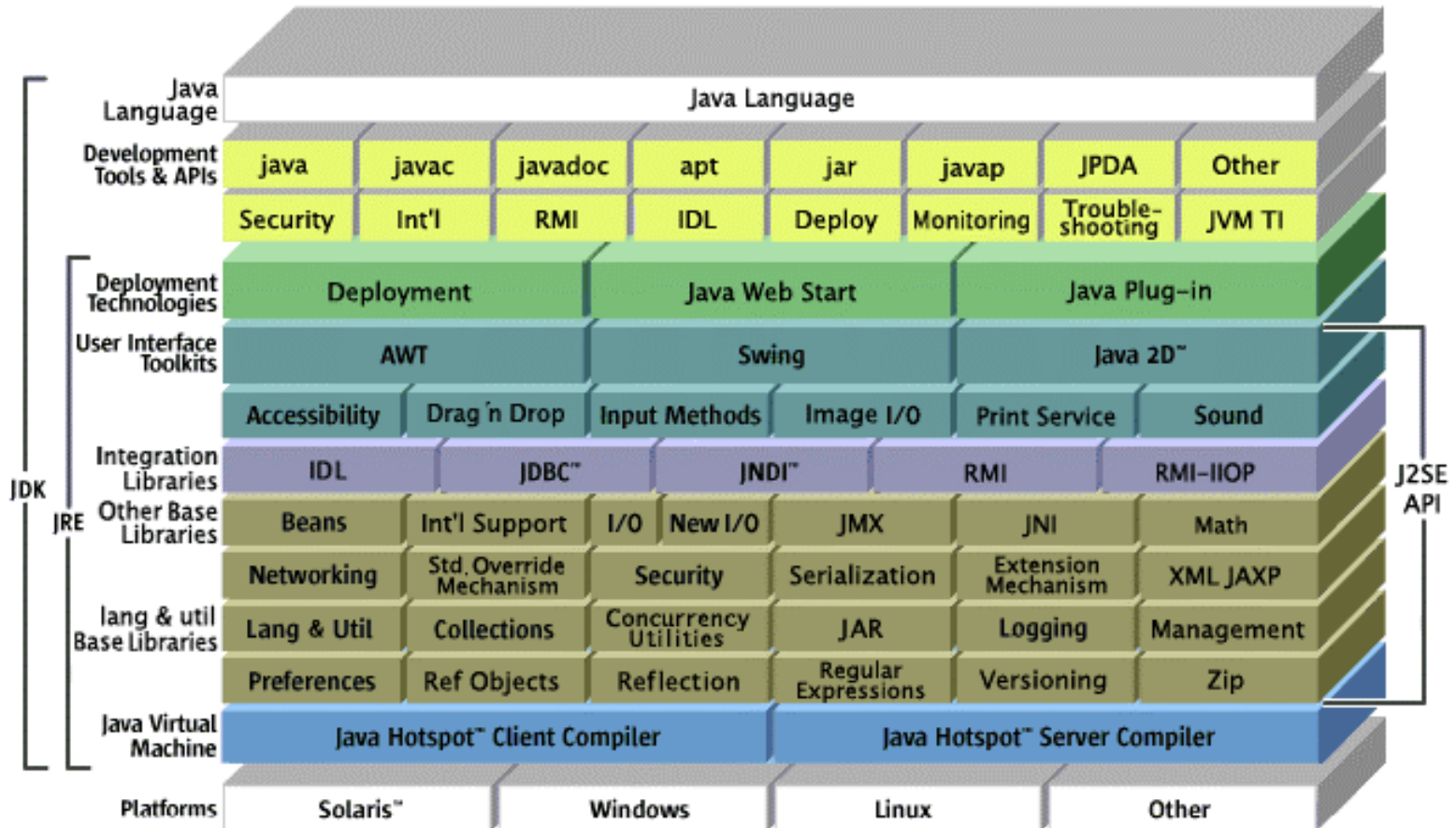


# Ambiente Java

Java™ 2 Platform Standard Edition 5.0



# Compilatori

- Sorgente -> bytecode
  - portabile
  - completamente decompilabile
  - interpretato
- Sorgente -> eseguibile nativo
  - non portabile
  - non facilmente decompilabile
  - piu' veloce di C e C++

# Bytecode e Interprete

- Intestazione fornisce versione
- Istruzioni della macchina virtuale
- Lunghezza costante
- Stack machine tipata

# JIT compiler

- JIT == Just In Time
- Compila il byte code al tempo dell'esecuzione
- produce codice macchina nativo (non portabile)
- migliora il tempo di esecuzione (tranne che la prima volta)
- Non effettua ottimizzazioni globali

