

Corso Web MVC

Java SE

Emanuele Galli

www.linkedin.com/in/egalli/

general purpose: non è pensato per problemi specifici. Imperativo perché dà ordini alla CPU, e ci obbliga a lavorare pensando il tutto come a classi e oggetti che interagiscono tra di loro. Multi platform perché funziona come i linguaggi interpretati e non compilati. E' stato inventato per lavorare bene su internet, quindi network centric.

Java

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

strumenti

jvm è un programma eseguito dal sistema operativo sull'hardware. lo installo e ricevo il programma java (x.class, diventato così da x.java file di testo, dopo essere stato compilato da javac e che adesso contiene il bytecode) lo posso eseguire, indipendentemente dalla piattaforma che ho. la jvm mi traduce il bytecode in linguaggio macchina specifico, altrimenti il bytecode non cambia, rimane invariato. Da qui il programma viene eseguito sulla CPU.

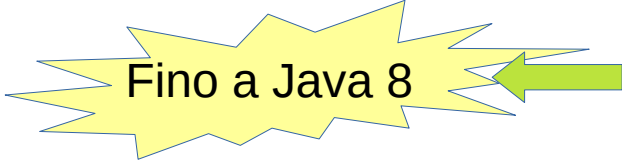
- **JVM**: Java Virtual Machine

ambiente per l'esecuzione di java, è per l'utente. E' sufficiente per eseguire i codici scritti da altri, quindi se si vuole solo eseguire basta questo.

- **JRE**: Java Runtime Environment

- **JDK**: Java Development Kit

se invece vuoi sviluppare, quindi il programmatore, deve avere la jdk. Un insieme di tool che comprende le altre due e mi permette di sviluppare il codice.



Fino a Java 8

Versioni

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



SE: Standard Edition

serve per il web

EE: Enterprise Edition

LTS: Long-Term Support

Link utili

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

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>

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

Java SE Downloads

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

<https://adoptopenjdk.net/>

quelle di oracle sono standard, ma per scaricarle bisogna registrarsi. L'ultimo invece è libero.

path: variabile che mi aiuta a ricercare un programma

Hello Java a riga di comando

- Generazione del **bytecode**.class

`javac Hello.java` codice sorgente

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

- Visualizzazione del bytecode disassemblato*

`javap -c Hello`

- Generazione del codice macchina ed esecuzione

`java Hello`

jvm: eseguiami la classe Hello!

il nome della classe deve essere uguale al nome del file. Hello-Hello, sempre con la maiuscola.

programmino per scrivere hello. ho bisogno della prima funzione: **public static void** (non ritorno niente) **main** (chiamato, invocato dalla java virtual machine). Il nome della funzione, più i parametri (input), fanno parte della signature. Args in questo caso è un array di stringhe. All'interno delle graffe c'è il *body*, con l'istruzione. Il *metodo* è una funzione all'interno della classe, si chiama così per abbreviare. Out sarà lo schermo, dove ricevo l'output. `println` vuol dire stampa e vai a capo (line) aspettando la prossima azione. Con la prima graffa chiudo la classe, con la seconda chiudi il main.

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



se nel prompt scrivo `java` (che è la virtual machine) e poi `Hello`, mi ha eseguito il `main` della classe trasformando il codice macchina e stampando `Hello!`

Version Control System (VCS)

- Obiettivi
 - Mantenere traccia dei cambiamenti nel codice; sincronizzazione del codice tra utenti
 - Cambiamenti di prova senza perdere il codice originale; tornare a versioni precedenti
- Architettura client/server (CVS, Subversion, ...) il più utilizzato
 - 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

voglio modificare un file nel repository lo blocco, lo metto in lock, così nessuno lo tocca, e una volta modificato lo libero, lo committo. il problema è che se il file è molto grosso nessuno ti può aiutare nella risoluzione. chiunque voglia accedere a quel file deve entrare nel repository e scaricarselo.

non c'è un file dominante rispetto all'altro. posso scaricarmi la copia e una volta che faccio il commit lo posso caricare sulla mia locale e non sul server/repository. dopo aver fatto il commit si fa il push, che sincronizza con la macchina a disposizione di tutti quanti. prima si fa il pull, ovvero mi metto giù tutte le nuove versioni del repository perché altra gente sta modificando. il git dopo il push cerca di mergiare la versione prima con la mia.

Git

- 2005 by 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
 - github.com, 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

dal git clono il repository nel mio computer, creando un folder con la directory .git (il vero repository, con dentro la storia del progetto); per lavorarci guardo i file che stanno fuori e sono comprensibili; quando ho finito di cambiarli faccio il commit che me li ributta nel repository locale. Da lì con il pull controllo che non ci siano stati altri cambiamenti, poi push me lo ributta in repository remoto.

