

INTRODUZIONE A JAVA

Pietro Di Lena

Università di Bologna

Credits: basate su slide create dal Prof. Angelo Di Iorio

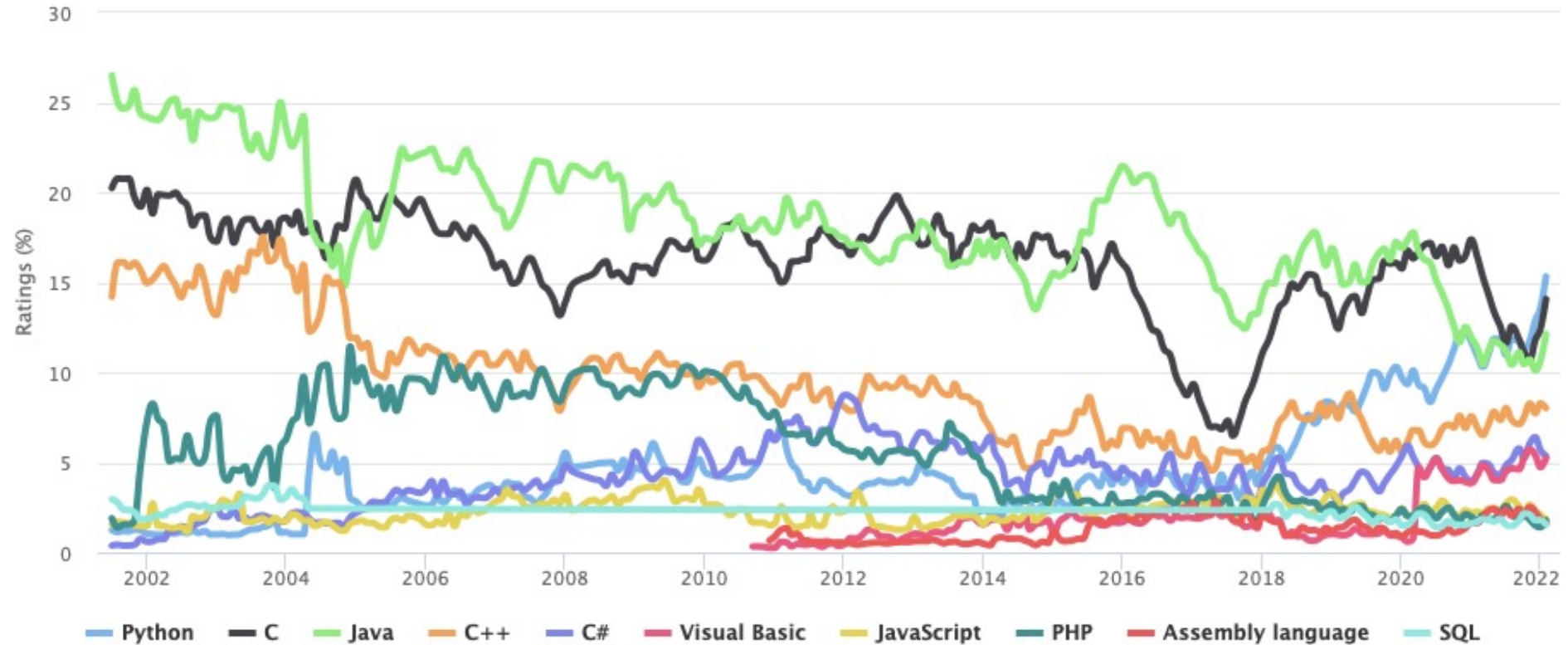
Java in questo corso

- Usiamo Java per implementare strutture dati e algoritmi
- Impariamo ad usare strutture dati built-in nel linguaggio
- Non è un corso completo su OOP e Java
 - trattiamo solo alcuni aspetti utili per il progetto
- Assumiamo conoscenze di base di:
 - programmazione e linguaggi procedurali
 - principi di programmazione a oggetti (ma approfondiremo)
- Ambiente di sviluppo: quello che preferite
 - preferenza del docente: terminale Unix
 - IDE suggerita: Eclipse (<http://www.eclipse.org/ide/>)

Java in queste prime lezioni























- HelloWorld e ambiente di sviluppo
- Programmazione Object-Oriented
- Organizzazione classi e packages
- Tipi di dato
- **Interfacce**
- **Java Generics**
- Java Collections

Perché Java: TIOBE index



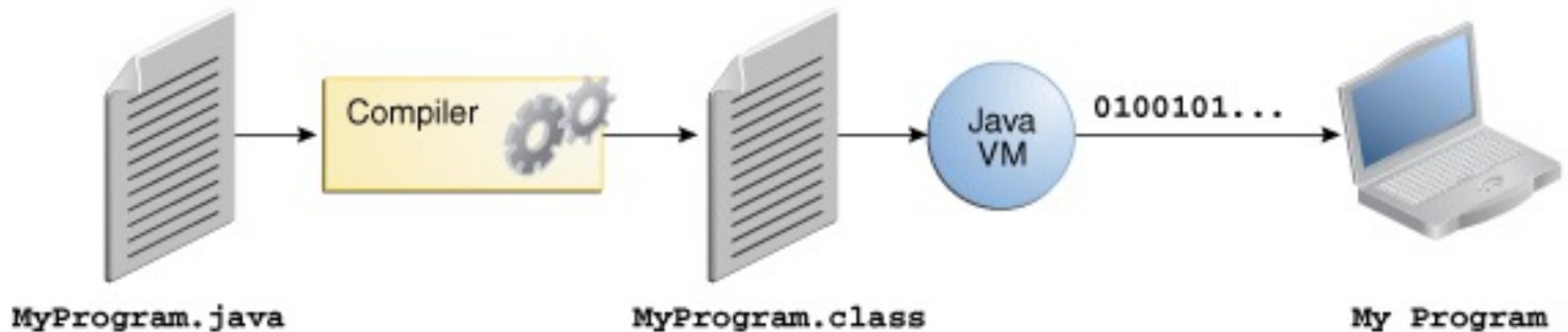
<https://www.tiobe.com/tiobe-index/>

Perché Java: IEEE spectrum

Rank	Language	Type	Score
1	Python [▼]	  	100.0
2	Java [▼]	  	95.4
3	C [▼]	  	94.7
4	C++ [▼]	  	92.4
5	JavaScript [▼]		88.1
6	C# [▼]	   	82.4
7	R [▼]		81.7
8	Go [▼]	 	77.7
9	HTML [▼]		75.4
10	Swift [▼]	 	70.4

