

# Corso Web MVC Java

Emanuele Galli

[www.linkedin.com/in/egalli/](http://www.linkedin.com/in/egalli/)

# Java

Linguaggio di programmazione general-purpose, multi-platform, network-centric, class-based, object-oriented progettato da James Gosling @ Sun Microsystems.

- JVM: Java Virtual Machine
- JRE: Java Runtime Environment
- JDK: Java Development Kit

# Versioni

- 23 maggio 1995: prima release
- 1998 1.2 (J2SE)
- 2004 1.5 (J2SE 5.0)
- 2011 Java SE 7
- 2014 Java SE 8 (LTS)
- 2019 Java SE 12

# Link utili

*The Java Language Specifications*

<https://docs.oracle.com/javase/specs/>

*Java Platform, Standard Edition Documentation*

<https://docs.oracle.com/en/java/javase/index.html>

*Java SE 8 API Specification*

<https://docs.oracle.com/javase/8/docs/api/index.html>

*The Java Tutorials*

<https://docs.oracle.com/en/java/javase/index.html>

# Say hello /1

```
// Hello.java
```

```
public class Hello {
```

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

```
        System.out.println("Hello!");
```

```
    }
```

```
}
```

# Say hello /2

- JDK (8) from Oracle (for Windows x64 or ...)

<https://www.oracle.com/technetwork/java/javase/downloads/index.html>

- javac: Hello.java → Hello.class

source code → bytecode

- java: Hello.class → "Hello!"

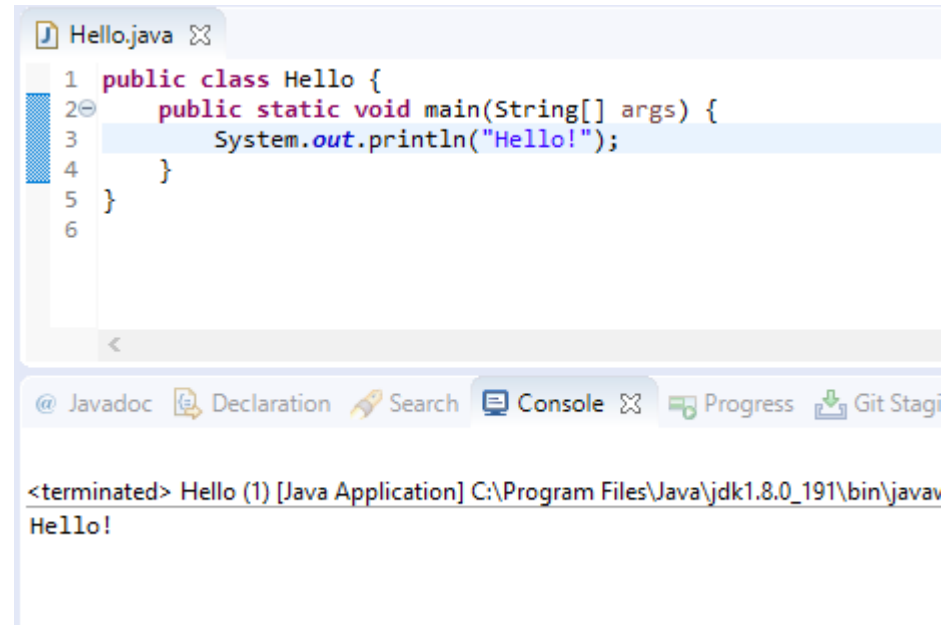
bytecode → machine code

# Say hello /3

## Integrated Development Environment (IDE)

- IntelliJ IDEA
- Eclipse IDE ← <https://www.eclipse.org/downloads/>
- Apache NetBeans
- ...

# Hello!



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

The screenshot shows an IDE window titled 'Hello.java'. The code is a simple Java class with a main method. The line 'System.out.println("Hello!");' is highlighted. Below the code editor, there is a toolbar with icons for Javadoc, Declaration, Search, Console, Progress, and Git Staging. The Console tab is active, showing the output: '<terminated> Hello (1) [Java Application] C:\Program Files\Java\jdk1.8.0\_191\bin\javav Hello!'.



# Struttura del codice /1

- Dichiarazioni
  - **Package** (collezione di classi)
  - **Import** (accesso a classi di altri package)
  - **Class** (solo una “public” per file)
- Commenti
  - **Multi-line**
  - **Single-line**
  - **Javadoc-style**

```
/*  
 * A simple Pi.java source file  
 */  
package dd.hello;  
  
import java.lang.Math; // not required  
  
/**  
 * @author manny  
 */  
public class Pi {  
    public static void main(String[] args) {  
        System.out.println(Math.PI);  
    }  
}  
  
class PackageClass {  
    // TBD  
}
```

# Struttura del codice /2

- Metodi
  - **main** (definito)
  - **println** (invocato)
- Parentesi
  - **Graffe** (blocchi, body di classi e metodi)
  - **Tonde** (liste parametri di metodi)
  - **Quadre** (array)
- **Statement** (sempre terminati da punto e virgola!)

```
/*
 * A simple Pi.java source file
 */
package dd.hello;

import java.lang.Math; // not required

/**
 * @author manny
 */
public class Pi {
    public static void main(String[] args) {
        System.out.println(Math.PI);
    }
}

class PackageClass {
    // TBD
}
```