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

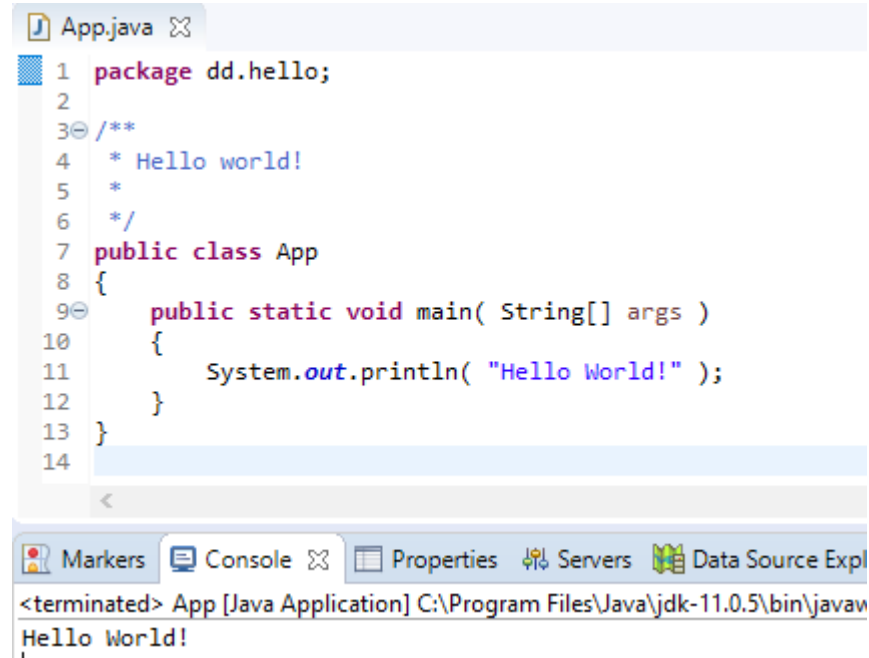
- clone <url>: clona un repository in locale
- add <filename(s)>: stage per commit
- commit -m "message": copia sul repository locale
- commit -am "message": add & commit
- status: lo stato del repository locale
- push: da locale a remoto
 - push --set-upstream origin <branch>
- pull: da remoto a locale
- 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)

ambienti di sviluppo: mi permette di rimanere in un programma ed effettuare operazioni in un colpo solo, più un supporto per velocizzare il tutto. è un'applicazione

- 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 with a file named '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 of the program:

```
<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

insieme a Gradle sono i due più importanti strumenti di sviluppo, Gradle è più per sviluppo android. Maven e questi tools ci permettono di costruire i file in modo automatico e tenerli sotto controllo. Invece di scrivere javac ogni volta per ogni mia operazione, lo dico a maven che lo fa per tutti i codici. Packaging: per unire tutti i file .class da dare al cliente, maven li riunisce in un Jar che poi sarebbe un file zip: tiene dentro la struttura dei miei file .class per l'eseguibile.

Maven è già integrato negli ambienti di sviluppo, gratis, per creare e compilare il jar ma anche per fare il test in background mentre lavoro.

xml documento che mi fa rappresentare informazioni in maniera molto rigorosa, con struttura ad albero, tramite parentesi angolari: <project> (< tag) vuol dire che sto creando la root del mio progetto; per chiuderlo </project>. non funziona così solo la radice, ma anche i "figli", cioè quello che c'è dentro, i nodi RICORDA sempre di chiuderle. xml lavora su stringhe. Il pom è uno di questi documenti, un file di configurazione che mi descrive come usare Maven per il progetto corrente. Sa già che voglio lavorare con Java, devo solo specificargli la versione.

librerie: raccolte di codici (funzionalità) scritte da qualcun altro; framework invece è più rigido rispetto alla libreria, mi dà la struttura da seguire.

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

, insieme alle dependency sono i due file principali all'interno del pom.

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

utf-8 tabella di conversione. Dopo la tabella Ascii. Per memorizzare numeri al posto di caratteri.

```
<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)

junit: libreria che mi
permette di fare il testing

```
<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

tutti i file java devono essere all'interno di un package, con nome (è il folder dove metto il mio file java).

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**

mi genera la documentazione automatica del file

la classe contiene il codice sorgente, a sua volta dentro metodi (blocchi di codice con un nome, che ci permettono di raggiungere un certo scopo. Grazie al nome posso chiamare la mia funzione in altre parti del codice, per implementare funzionalità. Grazie al parametro che do posso adattare la funzione al mio caso particolare).

Main è il metodo standard invocato dalla jvm, a cui passa gli argomenti; println viene invocato, cioè eseguito. Senza il main non si esegue.

Public: può essere usata dall'esterno, non è privata. In un file ci può essere una sola classe public che deve avere il nome del file stesso: le altre classi interne sono private e devono avere una certa utilità (raro).