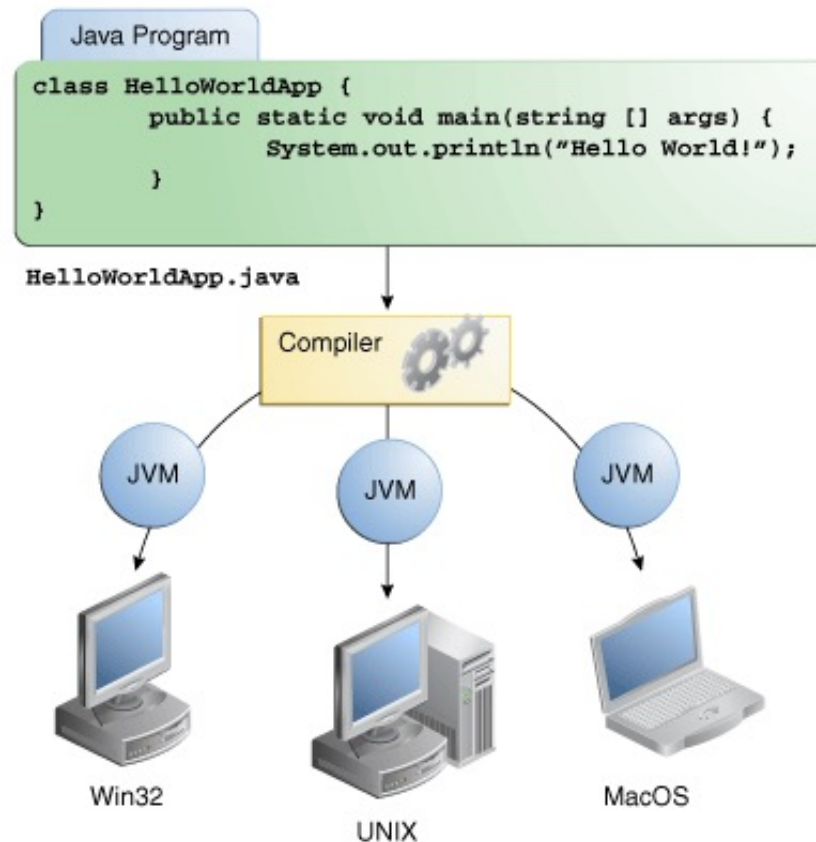
<https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020>

Java Development Process



<http://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html>

Java Virtual Machine



<http://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html>

Esercizio 0: Hello World in Java

Dichiarazione di classe

Tutti i programmi Java sono organizzati in classi.

`public class HelloWorld {`

Dichiarazione di metodo

Il metodo `main(...)` è invocato al "run" della classe e prende in input i parametri dell'esecuzione

`public static void main(String[] args) {
 System.out.println("Hello World!");
}`

Invocazione di metodo

`println(...)`

`}`

Operatori e strutture di controllo

- Sintassi simile al C per operatori e istruzioni di base:
 - operatori aritmetici, booleani, di confronto, etc.
 - strutture di controllo condizionali: `if`, `else`, `switch`, etc.
 - strutture di controllo iterative: `while`, `for`, etc.
 - Assegnamenti
- Impareremo la loro sintassi tramite esempi

Tipi in Java

- Java è un linguaggio **fortemente tipizzato** (strongly-typed)
 - ogni espressione ha un tipo, che il compilatore usa per controllare la correttezza delle operazioni eseguite
- Il tipo stabilisce l'insieme di valori possibili e di operazioni eseguibili su una variabile:
 - è obbligatorio indicarlo quando si dichiara una variabile
- Due tipi principali:
 - **tipi primitivi**: forniti dal linguaggio (dimensione fissa)
 - **tipi classe**: definiti dal programmatore, associati ad una classe. Descrivono come memorizzare gli oggetti di quella classe e i loro metodi

Sintassi Java

keyword per indicare
una "costante"

```
final int KINGS_OF_ROME = 7;
```

```
boolean found;
```

```
int a = 8;      inizializzazione di default (false)
```

```
int b = 2;
```

```
int sum;
```

```
short c = 10000; // 100000 ?
```

```
sum = a + b + c;
```

dichiarazione e assegnamento

classe HelloVariables

Commenti?

```
int n = 5;

somma = 0;

for (int i = 1; i <= n; i++) {
    somma += i;
}

System.out.println("Somma: " + somma);
```

Tipi nativi e possibili valori

Tipo	Dim. bit	Default	Min	Max
boolean	1	false	false	true
byte	8	0	-128	127
char	16	'\u0000'	'\u0000' (0)	'\uffff' ($2^{16}-1$)
short	16	0	-2^{15}	$2^{15}-1$
int	32	0	-2^{31}	$2^{31}-1$
long	64	0L	-2^{63}	$2^{63}-1$
float	32	0.0f	$\sim -3.4\text{E}+38$	$\sim 3.4\text{E}+38$
double	64	0.0d	$\sim -1.60\text{E}+308$	$\sim 1.60\text{E}+308$

Object-Oriented Programming

- Java è nativamente object-oriented
- La programmazione orientata agli oggetti (OOP) è un paradigma di programmazione pensato per rendere più semplice lo sviluppo, l'evoluzione, la manutenzione e il testing del software
- Un paradigma diverso da quello imperativo (occorre ragionare in modo diverso!)
- Fondamentale per scrivere applicazioni grandi ma utile anche per applicazioni piccole
- Favorisce il riuso

OOP: classi e oggetti

- Un programma è visto come un **insieme di oggetti** che:
 - contengono **valori** e le **funzioni (metodi)** che possono essere invocate per modificare tali valori
 - interagiscono tra di loro
- Un **oggetto** è un'**istanza** di una **classe**
- Una **classe** è un **tipo di dato** definito dal programmatore
 - definisce **campi** e **metodi** degli oggetti di quella classe
- Le classi consentono di:
 - definire il comportamento del dato
 - nascondere l'implementazione (***incapsulamento***)

Incapsulamento

- Un oggetto contiene una parte *pubblica* e una parte *privata* (anche una *protected*, ci torneremo a breve)
 - la parte pubblica è visibile da chi usa l'oggetto
 - la parte privata no
- La parte privata è incapsulata (*information hiding*)
- Si realizza così una netta separazione tra **interfaccia** e **implementazione**
- Se si modifica la parte privata lasciando inalterata quella pubblica si può continuare ad interagire con la parte pubblica senza dover apportare modifiche al codice

Visibilità

- I metodi, e in generale le classi e le variabili di istanza, possono avere diversi livelli di accesso, definiti con apposite keyword prima del nome del metodo:
 - **public**: il metodo può essere invocato da tutte le altre classi
 - **protected**: il metodo può essere invocato da:
 - sottoclassi della classe corrente (ereditarietà)
 - classi appartenenti allo stesso package
 - **private**: il metodo può essere invocato solo da altri metodi della stessa classe; né sottoclassi né altri classi possono accedervi