# Direttive di programma

- una o piu' linee
- un token non puo' stare su piu' di una linea
- terminate da ;
- `System.out.println("HELLO");`
  - equivale a
- `System`
- `.out`
- `.println`
- `("HELLO"`
- `);`

# Direttive di programma (Cont.)

- Ma non
- Sys
- tem.out.pr
- `intln(“HELLO”);`
- Comunque attenzione alla leggibilita’

# Concetti base di programmazione a oggetti

- Costruzione dei tipi
- Oggetti e istanze
- Incapsulazione delle caratteristiche
- Derivazione e specializzazione
- Manipolabilita'

# Classi

- Definisce un oggetto elencandone
  - le caratteristiche
  - le possibili manipolazioni
  - le relazioni con altre classi (Classi derivate)



# Classi derivate

- Java permette di definire classi derivate da altre classi
- Una classe derivata specifica ulteriormente la classe da cui deriva
- Una classe puo' derivare da una sola altra classe

# Tipi di dati

- Oggetti
  - definiti da classi
  - passati sempre per riferimento
- tipi primitivi
  - passati sempre per valore

# La libreria di classi di SDK 1.2

- Stesso principio di C e C++
- Non keyword come in pascal
- librerie standard
  - I/O
  - Networking
  - Utilities
  - Reflections
  - SQL (JDBC)
  - ...

# Tipi primitivi e classi relative

- `boolean` <-> `Boolean`
- `byte` <-> `Byte`
- `short` <-> `Short`
- `int` <-> `Integer`
- `long` <-> `Long`
- `float` <-> `Float`
- `double` <-> `Double`

# Aritmetica e Regole di conversione

- Aritmetica intera sempre almeno a 32 bit
- Upcast automatico
  - byte->short->int->long->float->double
- Downcast esplicito
  - short a,b,c;
  - a = b + c; // non compila
  - a = (short)(b + c) // ok
- ECCEZIONE operatori op=

# Operatori e loro precedenza

- Le stesse del C e C++

() , [] , . , X++ , X--

+ unario , = unario , ++X , --X , ~ , !

Cast esplicito , new

\* , / , %

+ , -

<< , >> , >>>

< , <= , > , >= , instanceof

== , !=

&

^

|

&&

||

?:

= , op=

- **CONSIGLIO:** se avete dubbi usate le parentesi

# Commenti e documentazione

- come c++
  - /\* ..... \*/
  - // sino a fine linea
- Commenti per documentazione automatica
- /\*\*
- \* Questo commento viene usato per creare
- \* automaticamente la documentazione della
- \* classe.
- \*/

# Confronti

- Confronti come nel C e C++
- $>$
- $>=$
- $==$
- $!=$
- $<=$
- $<$



# Costrutti di controllo

```
if(<boolean value>)
```

```
{
```

```
}
```

```
else
```

```
{
```

```
}
```

- Indentazione non codificata, dipende dallo stile personale

# Costrutti di controllo (Cont.)

```
if(<boolean value>)  
{  
    if(<boolean value>)  
    {  
    }  
    else  
    {  
    }  
}  
else if(<boolean value>)  
{  
}  
else if(<boolean value>) .... Ad libitum
```

# Costrutti di controllo (Cont.)

- Esiste l'operatore condizionale
- $a > b ? C : D;$
- Se volete partecipare al campionato “Offuscated Java code” ....

# Costrutti di controllo (Cont.)

```
switch(<expression>)
```

```
{
```

```
    case <value>:
```

```
        break;
```

```
    case <value>:
```

```
        break;
```

```
    case <value>:
```

```
        // fallthrough!!
```

```
    case <value>:
```

```
        break;
```

```
    default:
```

```
        break;
```

```
}
```

# Costrutti di controllo (Cont.)

```
for(<inizializzazione>;<condizione di uscita>;<azione iterativa>)  
{  
    <corpo>  
}
```

- **equivale**

```
<inizializzazione>  
while(!<condizione di uscita>)  
{  
    <corpo>  
    <azione iterativa>  
}
```

# Costrutti di controllo (Cont.)

do

{

<corpo eseguito almeno una volta>

}

while(<condizione di continuazione>)

# Controllo dei loop

## continue

passa direttamente all'iterazione successiva senza eseguire il resto del corpo

la condizione iterativa di un loop for viene eseguita

la condizione iterativa del loop while equivalente NON viene eseguita

## break

interrompe l'esecuzione del loop e passa alla prima istruzione fuori dal loop

# Controllo dei loop (Cont)

- continue e break possono usare etichette

Uscita:

```
for(i=0;i<1000;i++)  
    for(j=0;j<1000;j++)  
    {  
        <corpo 1>  
        if(condizione eccezionale)  
            continue Uscita;  
        <corpo 2>  
        if(condizione di errore)  
            break Uscita;  
    }
```