# Variabili e tipi di dato

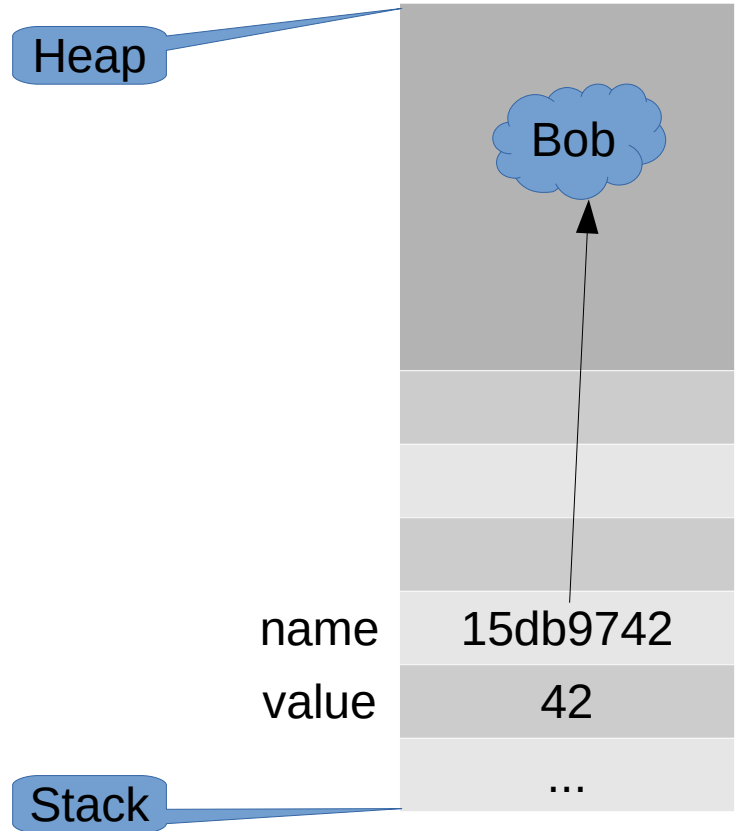
- Variabile: una locazione di memoria con un nome usato per accederla.
- Tipi di dato: determina valore della variabile e operazioni disponibili.
  - Primitive data type
  - Reference data type (class / interface)

# Tipi primitivi

bit			signed integer	floating point IEEE 754
1(?)	boolean	false true		
8			byte	
16	char	'\u0000' '\uFFFF'	short	
32			int	float
64			long	double

# Primitivi vs Reference

```
// primitivo  
int value = 42;  
  
// reference  
String name = "Bob";
```



# String

- Reference type che rappresenta una sequenza immutabile di caratteri
- StringBuilder, controparte mutabile, per creare stringhe complesse

```
String s = new String("hello");
```

Forma standard

```
String t = "hello";
```

Forma semplificata equivalente

# Operatori unari

++ incremento

-- decremento

prefisso: “naturale”

postfisso: ritorna il valore  
prima dell'operazione

+ positivo (useless)

- cambia il segno

```
int value = 1;
System.out.println(value);           // 1
System.out.println(++value);        // 2
System.out.println(--value);         // 1
System.out.println(value++);         // 1
System.out.println(value);           // 2
System.out.println(value--);         // 2
System.out.println(value);           // 1
System.out.println(+value);          // 1
System.out.println(-value);          // -1
```

# Operatori aritmetici

+ addizione

- sottrazione

\* moltiplicazione

/ divisione (intera)

% modulo

```
int a = 10;  
int b = 3;  
  
System.out.println(a + b); // 13  
System.out.println(a - b); // 7  
System.out.println(a * b); // 30  
System.out.println(a / b); // 3  
System.out.println(a % b); // 1
```



# Concatenazione di stringhe

- L'operatore + è overloaded per le stringhe.
- Se un operando è di tipo stringa, l'altro viene convertito a stringa e si opera la concatenazione.

```
System.out.println("Resistence" + " is " + "useless" );  
System.out.println("Solution: " + 42 );
```

# Operatori relazionali

<	Minore
<=	Minore o uguale
>	Maggiore
>=	Maggiore o uguale
==	Uguale
!=	Diverso

```
int alpha = 12;
int beta = 21;
int gamma = 12;

System.out.println("alpha < beta? " + (alpha < beta)); // true
System.out.println("alpha < gamma? " + (alpha < gamma)); // false
System.out.println("alpha <= gamma? " + (alpha <= gamma)); // true

System.out.println("alpha > beta? " + (alpha > beta)); // false
System.out.println("alpha > gamma? " + (alpha > gamma)); // false
System.out.println("alpha >= gamma? " + (alpha >= gamma)); // true

System.out.println("alpha == beta? " + (alpha == beta)); // false
System.out.println("alpha == gamma? " + (alpha == gamma)); // true

System.out.println("alpha != beta? " + (alpha != beta)); // true
System.out.println("alpha != gamma? " + (alpha != gamma)); // false
```

# Operatori logici (e bitwise)

&&	AND
	OR
!	NOT
&	AND
	OR
^	XOR

```
boolean alpha = true;
boolean beta = false;

System.out.println(alpha && beta);    // false
System.out.println(alpha || beta);    // true
System.out.println(!alpha);           // false
System.out.println(alpha & beta);      // false
System.out.println(alpha | beta);      // true

int gamma = 0b101;    // 5
int delta = 0b110;    // 6

System.out.println(gamma & delta);    // 4 = 0100
System.out.println(gamma | delta);    // 7 = 0111
System.out.println(gamma ^ delta);    // 3 = 0011
```

# Operatori di assegnamento

=	Assegnamento
+=	Aggiungi e assegna
-=	Sottrai e assegna
*=	Moltiplica e assegna
/=	Dividi e assegna
%=	Modulo e assegna
&=	AND e assegna
=	OR e assegna
^=	XOR e assegna

```
int alpha = 2;
```

```
alpha += 8;           // 10
```

```
alpha -= 3;           // 7
```

