Richiami di Java

Linguaggio Java

- Robusto
  - Non permette costrutti “pericolosi”
    - Eredità Multipla
    - Gestione della Memoria
- Orientato agli oggetti
  - Ogni cosa ha un tipo
  - Ogni tipo è un oggetto (quasi)
- Protegge e gestisce dagli errori
  - Verifica gli errori runtime
  - Gestisce esplicitamente gli errori

Caratteristiche Java - cont.

- Portabile
  - Astrae l'accesso alle risorse del SO
    - File, network, etc.
  - Astrae i tipi di dato
    - int, float, double
- Debug/Linking dinamico
  - Debug del bytecode
  - Il bytecode contiene le informazioni per il linking a runtime

Java Development Kit

- bin: contiene i file eseguibili
  - javac: compilatore
  - java: avvia la JVM ed esegue i programmi
  - jdb: esecuzione in modalità debug
  - javadoc: genera la documentazione
  - …
- demo: esempi (anche complicati) per mostrare le potenzialità di Java
- include e lib: codice binario usato da JDK
- jre: la Java Virtual Machine
- docs: documentazione delle librerie java

Java Virtual Machine

- È un programma installato nel vostro PC
- Crea l'ambiente esecutivo per i programmi
  - Li intefaccia con la macchina
- Avviata con "java"
- Non è legata al linguaggio Java
  - Esegue bytecode
- Il bytecode può essere generato in diversi modi
  - Il principale e javac

JVM e bytecode

- Il programma compilato è “teoricamente” portabile
  - Non dipende dal SO
  - Mi serve solo la JVM
- JVM astrae i SO
- La JVM non è portabile
  - SO diversi hanno JVM diverse
- Tutte le JVM dovrebbero avere le stesse caratteristiche
  - Spesso non è vero
  - Ecco il perché del “teoricamente”

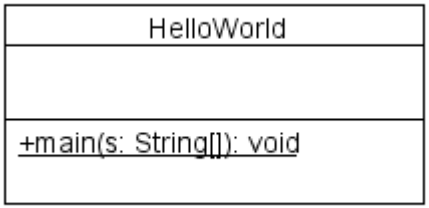
Linguaggio ed esecuzione

- Java
  - Linguaggio (è uno solo)
- Diverse piattaforme
  - J2SE (Standard Edition)
    - Applicazioni desktop
  - J2EE (Enterprise Edition)
    - Applicazioni server oriented
  - J2ME (Mobile Edition)
    - Applicazioni Mobile
  - …

Hello world

```
1 public class HelloWorld
2 {
3     public static void main(String args[])
4     {
5         System.out.println("Hello World!");
6     }
7 }
```

- Riga1: dichiarazione della classe
- Righe 2-7: parentesi graffe che contengono la classe
- Riga 3: dichiarazione del metodo main
- Riga 5: stampa la stringa “Hello World!” usando una libreria Java



Elementi di base di Java

1. Classe
2. Oggetto
3. Membro
   - Attributo
   - Metodo
4. Costruttore
5. Package

Classi ed Oggetti

- Una classe è una astrazione indicante un insieme di oggetti che condividono le stesse funzionalità

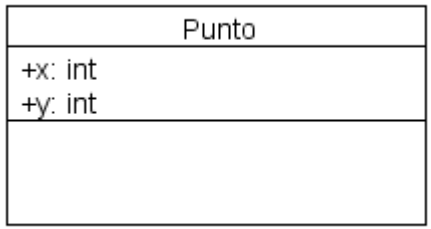
- Un oggetto è una istanza (fisica) di una classe

La classe Punto

- Astrazione del punto cartesiano bidimensionale

```
public class Punto
{
    public int x;
    public int y;
}
```

- Possiamo compilarlo
  - javac Punto.java
- Non possiamo eseguirlo
  - java Punto
- Abbiamo definito il "template" ma non l'oggetto
  - La sua realizzazione fisica nel PC



Gli oggetti Punto

```
1     public class Principale
2     {
3         public static void main(String args[])
4         {
5             Punto punto1;
6             punto1 = new Punto();
7             punto1.x = 2;
8             punto1.y = 6;
9             Punto punto2 = new Punto();
10            punto2.x = 0;
11            punto2.y = 1;
12            System.out.println(punto1.x);
13            System.out.println(punto1.y);
14            System.out.println(punto2.x);
15            System.out.println(punto2.y);
16        }
17    }
```

La main per poter eseguire il codice

Creo un oggetto

Creo un altro oggetto

Osservazioni

- Classe Punto
  - Definisce la struttura dati
  - La usiamo in compilazione
  - Sono gli oggetti che hanno un ruolo attivo
- A rigore
  - Le classi non dovrebbero possedere membri
    - Nota: sono gli oggetti che possiedono x e y
  - Infatti per accedere alla locazione di memoria
    - nomeOggetto.nomeVariabile
    - La variabile appartiene a punto1 e non a Punto

Osservazioni

- Prima Eccezione: La classe Principale
  - Esegue del codice nella classe
  - Non su un oggetto creato dalla classe
- Per storia:
  - Java non usa programmi "chiamanti" come il C++
  - Java avvia i programmi un metodo "statico" della classe: la main
  - Va sempre dichiarata così:

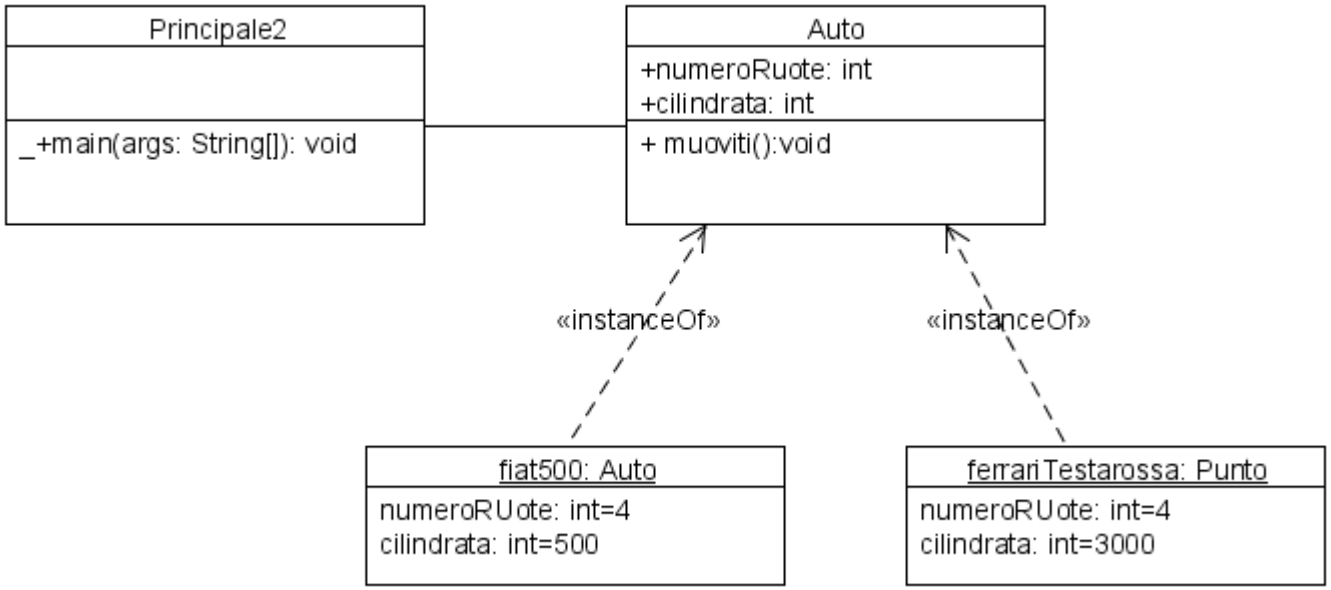
```
public static void main(String args[])
```

Astrarre la realtà

```
public class Auto
{
    public int numeroRuote = 4;
    public int cilindrata; // quanto vale?
    public void muoviti()
    {
        // implementazione del metodo...
    }
}
```

```
1    public class Principale2
2    {
3        public static void main(String args[])
4        {
5            Auto fiat500;
6            fiat500 = new Auto();
7            fiat500.cilindrata = 500;
8            fiat500.muoviti();
9            Auto ferrariTestarossa = new Auto();
10           ferrariTestarossa.cilindrata = 3000;
11           ferrariTestarossa.muoviti();
12       }
13   }
```

UML


Principale2
_+main(args: String[]): void
Auto
+numeroRuote: int
+cilindrata: int
+ muoviti():void
«instanceOf»
«instanceOf»
fiat500: Auto
numeroRUote: int=4
cilindrata: int=500
ferrariTestarossa: Punto
numeroRUote: int=4
cilindrata: int=3000

Dichiarare Metodi

```
[modificatori] tipo_di_ritorno nome_del_metodo
([parametri]) {corpo_del_metodo}
```

- Modificatori
  - cambiano le caratteristiche del metodo
  - Esempi: public, static
- Tipo di ritorno
  - Il tipo di dato che il metodo restituisce
  - Può essere un tipo primitivo (int) od un oggetto (String) o nulla (void)
- Nome del metodo
- Parametri
  - Dichiarazione di variabili che possono essere passate al metodo
  - Possono non esserci
  - Se più di uno vanno separati dalla virgola
- Corpo del metodo
  - Le istruzioni da eseguire

Esempio

```
public class Aritmetica
{
    public int somma(int a, int b)
    {
        return (a + b);
    }
}
```

```
1   public class Uno
2   {
3       public static void main(String args[])
4       {
5           Aritmetica oggetto1 = new Aritmetica();
6           int risultato = oggetto1.somma(5, 6);
7       }
8   }
```

Accesso a Metodi e Variabili

- Metodi:
  - nomeOggetto.nomeMetodo()

```
int risultato = oggetto1.somma(5, 6);
```