```
/*      commento si apre così e si chiude con asterisco e slash. Il commento con
 *      doppio slash finisce dove finisce il commento.
 * A simple Java source file
 */
package s028;

vogliamo usare funzionalità che arrivano da un'altra parte
import java.lang.Math; // not required

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

class PackageClass {
    // TBD    to be discussed, commento per i posteri
}
```

Struttura del codice /2

- Metodi

- **main** (definito)
- **println** (invocato)

il metodo può ricevere un input e dare un output

- Parentesi

per delimitare i blocchi: fondamentali

- **Graffe** (blocchi, body di classi e metodi)
- **Tonde** (liste di parametri per metodi)
- **Quadre** (array)

le istruzioni

- **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)

ad es: intero, stringa..dentro una variabile c'è una sequenza di bit, ma devo dichiararne il tipo

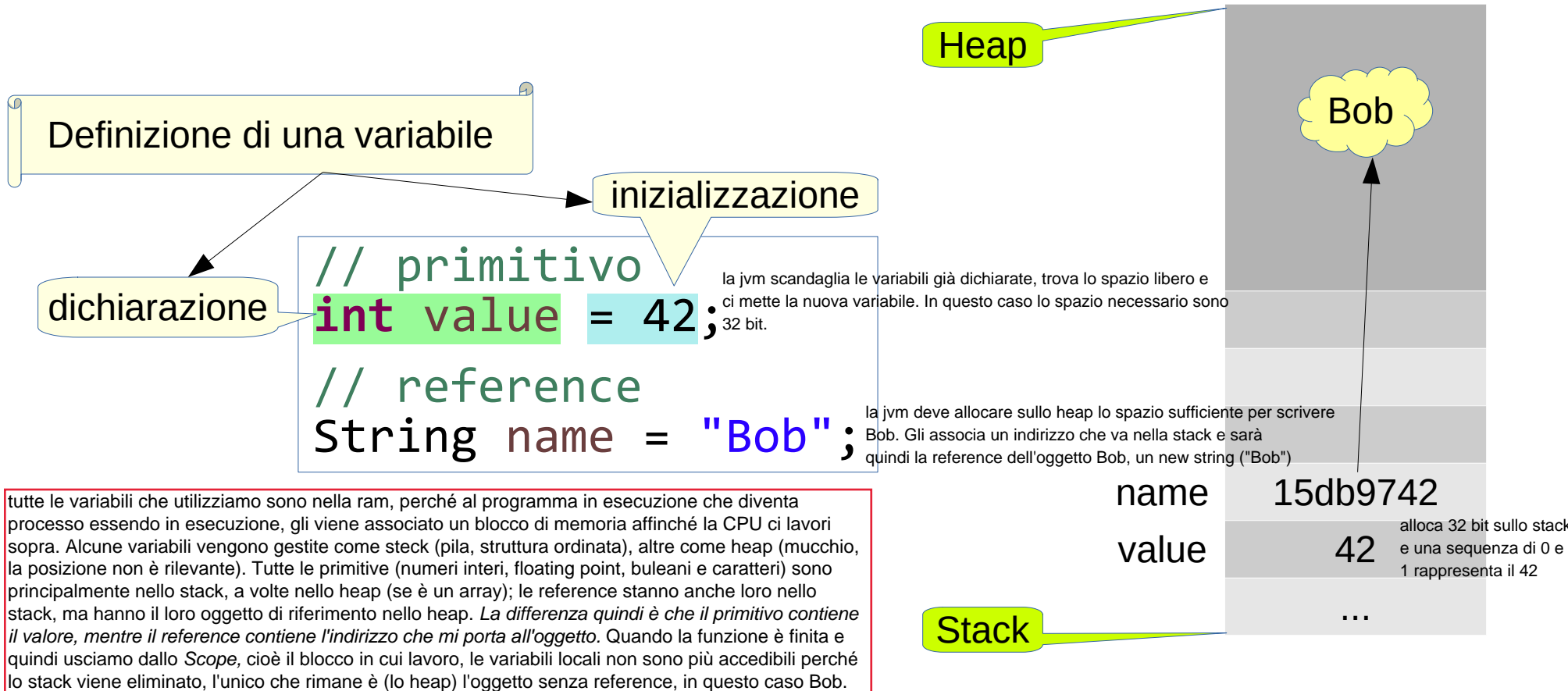
Tipi primitivi

bit	tutto minuscolo		signed integer	interi	floating point IEEE 754
1(?)	boolean	false true	1 bit è sempre riservato al segno, non lo posso usare per rappresentare il numero.		
8 bit			byte	-128 127 256 valori in tutto	
16	char	'\u0000' '\uFFFF'	short	-32,768 32,767	
32			int	-2^{31} $2^{31} - 1$	float avendo solo 32 bit non riesce a rappresentare i valori. Java pensa automaticamente che i numeri con vengano rappresentati da double
64			long	-2^{63} $2^{63} - 1$	



va messa in fondo al numero

Primitivi vs Reference



String

- Reference type che rappresenta una sequenza immutabile di caratteri

non è che non posso chiamare metodi su quella stringa, semplicemente me ne tornerà una nuova.

- StringBuilder, controparte mutabile, per creare stringhe complesse

è una classe. è come se fosse una bozza della mia stringa immutabile, ci lavoro sopra e la modifico fino a che non sono contenta

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

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

Forma standard

Forma semplificata equivalente

mi ricordo che ci sarebbe il new perchè essendo String immutabile non posso trattarla da primitivo

Operatori unari

lavora su un solo operando, lavora con una sola variabile. se ++ e -- stanno a sinistra del valore funzionano in modo "naturale"; se stanno a destra mi stampa il valore corrente, dopodiché lo incrementa. il + da solo non vuol dire niente, mentre il - cambia il valore corrente. è come se moltiplicassi per -1.

++ 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) se entrambi gli operandi sono interi

% modulo serve per rappresentare il riporto della divisione: 10/3 fa 3 e l'1 lo rappresento con % = quindi 10%3 fa 1

```
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

ad un operatore solo assegno più significati=overload (se uso il `+` per i numeri sommo, se ho le stringhe diventa una concatenazione)

è meglio convertire un numero in una stringa che il contrario. Sarebbe impossibile convertire una stringa in un numero. Così, se devo sommare una stringa ad un numero, ottengo una terza stringa senza complicazioni, grazie al metodo `toString`.

