

# Corso Web MVC

## Java SE

Emanuele Galli

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

# Java

non è pensato per fare qualcosa di specifico  
linguaggio imperativo  
ci permette di lavorare pensando a classi (Platone) e a oggetti

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

funziona su più piattaforme

pensata per lavorare bene sulle reti internet

- **JVM**: Java Virtual Machine

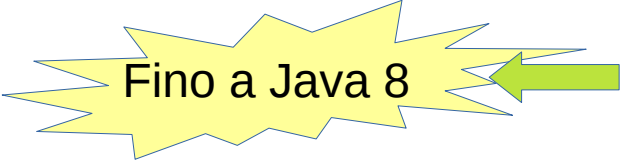
fa da interprete su ogni piattaforma (IOS, Windows, Linux..) per poter eseguire programmi Java

- **JRE**: Java Runtime Environment

contiene JVM, ma non viene più utilizzata (serve solo ad utilizzare programmi Java)

- **JDK**: Java Development Kit

per scrivere il codice; contiene JRE e quindi contiene strumenti che sviluppano il codice



Fino a Java 8

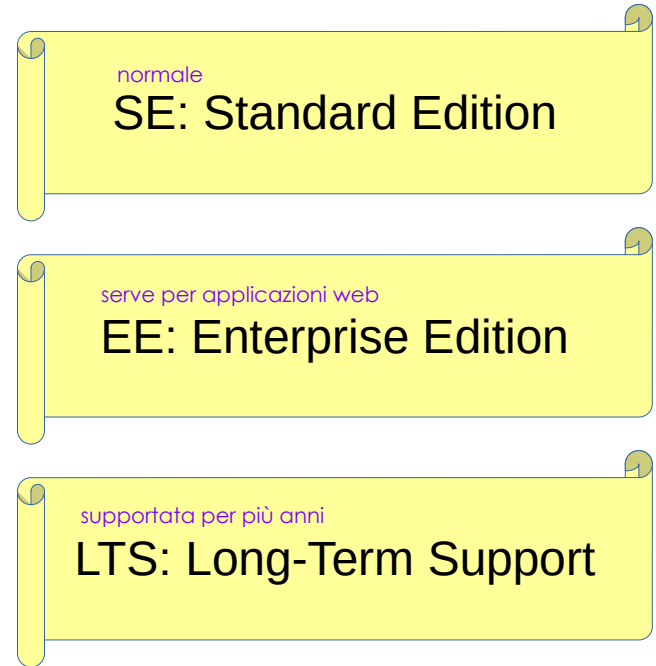
quella più utilizzata, ultimo aggiornamento Java13

# Versioni

- 23 maggio 1995: prima release
- 1998 1.2 (J2SE)
- 2004 1.5 (J2SE 5.0) Java classico
- 2011 Java SE 7
- 03/2014 Java SE 8 (LTS)
- 09/2018 Java SE 11 (LTS)
- 09/2019 Java SE 13



troppo avanzato



# Link utili

come è fatto Java, ma molto complicato

*Java Language Specifications:* <https://docs.oracle.com/javase/specs/>

come funziona

*Java SE 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>

più semplice

*The Java Tutorials (JDK 8):* <https://docs.oracle.com/javase/tutorial/>

*Java SE Downloads* più popolare (disponibili senza registrarsi)

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

<https://adoptopenjdk.net/>

# Hello Java a riga di comando

- Generazione del **bytecode**

`javac Hello.java`

- Visualizzazione del bytecode disassemblato*

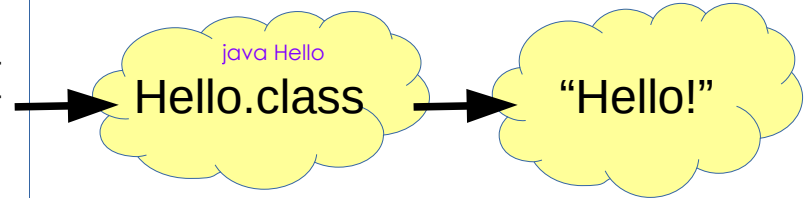
`javap -c Hello`

- Generazione del **codice macchina** ed **esecuzione**

`java Hello`

**bin** di jdk non nel system path!  
vedi: impostazioni di Windows  
path → variabili d'ambiente di sistema

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



# Version Control System (VCS)

il programmatore vuole salvare la storia delle versioni del codice per poterlo cambiare

- Obiettivi

- Mantenere traccia dei cambiamenti nel codice; sincronizzazione del codice tra utenti
- Cambiamenti di prova senza perdere il codice originale; tornare a versioni precedenti

i programmatori lavorano in gruppo e quindi condividono il codice  
un programmatore che si accorge di aver sbagliato può cancellarlo tranquillamente sapendo di cancellare solo la propria parte

modo di affrontare il problema

- Architettura client/server (CVS, Subversion, ...)

- Repository centralizzato con le informazioni del progetto  
(codice sorgente, risorse, configurazioni, documentazione, ...)
- check-out/check-in (lock del file), branch/merge (conflitti)

- Distributed VCS, architettura peer-to-peer (Git, Mercurial, ...)

- Repository clonato su tutte le macchine
- Solo push e pull richiedono connessione di rete

situazione preferita non c'è il lock  
tutti possono modificare contemporaneamente ma prima di caricare i cambiamenti bisogna fare il pull di verifica dell'aggiornamento del repository. è cmq una versione più elastica assomigliante a un social network (github)

# Git

- 2005 by creatore di Linux Linus Torvalds et al.
- 24 febbraio 2019: version 2.21
- Client ufficiale
  - <https://git-scm.com/> (SCM: Source Control Management)
- Supportato nei principali ambienti di sviluppo
- Siti su cui condividere pubblicamente un repository salvare il codice
  - [github.com](https://github.com), [bitbucket.org](https://bitbucket.org), ...
- Gli utenti registrati possono fare il **fork** di repository pubblici
  - Ad es: <https://github.com/egalli64/mpjp.git>, tasto “fork” in alto a destra