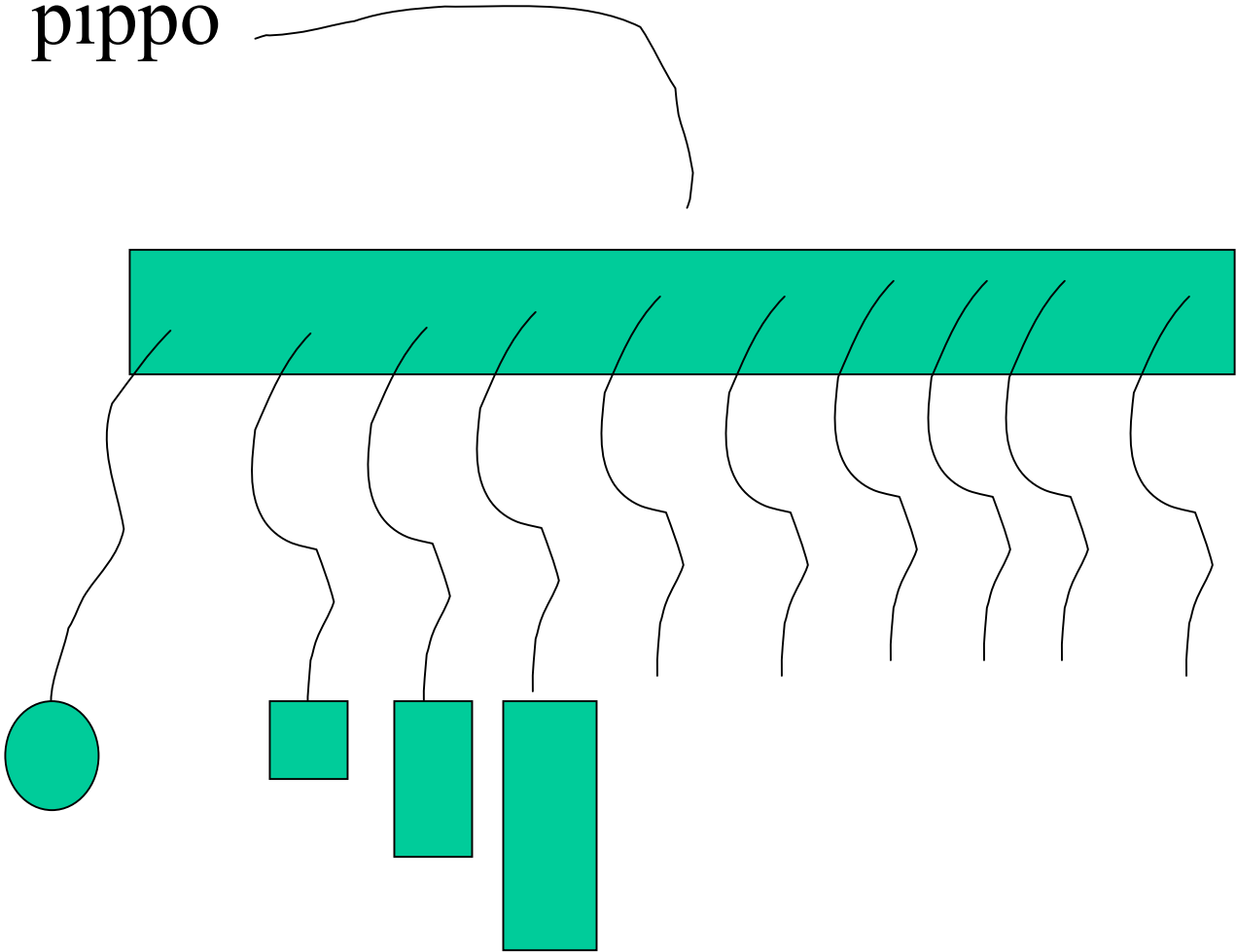
# Array

- Array sono veri oggetti, non solo un modo conveniente di indicizzare una zona di memoria come in C o C++
- Array in Java hanno una lunghezza definita
  - non si puo' sfiorare e massacrare i dati senza saperlo
  - overhead dovuto al controllo degli indici
- Dichiarazione != definizione
- Definizione del nome
- dichiarazione alloca la memoria
- `char[] myArray = new char[10];`

# Array (Cont.)

- `int[][] pippo;`
- `pippo = new int[10][];`
- `pippo[0] = new int[0];`
- `pippo[1] = new int[1];`
- ...
- `pippo[9] = new int[9];`
  
- `long pluto[][][];`
- etc.

- pippo



# Stringhe

- Le stringhe sono oggetti immutabili come le classi corrispondenti ai tipi primitivi
  - non ci sono problemi di sfioramento della memoria allocata
  - overhead di gestione degli oggetti
- Si opera su stringhe come oggetti tramite metodi specifici
- `String a = new String("pippo");`
- `a = a.concat(" pluto");`

# Stringhe (Cont)

`char charAt(int index)`

Returns the character at the specified index.