Esempio: Rettangolo

- Esercizio: definire una classe Java per descrivere un *rettangolo*, che ha una *larghezza* e un'*altezza* (per semplicità, valori interi) e del quale si vuole poter calcolare *perimetro* e *area*.
- Due possibili implementazioni nelle prossime slide

```
package rettangoli
```

```
public class Rettangolo {  
    private int l;  
    private int h;  
  
    public Rettangolo(int l, int h) {  
        this.h = h;  
        this.l = l;  
    }  
  
    public int getPerimetro() {  
        return (this.l+this.h)*2;  
    }  
  
    public int getArea() {  
        return (this.l*this.h);  
    }  
}
```

package rettangoli

```
public class Rettangolo2 {  
    private int area;  
    private int perimetro;  
  
    public Rettangolo2(int l, int h) {  
        this.area = l*h;  
        this.perimetro = (l+h)*2;  
    }  
  
    public int getPerimetro(){  
        return this.perimetro;  
    }  
  
    public int getArea(){  
        return this.area;  
    }  
}
```

package rettangoli

Istanziare oggetti: costruttore

- Il costruttore è un particolare metodo di una classe che viene invocato al momento della creazione di un oggetto e permette di inizializzare lo stato interno dell'oggetto
- In Java il costruttore deve avere lo stesso nome della classe e non bisogna specificare il tipo di ritorno (ritorna un riferimento all'oggetto appena creato)
- Possono esistere più costruttori con lo stesso nome a patto che abbiano un diverso numero di parametri
- La keyword `new` permette di istanziare un nuovo oggetto invocando il costruttore di una classe

Istanziare oggetti e invocare metodi

```
public class TestRettangoli {  
  
    public static void main(String[] args) {  
        Rettangolo r1 = new Rettangolo(20, 50);  
        Rettangolo2 r2 = new Rettangolo2(80, 10);  
  
        System.out.println("Area di r1: " + r1.getArea());  
  
        System.out.println("Perimetro di r2: " + r2.getPerimetro());  
    }  
}
```

package rettangoli

Tipi: wrapper e automatic (un)boxing

- In Java è spesso utile, e in alcuni casi obbligatorio, lavorare su oggetti e non su variabili di tipi primitivi
- Java definisce una classe *wrapper* per ogni tipo primitivo
- Queste classi espongono diversi metodi per operare sui dati

<i>Base Type</i>	<i>Class Name</i>	<i>Creation Example</i>	<i>Access Example</i>
boolean	Boolean	obj = new Boolean(true);	obj.booleanValue()
char	Character	obj = new Character('Z');	obj.charValue()
byte	Byte	obj = new Byte((byte) 34);	obj.byteValue()
short	Short	obj = new Short((short) 100);	obj.shortValue()
int	Integer	obj = new Integer(1045);	obj.intValue()
long	Long	obj = new Long(10849L);	obj.longValue()
float	Float	obj = new Float(3.934F);	obj.floatValue()
double	Double	obj = new Double(3.934);	obj.doubleValue()

```
System.out.println("Min Integer:" + Integer.MIN_VALUE);  
System.out.println("Max Float:" + Float.MAX_VALUE);
```

```
Integer a = 8; // boxing  
int      b = new Integer(2); // unboxing  
Integer c = Integer.parseInt("-10");  
Double  d = Double.parseDouble("3.14");
```

```
Double sum = a + b + c + d;
```

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

```
if (a.compareTo(b) > 0)      System.out.println("a > b");  
else if (a.compareTo(b) < 0) System.out.println("b > a");  
else                        System.out.println("a = b");
```


ORGANIZZAZIONE FILE E PACKAGE

Organizzazione file e packages

- Java usa un approccio semplice ma funzionale per organizzare i file e le classi di un programma: ogni classe è memorizzata in un file separato, con lo stesso nome ed estensione `.java`
- Classi e definizioni (es. enumerazioni) diverse possono essere raggruppate in **packages**
- I nomi dei package sono solitamente in minuscolo e tutti i file appartenenti allo stesso package sono nella stessa directory e contengono l'istruzione

```
package nomePackage;
```

- I package sono organizzati in subpackage, secondo un principio gerarchico

Import e accesso ai package

- Per fare riferimento ad un nome in un package si usa un nome qualificato tramite l'operatore .

```
System.out.println("Hello World")  
java.util.Scanner(...)
```

- Per includere classi da un altro package si usa la keyword `import`

```
import java.util.Scanner;
```

```
...
```

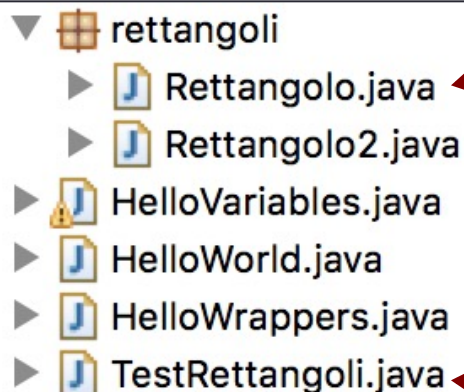
```
Scanner input = new Scanner(System.in);
```

- Si può importare un intero package se ci si aspetta di usare molte classi (più dispendioso)

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

Esempio

- Le classi `Rettangolo` e `Rettangolo2` sono organizzate in un package `rettangoli`
- Necessario:
 - Dichiarare l'appartenza al package in ogni classe
 - Importare le classi per usarle in altri package (es. in `TestRettangoli`)



```
package rettangoli;  
  
public class Rettangolo {  
    ...  
}
```

```
import rettangoli.Rettangolo;  
import rettangoli.Rettangolo2;  
  
public class TestRettangoli {  
    ...  
}
```

ALTRI TIPI DI DATO UTILI

Classe String

- Java ha una classe built-in per lavorare sulle stringhe: `String`
- `String` è usato per memorizzare sequenze di caratteri, eventualmente vuote
- Sulle istanze di `String` è possibile fare operazioni comuni:
 - concatenazione: operatore `+`
 - calcolo della lunghezza: `.length()`
 - estrazione di sottostringhe: `substring(...)`
 - etc.
- API e dettagli:
<https://docs.oracle.com/javase/8/docs/api/java/lang/String.html>

Enumeration

- Le enumerazioni sono usate per rappresentare scelte tra un set finito di valori

```
public enum Seme{ CUORI, QUADRI, PICCHE, FIORI };
```

- Una enumerazione è una classe speciale, quindi un tipo
- I valori elencati sono nomi di oggetti pubblici (e statici)

```
Seme s = Seme.CUORI;
```