# Configurazione di Git

- Vince il più specifico tra
  - Sistema: Programmi Git/mingw64/etc/gitconfig
  - Globale: Utente corrente .gitconfig
  - Locale: Repo corrente .git/config
- Set globale del nome e dell'email dalla shell di Git
  - `git config --global user.name "Emanuele Galli"`
  - `git config --global user.email egalli64@gmail.com`



# Nuovo repository Git locale

- Prima si crea il repository remoto → URL .git
- A partire da quella URL, copia locale del repository
  - Esempio: <https://github.com/egalli64/empty.git>
- Shell di Git, nella directory git locale:
  - `git clone <URL>`
- Possiamo clonare ogni repository pubblico
- Per il push dobbiamo avere il permesso

# Creare un file nel repository

Dalla shell di Git, nella directory del progetto

Crea il file hello.txt

Aggiorna la versione  
nel repository locale  
sincronizzandola  
con la copia di lavoro

```
echo "hello" > hello.txt  
git add hello.txt  
git commit -m "first commit"  
git push -u origin master
```

I cambiamenti nel file  
andranno nel repository

Aggiorna la versione  
nel repository remoto  
sincronizzandola  
con quella in locale

# File ignorati da Git

- Alcuni file devono restare locali
  - Configurazione
  - File compilati
- Per ignorare file o folder
  - Creare un file “.gitignore”
  - Inserire il nome del file, pattern o folder su una riga

Esempio di file  
.gitignore

```
/bin/  
/*.tmp
```

# git pull

- Per assicurarsi di lavorare sul codebase corrente, occorre sincronizzarsi col repository remoto via pull
- È in realtà la comune abbreviazione dei comandi fetch + merge origin/master

# Cambiamenti nel repository

- Se vogliamo che un nuovo file, o che un edit, venga registrato nel repository, dobbiamo segnalarlo col comando `git add`
- A ogni commit va associato un `messaggio`, che dovrebbe descrivere il lavoro compiuto  
`git commit -m ".classpath is only local"`
- Per l'editing, si può fondere add con commit, usando l'opzione "a"  
`git commit -am "hello"`
- La prima commit crea il branch "master", le successive aggiornano il branch corrente

# git push

- Commit aggiorna il repository locale
- Push aggiorna il repository remoto
- Per ridurre il rischio di conflitti, **prima pull**, dopo (e solo se non sono stati rilevati problemi) push

# Conflitti su pull

- Il file hello.txt ha una sola riga: “A”
- L’utente X aggiunge una riga “K” e committa
- L’utente Y fa una pull, aggiunge la riga “B”, committa e fa un push
- Ora, il pull di X causa un **auto-merging** di hello.txt con un **conflitto**
- Git chiede di risolverlo editando il file + **add/commit** del risultato

Cambiamento  
locale

Cambiamento  
remoto

```
A
<<<<<<< HEAD
K
=====
B
>>>>>>> 627ffd9686ef003803a1ecdd25d2a2f2a655a897
```

id della commit  
con conflitto

# Branching del repository

- `git branch`
  - Lista dei branch esistenti, evidenzia quello corrente
- `git branch <name>`
  - Crea un nuovo branch
  - Il primo push del nuovo branch deve creare un upstream branch
    - `git push --set-upstream origin <name>`
- `git checkout <name>`
  - Permette di scegliere il branch corrente
- `git merge <name>`
  - Eseguito dal branch principale, fusione con risoluzione degli eventuali conflitti



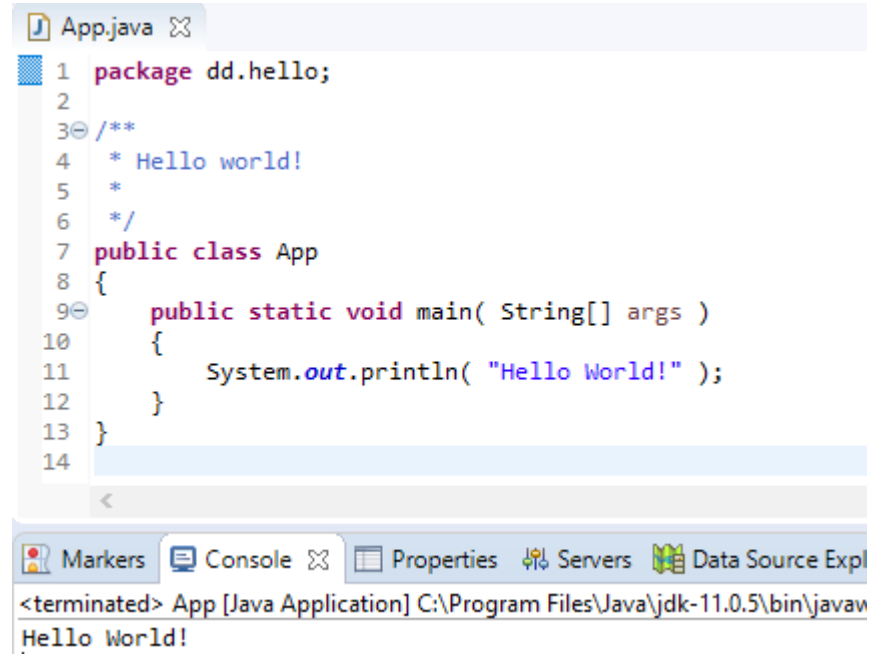
# Principali comandi Git in breve

- `clone <url>`: clona un repository in locale
- `add <filename(s)>`: stage per commit
- `commit -m "message"`: copia sul repository locale
- `commit -am "message"`: add & commit
- `status`: lo stato del repository locale
- `push`: da locale a remoto
  - `push --set-upstream origin <branch>`
- `pull`: da remoto a locale
- `log`: storico delle commit
- `reflog`: storico in breve
- `reset --hard <commit>`: il repository locale torna alla situazione del commit specificato
- `branch`: lista dei branch correnti
- `branch <branch>`: creazione di un nuovo ramo di sviluppo
- `checkout <branch>`: scelta del branch corrente
- `merge <branch>`: fusione del branch

# Integrated Development Environment (IDE)

- Semplifica lo sviluppo di applicazioni
  - IntelliJ IDEA
  - Eclipse IDE
    - Installer: <https://www.eclipse.org/downloads/>
    - Full: <https://www.eclipse.org/downloads/packages/>
    - STS – Spring Tool Suite
  - Apache NetBeans
  - Microsoft VS Code (“code editor”, più leggero di IDE)
  - ...