  - Nota: l'oggetto1 va creato con un new
- Variabile
  - nomeOggetto.nomevariabile

Esempio

```
public class Saluti
{
    public void stampaSaluto()
    {
        System.out.println("Ciao");
    }
}
```

```
1     public class Due
2     {
3         public static void main(String args[])
4         {
5             Saluti oggetto1 = new Saluti();
6             oggetto1.stampaSaluto();
7         }
8     }
```

Dichiarare Variabili

```
[modificatori] tipo_di_dato nome_della_variabile [ =
inizializzazione];
```

- Modificatori:
  - cambiano le caratteristiche della variabile
- Tipo di dato
  - Il tipo di dato della variabile
- Nome della Variabile
- Inizializzazione
  - Il valore a cui viene impostata la memoria di default

```
public class Quadrato
{
    public int altezza, larghezza;
    public final int NUMERO_LATI = 4;
}
```

Variabili

- Variabili d'istanza
  - Dichiarate nella classe ma fuori da un metodo
  - Fanno parte dell'oggetto
    - Vengono allocate con il new dell'oggetto
    - Vengono de-allocate quando l'oggetto non esiste più
- Variabili locali
  - Sono dichiarate all'interno dei metodi
  - Vengono allocate quando si esegue il metodo

Parametri o Argomenti

- Compaiono nella dichiarazione dei metodi

```
public int somma(int x, int y)
{
    int z = x + y;
    return z;
}
```

- Sono creati quando chiamiamo il metodo

```
int risultato = oggetto1.somma(5, 6);
```

```
int a = 5, b = 6;
int risultato = oggetto1.somma(a, b);
```

I Metodi Costruttori

- Metodi speciali con le seguenti proprietà
  - Hanno lo stesso nome della classe
  - Non hanno tipo di ritorno
  - Sono chiamati automaticamente se creo un oggetto della classe

```
public class Punto
{
    public Punto() //metodo costruttore
    {
        System.out.println("costruito un Punto");
    }
    int x;
    int y;
}
```

Creare oggetti

- Dichiarazione ed istanza

```
Punto punto1; //dichiarazione
punto1 = new Punto();  // istanza
```

```
Punto punto1 = new Punto(); //dichiarazione ed istanza
```

- Solo istanza

```
new Punto();
```

  - Non è utilizzabile mi manca il riferimento all'oggetto

Costruttori con parametri

```
public class Punto
{
    public Punto(int a, int b)
    {
        x = a;
        y = b;
    }
    public int x;
    public int y;
}
```

- Non posso più usare

```
Punto punto1 = new Punto();
```

- Creo l'oggetto con

```
Punto punto1 = new Punto(5,6);
```

Package

- Permette di raggruppare classi java
- In pratica:
  - È una cartella nel nostro PC
- Per crearlo
  - Almeno una Classe deve dichiarare l'appartenenza al package
  - Tale classe deve risiedere in tale cartella

```
package package1;
```

Stile di Codifica e Commenti

```
public class Classe {
    public int intero;
    public void metodo() {
        intero = 10;
        int unAltroIntero = 11;
    }
}
```

```
// Questo è un commento su una sola riga
```

```
/*
   Questo è un commento
   su più righe
*/
```

```
/**
   Questo commento permette di produrre
   la documentazione del codice
   in formato HTML, nello standard Javadoc
*/
```

Strutture dati

- Regole per gli identificatori
- Tipi di Dato
  - tipi primitivi: int, double, etc
  - tipi non primitivi: reference
- Classi di Java
  - String
- Array

Regole per gli identificatori

- Identificatori:
  - nomi di classi, metodi, variabili, package, etc.
- Primo carattere
  - A-Z, a-z, _, $
- Altri caratteri
  - A-Z, a-z, _, $, 0-9
- Non possono essere una keyword java!!
  - ex `new`, `class`, etc
  - Elenco delle keyword sul libro

Tipi Primitivi

- Tipi interi
  - byte, short, int, long
- Tipi a virgola mobile
  - float e double
- Tipo testuale
  - char
- Tipo logico-booleano
  - boolean

Tipi Primitivi

Tipo	Dimensione	Valori
boolean	1-bit	true/false
char	16-bit Unicode	‘\n’
byte	signed 8-bit	-128…+127
short	signed 16-bit	-32768 … +32767
int	signed 32-bit	-2^{31} … + $2^{31}-1$
long	signed 64-bit	-2^{63} … + $2^{63}-1$
float	32-bit (IEEE-754)	3.4e+38 (7 decim.)
double	64-bit (IEEE-754)	1.7e+308 (15 decim)

Esplicitare il tipo

- il tipo va esplicitato:
  - 10 è un int (*default*)
  - 10 è anche uno short
  - 10 è anche un byte
  - 10L è un long
  - 10F è un float
  - 10D è un double
  - 1.0F è un float
  - 1.0 è un double (*default*)

Conversioni di tipo

- Conversioni automatiche se non c'è perdita di precisione:
  - da numeri interi a numeri in virgola mobile
  - fra interi di cardinalità minore a interi di cardinalità maggiore (int >> long)
  - da float a double
- Negli altri casi devo usare un "*casting*" esplicito
  - double >> int
  - (int)(5.5+0.4) vale 5

Riferimenti ad Oggetti

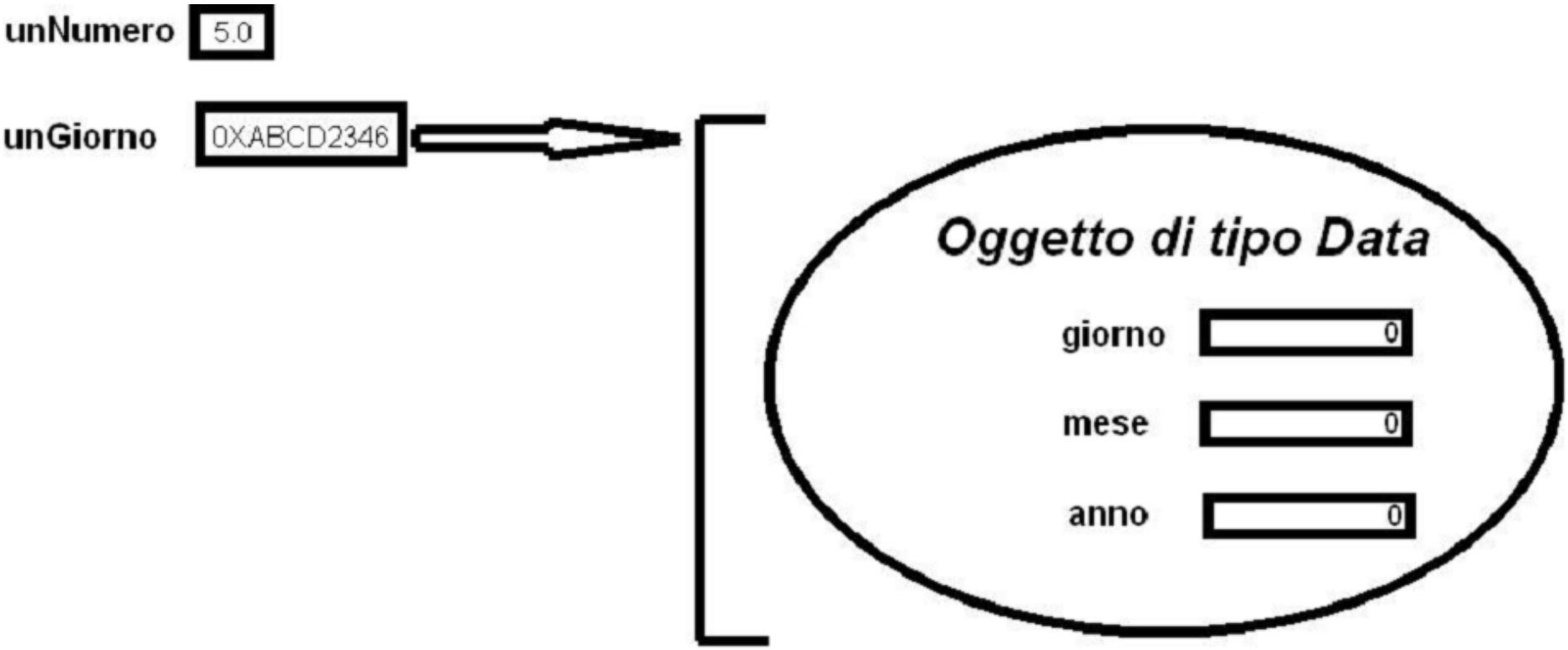
```
NomeClasse nomeOggetto;
```

- Simile alla dichiarazione di tipi primitivi
  - Il nomeOggetto è detto reference (riferimento)
  - Contiene un indirizzo di memoria

```
public class Data {
    public int giorno;
    public int mese;
    public int anno;
}
```