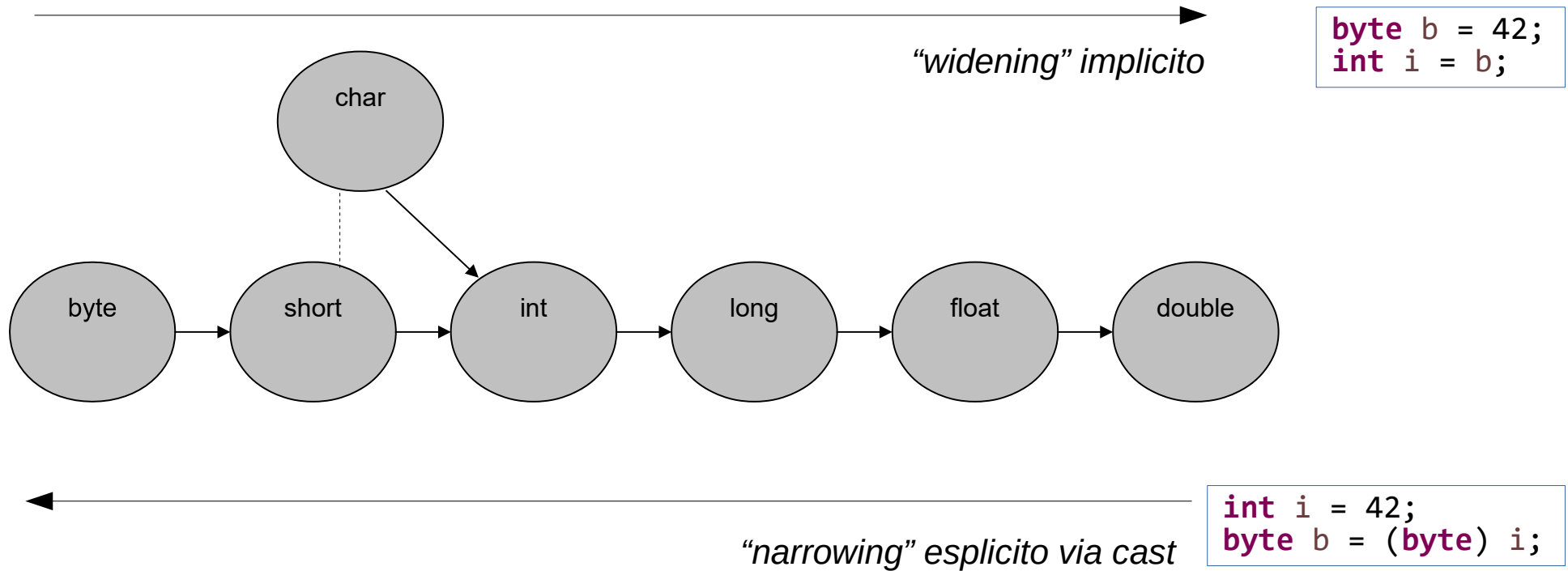
```
alpha *= 2;           // 14
```

```
alpha /= 2;           // 7
```

```
alpha %= 5;           // 2
```

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

# Cast tra primitivi



# Array

- Sequenza di “length” valori dello stesso tipo, memorizzati nello heap.
- Accesso per indice, a partire da 0.

```
int[] array = new int[12];  
array[0] = 7;  
  
int value = array[5];
```

```
int[] array = { 1, 4, 3 };  
  
if(array.length != 3) {  
    System.out.println("Unexpected");  
}
```

```
int[][] array2d = new int[4][5];  
  
int value = array2d[2][3];
```

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]
[1][0]	[1][1]	[1][2]	[1][3]	[1][4]
[2][0]	[2][1]	[2][2]	[2][3]	[2][4]
[3][0]	[3][1]	[3][2]	[3][3]	[3][4]

# if ... else if ... else

- Se la condizione è vera, si esegue il blocco associato.
- Altrimenti, se presente si esegue il blocco “else”.

```
if (condition) {  
    doSomething();  
}  
nextStep();
```

```
if (condition) {  
    doSomething();  
} else {  
    doSomethingElse();  
}  
nextStep();
```

```
if (condition) {  
    doSomething();  
} else if (otherCondition) {  
    doSomethingElse();  
} else {  
    doSomethingDifferent();  
}  
nextStep();
```

# switch

Scelta multipla su byte, short, char, int, String, enum

```
int value = 1;

// ...

switch (value) {
case 1:
    f();
    break;
case 2:
    g();
    break;
default:
    h();
    break;
}
```

```
String value = "1";

// ...

switch (value) {
case "1":
    f();
    break;
case "2":
    g();
    break;
default:
    h();
    break;
}
```

```
public enum WeekendDay {
    SATURDAY, SUNDAY
}

WeekendDay day = WeekendDay.SATURDAY;

// ...

switch (day) {
case SATURDAY:
    f();
    break;
case SUNDAY:
    g();
    break;
}
```



# loop

```
while (condition) {  
    // ...  
    if (something) {  
        condition = false;  
    }  
}
```

```
for (int i = 0; i < 10; i++) {  
    // ...  
    if (i == 4) {  
        continue;  
    }  
    // ...  
}
```

```
for (;;) {  
    // ...  
    if (something) {  
        break;  
    }  
    // ...  
}
```

```
do {  
    // ...  
    if (something) {  
        condition = false;  
    }  
} while (condition);
```

```
int[] array = new int[12];  
for (int value : array) {  
    System.out.println(value);  
}
```

# Classi e oggetti

- Classe:
  - Ogni classe è definita in un package
  - Descrive un nuovo tipo di dati, con proprietà e metodi
- Oggetto
  - Istanza di una classe, suo modello di riferimento

Reference a MyClass

Crea un oggetto MyClass

```
MyClass object = new MyClass();
```

# Metodo

- Blocco di codice associato ad un oggetto (default) o a una classe (static)
- Identificato da:
  - return type
  - nome
  - lista dei parametri

```
class MyClass {  
    static String h() {  
        return "Hi";  
    }  
    int f(int a, int b) {  
        // ...  
        return a * b;  
    }  
    void g(boolean flag) {  
        if (flag) {  
            System.out.println("Hello!");  
            return;  
        }  
        // ...  
        System.out.println("Goodbye");  
    }  
}
```

# Constructor (ctor)

- Metodo speciale, ha lo stesso nome della classe, è invocato durante la creazione di un oggetto via “new” per inizializzarne lo stato
- Non ha return type (nemmeno void)
- Se una classe non ha ctor, Java ne crea uno di default senza parametri (che non fa niente)

# Alcuni metodi di String

- char charAt(int)
  - int compareTo(String)
  - String concat(String)
  - boolean contains(CharSequence)
  - boolean equals(Object)
  - int indexOf(int)
  - int indexOf(String)
  - boolean isEmpty()
  - int lastIndexOf(int ch)
  - int length()
  - String replace(char, char)
  - String[] split(String)
  - String substring(int)
  - String toLowerCase()
  - String toUpperCase()
  - String trim()
- Tra i metodi statici:
- String format(String, Object...)
  - String join(CharSequence, CharSequence...)
  - String valueOf(Object)

# Alcuni metodi di StringBuilder

- `StringBuilder(int)`
- `StringBuilder(String)`
- `StringBuilder append(Object)`
- `char charAt(int)`
- `StringBuilder delete(int, int)`
- `void ensureCapacity(int)`
- `int indexOf(String)`
- `StringBuilder insert(int, Object)`
- `int length()`
- `StringBuilder replace(int, int, String)`
- `StringBuilder reverse()`
- `void setCharAt(int, char)`
- `void setLength(int)`
- `String toString()`