- Possono essere quindi passati come parametro

Esercizio

- Implementare in Java una classe `Persona`, caratterizzata da: nome, cognome e cittadinanza, dove la cittadinanza può essere scelta tra 'Italiana', 'EU', 'ExtraEU'.
- La cittadinanza deve essere implementata tramite l'enumerazione `Cittadinanza`.
- La classe `Persona` implementa due metodi
 - costruttore
 - `getInfo()`: ritorna una stringa di informazioni, ad es.
"Mi chiamo Mario Rossi e ho cittadinanza EU"

```
package persone
```


Vettori in Java

- Un array (vettore) è un contenitore:
 - di dimensione fissa
 - di elementi dello stesso tipo

- Per inizializzare un vettore in Java:

```
Integer[] integers = new Integer[10];
```

```
Integer[] integers = {10, 2, 4, 5, 6};
```

```
String[] words = {"cane", "gatto", "tacchino"};
```

- La classe *java.util.Arrays* (statica) espone metodi per accedere e manipolare vettori

[ci torneremo]

EREDITARIETA'

Ereditarietà

- Spesso è utile definire una classe **in relazione** ad un'altra
- Ci sono molti casi infatti in cui vale la relazione **is-a** tra classi:
 - uno studente è anche una persona
 - un felino è un animale
 - un'auto è un mezzo di trasporto
 - ...
- Se abbiamo già definito una classe `Persona` è utile/comodo riusare il codice già scritto in una nuova classe `Studente`:
 - `Studente` ha tutte proprietà di `Persona`
 - `Studente` ha tutti i metodi di `Persona`, eventualmente modificati
 - `Studente` può avere altri metodi e altre proprietà
- Uno studente ha nome, cognome, indirizzo, telefono, etc. (come persona) ma ha anche una lista di esami dati e una media voti

Esempio Football



Terminologia di base

- La classe che eredita attributi e metodi si chiama **classe derivata**, o **sottoclasse**, o **classe figlio**
- La classe di partenza è la **classe base**, o **classe padre**, o **superclasse**
- Nell'esempio:
 - classe base: `Person`
 - classe derivata: `FootballPlayer`
 - classe derivata: `Goalkeeper`
- Ogni sottoclasse può avere a sua volta sottoclassi, in modo da creare una gerarchia

In Java

- Java usa la keyword `extends` per indicare ereditarietà
- La keyword `super` permette di accedere a variabili istanza e metodi della classe padre
- Ogni classe Java eredita dalla classe `Object` (ci torneremo a breve)
- Java NON supporta ereditarietà multipla

Variabili di istanza e metodi ereditati

- Ogni **variabile** istanza della classe padre è anche una variabile istanza della sottoclasse
- NON bisogna dichiararle di nuovo
- **Anche se formalmente lecito (diventano variabili diverse) è visto come cattivo stile di programmazione.**
- Le variabili di istanza possono o meno essere accedute dai metodi delle sottoclassi a seconda del valore del loro modificatore (*private*, *protected* o *public*)
- Anche i metodi, ad eccezione di quelli privati, sono ereditati dalle sottoclassi che possono quindi invocarli direttamente

Overriding

- Se un metodo di una classe derivata ha **lo stesso nome e gli stessi parametri** (in termini di tipo, ordine e numero) di un metodo della superclasse, esso **ridefinisce** il comportamento della classe
- Java eseguirà il codice del metodo nella sottoclasse (con alcune regole ed eccezioni che vedremo)
- Questo comportamento si definisce **overriding** e va **distinto** da **overloading** che invece permette di definire uno stesso metodo ma con parametri diversi
- Come per i costruttori, il metodo della sottoclasse può invocare il corrispondente metodo della superclasse usando la keyword `super`

`super.method(...)`

Object ed ereditarietà

- In Java ogni classe deriva dalla classe `Object`
- `Object` definisce alcuni metodi che sono quindi ereditati da tutte le classi
 - `public String toString()`
 - `public boolean equals(Object obj)`
 - `public int hashCode()`
 - ...
- Questi metodi sono troppo generici e potrebbe quindi essere utile ridefinirli in modo appropriato
- **Attenzione alla signature del metodo: i tipi devono essere esattamente gli stessi**
 - `public boolean equals(Object obj){...}`
 - `public boolean equals(Persona persona){...}`

Entrambi metodi di
Persona

Metodo toString()

- Il metodo `toString()` non ha argomenti e restituisce tutte le informazioni relative ad un oggetto in una stringa
- Viene invocato ogni volta che si “stampa” un oggetto
- Il comportamento di default, ereditato da `Object`, produce un risultato poco utile e chiaro, per cui di solito se ne ridefinisce il comportamento (overriding)

toString() derivato da Object

```
public class TestPersone {  
  
    public static void main(String[] args) {  
  
        Persona mrossi    = new Persona("Mario Rossi");  
        Persona gbianchi = new Persona("Giuseppe Bianchi",  
                                         Cittadinanza.ExtraEU);  
  
        System.out.println(mrossi);  
        System.out.println(gbianchi);  
    }  
}
```

```
persone.Persona@677327b6  
persone.Persona@14ae5a5
```

Override di toString()

```
public String toString(){  
    return "Mi chiamo " + this.nome +  
           " e ho cittadinanza " + this.cittadinanza;  
}
```

```
public class TestPersone {  
    public static void main(String[] args) {  
        Persona mrossi = new Persona("Mario Rossi");  
        Persona gbianchi = new Persona("Giuseppe Bianchi",  
                                         Cittadinanza.ExtraEU);  
  
        System.out.println(mrossi);  
        System.out.println(gbianchi);  
        ...  
    }  
}
```

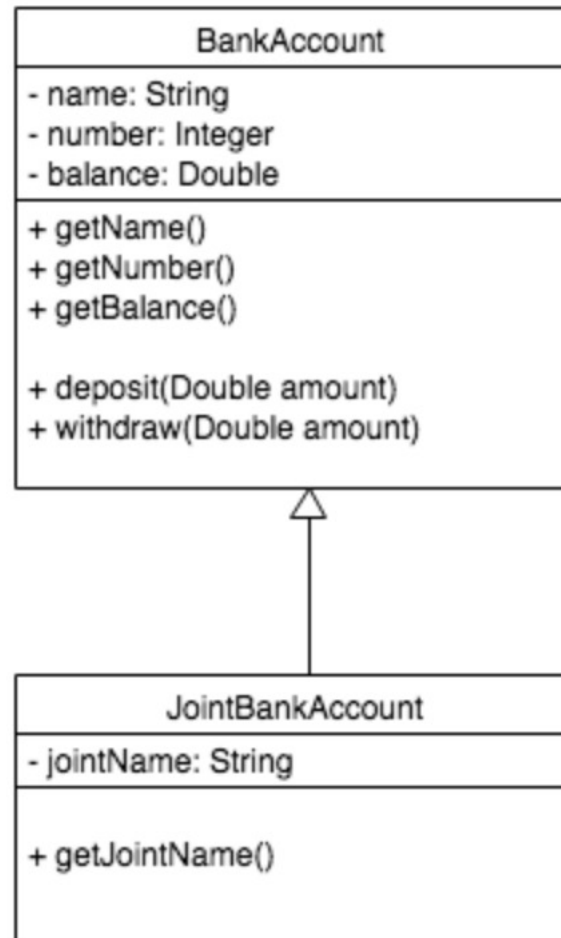
```
Mi chiamo Mario Rossi e ho cittadinanza Italiana  
Mi chiamo Giuseppe Bianchi e ho cittadinanza ExtraEU
```