# Hello World!



The screenshot shows an IDE window titled 'App.java'. The code is as follows:

```
1 package dd.hello;
2
3 /**
4  * Hello world!
5  *
6  */
7 public class App
8 {
9     public static void main( String[] args )
10    {
11        System.out.println( "Hello World!" );
12    }
13 }
14
```

Below the code editor, the 'Console' tab is active, displaying the output: '<terminated> App [Java Application] C:\Program Files\Java\jdk-11.0.5\bin\javav  
Hello World!'



# Build automation con Maven

- Build automation
  - Compilazione del codice sorgente
  - Packaging dell'eseguibile
  - Esecuzione automatica dei test
- UNIX make, Ant, Maven, Gradle
- Apache Maven, supportato da tutti i principali IDE per Java
  - pom.xml (POM: Project Object Model)
  - I processi seguono convenzioni stabilite, solo le eccezioni vanno indicate
  - Le dipendenze implicano il download automatico delle librerie richieste

# Nuovo progetto Maven in Eclipse

- Creare un progetto Maven
  - File, New, Maven Project
  - È necessario specificare solo **group id** e **artifact id**
  - Il progetto risultante è per Java 5
- Nel POM specifichiamo le nostre **variazioni** (vedi slide successive)
  - Properties
  - Dependencies
- A volte occorre forzare l'update del progetto dopo aver cambiato il POM
  - Alt-F5 (o right-click sul nome del progetto → Maven, Update project)

# Properties

- Definizione di costanti relative al POM
- Ad esempio:
  - Codifica nel codice sorgente
  - Versione di Java (source e target)

```
<properties>  
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
  <maven.compiler.source>11</maven.compiler.source>  
  <maven.compiler.target>11</maven.compiler.target>  
</properties>
```

# Aggiungere una dependency

- Ricerca su repository Maven (central e altri)
  - <https://search.maven.org/>,  
<https://mvnrepository.com/>
- Ad esempio:
  - JUnit (4.12 stabile), JUnit Jupiter engine (5.5.2)

```
<dependency>  
  <groupId>junit</groupId>  
  <artifactId>junit</artifactId>  
  <version>4.12</version>  
</dependency>
```

```
<dependency>  
  <groupId>org.junit.jupiter</groupId>  
  <artifactId>junit-jupiter-engine</artifactId>  
  <version>5.5.2</version>  
</dependency>
```

Tra le `<dependencies>`

Vogliamo usare Junit  
solo in test,  
perciò aggiungiamo:  
`<scope>test</scope>`

# Import di un progetto via Git

- Da Eclipse
  - File, Import ..., Git, Projects from Git (with smart import)
    - Clone URI
    - Fornita da GitHub. Ad es: <https://github.com/egalli64/mpjp>
    - Se Eclipse non lo riconosce come progetto Maven, va importato come “General Project” e poi “mavenizzato”
  - Oppure, se il repository è già stato clonato
    - import del progetto come Existing local repository



# Nuovo repository Git in Eclipse

- GitHub, creazione di un nuovo repository “xyz”
- Shell di Git, nella directory git locale:

```
git clone <url di xyz.git>
```

*(oppure si può clonarlo dalla prospettiva Git di Eclipse)*

- Eclipse: creazione di un nuovo progetto
  - Location: directory del repository appena clonato git/xyz
- Il nuovo progetto viene automaticamente collegato da Eclipse al repository Git presente nel folder

# Team per Git in Eclipse

- Right click sul nome del progetto, Team
  - Pull (o Pull... per il branch corrente)
  - Commit rimanda alla view “Git staging”
  - Push to upstream (per il branch corrente)
  - Switch To, New branch...
    - Basta specificare il nome del nuovo branch
  - Switch To, per cambiare il branch corrente
  - Merge branch, per fondere due branch

# .gitignore in Eclipse

- Per ignorare file o folder
  - Come già visto, file .gitignore
  - Oppure: right-click sulla risorsa, Team, Ignore
- Eclipse annota le icone di file e folder con simboli per mostrare come sono gestiti da Git
  - punto di domanda: risorsa sconosciuta
  - asterisco: risorsa staged per commit
  - più: risorsa aggiunta a Git ma non ancora tracked
  - assenza di annotazioni: risorsa ignorata

# Struttura del codice /1

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

```
/*  
 * A simple Java source file  
 */  
package s028;  
  
import java.lang.Math; // not required  
  
/**  
 * @author manny  
 */  
public class Simple {  
    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 di parametri per metodi)
  - **Quadre** (array)
- **Statement** (sempre terminati da punto e virgola!)

```
/*  
 * A simple Java source file  
 */  
package s028;  
  
import java.lang.Math; // not required  
  
/**  
 * @author manny  
 */  
public class Simple {  
    public static void main(String[] args) {  
        System.out.println(Math.PI);  
    }  
}  
  