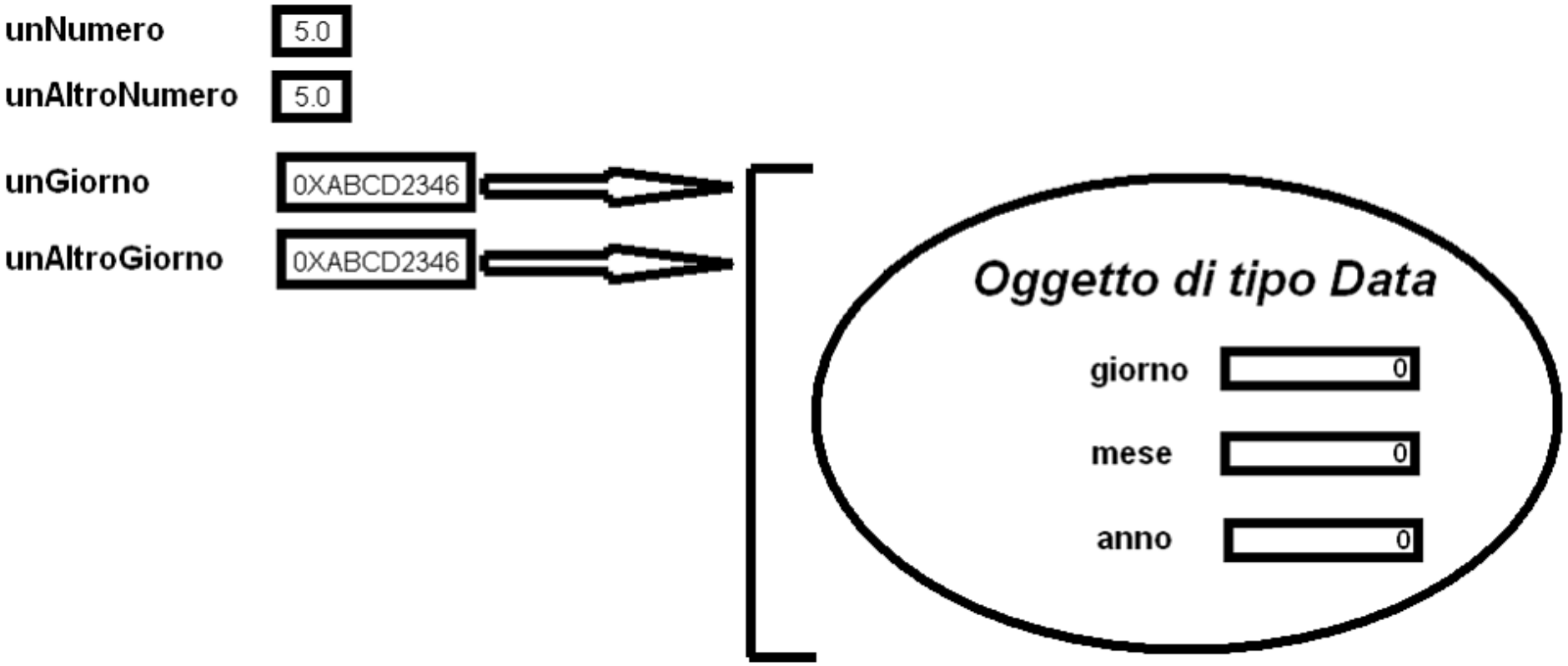
Schema di allocazione in memoria

```
double unNumero = 5.0;
Data unGiorno = new Data();
```



Schema di allocazione in memoria

```
double unNumero = 5.0;
double unAltroNumero = unNumero;
Data unGiorno = new Data();
Data unAltroGiorno = unGiorno;
```



Passaggio dei Parametri

- Il passaggio dei Parametri avviene per valore
  - Al metodo viene passato il valore della variabile

```
public class CiProvo
{
    public void cambiaValore(int valore)
    {
        valore = 1000;
    }
}
```

```
CiProvo ogg = new CiProvo();
int numero = 10;
ogg.cambiaValore(numero);
System.out.println("il valore del numero è " + numero);
```

```
il valore del numero è 10
```

Passaggio dei Parametri

```
public class CiProvoConIReference
{
    public void cambiaReference(Data data)
    {
        data = new Data();
        // Un oggetto appena istanziato
        // ha le variabili con valori nulli
    }
}
```

```
CiProvoConIReference ogg = new CiProvoConIReference();
Data dataDiNascita = new Data();
dataDiNascita.giorno = 26;
dataDiNascita.mese = 1;
dataDiNascita.anno = 1974;
ogg.cambiaReference(dataDiNascita);
System.out.println("Data di nascita = "
  + dataDiNascita.giorno + "-"+ dataDiNascita.mese
  + "-" +dataDiNascita.anno );
```

Passaggio dei Parametri

- Risultato

```
Data di nascita = 26-1-1974
```

- Non permetto ad un metodo di cambiare il "reference" dichiarato nel chiamamante
- Riscriviamo il metodo

```
public void cambiaReference(Data data)
{
    data.giorno=29; // data punta allo stesso indirizzo
    data.mese=7     // della variabile dataDiNascita
}
```

- Permetto al metodo di cambiare l'oggetto che è "puntato" dal reference

Librerie Standard

- java.io
  - contiene classi per realizzare l'input – output in Java
- java.awt
  - contiene classi per realizzare interfacce grafiche, (es. Button)
- java.net
  - contiene classi per realizzare connessioni, come Socket
- java.applet
  - contiene un'unica classe: Applet
- java.util
  - raccoglie classi d'utilità (es. Date)
- java.lang
  - è il package che contiene le classi nucleo del linguaggio, come System e String

Usare le Librerie

- Comando import

```
import java.util.Date;
```

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

- Osservazioni:
  - java.lang.* è importato di default
  - Lo * non importa i sottopackage

```
import java.awt.*;

public class FinestraConBottone {
  public static void main(String args[]) {
    Frame finestra = new Frame("Titolo");
    Button bottone = new Button("Cliccami");
    finestra.add(bottone);
    finestra.setSize(200,100);
    finestra.setVisible(true);
  }
}
```

La classe String

- Creare una stringa

```
String nome = new String("Mario Rossi");
```

- Oppure

```
String nome = "Mario Rossi";
```

- Le stringhe sono oggetti con metodi:
  - toUpperCase()
  - toLowerCase()
  - trim()
  - etc.

Oggetti Immutabili

```
String a = "claudio";
String b = a.toUpperCase();
System.out.println(a); // a rimane immutato
System.out.println(b); // b è la stringa maiuscola
```

- Produce

```
claudio
CLAUDIO
```

- I metodi di String non cambiano la stringa contenuta nell'oggetto:
  - Restituiscono un'altra stringa modificata
- Per gli altri metodi di String vedere la documentazione Java

Documentazione Java

- Comando javadoc
  - Genera codice html dal codice e dai commenti
- La documentazione Java è scritta con javadoc
  - Vedere http://java.sun.com/javase/6/docs/api/
- Esercizio
  - Scrivere commenti di tipo javadoc in una Classe
  - Eseguire
    - javadoc NomeClasse.java

```
/**
 Questo è un metodo!
*/
public void metodo(){
   . . .
}
```

Gli Array

- Dichiarare Array

```
char alfabeto [];          oppure          char [] alfabeto;
Button bottoni [];         oppure          Button [] bottoni;
```

- Creare Array

```
alfabeto = new char[21];
bottoni = new Button[3];
```

- Inizializzare Array

```
alfabeto [0] = 'a';
alfabeto [1] = 'b';
alfabeto [2] = 'c';
alfabeto [3] = 'd';
. . . . . .
alfabeto [20] = 'z';
```

```
bottoni [0] = new Button();
bottoni [1] = new Button();
bottoni [2] = new Button();
```

Gli Array

- Dichiarazione/Creazione/Inizializzatione

```
char alfabeto [] = {'a', 'b', 'c', 'd', 'e',…, 'z'};
Button bottoni [] = { new Button(), new Button(), new
Button()} ;
```

- Lunghezza di un Array

```
    alfabeto.length
```

Array Multidimensionali

- Sono Array di Array
  - Può non essere rettangolare!!!

```
int arrayNonRettangolare [][] = new int[4][];
arrayNonRettangolare [0] = new int[2];
arrayNonRettangolare [1] = new int[4];
arrayNonRettangolare [2] = new int[6];
arrayNonRettangolare [3] = new int[8];
arrayNonRettangolare [0][0] = 1;
arrayNonRettangolare [0][1] = 2;
arrayNonRettangolare [1][0] = 1;
. . . . . .
arrayNonRettangolare [3][7] = 10;
```

Limiti degli Array

- Non possono essere eterogenei
- Non si possono ridimensionare

- In realtà gli Array sono oggetti e quindi il seguente codice agisce solo sul riferimento

```
int mioArray [] = {1, 2, 3, 4};
mioArray = new int[6];
```

- Per copiare gli Array si usa arraycopy
  - Della classe System

- Vedremo classi Java che superano questi problemi

Elementi del linguaggio

- Operatori
- Istruzioni
  - If
  - while
  - for

Operatori

- Assegnazione

  =
- Aritmetici

  +, -, *, /, %

  +=, -=, *=, /=, %=
- Unari di pre e post incremento/decremento

  ++, --
- Bitwise

  ~, &, |, ^, <<, >>, >>>, &=, |=, ^=, <<=, >>=
- Relazionali

  ==, !=, >, <, <=, >=
- Booleani

  ! , &, |, ^, &&, ||, &=, |=, ^=

Operazioni su Stringhe

- Uguaglianza

```
String a = "Java";
String b = "Java";
String c = new String("Java");
System.out.println(a==b);
System.out.println(b==c);
```

```
System.out.println(a.equals(b));
System.out.println(b.equals(c));
```

- Concatenazione

```
String nome = "James ";
String cognome = "Gosling";
String nomeCompleto = "Mr. " + nome + cognome;
```

Istruzioni if, else

```
if (espressione-booleana) istruzione;
```

```
if (numeroLati == 3)
    System.out.println("Questo è un triangolo");
```

```
if (espressione-booleana) istruzione1;
else istruzione2;
```

```
if (numeroLati == 3)
    System.out.println("Questo è un triangolo");
else
    System.out.println("Questo non è un triangolo");
```

Istruzioni if, else

```
if (espressione-booleana) {
    istruzione_1;
    ..........;
    istruzione_k;
} else if (espressione-booleana){
    istruzione_k+1;
    ..............;
    istruzione_j;
} else if (espressione-booleana){
    istruzione_j+1;
    ..............;
    istruzione_h;
} else {
    istruzione_h+1;
    ..............;
    istruzione_n;
}
```

Operatore Ternario

- Può sostituire il costrutto if else

```
variabile = (espr-booleana) ? espr1 : espr2;
```