`int compareTo(Object o)`

Compares this String to another Object.

`int compareTo(String anotherString)`

Compares two strings lexicographically.

Etc.

# Definizione di classi

- Definire un tipo di oggetto e la semantica che gli si vuole dare tramite
  - campi
    - come i campi di una struttura C
  - metodi
    - non esiste parallelo diretto in C
  - blocchi di inizializzazione
    - eseguiti al caricamento, prima di generare qualunque istanza della classe, una sola volta
- I nomi delle classi cominciano con la maiuscola

# Classe DisneyCharacter

```
package it.unige.dist.laser.esempio;
public class DisneyCharacter
{
    int yob = -1;
    String name;
    protected DisneyCharacter(String theName, int theYob)
    {
        name = theName;
        yob = theYob;
    }
    public String isFunny()
    {
        return "tutti i personaggi Disney sono divertenti";
    }
    public String toString()
    {
        return new String(name + " " + String.valueOf(yob));
    }
    static
    {
        System.out.println("Hurra!");
    }
}
```

# Campi e tipi di campi

- campo di istanza
  - uno per ogni istanza della classe
  - tipo di default
- campo di classe
  - uno comune a tutte le istanze della classe
  - richiede la keyword “static”



# Metodi e tipi di metodi

- metodo di istanza
  - uno per ogni istanza della classe
  - tipo di default
- metodo di classe
  - uno comune a tutte le istanze della classe
  - richiede la keyword “static”

# Passaggio parametri

- parametri formali e parametri attuali
- tipi primitivi per valore
- Oggetti per indirizzo
  - si possono definire “final” e non possono essere cambiati (controllo a tempo di compilazione)

# Accesso a campi e metodi

- Un campo o un metodo di istanza richiede di avere una istanza attiva della classe per potervi fare riferimento

```
String s = new String("pippo");  
s.length(); // metodo di istanza
```

- Un campo o metodo di classe si riferisce al nome della classe stessa

```
int i = 10;  
String.valueOf(i);
```

# Regole di visibilita'

- Private
- Protected
- “Default”
- Public

# Costruttori

- I costruttori hanno il nome della classe
- I costruttori non definiscono il tipo di ritorno
- Il costruttore di default (senza parametri) se non definito esplicitamente viene generato automaticamente dal compilatore
- Un costruttore di default generato dal compilatore non fa nulla (a parte invocare il costruttore di default della classe madre se esiste)

# garbage collection

- Java non ha deallocazione esplicita
- un oggetto rimane in memoria almeno sino a che esiste un riferimento ad esso
  - in pratica una volta perso l'ultimo riferimento puo' passare del tempo
- non ci possono essere memory leak causati da programmi utente
- la garbage collection introduce due overhead
  - contare i riferimenti
  - effettuare la garbage collection vera e propria

# Overloading dei metodi

- Si possono avere piu' metodi con lo stesso nome purché la signature (numero e/o tipo dei parametri formali) sia diversa
- Il valore di ritorno non è sufficiente a distinguere

```
public class Esempio
```

```
{  
    int value(int a) {};  
    int value(long a) {};  
    long value(int a, long b) {};  
    long value(long a) {}; //errore al tempo di compilazione!  
}
```

# Ereditarieta'

- Una classe puo' estendere un'altra classe
- Eredita cio' che era gia' definito nella classe madre
  - campi e metodi
- il processo e' addittivo
  - a estende b estende c
  - implica che a contiene tutto cio' che e' in b e in a
- un tipo derivato e' sempre anche di ogni tipo piu' astratto, non e' vero il contrario
  - a e' di tipo a, di tipo b e di tipo c
  - b e' di tipo b e di tipo c
  - c e' solo di tipo c



# Esempio classe derivata

```
package it.unige.dist.laser.esempio;
public class Pippo extends DisneyCharacter
{
    public pippo()
    {
        super("Pippo", <non mi ricordo, acc...>)
    }
    public String isFunny()
    {
        return "tra i personaggi Disney Pippo e' uno dei piu' divertenti";
    }
    static
    {
        System.out.println("Yuk yuk!");
    }
}
```

# Classi astratte

- Una classe puo' essere "astratta", cioe' definire la necessita' di avere un tipo di comportamento al fine di appartenere ad una categoria logica

```
public abstract class Astratta
{
    public abstract int quanti();
}
```

- non puo' essere istanziata direttamente
- deve essere estesa e la classe derivata deve implementare tutti i metodi astratti della classe base