class PackageClass {  
    // TBD  
}
```

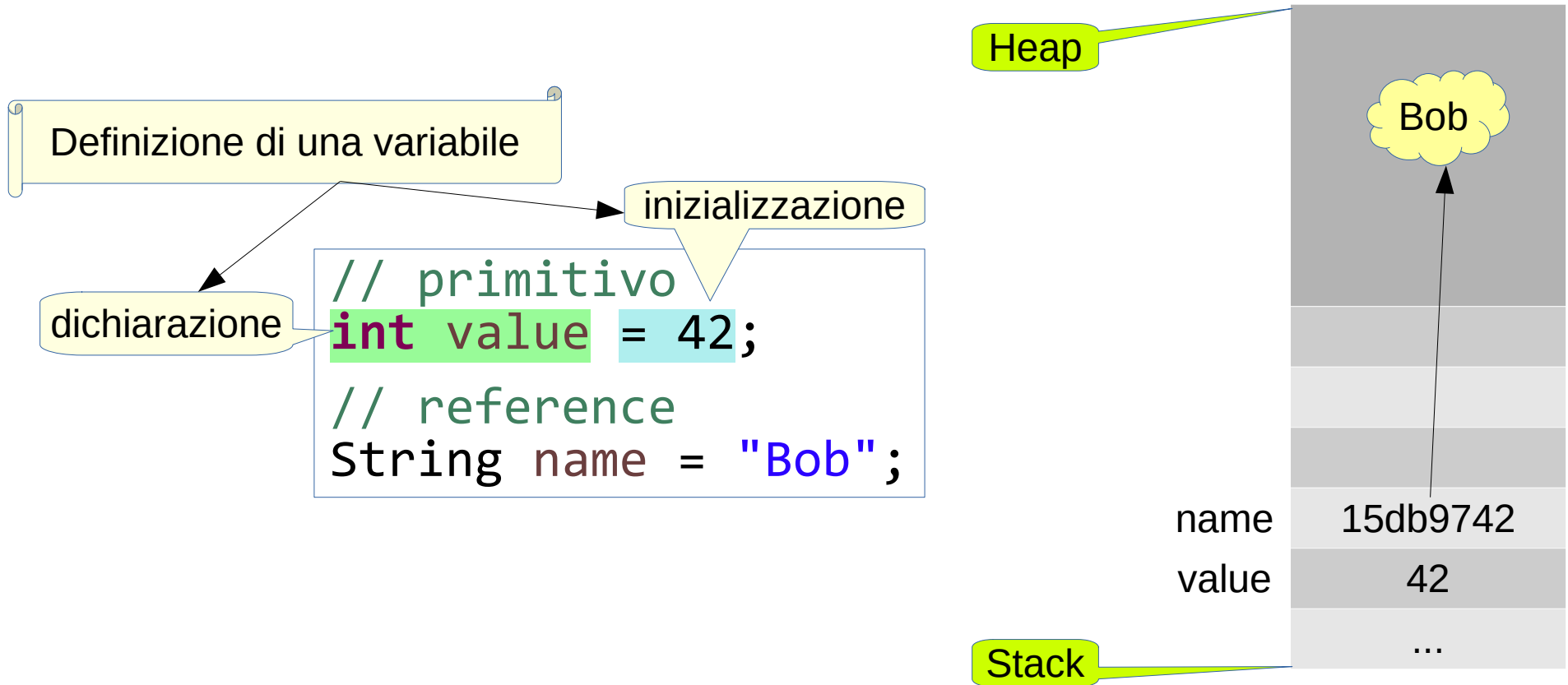
# Variabili e tipi di dato

- 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	-128	
				127	
16	char	'\u0000'	short	-32,768	
		'\uFFFF'		32,767	
32			int	-2^31	float 
				2^31 - 1	
64			long 	-2^63	double
				2^63 - 1	

# Primitivi vs Reference





# String

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

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

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

Forma standard

Forma semplificata equivalente

# Operatori unari

++ incremento

-- decremento

prefisso: “naturale”

postfisso: ritorna il valore  
prima dell'operazione

+ mantiene il segno corrente

- cambia il segno corrente

```
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.
- Da Java 11, `repeat()` è una specie di moltiplicazione per stringhe

```
System.out.println("Resistance" + " is " + "useless");  
System.out.println("Solution: " + 42);  
  
System.out.println("Vogons".repeat(3));
```

# 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)

“shortcut”  
preferiti

&&	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
```

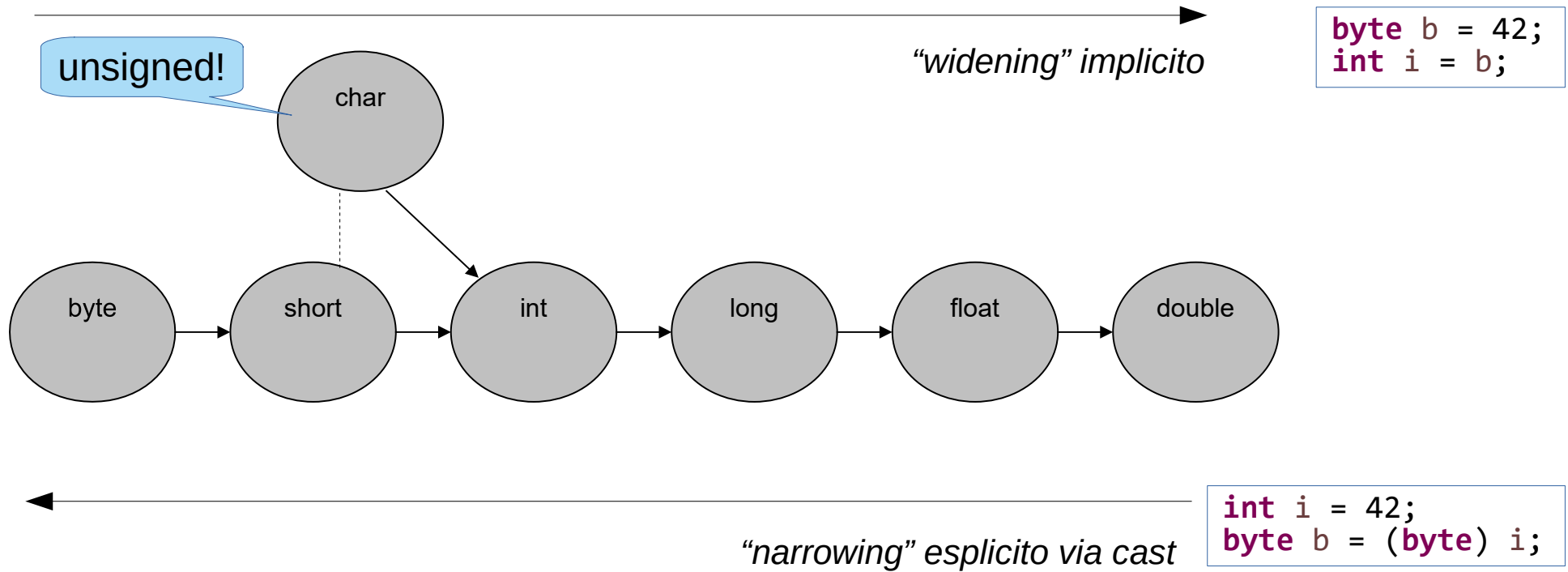
```
alpha *= 2;           // 14
```

```
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.
- Tentativo di accedere a un elemento esterno → `ArrayIndexOutOfBoundsException`