- true assegna expr1
- false assegna expr2

- Esempio

```
String query = "select * from table " +
    (condition != null ? "where " + condition : "");
```

Istruzione while

```
[inizializzazione;]
while (espr. booleana) {
    corpo;
    [aggiornamento iterazione;]
}
```

```
public class WhileDemo {
    public static void main(String args[]) {
        int i = 1;
        while (i <= 10) {
            System.out.println(i);
            i++;
        }
    }
}
```

Istruzioni do while

```
[inizializzazione;]
do {
    corpo;
    [aggiornamento iterazione;]
} while (espr. booleana);
```

```
public class DoWhile {
  public static void main(String args[]) {
    int i = 10;
    do {
      System.out.println(i);
    } while(i < 10);
  }
}
```

Istruzione for

```
for (inizializzazione; espr. booleana; aggiornamento)
{
    istruzione_1;
    ..........;
    istruzione_n;
}
```

```
public class ForDemo
{
    public static void main(String args[])
    {
        for (int n = 10; n > 0; n--)
        {
            System.out.println(n);
        }
    }
}
```

Istruzioni switch case

```
switch (variabile di test) {
    case valore_1:
        istruzione_1;
    break;
    case valore_2: {
        istruzione_2;
        ..........;
        istruzione_k;
    }
    break;
    case valore_3:
    case valore_4: {  //blocchi di codice opzionale
        istruzione_k+1;
        ..........;
        istruzione_j;
    }
    break;
    [default: {    //clausola default opzionale
        istruzione_j+1;
        ..........;
        istruzione_n;
    }]
}
```

Istruzioni break e continue

```
int i = 0;
while (true) //ciclo infinito
{
    if (i > 10)
        break;
    System.out.println(i);
    i++;
}
```

```
int i = 0;
do
{
    i++;
    if (i == 5)
        continue;
    System.out.println(i);
}
while(i <= 10);
```

Paradigmi della PO

- Incapsulamento
- Ereditarietà
- Polimorfismo

Incapsulamento

- Una classe contiene dati e metodi
- Filosofia
  - Ai dati si accede solo attraverso i metodi
  - I metodi devono prevenire dati non corretti
- Realizzazione
  - Si rendono privati gli attributi
  - Si antepone il modificatore private
  - Dei metodi public garantiscono l'accesso ai dati

Esempio

- Classe non incapsulata

```
public class Data {
    public int giorno;
    public int mese;
    public int anno;
}
```

- Uso degli attributi

```
...
Data unaData = new Data();
unaData.giorno = interfaccia.dammiGiornoInserito();
unaData.mese = interfaccia.dammiMeseInserito();
unaData.anno = interfaccia.dammiAnnoInserito();...
```

Esempio di Incapsulamento

```
public class Data {
    private int giorno;
    private int mese;
    private int anno;

    public void setGiorno(int g) {
       if (g > 0 && g <= 31) {
            giorno = g;
        }
        else {
            System.out.println("Giorno non valido");
        }
    }

    public int getGiorno() {
        return giorno;
    }

    public void setMese(int m) {
        if (m > 0 && m <= 12) {
            mese = m;
        }
        else {
            System.out.println("Mese non valido");
        }
    }

    public int getMese() {
        return mese;
    }

    public void setAnno(int a) {
        anno = a;
    }

    public int getAnno() {
        return anno;
    }
}
```

```
...
Data unaData = new Data();
unaData.setGiorno(interfaccia.dammiGiornoInserito());
unaData.setMese(interfaccia.dammiMeseInserito());
unaData.setAnno(interfaccia.dammiAnnoInserito());
...
```

Vantaggi dell'Incapsulamento

- Robustezza
  - Controllo l'accesso ai dati
- Indipendenza e Riusabilità
  - Le altre classi conoscono solo l'interfaccia
  - Non devono conoscere i dettagli interni
- Manutenzione
  - Posso riscrivere il corpo dei metodi senza cambiare l'interfaccia

```
public void setGiorno(int g) {
    if (g > 0 && g <= 31 && mese != 2) {
        giorno = g;
    }
    else {
        System.out.println("Giorno non valido");
    }
}
```

Altro esempio

```
public class Dipendente  {
    private String nome;
    private int anni; //intendiamo età in anni
    . . .
    public String getNome() {
        return nome;
    }
    public void setNome(String n) {
        nome = n;
    }
    public int getAnni() {
        return anni;
    }
    public void setAnni(int n) {
        anni = n;
    }
    public int getDifferenzaAnni(Dipendente altro) {
        return (anni - altro.anni);
   }
}
```

Incapsulamento funzionale

- Dichiaro dei metodi con il modificatore private
  - Possono essere visti solo negli altri metodi della classe

```
public class ContoBancario {
    . . .
    public String getContoBancario(int codiceDaTestare)
    {
        return controllaCodice(codiceDaTestare);
    }

    private String controllaCodice(int codiceDaTestare) {
        if (codiceInserito == codice) {
            return contoBancario;
        }
        else  {
            return "codice errato!!!";
        }
    }
}
```

Oggetti ed incapsulamento

- Se dichiaro un membro privato questo non è accessibile da altre classi
  - Ne dagli oggetti delle altre classi
- Due oggetti della stessa classe possono accedere ai membri privati in "modo pubblico"
  - È comunque preferibile non farlo
  - Usiamo comunque i getter

```
public int getDifferenzaAnni(Dipendente altro){
    return (getAnni() - altro.getAnni());
}
```

Il reference this

- La classe non conosce i reference che verranno dichiarati
- Per riferirsi all'oggetto corrente all'interno della definizione di una classe si usa this

```
public class Cliente
{
  . . .
  public void setCliente(String nome, String indirizzo,
  int numeroDiTelefono)
  {
    this.nome = nome;
    this.indirizzo = indirizzo;
    this.numeroDiTelefono = numeroDiTelefono;
  }
  . . .
}
```

Ereditarietà

- Partiamo da una classe "generale" e la estendiamo una o più volte particolareggiando le sue caratteristiche

```
public class Libro {
    public int numeroPagine;
    public int prezzo;
    public String autore;
    public String editore;
    . . .
}
public class LibroSuJava{
    public int numeroPagine;
    public int prezzo;
    public String autore;
    public String editore;
    public final String ARGOMENTO_TRATTATO = "Java";
    . . .
}
```

La parola chiave extends

```
public class LibroSuJava extends Libro {
    public final String ARGOMENTO_TRATTATO = "Java";
    . . .
}
```

- Tutti i membri pubblici della classe Libro sono ereditati dalla classe LibroSuJava
  - Posso usare numeroPagine, prezzo, …
- Si dice che
  - Libro è la superclasse (o classe padre) di LibroSuJava
  - LibroSuJava è la sottoclasse (o classe figlia) di Libro

Ereditarietà Multipla

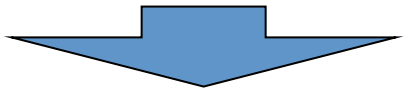
```
public class Idrovolante extends Nave, Aereo {
    . . .
}
```

- NON è permesso in Java estendere due classi!!
  - Potete estendere una sola classe
- Garantiamo la creazione di un albero
  - Questa scelta toglie della flessibilità al programmatore ma favorisce la robustezza
  - In C++ è permessa l'ereditarietà multipla

La Classe Object

- Nella libreria standard di java è definito il concetto di "oggetto generico"
  - java.lang.Object
- Ogni Classe java estende implicitamente la classe Object
  - Scopriremo di aver a disposizione dei metodi ereditati

```
public class Arte {
    . . .
}
```



In compilazione

```
public class Arte extends Object {
    . . .
}
```

Utilizzare l'ereditarietà

- Quando utilizzare l'ereditarietà?
  - Devo chiedermi se l'oggetto della candidata sottoclasse è un ("is a") oggetto della candidata superclasse
- Vediamo chi?

  Veicoli – Aerei

  Auto – Telaio

  Computer – Laptop

  Persona – Studente

  Quadrato – Rettangolo
- Generalizzazione
  - Se parto da alcune classi e raggruppo le caratteristiche comuni in una classe
- Specializzazione
  - Parto da una classe e derivo alcune sottoclassi

Ereditarietà ed Incapsulamento

- Se incapsulo una classe
  - Le variabili della classe non sono "visibili" della sottoclasse
  - Le variabili private non sono ereditate
- Se la superclasse espone i setter ed i getter
  - La sotto classe eredita questi metodi (se pubblici)
  - Accedo ai membri privati attraverso questi nella sottoclasse
- Quindi
  - La sottoclasse possiede degli attributi a cui può accedere attraverso dei metodi (ereditati)

Modificatore protected

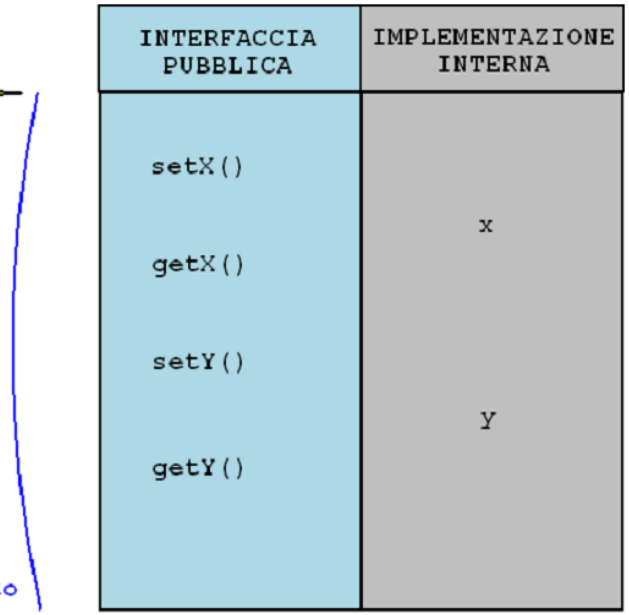
- Un membro dichiarato protected
  - È accessibile dalle classi del package
  - È accessibile dalle sottoclassi
    - Viene ereditato dalle sottoclassi

```
package package1;

public class Superclasse {
  protected void metodo(){

  }
}
```