# La classe Math

## ***Proprietà statiche***

- E – base del logaritmo naturale
- PI – pi greco

## ***Alcuni metodi statici***

- double abs(double) // int, ...
- int addExact(int, int) // multiply ...
- double ceil(double)
- double cos(double) // sin(), tan()
- double exp(double)
- double floor(double)
- double log(double)

## ***... alcuni metodi statici***

- double max(double, double) // int, ...
- double min(double, double) // int, ...
- double pow(double, double)
- double random()
- long round(double)
- double sqrt(double)
- double toDegrees(double) // approx
- double toRadians(double) // approx

# Tre principi OOP

- Incapsulamento per mezzo di classi
  - Visibilità pubblica / privata
- Ereditarietà in gerarchie di classi
  - Dal generale al particolare
- Polimorfismo
  - Una interfaccia, molti metodi



# Access modifier per data member

- Aiutano l'incapsulamento
  - Privato
  - Protetto (ereditarietà)
- Normalmente sconsigliati
  - Package (default)
  - Pubblico

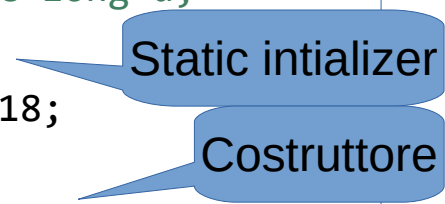
```
package my.package.example;

public class Sample {
    private int a;
    protected short b;
    static double c;
    // public long d;

    static {
        c = 18;
    }

    public Sample() {
        this.a = 42;
        this.b = 23;
    }

    // ...
}
```



# Access modifier per metodi

- Pubblico
- Package (usi speciali)
- Protetto / Privato (helper)

```
package my.package.example;

public class Sample {
    // ...

    static private double f() {
        return c;
    }

    void g() {
        f();
    }

    public int h() {
        return a / 2;
    }
}
```

# Lo “scope” delle variabili

- Locali
- Member (field, property)
  - instance (default)
  - static

```
public class Variables {  
    private static int staticMember = 5;  
    private int member = 5;  
  
    public void f() {  
        long local = 7;  
        if (staticMember == 2) {  
            short inner = 12;  
            staticMember = 1 + inner;  
            member = 3 + local;  
        }  
    }  
  
    public static void main(String[] args) {  
        double local = 5;  
        System.out.println(local);  
        staticMember = 12;  
    }  
}
```

# Inizializzazione delle variabili

- Esplicita per assegnamento (preferita)
  - primitivi: diretto
  - reference: via new
- Implicita by default (solo member)
  - primitivi
    - numerici: 0
    - boolean: false
  - reference: null

```
int i = 42;  
String s = new String("Hello");
```

```
int i;           // 0  
boolean flag;    // false  
String s;        // null
```

# Tipi wrapper

- Controparte reference dei tipi primitivi
  - Boolean, Character, Byte, Short, Integer, Float, Double
- Boxing esplicito
  - Costruttore
  - Static factory method (preferito)
- Unboxing esplicito
  - Metodi definiti nel wrapper
- Auto-boxing
- Auto-unboxing

```
Integer i = new Integer(1);  
Integer j = Integer.valueOf(2);  
  
int k = j.intValue();  
  
Integer m = 3;  
  
int n = k;
```

# interface

- **Cosa** deve fare una classe, **non come** deve farlo (fino a Java 8)
- Una class “**implements**” una interface
- Un’interface “**extends**” un’altra interface
- I metodi sono (implicitamente) **public**
- Le eventuali proprietà sono costanti **static final**

# interface vs class

```
interface Barker {  
    String bark();  
}
```

```
interface BarkAndWag extends Barker {  
    int AVG_WAGGING_SPEED = 12;  
  
    int tailWaggingSpeed();  
}
```

```
public class Fox implements Barker {  
    @Override  
    public String bark() {  
        return "yap!";  
    }  
}
```

extends vs implements

```
public class Dog implements BarkAndWag {  
    @Override  
    public String bark() {  
        return "woof!";  
    }  
  
    @Override  
    public int tailWaggingSpeed() {  
        return BarkAndWag.AVG_WAGGING_SPEED;  
    }  
}
```

# abstract class

- Una classe abstract non può essere istanziata
- Un metodo abstract non ha body
- Una classe che ha un metodo abstract deve essere abstract, ma non viceversa
- Una subclass di una classe abstract o implementa tutti i suoi metodi abstract o è a sua volta abstract



# Ereditarietà

- extends (is-a)
  - Subclasse che estende una già esistente
  - Eredita proprietà e metodi della superclass
  - p.es.: Mammal superclass di Cat e Dog
- Aggregazione (has-a)
  - Classe che ha come proprietà un'istanza di un'altra classe
  - p.es: Tail in Cat e Dog

# Ereditarietà in Java

- Single inheritance: una sola superclass
- Implicita derivazione da Object (che non ha superclass) by default
- Una subclass può essere usata al posto della sua superclass (is-a)
- Una subclass può aggiungere proprietà e metodi a quelli ereditati dalla superclass (attenzione a non nascondere proprietà della superclass con lo stesso nome!)
- Costruttori e quanto nella parte private della superclass non è ereditato dalla subclass
- Subclass transitivity: C subclass B, B subclass A  $\rightarrow$  C subclass A

# this vs super

- Reference all'oggetto corrente
  - this, come istanza della classe
  - super, come istanza della superclass
- ctor → ctor: (primo statement)
  - this() – nella classe
  - super() – nella superclass

# Esempio di ereditarietà

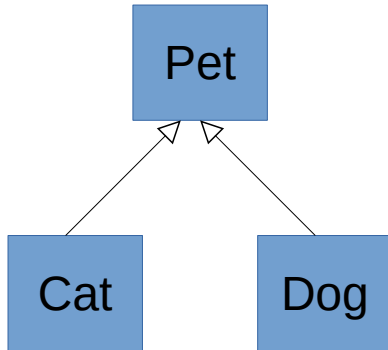
```
public class Pet {  
    private String name;  
  
    public Pet(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

```
Dog tom = new Dog("Tom", 2.42);  
  
// ...  
  