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

```
int[] array = { 1, 4, 3 };  
  
// array[array.length] = 21; // exception  
  
System.out.println(array.length); // 3
```

```
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 = 42;
// ...

switch (value) {
case 1:
    // ...
    break;
case 2:
    // ...
    break;
default:
    // ...
    break;
}
```

```
String value = "1";
// ...

switch (value) {
case "1":
    // ...
    break;
case "2":
    // ...
    break;
default:
    // ...
    break;
}
```

```
public enum WeekendDay {
    SATURDAY, SUNDAY
}

WeekendDay day = WeekendDay.SATURDAY;
// ...

switch (day) {
case SATURDAY:
    // ...
    break;
case SUNDAY:
    // ...
    break;
}
```

# loop

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

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

```
for (int i = 0; i < 5; i++) {  
    // ...  
    if (i == 2) {  
        continue;  
    }  
    // ...  
}
```

```
String[] array = new String[5];  
// ...  
for (String item : array) {  
    System.out.println(item);  
}
```

for each

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

forever

# Classi e oggetti

- Classe:
  - Ogni classe è definita in un package
  - Descrive un nuovo tipo di dato, che ha variabili e metodi
- Oggetto
  - Istanza di una classe, che è il suo modello di riferimento



# Metodo

- Blocco di codice che ha:
  - return type
  - nome
  - lista dei parametri
  - (lista eccezioni che può tirare)
- Associato a
  - una istanza (default)
  - o a una classe (static)

signature



```
public class Simple {  
    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");  
    }  
}
```

# Parametri

- In Java i valori sono passati a funzioni “by value”
- Primitivi:
  - Il parametro è una copia del valore passato. La sua eventuale modifica non si riflette sul valore originale
- Reference
  - Il parametro è una copia della reference passata. L'oggetto referenziato è lo stesso e dunque una eventuale modifica si riflette sul chiamante
  - Nota che:
    - immutabili, come String e wrapper, non possono essere modificati per definizione
    - ogni reference può essere null, va controllata prima dell'uso: `Objects.requireNonNull()`

# Constructor (ctor)

- Metodo speciale, con lo stesso nome della classe, invocato durante la creazione di un oggetto via “new” per inizializzarne lo stato
- Non ha return type (nemmeno void)
- Ogni classe può avere svariati ctor, ognuno dei quali deve essere distinguibile in base al numero/tipo dei suoi parametri
- 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 `substring(int, 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

# Unit Test

- Verifica (nel folder test) la correttezza di una “unità” di codice, permettendone il rilascio da parte del team di sviluppo con maggior confidenza
- Un unit test, tra l'altro:
  - dimostra che una nuova feature ha il comportamento atteso
  - documenta un cambiamento di funzionalità e verifica che non causi malfunzionamenti in altre parti del codice
  - mostra come funziona il codice corrente
  - tiene sotto controllo il comportamento delle dipendenze

# JUnit in Eclipse

- Right click sulla classe (Simple) da testare
  - New, JUnit Test Case
    - JUnit 4 o 5 (Jupiter)
    - Source folder dovrebbe essere specifica per i test
  - Se richiesto, add JUnit library to the build path
- Il wizard crea una nuova classe (SimpleTest)
  - I metodi che JUnit esegue sono quelli annotati @Test
  - Il metodo statico fail() indica il fallimento di un test
- Per eseguire un test case: Run as, JUnit Test

# Struttura di un test JUnit

- Ogni metodo di test dovrebbe
  - avere un nome significativo
  - essere strutturato in tre fasi
    - Preparazione
    - Esecuzione
    - Assert

```
public int negate(int value) {  
    return -value;  
}
```

Simple.java

SimpleTest.java

```
@Test  
public void negatePositive() {  
    Simple simple = new Simple();  
    int value = 42;  
  
    int result = simple.negate(value);  
  
    assertThat(result, equalTo(-42));  
}
```

# @BeforeEach

- I metodi annotati @BeforeEach (Jupiter) o @Before (4) sono usati per la parte comune di inizializzazione dei test
- Ogni @Test è eseguito su una nuova istanza della classe, per assicurare l'indipendenza di ogni test
- Di conseguenza, ogni @Test causa l'esecuzione dei metodi @BeforeEach (o @Before)

```
private Simple simple;

@BeforeEach
public void init() {
    simple = new Simple();
}

@Test
public void negatePositive() {
    int value = 42;

    int result = simple.negate(value);

    assertThat(result, equalTo(-42));
}
```

# JUnit assert

- Sono metodi statici definiti in `org.junit.jupiter.api.Assertions` (Jupiter) o `org.junit.Assert` (4)
  - `assertTrue(condition)`
  - `assertNull(reference)`
  - `assertEquals(expected, actual)`
  - `assertEquals(expected, actual, delta)`
- `assert Hamcrest-style`, usano `org.hamcrest.MatcherAssert.assertThat()` e `matcher` (`org.hamcrest.CoreMatchers`)  
`assertThat(T, Matcher<? super T>)` n.b: convenzione opposta ai metodi classici: `actual` – `expected`
  - `assertThat(condition, is(true))`
  - `assertThat(actual, is(expected))`
  - `assertThat(reference, nullValue())`
  - `assertThat(actual, startsWith("Tom"))`
  - `assertThat(name, not(startsWith("Bob")))`

```
assertEquals(.87, .29 * 3, .0001);
```

Per altri matcher (`closeTo`, ...) vedi hamcrest 2.1+



# Esercizi

- Implementare i seguenti metodi, verificarli con JUnit
  - speed(double distance, double time)
    - Distanza e tempo → velocità media
  - distance(int x0, int y0, int x1, int y1)
    - Distanza tra due punti (x0, y0) e (x1, y1) in un piano
  - engineCapacity(double bore, double stroke, int nr)
    - Alesaggio e corsa in mm, numero cilindri → cilindrata in cm cubi
  - digitSum(int value)
    - Somma delle cifre in un intero