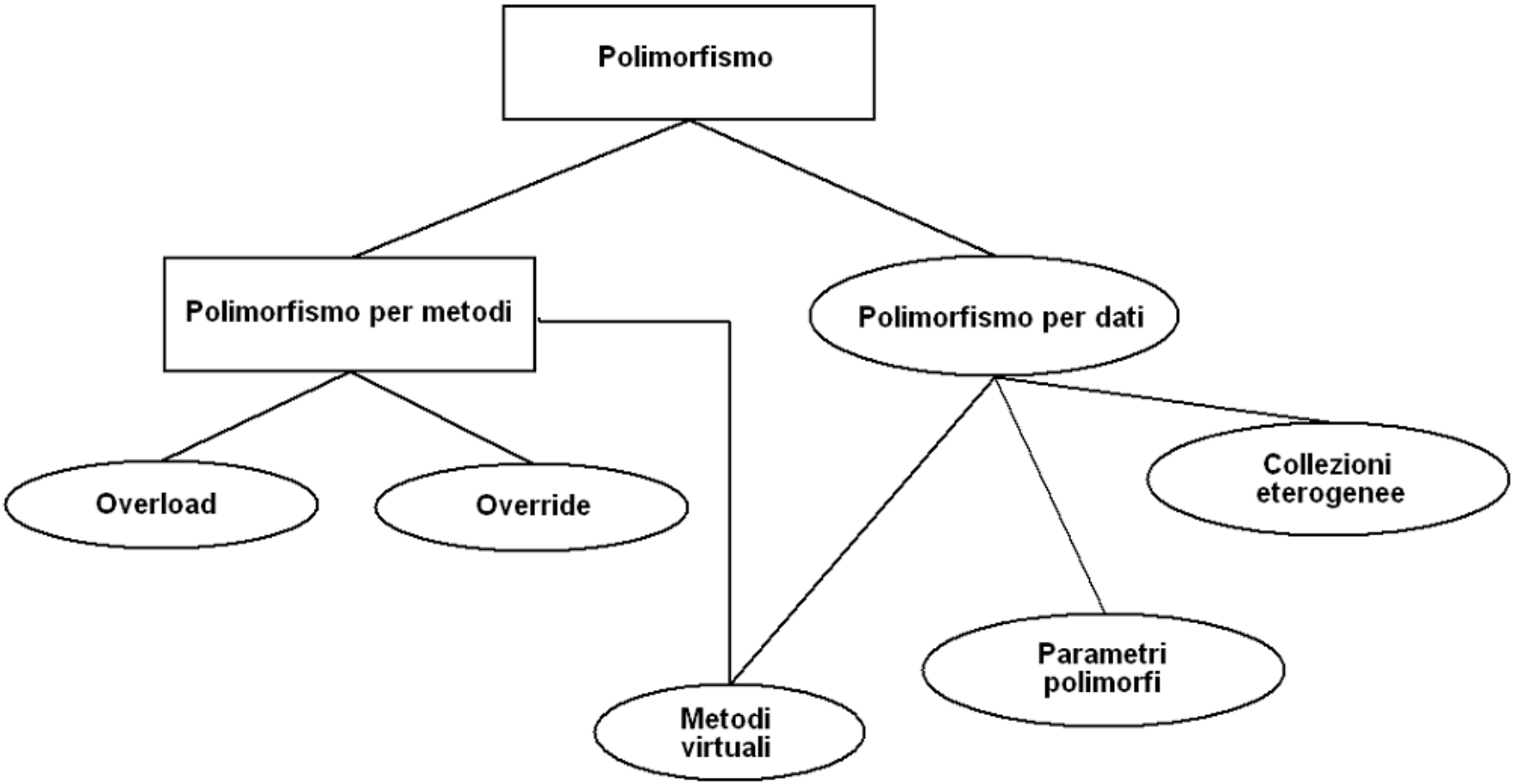
Ancora su i reference

```
public class Punto {
    private int x;
    private int y;
    public void setX(int x) {
        this.x = x;
    }
    public void setY(int y){
        this.y = y;
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
}
```

```
Punto ogg = new Punto();
```



Polimorfismo



Polimorfismo per Metodi: Overload

- Firma del metodo
  - La coppia identificatore – lista di parametri
  - Es. "public int somma(int a, int b)"
- Overload:
  - In una classe posso dichiarare metodi con lo stesso identificatore ma firma differente

- Tipicamente:
  - Sono metodi con la stessa funzionalità

Esempio di Overload

```
public class Aritmetica
{
    public int somma(int a, int b)
    {
        return a + b;
    }
    public float somma(int a, float b)
    {
        return a + b;
    }
    public float somma(float a, int b)
    {
        return a + b;
    }
    public int somma(int a, int b, int c)
    {
        return a + b + c;
    }
    public double somma(int a, double b, int c)
    {
        return a + b + c;
    }
}
```

Polimorfismo per Metodi: Override

- Override:
  - In una sottoclasse posso dichiarare metodi con la stessa firma della supercalsse

- Regole:
  - Utilizzare la stessa identica firma
    - altrimenti è un overload e non un override.
  - Il tipo di ritorno del metodo deve coincidere con quello del metodo che si sta riscrivendo.
  - Il metodo ridefinito non deve essere meno accessibile del metodo che ridefinisce

Classe Punto

```
public class Punto
{
    private int x, y;

    public void setX()
    {
        this.x = x;
    }
    public int getX()
    {
        return x;
    }
    public void setY()
    {
        this.y = y;
    }
    public int getY()
    {
        return y;
    }
    public double distanzaDallOrigine()
    {
        int tmp = (x*x) + (y*y);
        return Math.sqrt(tmp);
    }
}
```

Esempio di Override

```
public class PuntoTridimensionale extends Punto
{
     private int z;

     public void setZ()
     {
          this.z = z;
     }
     public int getZ()
     {
          return z;
     }

     public double distanzaDallOrigine()
     {
          int tmp = (getX()*getX()) + (getY()*getY())
            + (z*z); // N.B. : x ed y non sono ereditate
          return Math.sqrt(tmp);
     }
}
```

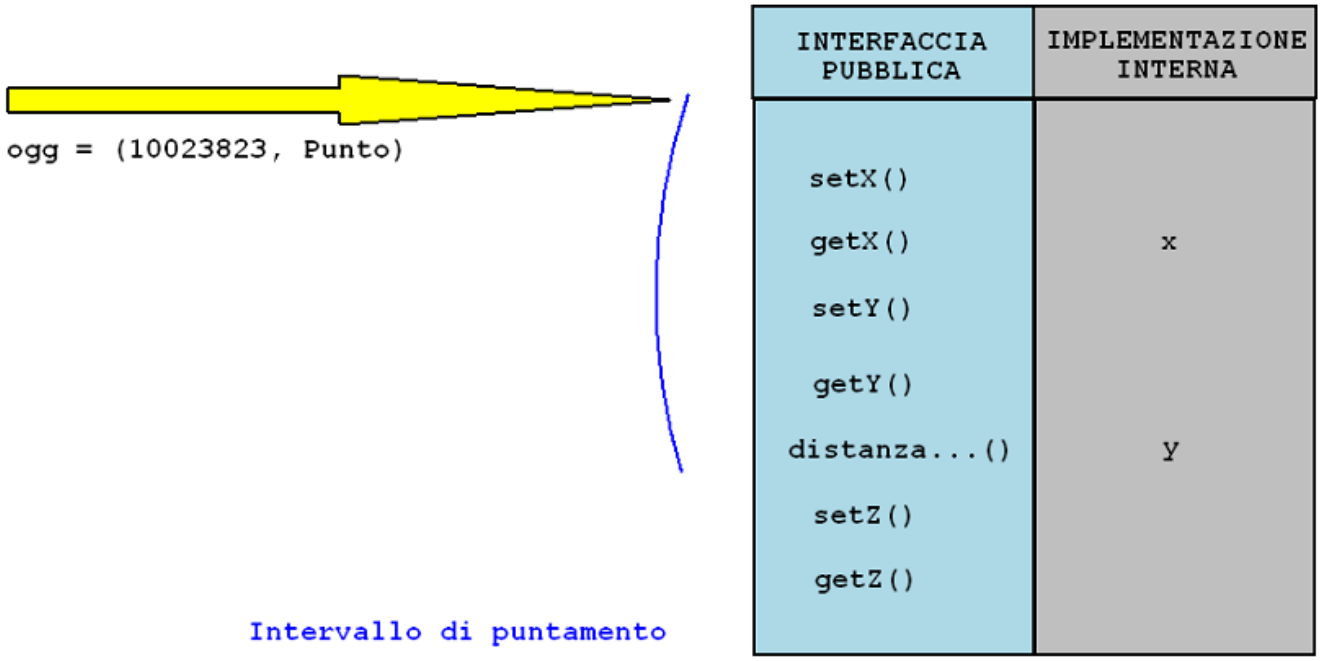
Override di Object

- toString()
  - Restituisce la stringa così formata: NomeClasse@IndirizzoInEsadecimale
- clone()
  - Duplica l'oggetto
- equals()
  - Controlla l'uguaglianza tra due oggetti
- hashCode()
  - Restituisce un int unico per ogni oggetto

Polimorfismo per Dati

- Possiamo assegnare ad un reference della superclasse una istanza della sottoclasse

```
Punto ogg = new PuntoTridimensionale();
```



- Non è lecito:

```
ogg.setZ(5);
```

Parametri polimorfici

- Quando chiamo un metodo il valore del riferimento viene passato al metodo
  - All'indirizzo passato può corrispondere un oggetto della sottoclasse