String name = tom.getName();  
double speed = tom.getSpeed();
```

```
public class Dog extends Pet {  
    private double speed;  
  
    public Dog(String name) {  
        this(name, 0);  
    }  
  
    public Dog(String name, double speed) {  
        super(name);  
        this.speed = speed;  
    }  
  
    public double getSpeed() {  
        return speed;  
    }  
}
```

# Reference casting

- Upcast: da subclass a superclass (sicuro)
- Downcast: da superclass a subclass (!?)
  - Protetto con l'uso di `instanceof`



```
// Cat cat = (Cat) new Dog(); // Cannot cast from Dog to Cat

Pet pet = new Dog();
Dog dog = (Dog) pet; // OK
Cat cat = (Cat) pet; // trouble at runtime
if(pet instanceof Cat) {
    Cat tom = (Cat) pet;
}
```

# Eccezioni

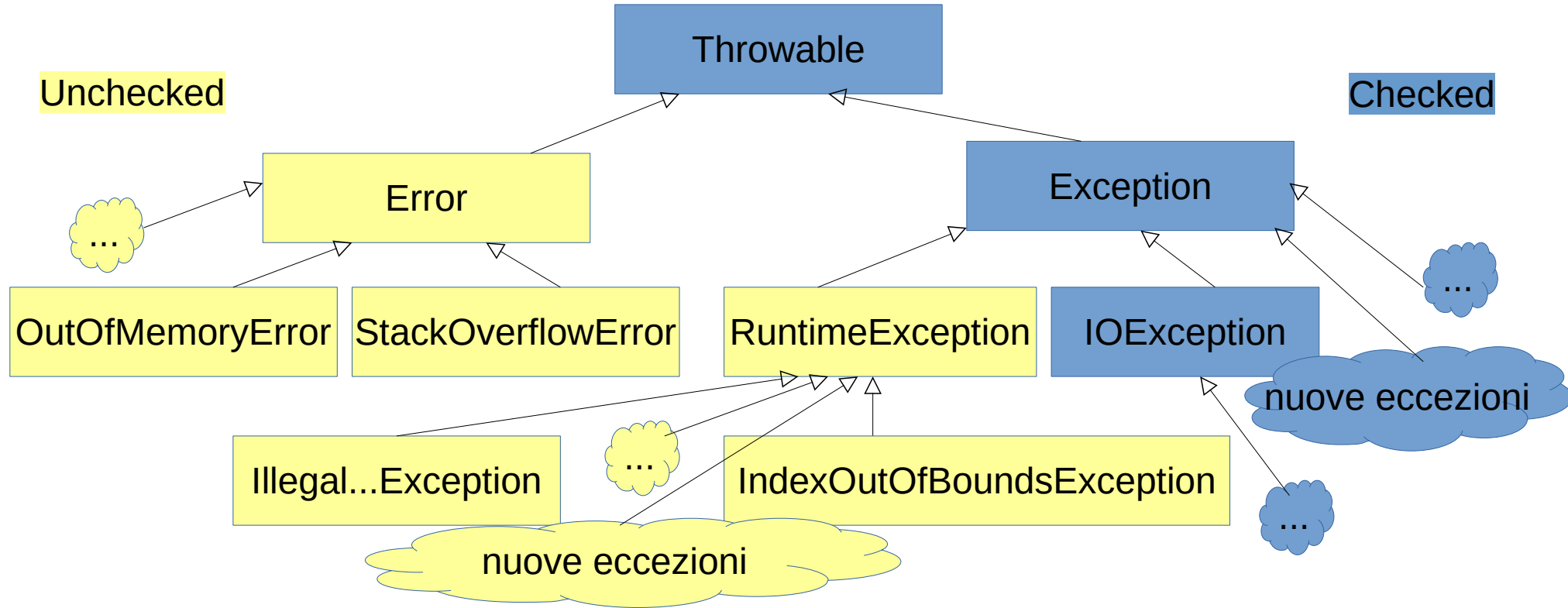
- Obbligano il chiamante a gestire gli errori
  - Unhandled exception → terminazione del programma
- Evidenziano il flusso normale di esecuzione
- Semplificano il debug esplicitando lo stack trace
- Possono chiarire il motivo scatenante dell'errore
- Checked vs unchecked

# try – catch – finally

- try: esecuzione protetta
- catch: gestisce uno o più possibili eccezioni
- finally: sempre eseguito, alla fine del try o dell'eventuale catch
- Ad un blocco try deve seguire almeno un blocco catch o finally
- “throws” nella signature, “throw” per “tirare” una eccezione.

```
void f() {  
    try {  
        g();  
    } catch (Exception ex) {  
        // ...  
    } finally {  
        cleanup();  
    }  
}  
  
// ...  
  