POLIMORFISMO

Polimorfismo

- Il concetto di **ereditarietà** è strettamente legato a quello di **polimorfismo**
- Letteralmente, **polimorfismo** vuol dire **molte forme**
- Qui usiamo il termine per indicare la possibilità di associare più significati (forme) al nome di un metodo
- In una gerarchia di classi, lo stesso metodo infatti può avere **forme diverse**, una per ogni sottoclasse che lo sovrascrive (overriding)
- Questi **metodi** sono detti **polimorfici**
- **Un oggetto può comportarsi come se fosse di più tipi in virtù dell'ereditarietà**

Esempio BankAccount



- Nota: il diagramma non mostra i costruttori e la classe `Object` da cui derivano tutte le classi Java

```
public class BankAccount {  
  
    private int number;  
    private String name;  
    private double balance;  
  
    public BankAccount(int accountNumber, String ownerName,  
                        double initialBalance) {  
        this.number = accountNumber;  
        this.name = ownerName;  
        this.balance = initialBalance;  
    }  
  
    public void deposit(double amount) {  
        this.balance += amount;  
    }  
    public void withdraw(double amount) {  
        // Withdraw anche da conti in rosso  
        this.balance -= amount;  
    }  
  
    public double getBalance() { return this.balance;}  
    // altri Getter omessi
```

...

package bank


```
public class JointBankAccount extends BankAccount {  
    private String jointName;  
  
    public JointBankAccount(int accountNumber,  
                            String ownerName, String jointOwner,  
                            double initialBalance) {  
        super(accountNumber, ownerName, initialBalance);  
        this.jointName = jointOwner;  
    }  
  
    public String getJointName() {  
        return jointName;  
    }  
    ...  
}
```

Metodo polimorfo toString()

- Nella gerarchia dei conti bancari ci sono tre versioni (forme) del metodo `toString`
 - quella della classe `Object`. È usata se non facciamo override di nelle sottoclassi
 - una per la classe `BankAccount`
 - una per la classe `JointBankAccount`
- Java decide a run-time la versione da eseguire

Metodo polimorfo toString()

```
public String toString() {
```

BankAccount.java

```
    return "BankAccount [" + number + ", " + name +  
        balance() + "$ ]";
```

```
}
```

```
public String toString() {
```

JointBankAccount.java

```
    return "JointBankAccount [" +  
        super.getNumber() + ", " + super.getName() + ", " +  
        jointName + ", " + super.getBalance() + "$ ]";
```

```
}
```

Cosa stampa?

```
public class BankAccountsDemo {  
    public static void main(String[] args) {  
  
        BankAccount fred = new BankAccount(123, "Fred", 345.50);  
  
        JointBankAccount fredMary = new JointBankAccount(345,  
                                                            "Fred", "Mary", 450.65);  
  
        System.out.println(fred);  
        System.out.println(fredMary);  
    }  
}
```

BankAccount [123, Fred, 345.5\$]

JointBankAccount [345, Fred, Mary, 450.65\$]

E' lecito?

```
public class BankAccountsDemo {  
    public static void main(String[] args) {  
  
        BankAccount fred = new BankAccount(123, "Fred", 345.50);  
  
        BankAccount fredMary = new JointBankAccount(345,  
                                                    "Fred", "Mary", 450.65);  
  
        System.out.println(fred);  
        System.out.println(fredMary);  
    }  
}
```

BankAccount [123, Fred, 345.5\$]

JointBankAccount [345, Fred, Mary, 450.65\$]

Dynamic binding (utile?)

- Nell'esempio precedente anche se il **tipo statico** al momento della compilazione è **BankAccount** il **tipo dinamico** a run-time è **JointBankAccount**
- Java esegue quindi il metodo `toString()` della sottoclasse `JointBankAccount`
- Polimorfismo e dynamic binding sono fortemente collegati (non sinonimi)

Ciclo polimorfo

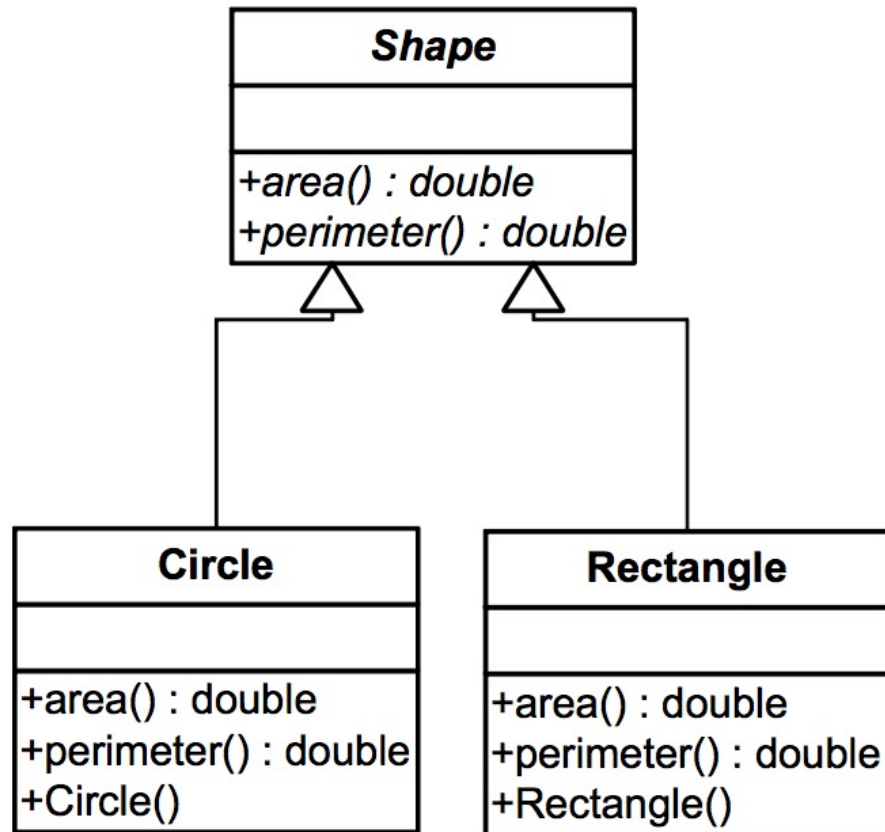
```
public class BankAccountsDemo {  
  
    public static void main(String[] args) {  
        BankAccount fred = new BankAccount(123, "Fred", 345.50);  
  
        BankAccount fredMary = new JointBankAccount(345,  
                                                    "Fred", "Mary", 450.65);  
  
        BankAccount[] accounts = {fred, fredMary};  
  
        for (BankAccount b : accounts) {  
            System.out.println(b);  
        }  
    }  
}
```