- Il metodo println accetta parametri della classe Object

```
Punto p1 = new Punto();
System.out.println(p1);
```

- Il corpo del metodo println invoca il metodo toString()
  - Se lo ho riscritto viene chiamato quello di Punto

Collezioni Eterogenee

- Insiemi di oggetti diversi
  - Es Array di Object

```
Object arr[] = new Object[3];
arr[0] = new Punto();     //arr[0], arr[1], arr[2]
arr[1] = "Hello World!"; //sono reference ad Object
arr[2] = new Date();      //che puntano ad oggetti
                          //istanziati da sottoclassi
```

  - oppure

```
Object arr[]={new Punto(),"Hello World!",new Date()};
```

Esempio di Collezione

```
public class Dipendente {
    public String nome;
    public int stipendio;
    public int matricola;
    public String dataDiNascita;
    public String dataDiAssunzione;
}

public class Programmatore extends Dipendente {
    public String linguaggiConosciuti;
    public int anniDiEsperienza;
}

public class Dirigente extends Dipendente {
    public String orarioDiLavoro;
}

public class AgenteDiVendita extends Dipendente {
    public String [] portafoglioClienti;
    public int provvigioni;
}
. . .
```

```
Dipendente [] arr = new Dipendente [180];
arr[0] = new Dirigente();
arr[1] = new Programmatore();
arr[2] = new AgenteDiVendita();
. . .
```

instanceof

- Fornisce la Classe dell'istanza

```
public void pagaDipendente(Dipendente dip) {
    if (dip instanceof Programmatore) {
        dip.stipendio = 1200;
    }
    else if (dip instanceof Dirigente){
        dip.stipendio = 3000;
    }
    else if (dip instanceof AgenteDiVendita) {
        dip.stipendio = 1000;
    }
    . . .
}
```

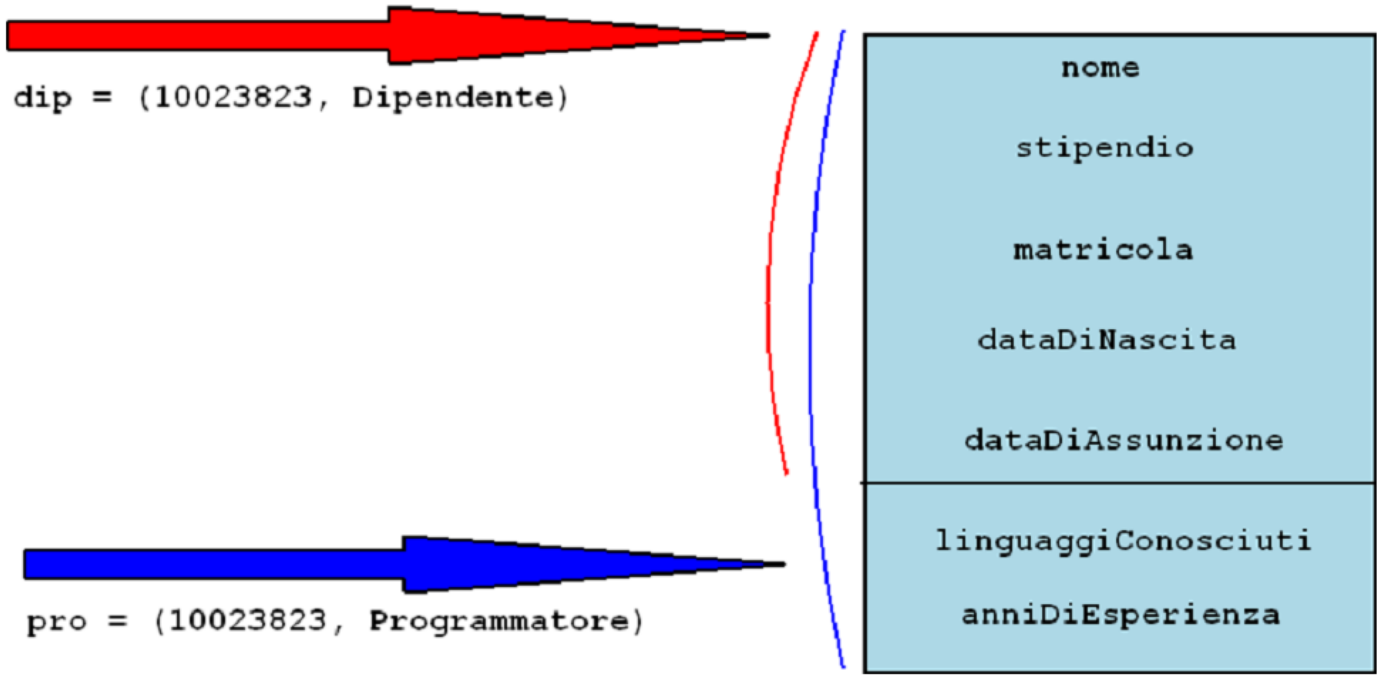
```
. . .
for (Dipendente dipendente : arr) {
    pagaDipendente(dipendente);
    . . .
}
```

Casting di Oggetti

- È lecito questo costrutto??

```
if (dip instanceof Programmatore) {
    Programmatore pro = (Programmatore) dip;
    . . .
```

- Associo un reference della sottoclasse all'oggetto



Metodi Virtuali

- Situazione
  - B è una sottoclasse di A
  - B ridefinisce (override) un metodo *m* di A
  - Ho creato una istanza di B
  - Invoco *m* con un reference di classe A
- Penso di invocare un metodo di A ma in realtà sto invocando il metodo di B
- Esempio

```
. . .
Object obj = new Date();
String s1 = obj.toString();
. . .
```

Inizializzare oggetti

```
public class Cliente
{
    private String nome;
    private String indirizzo;
    private String numeroDiTelefono;
    public void setNome(String n)
    {
        nome = n;
    }
    public void setIndirizzo(String ind)
    {
        indirizzo = ind;
    }
    public void setNumeroDiTelefono(String num)
    {
        numeroDiTelefono = num;
    }
    public String getNome()
    {
        return nome;
    }
    public String getIndirizzo()
    {
        return indirizzo;
    }
    public String getNumeroDiTelefono()
    {
        return numeroDiTelefono;
    }
}
```

Uso della Classe

```
Cliente cliente1 = new Cliente();
cliente1.setNome("James Gosling");
cliente1.setIndirizzo("Palo Alto, California");
cliente1.setNumeroDiTelefono("0088993344556677L");
```

Costruttori e polimorfismo

```
public class Cliente
{
    private String nome;
    private String indirizzo;
    private String numeroDiTelefono;
    //il seguente è un costruttore
    public Cliente(String n, String ind, String num)
    {
        nome = n;
        indirizzo = ind;
        numeroDiTelefono = num;
    }
    public void setNome(String n)
    {
        . . . . . .
```

Uso della Classe

```
Cliente cliente1 = new Cliente("James Gosling", "Palo
Alto, California",
  "0088993344556677");
```

Un codice migliore

```
public Cliente(String nome, String indirizzo, String
numeroDiTelefono)
{
    this.setNome(nome);
    this.setIndirizzo(indirizzo);
    this.setNumeroDiTelefono(numeroDiTelefono);
}
```

Costruttori ed Ereditarietà

```
public class Punto
{
    private int x,y;
    public Punto()
    {
        System.out.println("Costruito punto
bidimensionale");
    }
    . . .
    // inutile riscrivere l'intera classe
}

public class Punto3D extends Punto
{
    private int z;
    public Punto3D()
    {
        System.out.println("Costruito punto " +
          "tridimensionale");
    }
    . . .
    // inutile riscrivere l'ir
}
```

```
new Punto3D(); /* N.B. :L'assegnazione di un reference
                  non è richiesta per istanziare un
                  oggetto */
```

- Output:

  Costruito punto bidimensionale

  Costruito punto tridimensionale

Tutte le proprietà dei Costruttori

1. Hanno lo stesso nome della classe
2. Non hanno tipo di ritorno
3. Sono chiamati automaticamente (e solamente) ogni volta che viene istanziato un oggetto della classe cui appartengono, relativamente a quell'oggetto.
4. Sono presenti in ogni classe.
5. non sono ereditati dalle sottoclassi
6. un qualsiasi costruttore (anche quello di default), come prima istruzione, invoca sempre un costruttore della superclasse.

Il riferimento super

```
public class Persona
{
    private String nome, cognome;
    public String toString()
    {
        return nome + " " + cognome;
    }
    . . .
    //accessor e mutator methods (set e get)
}

public class Cliente extends Persona
{
    private String indirizzo, telefono;
    public String toString()
    {
        return super.toString()+ "\n"+
        indirizzo + "\nTel:" + telefono;
    }
    . . .
    //accessor e mutator methods (set e get)
}
```

- super è un reference implicito all'intersezione tra l'oggetto corrente e la sua superclasse
- Permette di accedere ai membri della superclasse anche se riscritti

Costruttori e super

```
public class Punto
{
    private int x, y;
    public Punto()
    {
        super(); //inserito dal compilatore
    }
    public Punto(int x, int y)
    {
        super(); //inserito dal compilatore
        setX(x); //riuso di codice già scritto
        setY(y);
    }
    . . .
}
public class Punto3D extends Punto
{
    private int z;
    public Punto3D()
    {
        super(); //inserito dal compilatore
    }
    public Punto3D(int x, int y, int z)
    {
        super(x,y); //Chiamata esplicita al
        //costruttore con due parametri interi
        setZ(z);
    }
    . . .
}
```

La chiamata al costruttore della superclasse è operata con super()

Posso inserire esplicitamente la chiamata con super(parametri) se devo selezionare il costruttore da chiamare

- Deve essere la prima istruzione nel costruttore della sottoclasse
- Se non esiste un costruttore di default sono obbligato a esplicitare la chiamata

