

- Sotto quali ipotesi il seguente blocco compila correttamente?

```
class A {
      public void m() throws E1 { }
  }
  class B extends A {
      public void m() throws E2 {  throw new E1(); }
  }
```

Risposta

In nessun caso

- Poiché `B.m()` compila, deve valere la relazione `E1 <: E2`:
- D'altra parte, affinché `B` sia una sottoclasse legale di `A` , deve essere `E2 <: E1`.
- Quindi dovremmo avere `E1 = E2`
- Impossibile

Domanda

- **Che relazione deve valere tra `E1` ed `E2` perchè il blocco lasci sfuggire una eccezione?**

```
try
{
     try
     {
          throw new E1( );
     }
     catch (E2 e)
     {
          throw new E2( );
     }
}
catch (E1 e)
{
     . . .
}
```

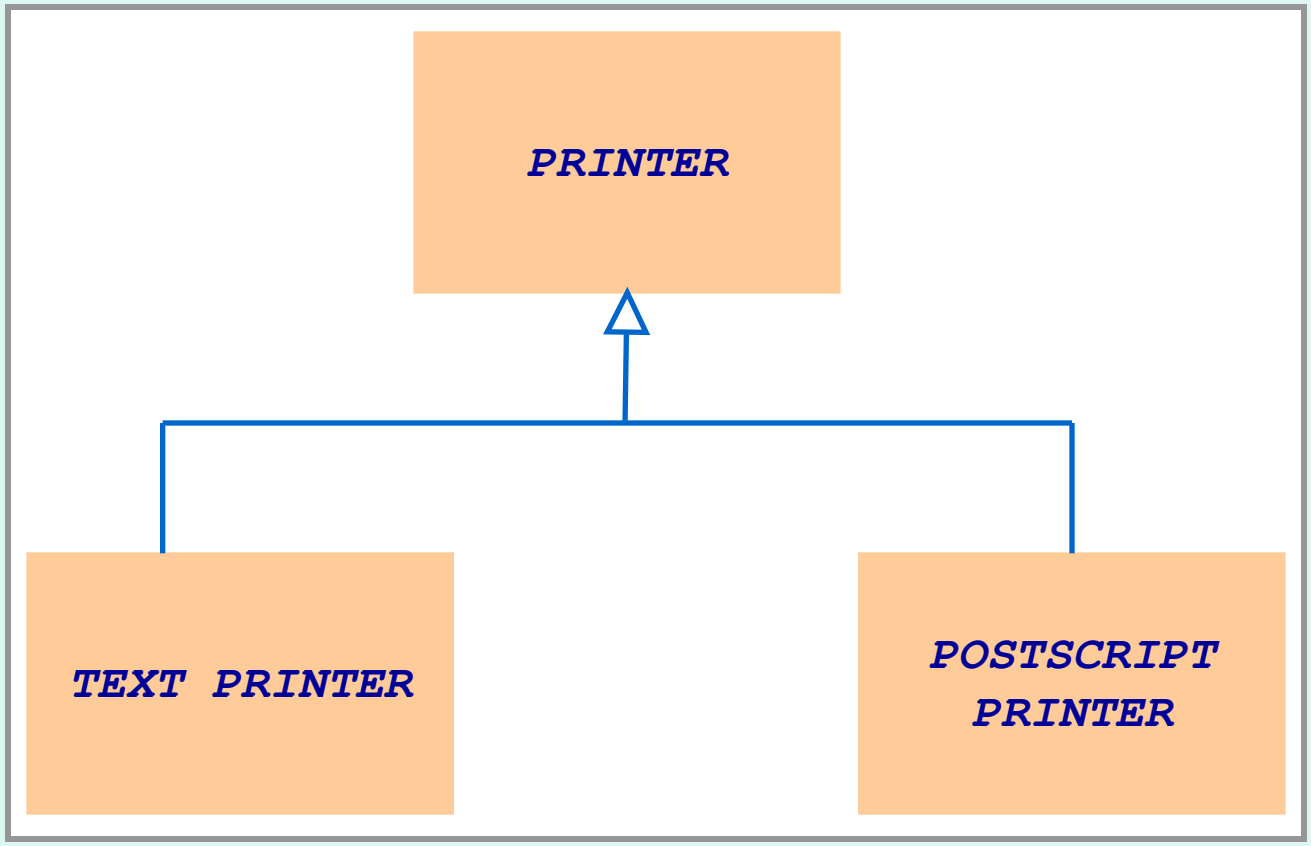
Risposta

Lascia sfuggire `E2` solo se `E1 <: E2`

- **se `E2 <:E1`, l'eccezione `E1` lanciata nel blocco interno viene catturata dal `catch` esterno**
- **se `E1 <:E2`, l'eccezione `E1` viene catturata dal `catch` interno, che rilancia `E2`. Poichè `E1 <:E2` questa seconda eccezione non viene catturata.**