Lo statement è il comando fino al ;

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

la `b` barrata rappresenta uno spazio

Operatori relazionali

La jvm si limita a riportare la prima stringa `alpha < beta`, mentre mettendo l'operazione con parentesi tonde riesce a risolverla perché è un booleano e quindi al posto del risultato metterà la stringa "true" che poi si va a concatenare con la stringa precedente.

<	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

or esclusivo

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

per avere true devono essere entrambi true

```
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
```

fa 5 perché parto da destra e faccio 2 alla 0, 2 alla 1 e 2 alla 2= 5 (la base è 2 perché per la macchina è comodo lavorare sul sistema 0 e 1) 0b mi fa capire che sto lavorando in binario

se sono due interi NON posso usare gli shortcut

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

per comodità, metto il doppio operatore || di modo che la macchina si fermi nel fare l'operazione, velocizzandosi: nel secondo esempio, alpha è true quindi il risultato sarà per forza true, non c'è bisogno che guardi il valore di beta. Se invece voglio che la jvm continui con la sua operazione, perché voglio chiamare entrambe le funzioni, metto la stanghetta unica |. Si preferisce quindi l'uso degli shortcut. I doppi sono

Operatori di assegnamento

=	Assegnamento <small>diverso dal doppio uguale, che è il confronto</small>
+=	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; (come dire a= a+2) // 10
```

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

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

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

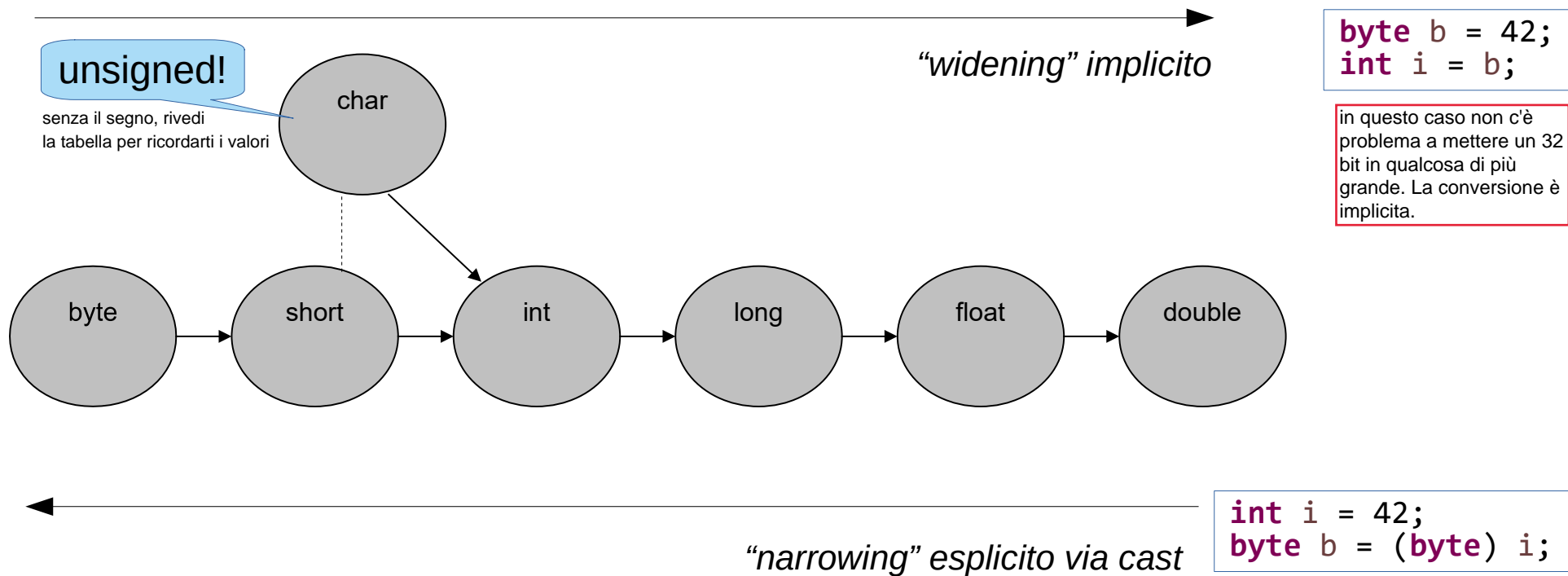
```
alpha %= 5; // 2
```

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

print è la chiamata a funzione

l'intero 42 ha 32 bit, ma se io voglio metterlo in un byte e quindi in 8 bit come faccio? con la dicitura cast cioè (byte) gli sto dicendo di eliminare i bit che non mi interessano (quelli che stanno a sinistra, con l'esponente più alto e sarebbero i più significativi) e prendere solo quello che mi interessa, mettendolo in 8 bit.

Cast tra primitivi



*abbiamo definito la variabile array tramite l'operatore new, affinché java allochi uno spazio di memoria nello heap che mi contenga 12 elementi. Quando un array ha una certa dimensione, non si può più modificare perché lo spazio di memoria a lui riservato ormai è quello. Se ho già i valori come nel secondo riquadro, creo un nuovo array con già i valori dentro e la lunghezza quindi non deve essere inizializzata, ce l'ho già.

gli array sono reference, il vero array è nello heap

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`

"è un array di interi di 12 elementi", Java li inizializza automaticamente a 0.