# Esercizi /2

- checkSign(int value)
  - “positive”, “negative”, o “zero”
- isOdd(int value)
  - Il valore passato è pari o dispari?
- asWord(int value)
  - “zero”, “one” ... “nine” per [0..9], altrimenti “other”
- vote(double percentile)
  - $F \leq 50$ , E in (50, 60], D in (60, 70], C in (70, 80], B in (80, 90],  $A > 90$
- isLeapYear(int year)
  - Anno bisestile?
- sort(int a, int b, int c)
  - Ordina i tre parametri

# Esercizi /3

- `sum(int first, int last)`
  - somma tutti i valori in `[first, last]` (o zero), p.es:  $(1, 3) \rightarrow 6$  e  $(3, 1) \rightarrow 0$
- `sumEven(int first, int last)`
  - somma tutti i numeri pari nell'intervallo
- Per un (piccolo) intero, scrivere metodi che calcolano:
  - il fattoriale
  - il numero di Fibonacci (0, 1, 1, 2, 3, 5, 8, ...)
  - la tavola pitagorica (ritornata come array bidimensionale)

# Esercizi /4

- reverse(String s)
  - Copia ribaltata
- isPalindrome(String s)
  - È un palindromo?
- removeVowels(String s)
  - Copia ma senza vocali
- bin2dec(String s)
  - Dalla rappresentazione binaria di un intero al suo valore
- reverse(int[] data)
  - Copia ribaltata
- average(int[] data)
  - Calcolo della media
- max(int[] data)
  - Trova il massimo

# Tre principi OOP

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

# Lo “scope” delle variabili

- **Locali** (automatiche)
  - Esistenza limitata
    - a un metodo
    - a un blocco interno
- Member (field, property)
  - **di istanza** (default)
  - **di classe** (static)

```
public class Scope {  
    private static int staticMember = 5;  
    private long 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;  
    }  
}
```

# Access modifier per data member

- Aiuta l'incapsulamento
  - Privato
- Dubbio
  - Protetto
- Normalmente sconsigliati
  - Package (default)
  - Pubblico

Static initializer

Costruttore

```
public class Access {  
    private int a;  
    protected short b;  
    static double c;  
    // public long d;  
  
    static {  
        c = 18;  
    }  
  
    public Access() {  
        this.a = 42;  
        this.b = 23;  
    }  
  
    // ...  
}
```

# Access modifier per metodi

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

```
public class Access {  
    // ...  
  
    static private double f() {  
        return c;  
    }  
  
    void g() {  
        f();  
    }  
  
    public int h() {  
        return a / 2;  
    }  
}
```



# 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");
```

```
private int i;           // 0  
private boolean flag;    // false  
private String t;        // null
```

# Final

- Costante primitiva

```
final int SIZE = 12;
```

- Reference che non può essere riassegnata

```
final StringBuilder sb = new StringBuilder("hello");
```

- Metodo di istanza che non può essere sovrascritto nelle classi derivate

```
public final void f() { // ...
```

- Metodo di classe che non può essere nascosto nelle classi derivate

```
public static final void g() { // ...
```

- Classe che non può essere estesa

```
public final class FinalSample { // ...
```

# Tipi wrapper

- Controparte reference dei tipi primitivi
  - Boolean, Character, Byte, Short, Integer, Float, Double
- Boxing esplicito
  - Costruttore (deprecato da Java 9)
  - Static factory method
- 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 = j;
```

# Alcuni metodi statici dei wrapper

- Boolean
  - valueOf(boolean)
  - valueOf(String)
  - parseBoolean(String)
- Integer
  - parseInt(String)
  - toHexString(int)
- Character
  - isDigit(char)
  - isLetter(char)
  - isLetterOrDigit(char)
  - isLowerCase(char)
  - isUpperCase(char)
  - toUpperCase(char)
  - toLowerCase(char)

# 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;  
    }  
}
```

# L'annotazione Override

- Annotazione: informazione aggiuntiva su di un elemento
- @Override
  - Annotazione applicabile solo ai metodi, genera un errore di compilazione se il metodo annotato non definisce un override
- **Override**: il metodo definito nella classe derivata ha la stessa signature e tipo di ritorno di un metodo super (che non deve essere final). La visibilità dell'override non può essere inferiore del metodo super
- **Overload**: metodi con stesso nome ma signature diversa
- Signature di un metodo: nome, numero, tipo e ordine dei parametri

# 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

- **this** è una reference all'oggetto corrente
- **super** indica al compilatore che si intende accedere ad un metodo di una superclass dal contesto corrente
- ctor → ctor: (primo statement)
  - **this()** – nella classe
  - **super()** – nella superclass



# Esempio di ereditarietà

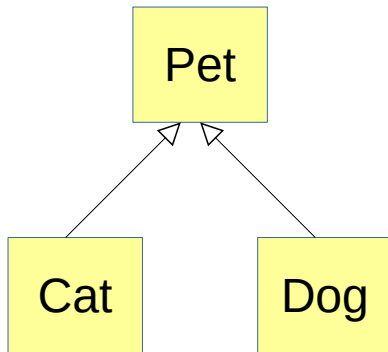
```
public class Pet {  
    private String name;  
  