Classi Innestate o Inner

```
public class Outer
{
    private String messaggio = "Nella classe ";
    private void stampaMessaggio()
    {
        System.out.println(messaggio + "Esterna");
    }
/* la classe interna accede in maniera naturale ai membri
   della classe che la contiene */
    public class Inner // classe interna
    {
        public void metodo()
        {
            System.out.println(messaggio + "Interna");
        }
        public void chiamaMetodo()
        {
            stampaMessaggio();
        }
        . . .
    }
    . . .
}
```

- Sono definite all'interno di un'altra classe
  - Anche in un metodo
- Caratteristiche
  - Hanno accesso alle variabili d'istanza
- Trovano impiego nelle GUI
- Proprietà sul libro

Classi Anonime

- Sono delle classi innestate senza "nome"
- Caratteristiche
  - Non hanno costruttore
  - Estendono un' altra classe
  - Si dichiarano quando si istanziano

```
public class Outer
{
    private String messaggio = "Nella classe ";
    private void stampaMessaggio()
    {
        System.out.println(messaggio+"Esterna");
    }
    //Definizione della classe anonima e sua istanza
    ClasseEsistente ca = new ClasseEsistente()
    {
        public void metodo()
        {
            System.out.println(messaggio+"Interna");
        }
    }; //Notare il ";"
    . . .
}
//Superclasse della classe anonima
public class ClasseEsistente
{
. . .
}
```

Modificatori

MODIFICATORE	CLASSE	ATTRIBUTO	METODO	COSTRUTTORE	BLOCCO DI CODICE
public	sì	sì	sì	sì	no
protected	no	sì	sì	sì	no
(default)	sì	sì	sì	sì	sì
private	no	sì	sì	sì	no
abstract	sì	no	sì	no	no
final	sì	sì	sì	no	no
native	no	no	sì	no	no
static	no	sì	sì	no	sì
strictfp	sì	no	sì	no	no
synchronized	no	no	sì	no	no
transient	no	sì	no	no	no

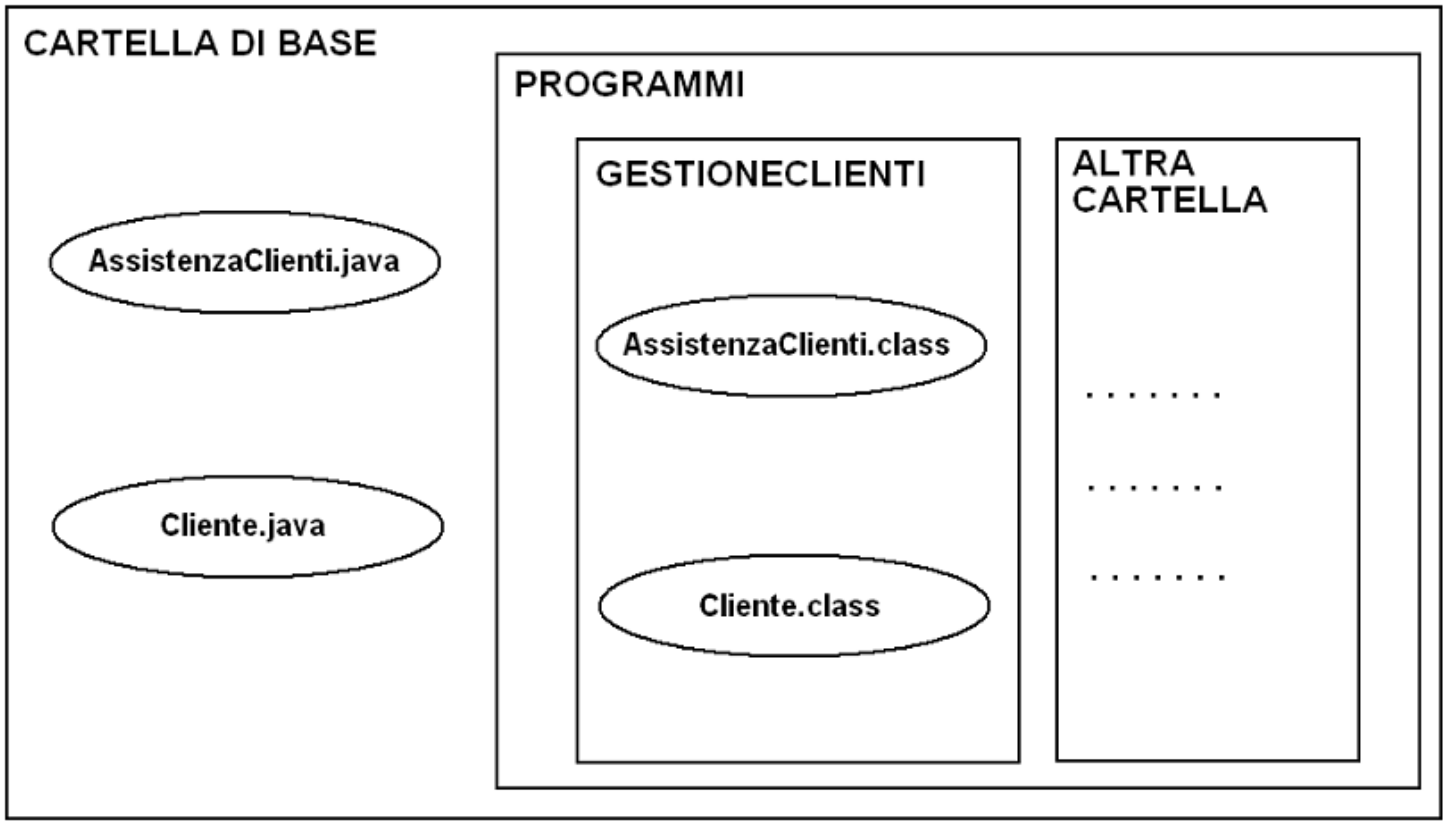
Modificatori di accesso

- public, (Default), protected e private

MODIFICATORE	STESSA CLASSE	STESSO PACKAGE	SOTTOCLASSE	DAPPERTUTTO
public	sì	sì	sì	sì
protected	sì	sì	sì	no
(default)	sì	sì	no	no
private	sì	no	no	no

Package

```
package programmi.gestioneClienti;
public class AssistenzaClienti
{
    . . . . .
```



java programmi.gestioneClienti.AssistenzaClienti

classpath

- È una variabile di ambiente
- Viene utilizzata in esecuzione per cercare classi
  - Devo dire alla JVM dove stanno le classi che uso nel programma
- Posso impostarla:
  - Per singola applicazione
  - Per tutte le applicazioni

```
java –cp .; C:\AltreClassiDaUtilizzare miopackage.MiaClasseConMain
```

File jar

- I file jar sono degli archivi java
  - Java ARchive
  - Fondamentalmente un file zip che contiene la cartella META-INF
- Per creare dei jar

```
jar cvf libreria.jar MiaCartella
```

- Includere nel classpath file jar

```
java –cp .; C:\cartellaConFIleJAR\MioFile.jar miopackage.MiaClasseConMain
```

Modificatore final

- Caratteristiche:
  - una variabile dichiarata final diviene una costante
  - un metodo dichiarato final non può essere riscritto in una sottoclasse (non è possibile applicare l'override)
  - una classe dichiarata final non può essere estesa

- Posso dichiarare final anche parametri e variabili locali di metodi

Modificatore static

- Se dichiaro del codice static questo è:
  - “condiviso da tutte le istanze della classe”
  - oppure diciamo solo “della classe”
- Per accedere a membri statici

```
NomeClasse.nomeMembro
```

- Esempio metodo

```
Math.sqrt(numero)
```

- Un metodo statico non può accedere alle variabili di istanza senza referenziarle

Variabili statiche

- Contiamo le istanze

```
public class Counter {
  private static int counter = 0;
  private int number;
  public Counter() {
    counter++;
    setNumber(counter);
  }
  public void setNumber(int number) {
    this.number = number;
  }
  public int getNumber() {
    return number;
  }
}
```

```
Counter c1 = new Counter();
Counter c2 = new Counter();
```

Inizializzatori statici

- Blocchi di codice definiti nella classe
  - È chiamato quando la classe (non gli oggetti) è caricata in memoria

```
public class EsempioStatico
{
    private static int a = 10;
    public EsempioStatico()
    {
        a += 10;
    }
    static
    {
        System.out.println("valore statico = " + a);
    }
}
```

```
EsempioStatico ogg = new EsempioStatico();
```

```
valore statico = 10
```

Modificatore abstract

- Metodi astratti
  - Non dichiarano il corpo del metodo
  - Lo deve dichiarare la sottoclasse
  - Esistono solo nelle classi astratte
- Classi Astratte
  - Non si possono istanziare

```
public abstract class Pittore
{
    . . .
   public abstract void dipingiQuadro();
    . . .
}
```

Interfacce

- Un interfaccia possiede :
  - tutti i metodi dichiarati public e abstract
  - tutte le variabili dichiarate public, static e final

```
public interface Saluto
{
    String CIAO = "Ciao";
    String BUONGIORNO = "Buongiorno";
    . . .
    void saluta();
}
```

```
public class SalutoImpl implements Saluto
{
    public void saluta()
    {
        System.out.println(CIAO);
    }
}
```

Ereditarietà Multipla

- Java simula l'ereditarietà multipla mediante le interfacce

```
public class MiaApplet extends Applet implements
MouseListener, Runnable
{
      . . .
}
```

Conversione di tipo

- Esempio
  - A e B sono due interfacce
  - B estende l'interfaccia A
  - C è una classe che implementa B
- Posso scrivere

```
B b = new C();
A a = b;
```

Differenze tra Classi Astratte e Interfacce

- Sia interfacce che classi astratte obbligano ad implementare dei comportamenti nelle sottoclassi
- Nelle classi astratte posso mettere del codice per le sottoclassi
- L'ereditarietà multipla si simula solo con le interfacce