```
*int[] array = new int[12];  
array[0] = 7;
```

5 è la posizione

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

in questo caso invece avrò il primo elemento che è un 7 e glielo dico, gli altri saranno tutti zeri. Se non ho nessun dettaglio del mio array inizializza a "null"

l'ultimo sarà 11 perché si parte sempre da 0. 12 non va bene.

```
int[] array = { 1, 4, 3 }; l'ultimo elemento è length-1
```

```
// array[array.length] = 21; // exception
```

```
System.out.println(array.length); // 3
```

length=quanti elementi ci sono dentro

array bidimensionale. è un array di array

```
int[][] array2d = new int[4][5];
```

```
int value = array2d[2][3]; prima le righe e poi le colonne
```

[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

come java implementa le operazioni condizionali, quando c'è da fare un branch nel codice piuttosto che un altro.

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

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

istruzione

nextStep();

se la condizione è falsa vado direttamente al next step.

-> parentesi graffa con altgr+shift+[

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

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

se le altre 2 sono false, allora fai questo...

nextStep();

switch

modo alternativo per fare gli "if". in questo caso il break è fondamentale, altrimenti il flusso non si ferma e va avanti

per definire costanti

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

```
int value = 42;

// ...
"guardami value"
switch (value) {
case 1:
    // ...
    break;
case 2:
    // ...
    break;
default: value è 1? no; è 2? no;
    // ... default allora guarda
    // ... tutti gli altri casi
    break;
}
```

etichetta ->

```
String value = "1";

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

```
public enum WeekendDay {
    SATURDAY, SUNDAY
}
```

costanti scritte sempre in maiuscolo
prima di usare l'enum lo devo dichiarare, come le classi. questo perché gli do un determinato range di valori, oltre ai quali non si va.

```
WeekendDay day = WeekendDay.SATURDAY;

// ...

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

loop

"fino a che la condizione è vera, ripeti questo blocco di codice"

```
while (condition) {
    // ...

    if (something) {
        condition = false;
    }
}
```

il do while, almeno una volta esegue il blocco, è un while al contrario

```
do {
    // ...

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

indice

condizione

incrementa i di 1

```
for (int i = 0; i < 5; i++) {
    // ...
    // variabile di loop

    if (i == 2) {
        continue;
    }

    // ...
}
```

istruzione che c'è solo nel loop.
fatto il check sulla condizione,
se è falsa esco dal blocco ed
eseguo lo statement successivo.
se è vera, incrementa i, verifica la
condizione e si ricomincia.

forever

```
for (;;) {
    // ...

    if (something) {
        break;
    }

    // ...
}
```

5 reference a stringhe, che inizializza a "null". Sarebbe stato 0 se fosse stato un array di interi

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

for each

l'item conterrà ogni volta il
reference per l'elemento corrente
non lavoriamo sull'indice, ma sulla reference al mio array nello heap

è il modello su cui costruisco l'oggetto

il particolare , è modellato sulla classe. Qui utilizzo effettivamente i metodi.

Classi e oggetti

- Classe:

- Ogni classe è definita in un package
- Descrive un nuovo tipo di dato, che ha variabili e metodi

un folder, directory.
quando creiamo una
classe definiamo un
nuovo tipo di dato

ecco perché c'è la
parentesi graffa aperta
che si chiude in fondo in
fondo

- Oggetto

- Istanza di una classe, che è il suo modello di riferimento

Le proprietà (chiamate anche field) sono variabili definite all'interno della classe e mi rappresentano lo stato della classe o dell'oggetto; i metodi mi rappresentano le funzionalità della classe

Reference a MyClass

Crea un oggetto MyClass

keyword
`MyClass reference = new MyClass();`

istanziare la classe, cioè creare l'oggetto

la jvm crea l'oggetto per lo heap, mentre il mio reference dell'oggetto viene allocato sullo stack. Ricordati le lettere maiuscole.

non posso fare import in una classe da un'altra, se sono nello stesso package.

una funzione definita, all'interno di una classe. lo riconosco dalle parentesi tonde

dopo nome, tipo, parametri

Metodo

definiamo la classe:

Return è la parola chiave con cui il metodo ritorna il risultato al chiamante. I metodi chiamano e vengono chiamati.