# Polimorfismo

- La capacità di identificare solo a runtime l'effettivo comportamento del programma

```
Public static void main(String[] argv)
```

```
{
```

```
    Astratta a = <leggi da canale di input>;
```

```
    Sstem.out.println(String.valueOf(a.quantità()));
```

```
}
```

# Overwrite dei metodi

- Si usa per implementare il polimorfismo
- I metodi in Java sono virtuali per default
- il metodo invocato sara' sempre quello della classe piu' derivata possibile purché
  - il metodo invocato deve essere presente anche nella classe base
  - la signature del metodo deve essere la stessa in classe base e classe derivata **INCLUSO IL TIPO RITORNATO**
- non e' possibile restringere l'accesso a un metodo derivato

# Modificatore final

- serve per prevenire la modificazione della semantica di una classe
- i metodi di Java sono “virtuali”
- se io derivo una classe viene eseguito il codice della classe derivata
- posso alterare la semantica di un oggetto ridefinendone i metodi in una classe derivata
- se la classe e’ definita “final” questo e’ impossibile
- e’ possibile anche definire final singoli metodi o campi

# Interfacce

- Guerra di religione Distributed OOP:
- Implementation Inheritance vs. Delegation and Containment
- Derivazione da classi base vs. Definizione di Interfacce e loro implementazione

# Uso delle Interfacce

contratto tra utilizzatore e fornitore di servizi

permette di nascondere completamente

l'implementazione e cristallizzare la modalita' di interazione

metodologia usata pesantemente in quasi tutti gli schemi di distributed OOP (e.g. CORBA, RMI e DCOM)

**ATTENZIONE:** contratto sintattico, non si dice nulla in proposito alla semantica!!!

# Definire un'Interfaccia

```
public interface Button
{
    public String getName();
    public void press();
    public boolean isPressed();
}
```



# Implementare un'Interfaccia

```
public class MyButton implements Button
{
    String name = "Innominato";
    boolean state = false;
    public MyButton(String theName)
    {
        name = theName;
    }

    public String getName()
    {
        return new String("Il mio bottone si chiama " + name);
    }
    public void press()
    {
        state ^= true;
        state = !state;
    }
    public boolean isPressed()
    {
        return state;
    }
}
```

# Concetto e uso dei Package

- package == insieme di classi legate da un comune attributo
  - implementano un servizio
  - sono logicamente simili e.g. tutte le classi che implementano un formato di immagine
  - ...
  - stesso programmatore?
- sono l'unita' di visibilita' di default

# Concetto e uso dei Package

(Cont)

- I nomi dei package cominciano con la minuscola
- I nomi dei package dovrebbero seguire la convenzione contraria a quella degli URL

```
package it.unige.dist.laser.pippo;
```

```
package com.sun.pluto;
```

```
import java.util.*;
```

# Da “I 10 comandamenti del programmatore C”

**6 If a function be advertised to return an error code in the event of difficulties, thou shalt check for that code, yea, even though the checks triple the size of thy code and produce aches in thy typing fingers, for if thou thinkest “it cannot happen to me”, the gods shall surely punish thee for thy arrogance.**

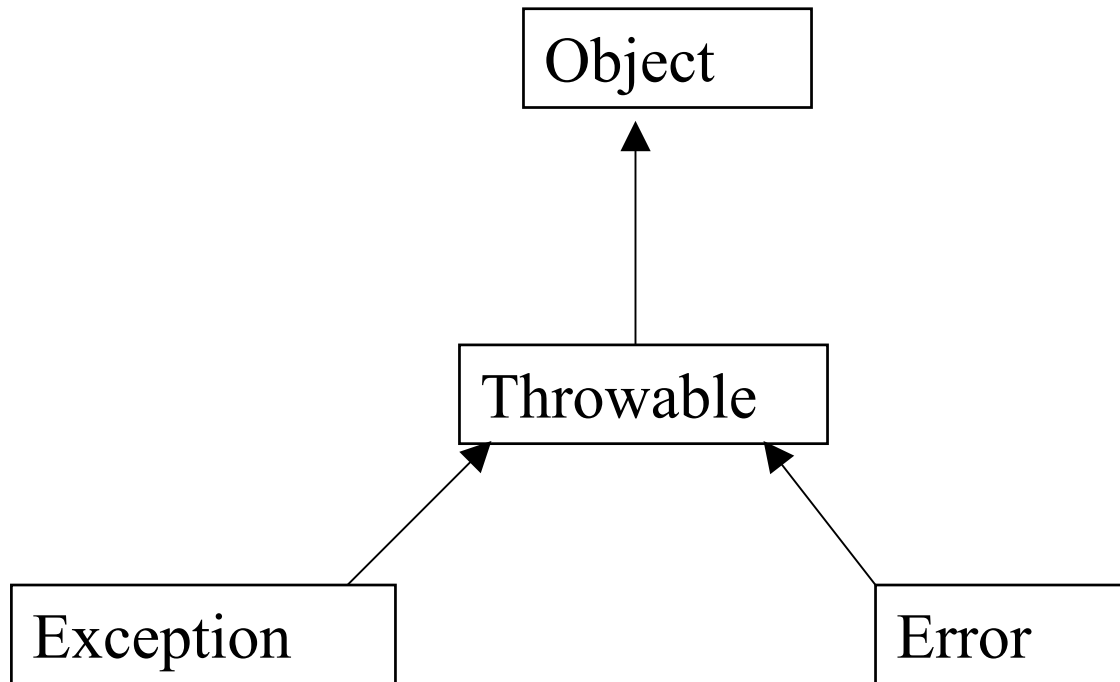
All true believers doth wish for a better error- handling mechanism, for explicit checks of return codes are tiresome in the extreme and the temptation to omit them is great. But until the far-off day of deliverance cometh, one must walk the long and winding road with patience and care, for thy Vendor, thy Machine, and thy Software delight in surprises and think nothing of producing subtly meaningless results on the day before thy Thesis Oral or thy Big Pitch To The Client.