CLASSI ASTRATTE E INTERFACCE

Classi Astratte

- Ci sono diverse situazioni in cui le classi viste finora non modellano la realtà in modo completamente corretto
- Si consideri la gerarchia in cui le classi `Rectangle` e `Circle` derivano dalla classe `Shape`
- Osservazioni:
 - Non ha molto senso creare istanze di figura geometrica senza sapere di quale figura concreta si tratta
 - Non è possibile calcolare perimetro e area di una figura geometrica senza sapere di quale figura si tratta
 - E' necessario però sapere calcolare perimetro e area di ogni figura geometrica
- Si potrebbe utilizzare ereditarietà e fornire implementazioni vuote per questi metodi

Esempio figure geometriche



```
public class Shape {  
    public double getArea(){ return 0; }  
    public double getPerimeter(){ return 0;}  
    ...  
}
```

```
public class Rectangle extends Shape {  
    //variabili e costruttore omessi  
    @Override  
    public double getArea() {return s1 * s2;}  
  
    @Override  
    public double getPerimeter() {return (s1 + s2) * 2;}  
    ...  
}
```

```
public class Circle extends Shape {  
    //variabili e costruttore omessi  
    @Override  
    public double getArea() {return r * r * 3.14;}  
  
    @Override  
    public double getPerimeter() {return 2 * 3.14 * r;}  
    ...  
}
```

Classi Astratte (serve??)

- Chi garantisce che le sottoclassi implementeranno i metodi per calcolare area e perimetro?
- E' corretto dire che una qualunque figura geometrica ha perimetro e area uguali a 0?
- Abbiamo bisogno di un “segnaposto” che definisce comportamenti in modo incompleto e demanda le sottoclassi a completarli
- Usiamo classi **astratte** che indicano **cosa dovrebbe essere comune alle sottoclassi**

Classi Astratte

- Una classe si dice **astratta** se dichiara almeno un metodo senza fornire la sua implementazione
- Questa implementazione **DEVE** essere fornita da ogni sua sottoclasse **concreta**
- La keyword `abstract` permette di dichiarare un metodo astratto
- Se una classe Java ha un metodo astratto, anche solo uno, deve essere dichiarata astratta (con `abstract`)
- Non possiamo creare un oggetto da una classe astratta

Interfacce

- Uno dei punti chiave della programmazione OOP è la separazione tra:
 - **Interfaccia di una classe:** cosa una classe permette di fare e come può essere utilizzata (quali sono i metodi e che *signature* hanno)
 - **Implementazione di una classe:** come una classe funziona internamente
- Conoscendo l'interfaccia di una classe è possibile invocarne i metodi **senza conoscere la loro implementazione**
- In Java le interfacce sono entità a sé stanti che possono essere “usate” da diverse classi

Java Interface

```
public interface IShape {  
  
    //Calcola area  
    public double getArea();  
  
    //Calcola perimetro  
    public double getPerimeter();  
  
}
```

package geometry

Java Interface

- Un'interfaccia contiene le intestazione dei metodi pubblici
- Può anche definire costanti pubbliche
- Può, anzi dovrebbe, contenere anche commenti per gli utilizzatori dell'interfaccia stessa
- Si definisce con la keyword `interface` (al posto di `class`)
- Per convenzione il nome di un'interfaccia inizia con una lettera maiuscola (come per le classi)

Interfaccia

- Le interfacce contengono solo **prototipi di metodi**
- **NON** contengono nessuna implementazione
- Sono tipi di **classi completamente astratte**
- Una classe può implementare una interfaccia
- Si usa la keyword `implements` per indicare che una classe implementa un'interfaccia
- **Una classe che implementa un'interfaccia DEVE implementare TUTTI i metodi dell'interfaccia**

Implementare un'interfaccia

```
public class Rectangle implements IShape {  
    //variabili e costruttore omessi  
    @Override  
    public double getArea() {return s1 * s2;}  
  
    @Override  
    public double getPerimeter() {return (s1 + s2) * 2;}  
    ...  
}
```

```
public class Circle implements IShape {  
    //variabili e costruttore omessi  
    @Override  
    public double getArea() {return r * r * 3.14;}  
  
    @Override  
    public double getPerimeter() {return 2 * 3.14 * r;}  
    ...  
}
```

package geometry

Interfacce multiple

- Una stessa interfaccia può essere implementata da più classi
- Una **classe** in Java può estendere solo un'altra classe ma **può implementare più interfacce**
- Se classi diverse implementano la stessa interfaccia, allora rispetto all'interfaccia gli oggetti sono dello stesso tipo
- *NOTA: se una classe implementa più interfacce che definiscono lo stesso metodo (esattamente, stessa signature e stessi parametri) non c'è conflitto*

Interfacce multiple

```
public enum Color {BLACK, WHITE, YELLOW, RED, GREEN}

public interface IColorable {

    public Color getColor();

    public void setColor(Color c);

}
```

package geometry

Implementare più interfacce

```
public class Rectangle implements IShape, IColorable {  
    //variabili e costruttore omessi  
  
    @Override  
    public double getArea() {return s1 * s2;}  
  
    @Override  
    public double getPerimeter() {return (s1 + s2) * 2;}  
  
    @Override  
    public Color getColor() { return this.c; }  
  
    @Override  
    public void setColor(Color c) {this.c = c;}  
  
    ...  
}
```

package geometry

Interfacce e Classi astratte: principali differenze

Proprietà	Interfacce	Classi astratte
Ereditarietà	Una classe può implementare più interfacce	Una classe può avere una sola classe padre
Implementazione	Un'interfaccia non implementa nessun metodo; è completamente astratta	Una classe astratta può fornire implementazioni complete o parziali
Modificatori	Un'interfaccia definisce solo metodi pubblici	Una classe astratta può contenere metodi con diverse visibilità
Core vs. Periferiche (Omogeneità)	Un'interfaccia è solitamente usata per definire proprietà <i>non-core</i> e condivise da classi diverse (anche molto diverse tra loro)	Una classe astratta è solitamente usata per definire proprietà <i>core</i> e condivise dalle sottoclassi (omogenee tra loro)