- Blocco di codice che ha:

- **return type** in questo caso voglio che mi ritorni una stringa, il tipo indica cosa voglio che mi ritorni
- **nome** signature
- **lista dei parametri** signature
- (lista eccezioni che può tirare)

parametro: variabile locale associata al metodo, inizializzata dal chiamante (altre parti del codice).

il punto è l'operatore di deferenziazione, serve a chiamare la funzione.

- Associato a
 - una istanza (default)
 - o a una classe (**static**)

metodi che non sono assegnati all'oggetto, ma alla classe
math è un metodo statico, come main

nel file sarà Simple.java

```
public class Simple {  
    static String h() { metodo statico, se non creo l'oggetto posso usare solo questo.  
        return "Hi";  
    }  
  
    int f(int a, int b) { metodo di istanza, è legato a un oggetto  
        return a * b;  
    }  
  
    void g(boolean flag) { "nulla", g non ritorna niente, non posso assegnare il risultato a qualcosa  
        if (flag) { metodo di istanza  
            System.out.println("Hello");  
        }  
        System.out.println("Goodbye");  
    }  
}
```

il costruttore è un metodo che non ritorna niente. è un void.

Parametri

metodi

- In Java i valori sono passati a funzioni “by value” è il modo in cui il parametro viene inizializzato, cioè copiando l'x del chiamante.
- Primitivi:
 - Il parametro è una copia del valore passato. La sua eventuale modifica non si riflette sul valore originale
- Reference l'array è un reference. l'array vero e proprio è nello heap, mentre il reference sta nello stack.
 - Il parametro è una copia della reference passata. L'oggetto referenziato è lo stesso e dunque una eventuale modifica si riflette sul chiamante diversamente dai primitivi, in cui il chiamato può fare tutte le modifiche del caso, perché il chiamante non le vedrà
 - 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()` se tira l'eccezione vuol dire che c'è un null

Constructor (ctor)

per inizializzare le proprietà della mia classe, altrimenti c'è uno spazio vuoto in memoria con un "null". Con "new" creo, col costruttore inizializzo. In seguito nello spazio ci sarà un reference ad un'altra stringa che gli darà il nome. Bisogna invocare una variabile temporale che, uscita dal costruttore, scompare. Viene così creata una stringa nuova nello heap.

- Metodo speciale, con lo stesso nome della classe, invocato durante la creazione di un oggetto via “new” per inizializzarne lo stato si può anche chiamare con super () /this () senza utilizzare il nome del costruttore = particolare
- 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) solo se non c'è nessun altro costruttore, il compilatore ne crea uno di default.

N.B= il For funziona bene con gli array, ma può essere usato anche per utilizzare la "i" come contatore, di quante volte voglio fare un loop in generale. La formula è la stessa, ma al posto di i<array.length, c'è i< numero di volte in cui voglio che venga fatto il loop.

Alcuni metodi di String

sostituisce le [] quando nell'array voglio cercare il carattere in una determinata posizione.
su un array scriverei [1], quindi scrivo "Bob".charAt(1) per prendere il carattere in posizione 1

- **char charAt(int)**

se voglio dire che il primo oggetto viene prima e il secondo viene dopo, ritornerò un -1; se voglio che abbiano stesso valore metto 0; se voglio mettere prima il secondo ritorno il +1

- **int compareTo(String)**

- **String concat(String)** di solito si usa il + che è più comodo

- **boolean contains(CharSequence)**

- **boolean equals(Object)** (equals per stringhe, per primitivi ==)

- **int indexOf(int)** (anche 'c' invece che int)

- **int indexOf(String)**

- **boolean isEmpty()** la lunghezza è 0

- **int lastIndexOf(int ch)**

- **int length()**

se parlo di reference, è rischioso
fare comparazioni perché magari
sono reference che sembrano uguali
ma guardano a oggetti diversi.

cerca all'interno della stringa il carattere che passo e sostituisco con un altro

- **String replace(char, char)**

- **String[] split(String)**

- **String substring(int), String substring(int, int)** mi prende e ritorna la sottostringa

- **String toLowerCase()**

- **String toUpperCase()**

- **String trim()**

Tra i metodi statici:

metodi che si possono chiamare a
prescindere dall'oggetto stringa.

- **String format(String, Object...)**

unisce una sequenza di caratteri in una stringa sola

- **String join(CharSequence, CharSequence...)**

- **String valueOf(Object)**

converte un oggetto in stringa, qualunque esso sia

Alcuni metodi di StringBuilder

- `StringBuilder(int)`
- `StringBuilder(String)`
- modifica lo string builder corrente `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()`

mi serve per calcolare
funzioni matematiche:
"voglio calcolare un
coseno?" guardo su math

non creiamo oggetti math

math è più una collezione di metodi statici. questi
metodi statici ci permettono di generare casi
particolari. i più usati sono min e max.

La classe Math

è un raccoglitore di metodi statici

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

se utilizzo un'altra libreria per il mio codice, devo verificare tramite lo unit test che io possa effettivamente continuare a utilizzare il mio codice.