Occasionally, as with the `ferror()` feature of `stdio`, it is possible to defer error checking until the end when a cumulative result can be tested, and this often produceth code which is shorter and clearer. Also, even the most zealous believer should exercise some judgement when dealing with functions whose failure is totally uninteresting... but beware, for the cast to void is a two-edged sword that sheddeth thine own blood without remorse.

# Eccezioni

- la generazione di un'eccezione segnala un evento eccezionale (!)
- separa il cammino logico standard dalla gestione delle situazioni inattese
- Java tramite le eccezioni permette di concentrare il codice che gestisce le condizioni di errore o comunque anormali in modo centralizzato
- limita la complessita' della struttura e del flusso di programma
- **ATTENZIONE:** richiedono un grosso overhead computazionale

# Eccezioni (Cont.)



Possono essere gestite

Non dovrebbero essere gestite

# Blocco Try Catch

```
try
{
    <apri file>
    <leggi file>
    <elabora dati>
    <chiudi file>
}
catch(Exception e)
{
    <azione correttiva o semplice segnalazione>
}
catch(IOException ioe)
{ ioe.printStackTrace();
}
}
```

# Metodi ed Eccezioni

- si possono specificare le eccezioni lanciate da un metodo
- si forza il codice che invoca il metodo ad essere in grado di gestire le eccezioni elencate (controllo al tempo della compilazione)
- il compilatore controlla anche che le eccezioni elencate possono effettivamente essere generate



# Rilanciare le eccezioni

- Le eccezioni catturate con un blocco catch possono essere rilanciate
  - permette di effettuare clean-up locale e propagare comunque l'avvenimento eccezionale
  - permette di specificare ulteriormente cosa è accaduto

```
try
{
    <azione>
}
catch(Exception e)
{
    String date = (new java.util.Date()).toString();
    String message = e.toString();
    message = message.concat(" ");
    message = message.concat(date);
    Exception newE = new Exception(message);
    throw newE;
```

```
Exception newE = new Exception(e.toString() + " " + (new
```