void g() throws Exception {  
    // ...  
    if (somethingUnexpected()) {  
        throw new Exception();  
    }  
}
```

# Gerarchia delle eccezioni



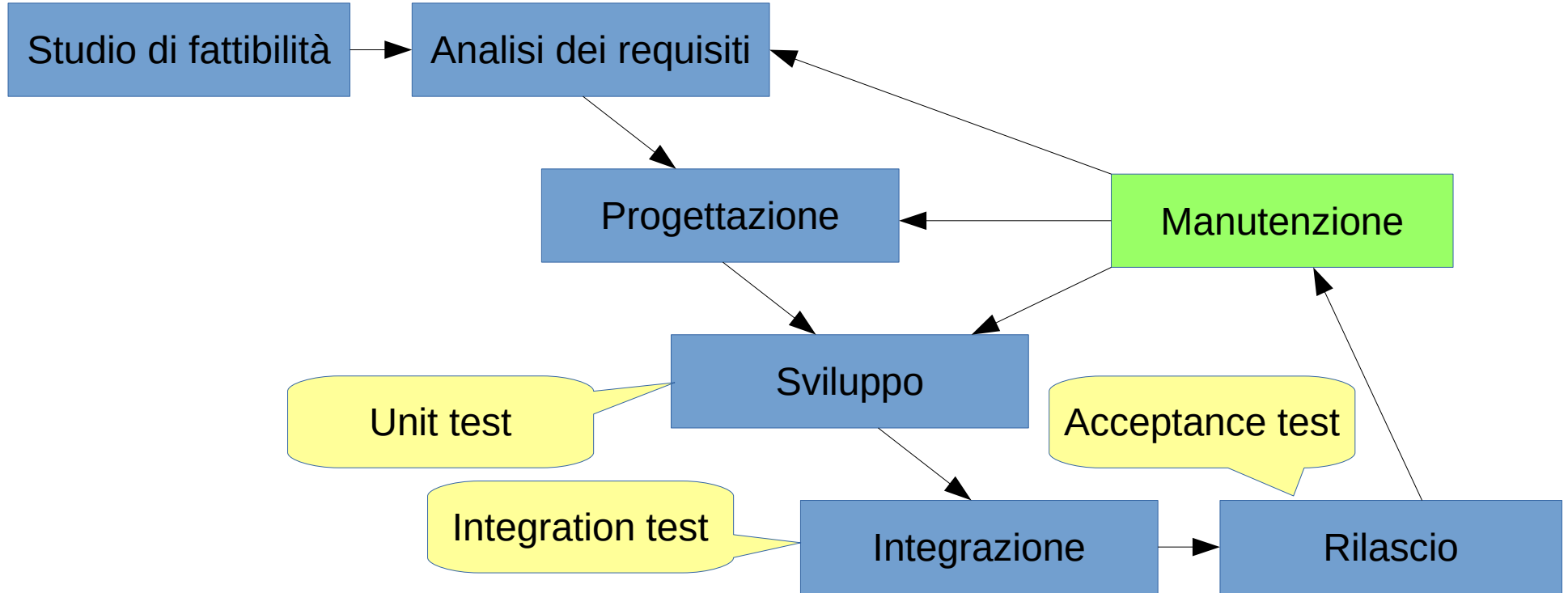


# Ciclo di vita del software

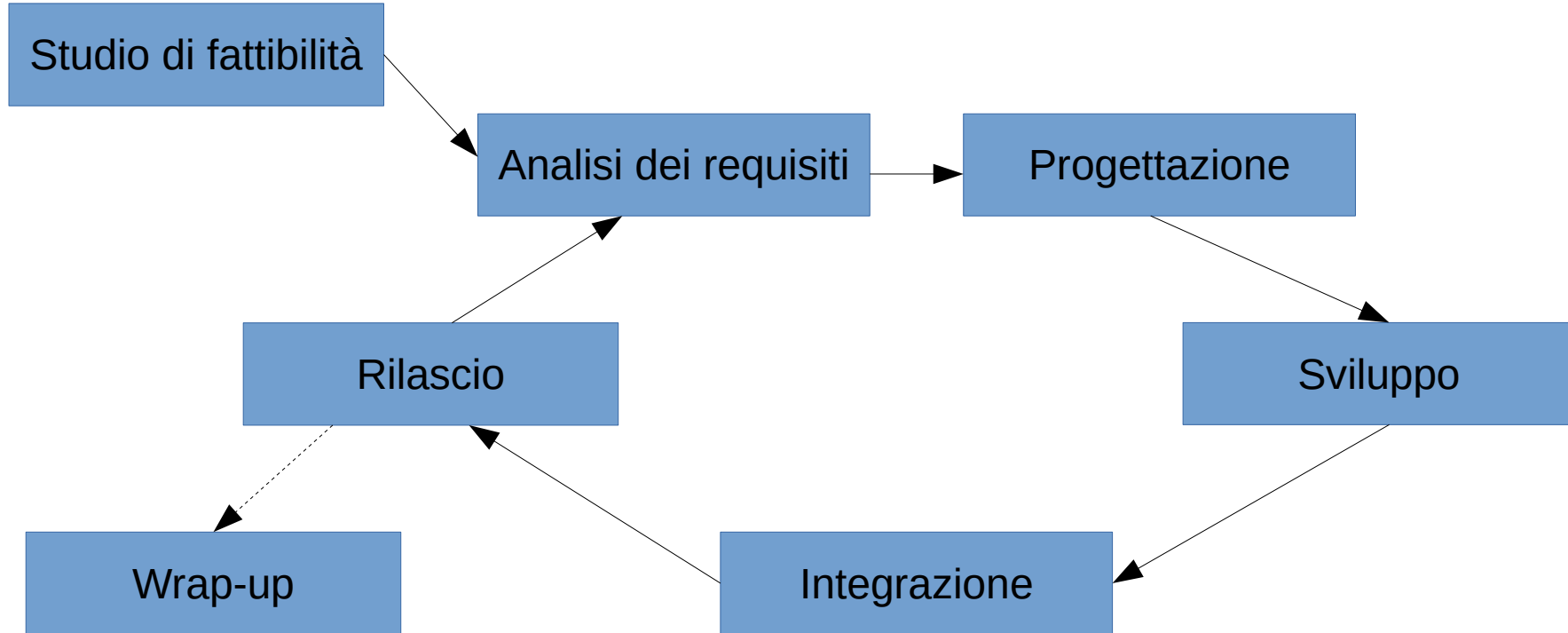
Come gestire la complessità di un progetto?

- Divide et impera
- Struttura
- Documentazione
- Milestones
- Comunicazione e interazione tra partecipanti

# Modello a cascata (waterfall)



# Modello agile



# Input con Scanner

- Legge input formattato e lo converte in formato binario
- Può leggere da `InputStream`, `File`, `String`, o altre classi che implementano `Readable` o `ReadableByteChannel`
- Uso generale di `Scanner`:
  - Il ctor associa l'oggetto scanner allo stream in lettura
  - Loop su `hasNext...()` per determinare se c'è un token in lettura del tipo atteso
    - Con `next...()` si legge il token
  - Terminato l'uso, ricordarsi di invocare `close()` sullo scanner

# Un esempio per Scanner

```
import java.util.Scanner;

public class Sample {
    public static void main(String[] args) {
        System.out.println("Please, enter a few numbers");
        double result = 0;

        Scanner scanner = new Scanner(System.in);
        while (scanner.hasNext()) {
            if (scanner.hasNextDouble()) {
                result += scanner.nextDouble();
            } else {
                System.out.println("Bad input, discarded: " + scanner.next());
            }
        }
        scanner.close(); // see try-with-resources
        System.out.println("Total is " + result);
    }
}
```

# try-with-resources

Per classi che implementano AutoCloseable