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

devo verificare il mio risultato,
asserisco che deve essere così

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

Simple.java

SimpleTest.java

```
@Test  
public void negatePositive() {  
    Simple simple = new Simple();  
    int value = 42;  
    chiamo il metodo  
    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)` il reference deve essere null, sennò fallisco
 - `assertEquals(expected, actual)` il loro contenuto deve essere uguale
 - `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

domande: il tempo può essere negativo? se il tempo è 0 ci sono problemi?

- 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

voglio che mi torni una stringa a seconda del segno del numero

- `checkSign(int value)`
 - “positive”, “negative”, o “zero”
- `isOdd(int value)` true or false?
 - Il valore passato è pari o dispari?
- `asWord(int value)` la stringa corrispondente
 - “zero”, “one” ... “nine” per [0..9], altrimenti “other”
- `vote(double percentile)` converti il numero in abcdef
 - F ≤ 50, E in (50, 60], D in (60, 70], C in (70, 80], B in (80, 90], A > 90
- `isLeapYear(int year)`
 - Anno bisestile? se è divisibile per 4
- `sort(int a, int b, int c)`
 - Ordina i tre parametri

Esercizi /3

va fatto sicuramente un loop

- `sum(int first, int last)`
compreso
– somma tutti i valori in `[first, last]` (o zero), p.es: `(1, 3) → 6` e `(3, 1) → 0`
somma di 1,2,3
first è più grande di last allora torna 0
- `sumEven(int first, int last)`
– somma tutti i numeri pari nell'intervallo
- Per un (piccolo) intero, scrivere metodi che calcolano:
 - il fattoriale non occorre utilizzare l'1 perché nella moltiplicazione è il valore neutro
 - il numero di Fibonacci (0, 1, 1, 2, 3, 5, 8, ...)
 - la tavola pitagorica (ritornata come array bidimensionale) la tabellina

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
"010101" da trasformare in un numero es.
- reverse(int[] data)
 - Copia ribaltata
- average(int[] data) se devo calcolare la media non può tornare un intero, lo devo trasformare in double per forza.
 - Calcolo della media
- max(int[] data)
 - Trova il massimo

Tre principi OOP

object oriented programming. Lo scopo delle classi è tenere insieme elementi che si riferiscono ad una stessa classe. es: la classe math.

Il modo in cui noi scriviamo le classi. La classe (è come uno schema) raccoglie al suo interno quello che mi serve per definire gli oggetti (implementazioni della classe). Una parte di incapsulamento è il racchiudere metodi e proprietà all'interno di una classe, l'altra include il concetto di visibilità.

- Incapsulamento per mezzo di classi

(black box: non so cosa c'è dentro, ho solo delle funzionalità per accederle)

- Visibilità pubblica (metodi) / privata (proprietà)

lo stato del mio oggetto

per indicare se un metodo/proprietà può essere chiamato al di fuori della classe o meno. In genere le proprietà, i dati, sono privati, non modificabili. I metodi invece, le funzionalità, sono accessibili.

- Ereditarietà in gerarchie di classi

- Dal generale al particolare

se ho qualcosa di comune a più classi la astraggo (funzionalità e proprietà), la rendo standard, per gestire al meglio il mio programma e non avere ripetizioni. "Object" è alla base di tutta la gerarchia delle classi.

- Polimorfismo

- Una interfaccia, molti metodi (override)

anche se ho lo stesso metodo, può essere ridefinito (cioè con un nome) a seconda dell'istanza particolare del mio oggetto. Posso descrivere uno stesso metodo su più classi della stessa gerarchia, comportandosi in modo diverso a seconda dell'oggetto.

Lo “scope” delle variabili

gestite automaticamente dalla CPU, inserite e rimosse dallo stack.

i reference invece non sono automatici

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

può esistere anche se non esistono oggetti in riferimento alla classe

```
public class Scope {  
    private static int staticMember = 5;  
    private long member = 5; ogni volta che creo un oggetto avrò member come  
                                proprietà, diversa a seconda dei casi  
  
    public void f() { più lo scope in cui lavora la variabile è limitato, più è facile tenerla  
                    sotto controllo e sapere chi ci sta lavorando  
        long local = 7;  
        if (staticMember == 2) {  
            short inner = 12; nasce e muore qui, all'interno di questo blocco  
            staticMember = 1 + inner;  
            member = 3 + local;  
        } è sottointeso this.member  
    }  
  
    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

protected: privato per tutti, ma accessibile dalla classe stessa o anche sui derivati della classe, o sui metodi che appartengono allo stesso package. Rimane privato solo per una classe che non c'entra proprio niente.

- Normalmente sconsigliati

- Package (default)

- Pubblico

ma solo per le proprietà. System.OUT.println, mi fa capire che è pubblica. ormai è così e non si può cambiare.