# Errori

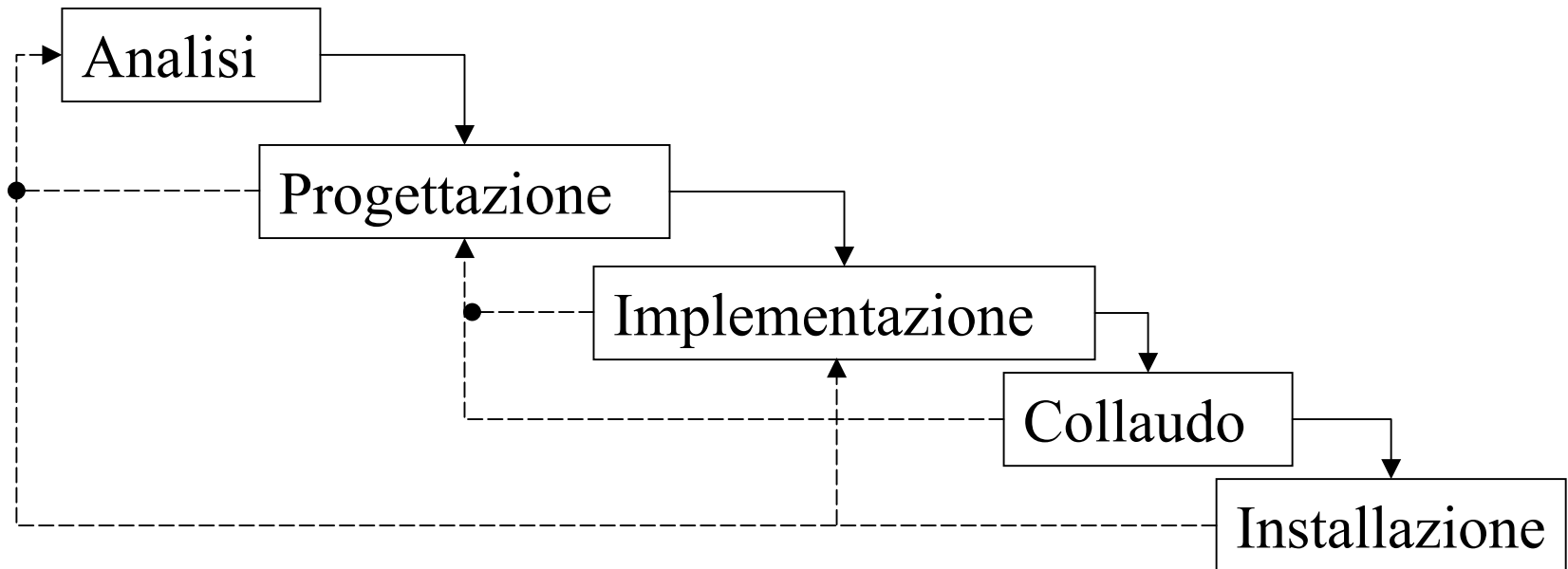
- Segnalano situazioni che di solito sono o irrecuperabili a livello di macchina virtuale
  - errori di link (problemi di caricamento di un file di classe)
  - errori interni alla macchina virtuale
- o servono ad effettuare il clean-up della macchina virtuale
  - morte di Thread (segnala la terminazione di un Thread)

# Ciclo di vita SW 1

- Modello a cascata
  - Analisi
  - Progettazione
  - Implementazione
  - Collaudo
  - Installazione