```
double result = 0;

// try-with-resources
try(Scanner scanner = new Scanner(System.in)) {
    while (scanner.hasNext()) {
        if (scanner.hasNextDouble()) {
            result += scanner.nextDouble();
        } else {
            System.out.println("Bad input, discarded: " + scanner.next());
        }
    }
}

System.out.println("Total is " + result);
```

# Java Util Logging

```
public static void someLog() {  
    Logger log =  
        Logger.getLogger("sample");  
  
    log.finest("finest message");  
    log.finer("finer message");  
    log.fine("fine message");  
    log.config("config message");  
    log.info("info message");  
    log.warning("warning message");  
    log.severe("severe message");  
}
```

```
public static void main(String[] args) {  
    Locale.setDefault(new Locale("en", "EN"));  
    Logger log = Logger.getLogger("sample");  
  
    someLog();  
  
    ConsoleHandler handler = new ConsoleHandler();  
    handler.setLevel(Level.ALL);  
    log.setLevel(Level.ALL);  
    log.addHandler(handler);  
    log.setUseParentHandlers(false);  
  
    someLog();  
}
```

# Generic

- Supporto ad algoritmi generici che operano allo stesso modo su tipi differenti (es: collezioni)
- Migliora la type safety del codice
- In Java è implementato solo per references
- Il tipo (o tipi) utilizzato dal generic è indicato tra parentesi angolari (minore, maggiore)



# Inner class

- Nested class: classe definita all'interno di un'altra classe
- La nested class ha accesso a tutti i membri della classe in cui è definita
- È possibile definirla come locale ad un blocco
- Inner class: non-static nested class
- Utili (ad es.) per semplificare la gestione di eventi

# Reflection

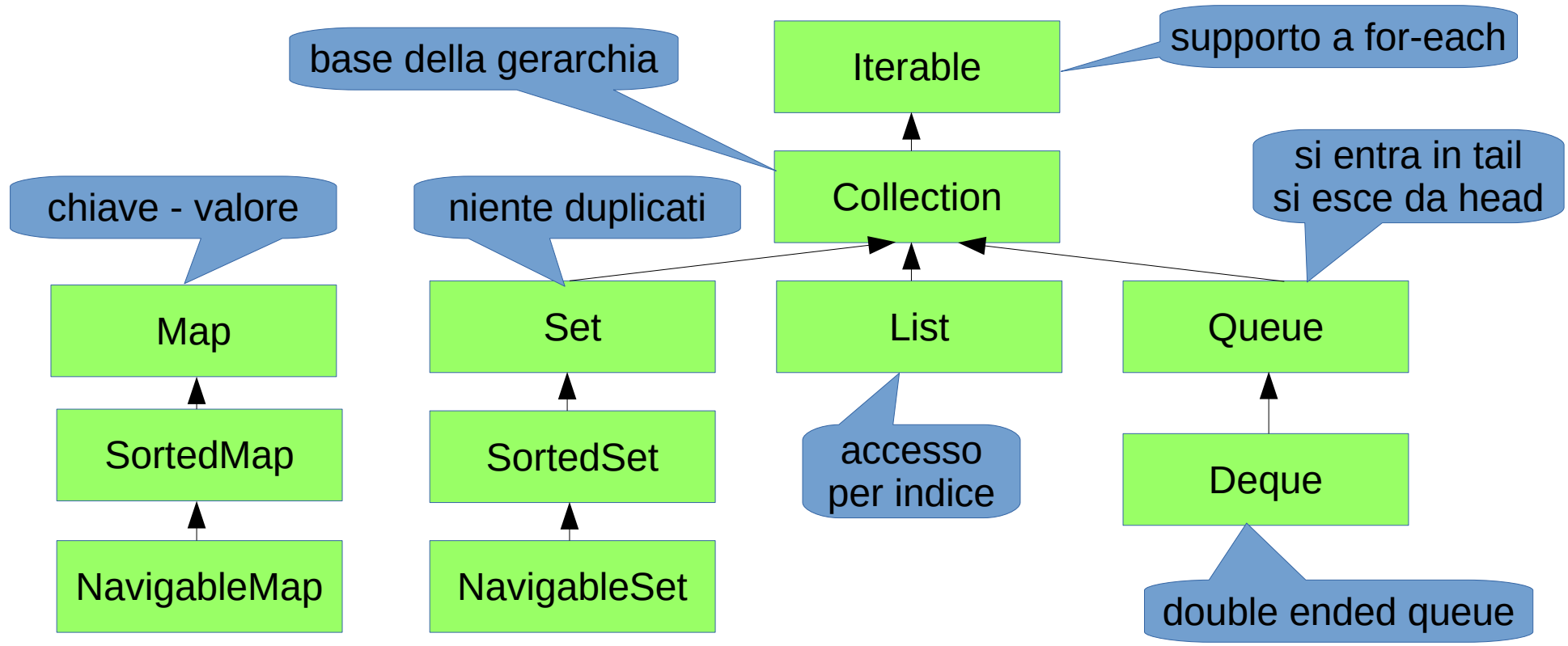
- Package `java.lang.reflect`
- Permette di ottenere a run time informazioni su di una classe
- “Class” è la classe che rappresenta una certa classe (!)
- Ad es: con `getMethods()` si possono ottenere tutti i metodi di una classe