    public Pet(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

```
Dog tom = new Dog("Tom");  
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 (rischioso)
  - Protetto con l'uso di `instanceof`



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

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

# Eccezioni

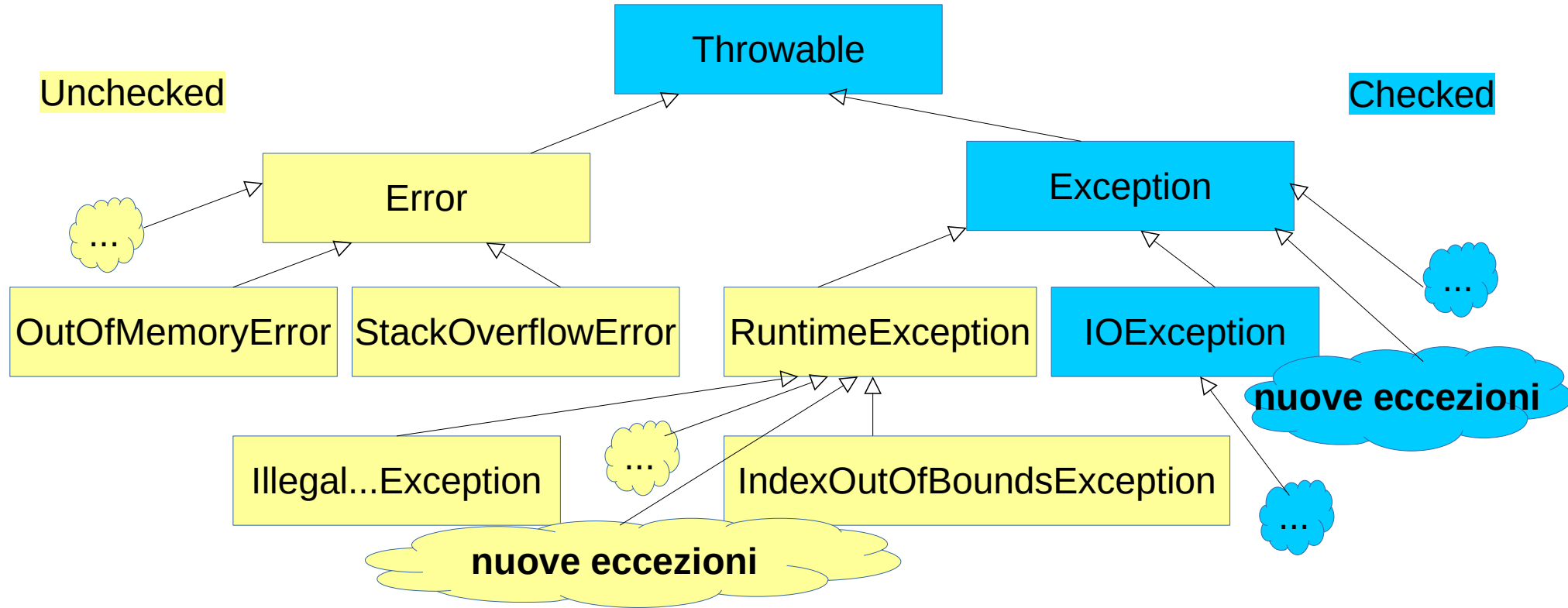
- Obbligano il chiamante a gestire gli errori
  - Unhandled exception → terminazione del programma
- Evidenziano il flusso normale di esecuzione
- Semplificano il debug esplicitando lo stack trace
- Possono chiarire il motivo scatenante dell'errore
- Checked vs unchecked

# 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.

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

# Gerarchia delle eccezioni





# Test eccezioni in JUnit 3

Math.abs() di  
Integer.MIN\_VALUE  
è  
Integer.MIN\_VALUE!

```
public int negate(int value) {  
    if(value == Integer.MIN_VALUE) {  
        throw new IllegalArgumentException("Can't negate MIN_VALUE");  
    }  
    return -value;  
}
```

```
@Test  
void negateException() {  
    Simple simple = new Simple();  
  
    try {  
        simple.negate(Integer.MIN_VALUE);  
    } catch (IllegalArgumentException iae) {  
        String message = iae.getMessage();  
        assertThat(message, is("Can't negate MIN_VALUE"));  
        return;  
    }  
    fail("An IllegalArgumentException was expected");  
}
```

# JUnit 4.7 ExpectedException

```
@Rule
public ExpectedException thrown = ExpectedException.none();

@Test
public void negateMinInt() {
    thrown.expect(IllegalArgumentException.class);
    thrown.expectMessage("Can't negate MIN_VALUE");

    Simple simple = new Simple();
    sample.negate(Integer.MIN_VALUE);
}
```

Nel @Test  
si dichiara  
quale eccezione  
e messaggio  
ci si aspetta

Si definisce una  
variabile di istanza  
ExpectedException  
taggata come @Rule

# JUnit 5 `assertThrows()`

Il metodo fallisce se quanto testato non tira l'eccezione specificata

L'eccezione attesa viene tornata per permettere ulteriori test

```
@Test
public void negateMinInt() {
    IllegalArgumentException exc = assertThrows(IllegalArgumentException.class, //
        () -> simple.negate(Integer.MIN_VALUE));
    assertThat(exc.getMessage(), is("Can't negate MIN_VALUE"));
}
```

L'assertion è eseguita su di un Executable, interfaccia funzionale definita in Jupiter

# Date e Time

- java.util
  - Date
  - DateFormat
  - Calendar
    - GregorianCalendar
  - TimeZone
    - SimpleTimeZone
- java.time (JDK 8)
  - LocalDate
  - LocalTime
  - LocalDateTime
  - DateTimeFormatter, FormatStyle
  - Instant, Duration, Period
- java.sql.Date



implementazioni  
più chiare,  
immutabili e  
thread-safe

# LocalDate e LocalTime

- Non hanno costruttori pubblici
- Factory methods: `now()`, `of()`
- Formattazione via `DateTimeFormatter` con `FormatStyle`
- `LocalDateTime` aggrega `LocalDate` e `LocalTime`

```
LocalDate date = LocalDate.now();
System.out.println(date);
System.out.println(LocalDate.of(2019, Month.JUNE, 2));
System.out.println(LocalDate.of(2019, 6, 2));
System.out.println(date.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)));

LocalTime time = LocalTime.now();
System.out.println(time);