Interfaccia come tipo

- Un'interfaccia definisce un tipo
- E' possibile quindi scrivere metodi che hanno uno o più parametri di tipo interfaccia
- Sarà poi possibile invocare questi metodi passandogli oggetti che implementano quell'interfaccia
- **Anche oggetti di classi diverse!**

Interfaccia come tipo

```
public class GeometryDemoInterface {  
    private static void colorWhite(IColorable s){  
        s.setColor(Color.WHITE);  
    }  
  
    public static void main(String[] args) {  
        Rectangle rect1 = new Rectangle(2, 3);  
        Circle circle1 = new Circle(5);  
  
        colorWhite(rect1);  
        colorWhite(circle1);  
    }  
}
```

OK se entrambe le classi
implementano IColorable

Interfaccia Comparable

- Java fornisce diverse interfacce predefinite implementate da diverse classi (ad esempio per le strutture dati)
- Una molto utilizzata è `Comparable` usata per imporre un ordinamento tra gli oggetti della classe che la implementa
- In realtà `Comparable` definisce un solo metodo:

```
public int compareTo(Object o);
```

- Le classi che lo implementano devono ritornare:
 - **numero negativo** se l'oggetto su cui viene chiamato “**precede**” il parametro `o`
 - **zero** se l'oggetto “**è uguale**” al parametro
 - **numero positivo** se “**è successivo**” al parametro

Esempio libri

```
public class Book implements Comparable {  
  
    private String title;  
    private Integer pubYear;  
  
    // Costruttore e metodi setter/getter omessi  
  
    public int compareTo (Object o) {  
        return 0;  
    }  
}
```

package books

```
import java.util.Arrays;

public class ComparablesDemo {

    public static void main(String[] args) {

        Book[] books = new Book[3];

        books[0] = new Book("Harry Potter ...", 1997);
        books[1] = new Book("The Lord of the Rings", 1954);
        books[2] = new Book("Don Quixote", 1605);

        Arrays.sort(books);

        for (Book b : books) {
            System.out.println(b);
        }

        Book [title=Harry Potter ..., pubYear=1997]
        Book [title=The Lord of the Rings, pubYear=1954]
        Book [title=Don Quixote, pubYear=1605]
    }
```

Richiama compareTo()
per fare i confronti



package books

Metodo compareTo()

```
public int compareTo(Object o) {  
    if ( (o != null) && (o instanceof Book)) {  
        Book nb = (Book) o; // type casting necessario  
  
        if (this.pubYear > nb.pubYear)  
            return 1;  
        else if (this.pubYear < nb.pubYear)  
            return -1;  
        else  
            return 0;  
    }  
    return -1; // default se si confronta  
               // Book con null o altre classi  
}
```

Diversi ordinamenti

- Crescente per anno di pubblicazione

```
Book [title=Don Quixote, pubYear=1605]  
Book [title=The Lord of the Rings, pubYear=1954]  
Book [title=Harry Potter ..., pubYear=1997]
```

- Per titolo in ordine alfabetico

```
Book [title=Harry Potter ..., pubYear=1997]  
Book [title=Don Quixote, pubYear=1605]  
Book [title=The Lord of the Rings, pubYear=1954]
```

ECCEZIONI

(molto velocemente)

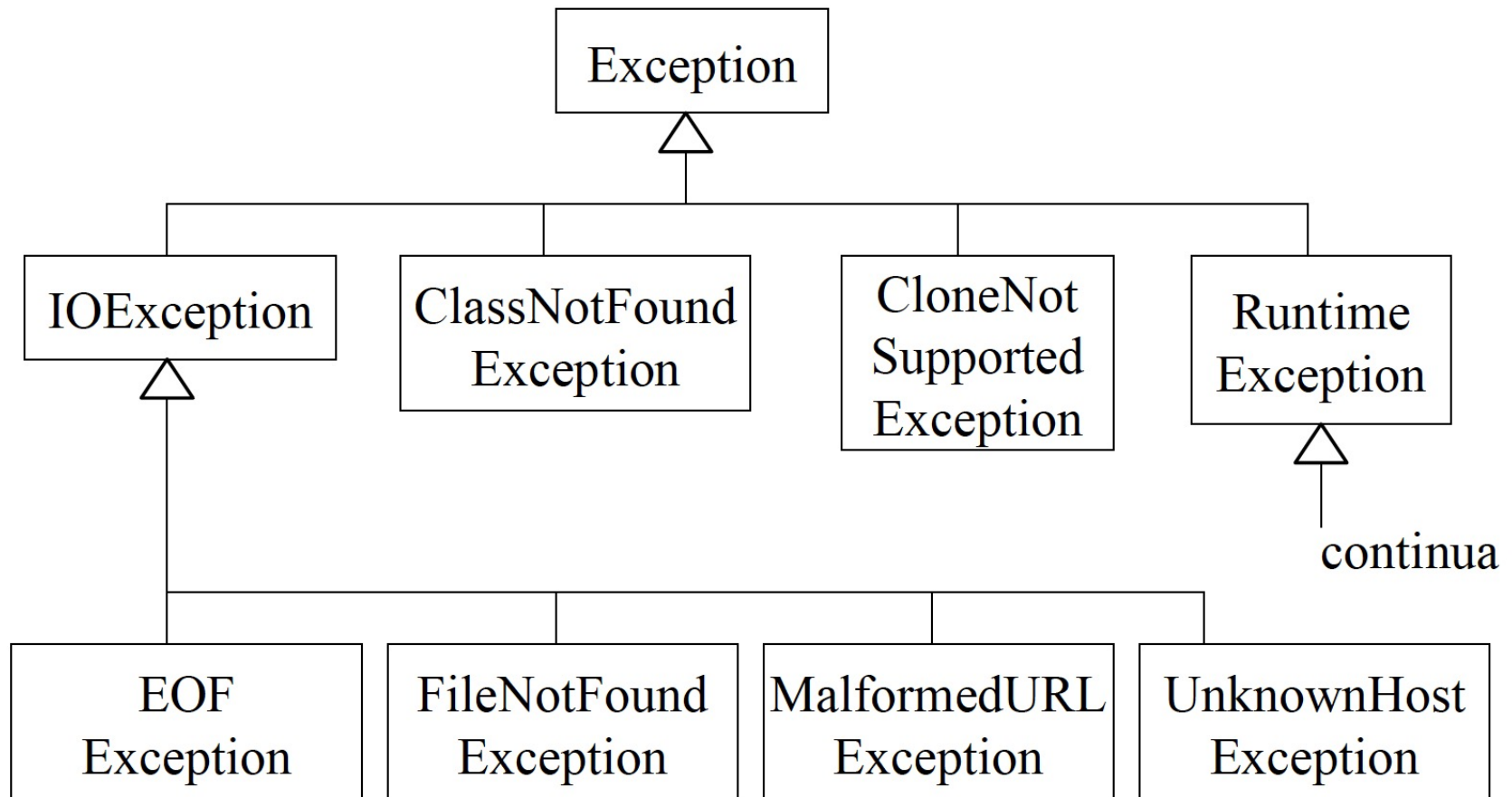
Eccezioni

- Il meccanismo delle eccezioni fornito da Java è un modo flessibile per realizzare una corretta gestione degli errori
- Idea di base: **passare il controllo** dal punto in cui si **verifica l'errore** direttamente a un altro punto dove **l'errore può essere gestito**
- Le eccezioni sono state progettate in modo che:
 - Le eccezioni **non** devono poter essere **trascurate**
 - Le eccezioni devono poter essere **gestite** da un **gestore competente**, non semplicemente dal chiamante del metodo che fallisce