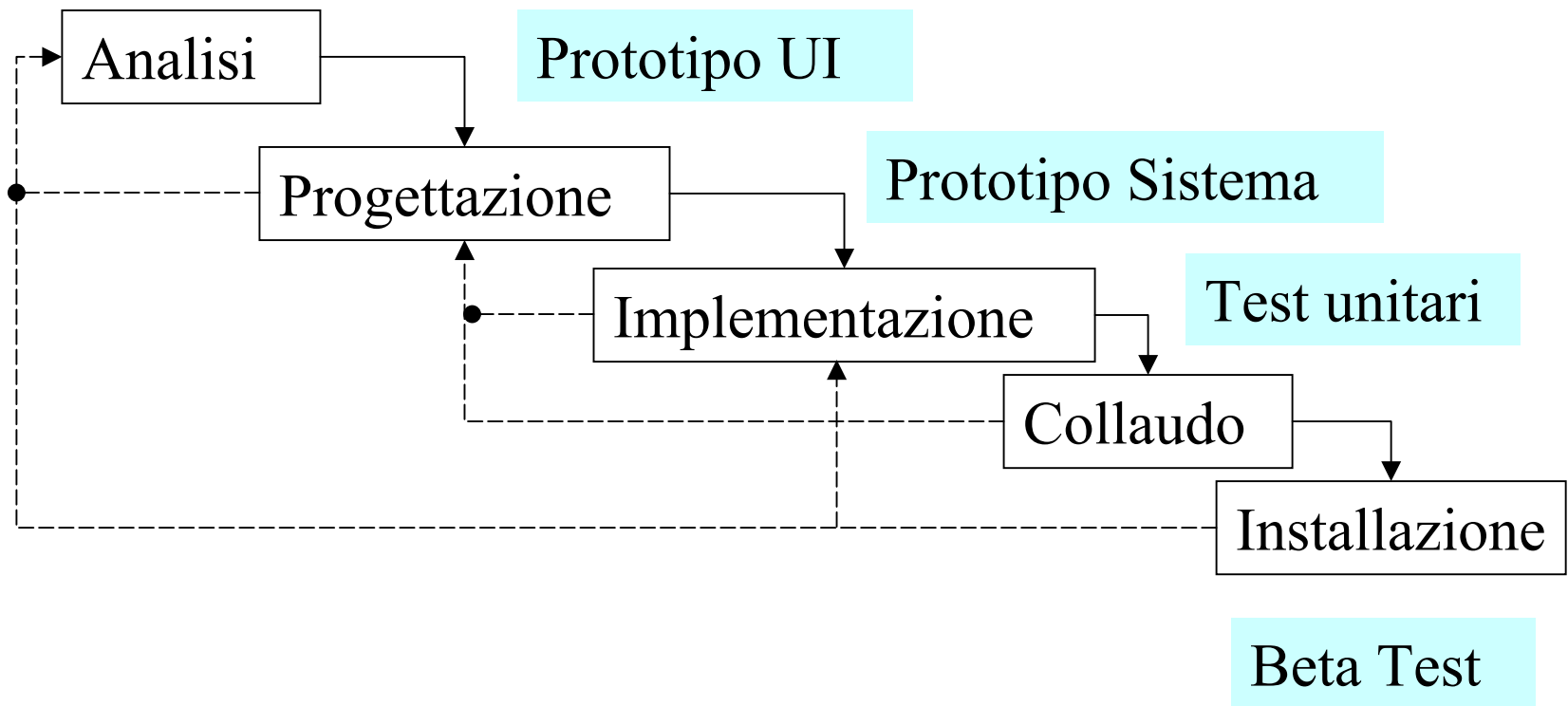
# Ciclo di vita SW 2

- In bella sequenza? Magari!



# Ciclo di vita SW 2

- Inoltre: anticipazioni!



# Analisi

- Scoprire gli oggetti
  - Definire le classi
- Schede Classe, Responsabilita', Collaboratore (CRC)

# Scheda CRC

Classe (nome dell'oggetto)

Responsabilita' (funzione)	Collaboratori (Oggetti contattati per eseguire la funzione)

# Esempio uso scheda CRC

- Stampa di una fattura

## FATTURA

Nome e dati cliente

Nome Articolo	Quantita'	Prezzo Unitario	Totale

Totale Finale



# Classi

- Fattura
  - DOH!
- Cliente
- Articolo

# Fattura

Stampa	Cliente, Articolo
Calcola Importo Dovuto	Articolo

# Cliente

Fornisci dati cliente	
-----------------------	--

# Articolo

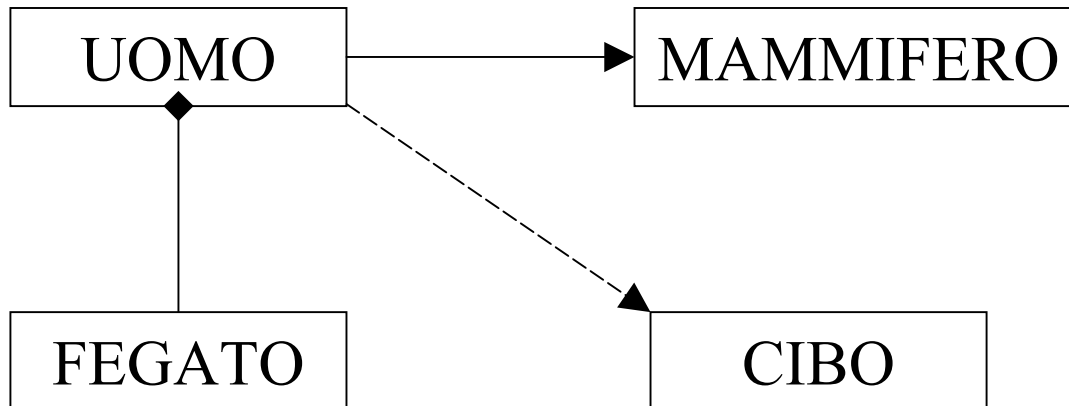
Calcola prezzo totale	
--------------------------	--

# Relazioni fra classi

- Derivazione
  - Uomo deriva da Mammifero deriva da Animale
  - Fegato, Cuore e polmone derivano da Organo
- Aggregazione
  - Uomo aggrega organi
    - Contiene un'istanza di Fegato
    - Contiene un'istanza di Cuore
    - Contiene due istanze di Polmone
- Dipendenza
  - Uomo dipende da Cibo

# Unified Modeling Language

- UML
  - Linguaggio grafico per rappresentare le relazioni tra classi



# Abstract Data Type

- **Definition:** A mathematically specified collection of data-storing entities with operations to create, access, change, etc. instances.
- *Note: Since the collection is defined mathematically, rather than as an implementation in a computer language, we may reason about effects of the operations, relations to other abstract data types, whether a program implements the data type, etc.*

# Esempio: Stack

- *One of the simplest abstract data types is the stack. The operations  $new()$ ,  $push(v, S)$ ,  $top(S)$ , and  $pop(S)$  may be defined with the following.*
- *$new()$  returns a stack*  
 *$pop(push(v, S)) = S$*   
 *$top(push(v, S)) = v$*   
*where  $S$  is a stack and  $v$  is a value.*



# Stack

- *From the previous slide axioms, one may define equality between stacks, define a pop function which returns the top value in a non-empty stack, etc. For instance, the predicate  $isEmpty(S)$  may be added and defined with the following two axioms.*  
 $isEmpty(new()) = true$   
 $isEmpty(push(v, S)) = false$