LocalDateTime ldt = LocalDateTime.of(date, time);
System.out.println(ldt);
```

# java.sql Date, Time, Timestamp

- Supporto JDBC a date/time SQL
  - Date, Time, Timestamp
- Conversioni
  - \*.valueOf(Local\*)
  - Date.toLocalDate()
  - Time.toLocalTime()
  - Timestamp.toLocalDateTime()
  - Timestamp.toInstant()

# La libreria java.io

- Supporto a operazioni di input e output
- In un programma solitamente i dati sono
  - Letti da sorgenti di input
  - Scritti su destinazioni di output
- Basata sul concetto di stream
  - Flusso sequenziale di dati
    - binari (byte)
    - testuali (char)
  - Aperto in lettura o scrittura prima dell'uso, va esplicitamente chiuso al termine
  - Astrazione di sorgenti/destinazioni (connessioni di rete, buffer in memoria, file su disco ...)

# File

- Accesso a file e directory su memoria di massa
- I suoi quattro costruttori
  - `File dir = new File("/tmp");`
  - `File f1 = new File("/tmp/hello.txt");`
  - `File f2 = new File("/tmp", "hello.txt");`
  - `File f3 = new File(dir, "hello.txt");`
  - `File f4 = new File(new URI("file:///C://tmp/hello.txt"));`

Forward slash anche per Windows



# Metodi per il test di File

- exists()
- isFile()
- isDirectory()
- isHidden()
- canRead()
- canWrite()
- canExecute()
- isAbsolute()

# Alcuni altri metodi di File

- `getName()` // "hello.txt"
- `getPath()` // "\\tmp\\hello.txt"
- `getAbsolutePath()` // "D:\\tmp\\hello.txt"
- `getParent()` // "\\tmp"
- `lastModified()` // 1559331488083L
- `length()` // 4L
- `list()` // ["hello.txt"]

usa separatore (`File.separator`)  
e formato del SO corrente

UNIX time in milliseconds

se invocato su una directory:  
array dei nomi dei file contenuti

# Scrittura in un file di testo

- Gerarchia basata sulla classe astratta **Writer**
  - **OutputStreamWriter** fa da bridge tra stream su caratteri e byte
    - Ridefinisce i metodi `write()`, `flush()`, `close()`
  - **FileWriter** costruisce un `FileOutputStream` da un `File` (o dal suo nome)
  - **PrintWriter** gestisce efficacemente l'`OutputStream` passato con i metodi `print()`, `println()`, `printf()`, `append()`
- 

```
File f = new File("/tmp/hello.txt");
PrintWriter pw = new PrintWriter(new FileWriter(f));
pw.println("hello");
pw.flush();
pw.close();
```

# Lettura da un file di testo

- Gerarchia basata sulla classe astratta Reader
- InputStreamReader fa da bridge tra stream su caratteri e byte
  - Ridefinisce i metodi read() e close()
- FileReader costruisce un FileInputStream da un File (o dal suo nome)
- BufferedReader gestisce efficacemente l'InputStream passato con un buffer e fornendo metodi come readLine()

```
File f = new File("/tmp/hello.txt");  
BufferedReader br = new BufferedReader(new FileReader(f));  
String line = br.readLine();  
br.close();
```

# Input con Scanner

- Legge input formattato con funzionalità per convertirlo anche in formato binario
- Può leggere da input Stream, 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 Adder {
    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();
}
```



# Inner class

- Nested class: classe definita all'interno di un'altra classe
- La nested class ha accesso diretto ai 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

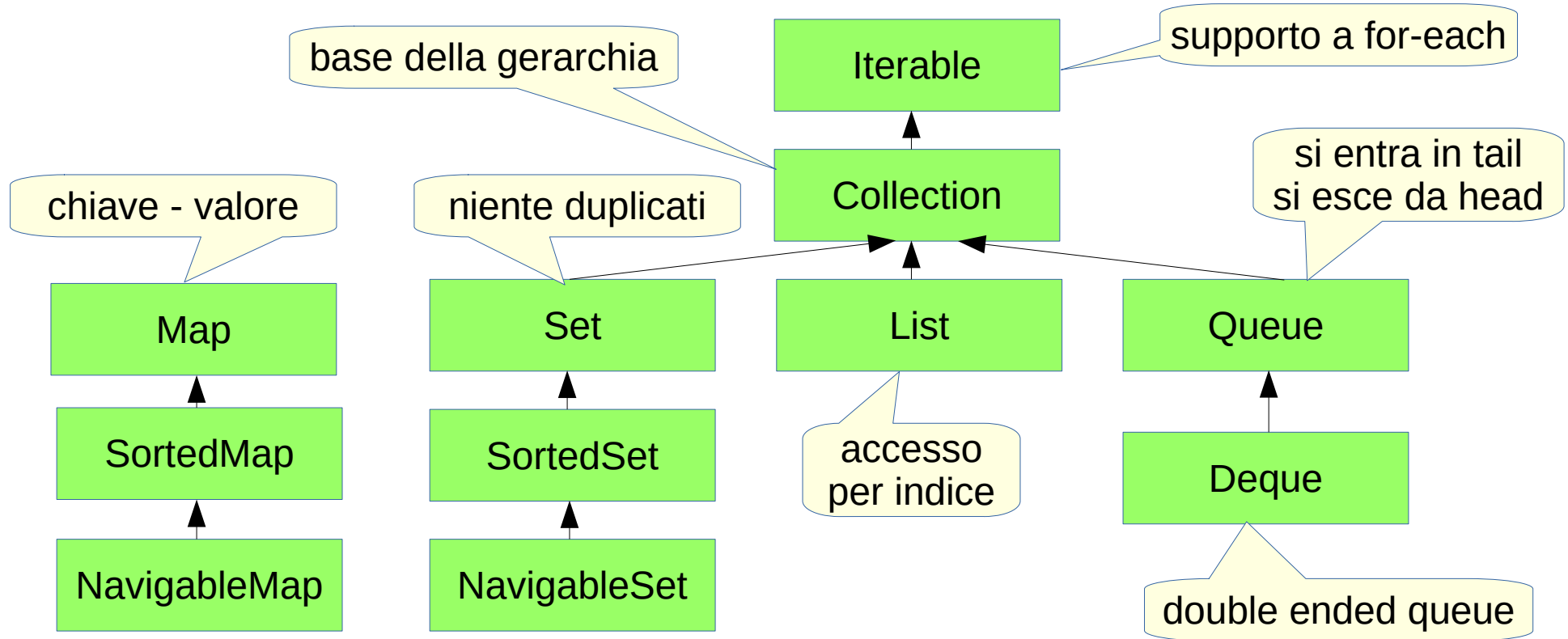
# Generic

- 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 reference types
- Il tipo (o tipi) utilizzato dal generic è indicato tra parentesi angolari (minore, maggiore)

# 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

# Reflection

- Package `java.lang.reflect`
- Permette di ottenere a run time informazioni su di una classe
- “Class” è la classe che rappresenta una classe
- “Field” rappresenta una proprietà, “Method” un metodo, ...

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

Tutti i metodi

Una specifica proprietà

```
Field field = ArrayList.class.getDeclaredField("elementData");  
field.setAccessible(true);  
Object[] data = (Object[]) field.get(al);
```

# 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()