- L'uso di entrambe le soluzioni è legato al polimorfismo

Enumerazioni

- Non le trattiamo

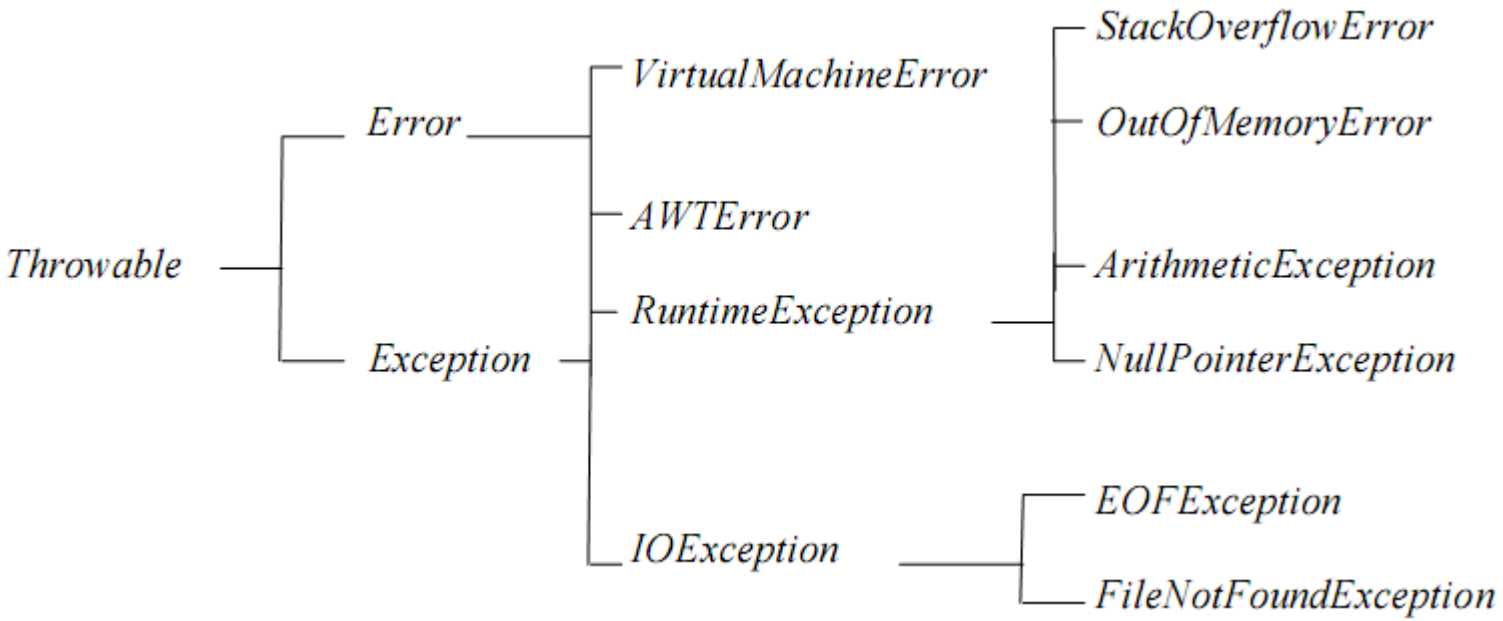
Modificatori di uso raro

- native
  - È legato a chiamate native al SO
- volatile
  - Serve per la gestione dei thread
- strictfp
  - Legato alle operazioni in virgola mobile

Eccezioni, errori ed asserzioni

- Le eccezioni sono delle situazioni impreviste che il flusso di una applicazione può incontrare
  - Gestite mediante try, catch, throws, throw, finally
  - Concetto implementato mediante **Exception**
- Gli errori sono delle situazioni impreviste che non dipendono dal programmatore.
  - Non sono gestibili
  - Implementate mediante **Error**
- Le asserzioni sono condizioni che devono essere verificate perché una parte di codice sia corretta
  - Gestite dalla parola chiave assert

Classi per eccezioni ed errori



Gestione delle Eccezioni

```
public class Ecc1 {
    public static void main(String args[]) {
        int a = 10;
        int b = 0;
        int c = a/b;
        System.out.println(c);
    }
}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Ecc1.main(Ecc1.java:6)
```

- Viene lanciata l'eccezione ma il programma è terminato

Catturare l'eccezione

```
public class Ecc2 {
    public static void main(String args[]) {
        int a = 10;
        int b = 0;
        try {
            int c = a/b;
            System.out.println(c);
        }
        catch (ArithmeticException exc) {
            System.out.println("Divisione per zero...");
        }
    }
}
```

- Stampare il contenuto di una eccezione

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (ArithmeticException exc) {
    exc.printStackTrace();
}
```

Eccezioni generiche

- Se gestisco l'eccezione "sbagliata" il programma termina

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (NullPointerException exc) {
    exc.printStackTrace();
}
```

- Posso usare il polimorfismo

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (Exception exc) {
    exc.printStackTrace();
}
```

Gestire più eccezioni

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (ArithmeticException exc) {
    System.out.println("Divisione per zero...");
}
catch (NullPointerException exc) {
    System.out.println("Reference nullo...");
}
catch (Exception exc) {
    exc.printStackTrace();
}
```

- La prima che corrisponde viene eseguita
  - Conta l'ordine in cui le scriviamo

finally

```
public class Ecc4 {
    public static void main(String args[]) {
        int a = 10;
        int b = 0;
        try {
            int c = a/b;
            System.out.println(c);
        }
        catch (ArithmeticException exc) {
            System.out.println("Divisione per zero...");
        }
        catch (Exception exc) {
            exc.printStackTrace();
        }
        finally {
            System.out.println("Tentativo di
operazione");
        }
    }
}
```

- Viene eseguito sempre
  - Serve per garantire il funzionamento di un blocco di codice

Creare eccezioni

- È possibile creare delle eccezioni
  - Si estende la classe Exception

```
public class PrenotazioneException extends Exception {
    public PrenotazioneException() {
// Il costruttore di Exception chiamato inizializza la
// variabile privata message
        super("Problema con la prenotazione");
    }
    public String toString() {
        return getMessage() + ": posti esauriti!";
    }
}
```

  - Devo dire a java quando lanciare questa eccezione

Lanciare eccezioni

- Creare l'oggetto eccezione
- Lanciarla con throw - sintassi:

```
PrenotazioneException exc = new PrenotazioneException();
throw exc;
```

- Esempio

```
try {
        //controllo sulla disponibilità dei posti
    if (postiDisponibili == 0) {
        //lancio dell'eccezione
        throw new PrenotazioneException();
    }
        //istruzione eseguita
        // se non viene lanciata l'eccezione
    postiDisponibili--;
}
catch (PrenotazioneException exc){
    System.out.println(exc.toString());
}
```

Esempio di propagazione

```
public class Botteghino {
  private int postiDisponibili;

  public Botteghino() {
    postiDisponibili = 100;
  }

  public void prenota() {
    try {
        //controllo sulla disponibilità dei posti
      if (postiDisponibili == 0) {
        //lancio dell'eccezione
        throw new PrenotazioneException();
      }
        //metodo che realizza la prenotazione
        // se non viene lanciata l'eccezione
      postiDisponibili--;
    }
    catch (PrenotazioneException exc){
      System.out.println(exc.toString());
    }
  }
}
```

Spesso dove viene generata una eccezione non si sa come gestirla

```
public class GestorePrenotazioni {
  public static void main(String [] args) {
    Botteghino botteghino = new Botteghino();
    for (int i = 1; i <= 101; ++i){
      botteghino.prenota();
      System.out.println("Prenotato posto n° " + i);
    }
  }
}
```

Esempio di propagazione

```
public class GestorePrenotazioni {
  public static void main(String [] args) {
    Botteghino botteghino = new Botteghino();
    try {
      for (int i = 1; i <= 101; ++i){
        botteghino.prenota();
        System.out.println("Prenotato posto n° " + i);

      }
    }
    catch (PrenotazioneException exc) {
      System.out.println(exc.toString());
    }
  }
}
```

Dichiaro il metodo per lanciare eccezioni

```
public void prenota() throws PrelievoException {
      //controllo sulla disponibilità dei posti
    if (postiDisponibili == 0) {
      //lancio dell'eccezione
      throw new PrenotazioneException();
    }
      //metodo che realizza la prenotazione
      // se non viene lanciata l'eccezione
    postiDisponibili--;
}
```

Eccezioni ed override

- Riscrivendo un metodo
  - Non posso aggiungere clausole throws
  - Se c'è la devo includere
  - Se c'è posso specificare come eccezione una sottoclasse dell'eccezione dichiarata

```
public class ClasseBase {
    public void metodo() throws java.io.IOException { }
}

class SottoClasseCorretta1 extends ClasseBase {
    public void metodo() throws java.io.IOException {}
}

 class SottoClasseCorretta2 extends ClasseBase {
    public void metodo() throws
java.io.FileNotFoundException {}
}

 class SottoClasseCorretta3 extends ClasseBase {
    public void metodo() {}
}

 class SottoClasseScorretta extends ClasseBase {
   public void metodo() throws java.sql.SQLException {}
}
```

Asserzioni

- Sintassi

```
1. assert espressione_booleana;
2. assert espressione_booleana: espressione_stampabile;
```

- Esempio

```
assert b > 0;
```

- Che è equivalente a

```
if (!(b>0)) {
   throw new AssertionError();
}
```

- Altro esempio

```
assert b > 0: "il valore di b è " + b;
```

Abilitare le asserzioni

- Per abilitare le asserzioni in esecuzione

```
java –ea MioProgrammaConAsserzioni
```

- Normalmente sono disabilitate

- Abilitazione parziale

```
java –ea –da:miopackage... MioProgramma
```