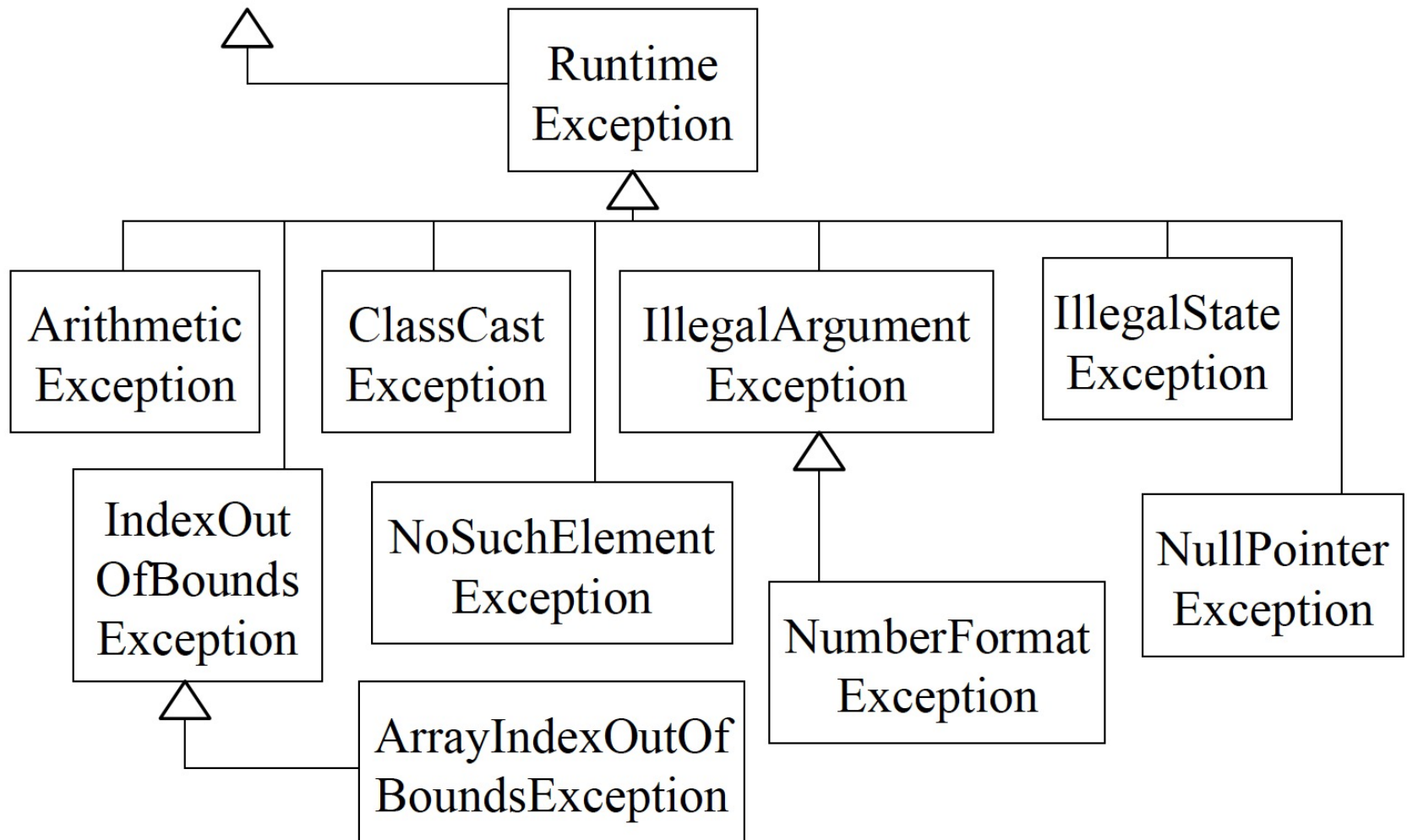
Segnalare un'eccezione

- Se ad un certo punto del codice si verifica un errore viene generata un'eccezione
- Un'eccezione è un oggetto di una classe
- Un'eccezione può essere lanciata esplicitamente con il comando `throw` seguito da un riferimento a un oggetto di una classe appropriata
- Java mette a disposizione una gerarchia di classi che rappresenta eccezioni di diversa natura
- Basta scegliere quella che fa di più al caso nostro, creare l'oggetto eccezione e lanciarlo

Gerarchia delle eccezioni (incompleta)



Gerarchia delle eccezioni (incompleta)



Gestire eccezioni: blocchi try-catch

- **Tutte le eccezioni** che si possono verificare in una certa applicazione dovrebbero essere **gestite da qualche parte**
- Se un'eccezione non ha nessun gestore allora il programma termina stampando la pila di attivazioni che l'eccezione ha attraversato prima di far terminare il programma
- Se un comando che si trova all'interno di un blocco **try** lancia un'eccezione allora il tipo dell'eccezione viene confrontato con i tipi elencati nelle clausole **catch** associate al blocco **try**
- Se uno dei tipi indicati nelle clausole **catch** è compatibile (uguale o superclasse) con il tipo dell'eccezione lanciata allora la propagazione dell'eccezione viene fermata, viene eseguito il blocco di codice associato alla **catch** e l'esecuzione continua con ciò che segue il blocco **try-catch**

```
public class QuickTestExceptions {  
  
    public static void main(String[] args) {  
  
        Integer[] a = {1, 4, 5, 76};  
  
        try {  
            System.out.println(a[10]);  
            //throw new IllegalArgumentException("SOS");  
        }  
        catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("Houston, abbiamo un problema.");  
        }  
        catch (IllegalArgumentException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

Definire nuove eccezioni

```
package exceptions;

public class BadException extends Exception {

    private static final long serialVersionUID = 1L;

    public String getMessage() {
        return "This is a really bad exception...";
    }

}
```

classe BadException

Altri aspetti di Java

- Non ne parliamo a lezione ma sono molto utili:
 - Javadoc
 - Basic I/O
 - Utilities
 - ...
- Torneremo su:
 - Java Generics