Esercizio

- **Considerate il seguente diagramma di classi che descrive un servizio di stampa**



Esercizio (contd)

- **`PRINTER` ha un servizio `print(File f)` che stampa qualunque file**
- **`TEXT_PRINTER` ridefinisce il servizio di stampa, richiedendo che i file da stampare siano solo di tipo *text***
- **`POSTSCRIPT_PRINTER`, a sua volta, ridefinisc il servizio di stampa, stampando solo file di tipo postscript.**
- **Progettate le classi definendone opportunamente i contratti**

Un esempio completo

- **Un programma**
  - Chiede all'utente il nome di un file dati con la seguente struttura attesa:
    - La prima linea contiene il numero di valori
    - Le linee successive contengono i valori veri e propri, che si presume siano valori reali
  - Esempio

```
3
1.45
 -2.1
0.05
```

Un esempio completo

- **Quali sono le eccezioni possibili?**
  - **`FileNotFoundException`** per un file non esistente
  - **`IOException`** per errori nell'interazione con il file, ad esempio al momento della **`close()`**
  - **`BadDataException`**, errori nel formato dei dati attesi

Continua…

Un esempio completo

- **Chi può/deve gestire gli errori riportati dalle eccezioni?**
  - Solo il metodo **`main`** dell'applicazione interagisce con l'utente e può quindi
    - Catturare le eccezioni
    - Stampare opportuni messaggi di errore
    - Offrire all'utente nuove possibilità di interazione

File DataSetTester.java

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Scanner;

public class DataSetTester
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      boolean done = false;
      while (!done)
      {
         try
         {
```

Continued…

File DataSetTester.java

```
      System.out.println("Please enter the file name: ");
      String filename = in.next();

      double[] data = readFile(filename);
      double sum = 0;
      for (double d : data) sum = sum + d;
      System.out.println("The sum is " + sum);
      done = true;
   }
   catch (FileNotFoundException exception)
   {
      System.out.println("File not found.");
   }
   catch (BadDataException exception)
   {
      System.out.println
         ("Bad data: " + exception.getMessage());
```

Continua…

File DataSetTester.java

```
            }
            catch (IOException exception)
            {
               exception.printStackTrace();
            }
        }
    }
}
```

Il metodo `readFile()`

- **Qualunque problema con il file di input viene passato al chiamante (il main)**

```
public double[] readFile(String filename)
      throws IOException, BadDataException
      // FileNotFoundException è una IOException
{
   FileReader reader = new FileReader(filename);
   try {
      Scanner in = new Scanner(reader);
      return readData(in);
   }
   finally {
      reader.close();
   }
}
```

Continua…

Il metodo `readData()`

- **Legge il primo intero, costruisce un array e invoca `readValue()` sui valori successivi**
- **Controlla due potenziali errori**
  - Il file non contiene un intero come primo elemento
  - Il file ha più valori di quelli dichiarati
- **Anche in questo caso passa le eccezioni al chiamante**

Il metodo `readData()`

```
private double[] readData(Scanner in) throws BadDataException
{
   if (!in.hasNextInt())
      throw new BadDataException("Length expected");

   int numberOfValues = in.nextInt();
   double[] data = new double[numberOfValues];

   for (int i = 0; i < numberOfValues; i++)
      readValue(in, data, i);

   if (in.hasNext())
      throw new BadDataException("End of file expected");

   return data;
}
```

Il metodo `readValue()`

```
private void readValue(Scanner in, double[] data, int i)
      throws BadDataException
{
   if (!in.hasNextDouble())
      throw new BadDataException("Data value expected");
   data[i] = in.nextDouble();
}
```

Scenario

1. `DataSetTester.main` chiama `DataSetReader.readFile`
2. `readFile` chiama `readData`
3. `readData` chiama `readValue`
4. `readValue` trova un valore inatteso e lancia `BadDataException`
5. `readValue` non ha alcun handler e termina

Continued…

Scenario

6. `readData` non ha handlers e termina a sua volta
7. `readFile` non ha handler e termina ma dopo aver eseguito la clausola finally
8. `DataSetTester.main` gestisce l'eccezione

Domanda

- **Supponiamo che l'utente specifichi un file che esiste ma è vuoto. Quale è il flusso di esecuzione?**

Risposta

- `DataSetTester.main` chiama `DataSetReader.readFile`,
- Che a sua volta chiama `readData`.
- `in.hasNextInt()` restituisce `false`, e `readData` lancia `BadDataException`.
- Il metodo `readFile` non la cattura e la propaga al `main`, dove viene catturata