```
Class<?> c = Integer.class;
Method[] methods = c.getMethods();
for(Method method: methods) {
    System.out.println(method);
}
```

# Java Collections Framework

- Lo scopo è memorizzare e gestire gruppi di oggetti (solo reference, no primitive)
- Enfasi su efficienza, performance, interoperabilità, estensibilità, adattabilità
- Basate su alcune interfacce standard
- La classe Collections contiene algoritmi generici
- L'interfaccia Iterator dichiara un modo standard per accedere, uno alla volta, gli elementi di una collezione

# Interfacce per Collection



# Alcuni metodi in Collection<E>

- boolean add(E)
- boolean addAll(Collection<? extends E>)
- void clear()
- boolean contains(Object);
- boolean equals(Object);
- boolean isEmpty();
- Iterator<E> iterator();
- boolean remove(Object);
- boolean retainAll(Collection<?>);
- int size();
- Object[] toArray();
- <T> T[] toArray(T[]);

# Alcuni metodi in List<E>

- void add(int, E)
- E get(int)
- int indexOf(Object)
- E remove(int)
- E set(int, E)

# Alcuni metodi in SortedSet<E>

- E first()
- E last()
- SortedSet<E> subSet(E, E)

# Alcuni metodi in NavigableSet<E>

- E ceiling(E), E floor(E)
- E higher(E), E lower(E)
- E pollFirst(), E pollLast()
- Iterator<E> descendingIterator()
- NavigableSet<E> descendingSet()



# Alcuni metodi in Queue<E>

- boolean offer(E e)
- E element()
- E peek()
- E remove()
- E poll()

# Alcuni metodi in Deque<E>

- void addFirst(E), void addLast(E)
- E getFirst(), E getLast()
- boolean offerFirst(E), boolean offerLast(E)
- E peekFirst(), E peekLast()
- E pollFirst(), E pollLast()
- E pop(), void push(E)
- E removeFirst(), E removeLast()

# Alcuni metodi in Map<K, V>

Map.Entry<K,V>

- K getKey()
- V getValue()
- V setValue(V)

- void clear()
- boolean containsKey(Object)
- boolean containsValue(Object)
- Set<Map.Entry<K, V>> entrySet()
- V get(Object)
- V getOrDefault(Object, V)
- boolean isEmpty()
- Set<K> keySet()
- V put(K, V)
- V putIfAbsent(K, V)
- V remove(Object)
- boolean remove(Object, Object)
- V replace(K key, V value)
- int size()
- Collection<V> values()

# Metodi in NavigableMap<K, V>

- Map.Entry<K,V> ceilingEntry(K)
- K ceilingKey(K)
- Map.Entry<K,V> firstEntry()
- Map.Entry<K,V> floorEntry(K)
- K floorKey(K)
- NavigableMap<K,V> headMap(K, boolean)
- Map.Entry<K,V> higherEntry(K)
- K higherKey(K key)
- Map.Entry<K,V> lastEntry()
- Map.Entry<K,V> lowerEntry(K)
- K lowerKey(K)
- NavigableSet<K> navigableKeySet()
- Map.Entry<K,V> pollFirstEntry()
- Map.Entry<K,V> pollLastEntry()
- SortedMap<K,V> subMap(K, K)
- NavigableMap<K,V> tailMap(K, boolean)

# ArrayList<E>

- implements List<E>
- Array dinamico vs standard array (dimensione fissa)
- Ctors
  - ArrayList() // capacity = 10
  - ArrayList(int) // set capacity
  - ArrayList(Collection<? extends E>) // copy

# LinkedList<E>

- implements List<E>, Deque<E>
- Lista doppiamente linkata
- Accesso diretto solo a head e tail
- Ctors
  - LinkedList() // vuota
  - LinkedList(Collection<? extends E>) // copy

# HashSet<E>

- implements Set<E>
- Basata sull'ADT hash table,  $O(1)$ , nessun ordine
- Ctors:
  - HashSet() // vuota, capacity 16, load factor .75
  - HashSet(int) // capacity
  - HashSet(int, float) // capacity e load factor
  - HashSet(Collection<? extends E>) // copy

# LinkedHashSet<E>

- extends HashSet<E>
- Permette di accedere ai suoi elementi in ordine di inserimento
- Ctors:
  - `LinkedHashSet()` // capacity 16, load factor .75
  - `LinkedHashSet(int)` // capacity
  - `LinkedHashSet(int, float)` // capacity, load factor
  - `LinkedHashSet(Collection<? extends E>)` // copy



# TreeSet<E>

- implements NavigableSet<E>
- Basata sull'ADT albero → ordine,  $O(\log(N))$
- Gli elementi inseriti devono implementare Comparable ed essere tutti mutualmente comparabili
- Ctors:
  - TreeSet() // vuoto, ordine naturale
  - TreeSet(Collection<? extends E>) // copy
  - TreeSet(Comparator<? super E>) // sort by comparator
  - TreeSet(SortedSet<E>) // copy + comparator

# TreeSet e Comparator

ordine naturale

comparator

plain

reversed

Java 8 lambda

```
List<String> data = Arrays.asList("alpha", "beta", "gamma", "delta");

TreeSet<String> ts = new TreeSet<>(data);

class MyStringComparator implements Comparator<String> {
    public int compare(String s, String t) {
        return s.compareTo(t);
    }
}

MyStringComparator msc = new MyStringComparator();

TreeSet<String> ts2 = new TreeSet<>(msc);
ts2.addAll(data);

TreeSet<String> ts3 = new TreeSet<>(msc.reversed());
ts3.addAll(data);

TreeSet<String> ts4 = new TreeSet<>((s, t) -> t.compareTo(s));
ts4.addAll(data);
```

# HashMap<K, V>

- implements Map<K,V>
- Basata sull'ADT hash table, O(1), nessun ordine
- Mappa una chiave K (unica) ad un valore V
- Ctors:
  - HashMap() // vuota, capacity 16, load factor .75
  - HashMap(int) // capacity
  - HashMap(int, float) // capacity e load factor
  - HashMap(Map<? extends K, ? extends V>) // copy

# TreeMap<K,V>

- implements NavigableMap<K,V>
- Basata sull'ADT albero → ordine,  $O(\log(N))$
- Gli elementi inseriti devono implementare Comparable ed essere tutti mutualmente comparabili
- Ctors:
  - TreeMap() // vuota, ordine naturale
  - TreeMap(Comparator<? super K>) // sort by comparator
  - TreeMap(Map<? extends K, ? extends V>) // copy
  - TreeMap(SortedMap<K, ? extends V>) // copy + comparator

# Multithreading

- Multitasking process-based vs thread-based
- L'interfaccia Runnable dichiara il metodo run()
- La classe Thread:
  - Ctors per Runnable
  - In alternativa, si può estendere Thread e ridefinire run()
  - start() per iniziare l'esecuzione

# synchronized

- Metodo: serializza su this
- Blocco: serializza su oggetto specificato

## comunicazione tra thread

- wait()
- notify() / notifyAll()