Static initializer

Costruttore

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

```
    static { per i data member statici. rari.
        c = 18;
    }
```

```
    public Access() {
        this.a = 42;
        this.b = 23;
    } "l'oggetto corrente", sto mettendo 42 nella proprietà
    dell'oggetto corrente, tramite costruttore.
```

```
    // ...
```

```
}
```

Access modifier per metodi

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

se ho un metodo lunghissimo perché devo fare cose complicate, lo spezzo in una serie di metodi che non possono essere chiamati al di fuori della classe, quindi dico che sono privati.

```
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

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

- Implicita by default (solo member)

inizializzata automaticamente, in base a quello che è

- primitivi
 - numerici: 0
 - boolean: false
- reference: null

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

Final

significato principale: per identificare le costanti, che non possono più essere modificate (non nel caso reference) e sono scritte in caps lock se non sono stringhe.

Altri significati:

-se un metodo è final, nessuno lo può ridefinire

-lassa perde

-se una classe è final, non voglio che venga estesa e quindi non voglio gerarchie.

- Costante primitiva

`final int SIZE = 12;` la sequenza di bit contenuta non cambia

- Reference che non può essere riassegnata

`final StringBuilder sb = new StringBuilder("hello");` il contenuto può cambiare, ma sb no, perché è il reference ad essere final.

- 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 prendo il primitivo e lo metto nella scatola wrapper, unboxing faccio il contrario
 - Costruttore (deprecato da Java 9)
 - Static factory method
- Unboxing esplicito
 - Metodi definiti nel wrapper
- Auto-boxing
- Auto-unboxing

è una tipologia di classe legata ai primitivi, è la controparte es: voglio avere metodi per fare qualcosa con gli interi? chiamo la classe Integer che, essendo classe, può includere metodi, cosa che con un int primitivo non posso fare.
per `int i=7` creo la classe `Integer i2=7`, ma `i2` non è il primitivo, è un reference all'oggetto che conterrà 7 (il vero int) nello heap.
Avere i primitivi mi permette di risparmiare tempo e memoria perché l'accesso è diretto all'elemento.

```
Integer i = new Integer(1);
```

fino a Java 8

```
Integer j = Integer.valueOf(2);
```

da Java 9, questo è lo standard

```
int k = j.intValue();
```

unboxing

```
Integer m = 3;
```

stessa cosa dello standard, ma più breve e implicito

```
int n = j;
```

stessa cosa in questa unboxing perché j è un reference e viene assegnato a un primitivo

Alcuni metodi statici dei wrapper

- Boolean

quando c'è "value of" sto creando un oggetto

- `valueOf(boolean)`
- `valueOf(String)` la stringa contiene "false" ma io voglio mi ritorni il booleano.
- `parseBoolean(String)`

- Integer

- `parseInt(String)` trasformare la stringa contenente "418" in un intero 418
- `toHexString(int)` torna l'intero in una stringa in base esadecimale, usando 16 simboli. (totalmente inutile)

- Character

- `isDigit(char)` se un carattere è una cifra torna true
- `isLetter(char)`
- `isLetterOrDigit(char)`
- `isLowerCase(char)`

- `isUpperCase(char)` quando c'è "is" vuol dire che se il carattere è maiuscolo è true. si comporta come un booleano
- `toUpperCase(char)` un singolo carattere diventa maiuscolo, grazie al "to", mi ritorna.
- `toLowerCase(char)`

interface

una specie di classe in cui non ci sono proprietà né definizione dei metodi, c'è solo la loro dichiarazione. Infatti il metodo che chiamo non ha parentesi graffa, non apre nessun gruppo di codice: mi dice solo cosa devo fare, quale metodo chiamare. Così facendo posso chiamare il metodo appartenente all'interface in questione, in modo da sapere quale metodo contiene. Una volta fatto lo definisco nella classe che mi interessa, con le graffe. (Implementa, la relazione che c'è tra classe e l'interface a cui mi riferisco).

- Cosa deve fare una classe, non come deve farlo (fino a Java 8) così facendo posso far fare la stessa cosa anche a due classi diverse, che in realtà non potrebbero. fanno riferimento alla stessa interfaccia
- Una class “implements” una interface
- Un'interface “extends” un'altra interface esistono gerarchie anche di interface, un'interfaccia deriva da un'altra.
- 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();  
}
```

anche se non c'è scritto questa proprietà è statica

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

il metodo è già stato definito da qualcun altro (è un aiuto ma non fondamentale)

extends vs implements

la classe implementa l'interfaccia. le interfacce estendono altre interfacce

```
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 come @test
- **@Override** sto ridefinendo un metodo già esistente: quel metodo è quindi polimorfico
 - 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** i parametri cambiano (ad es. int, stringa, ecc, che aggiungo magari al mio metodo che avevo precedentemente)
- Signature di un metodo: nome, numero, tipo e ordine dei parametri

abstract class

posso decidere di crearla quando non
mi serve quella classe nel mio codice

non posso creare oggetti in questa classe, non è
istanziabile. Il metodo all'interno è concreto. infatti
l'abstract class è una via di mezzo tra la classe e
l'interface.

- Una classe abstract non può essere istanziata
- Un metodo abstract non ha body non ha le graffe
- Una classe che ha un metodo abstract deve essere abstract, ma non viceversa se il metodo è abstract anche la classe deve esserlo
- 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)** includo nella classe la proprietà che ha, non esiste un extend o altre parole. NO parole chiave.
 - Classe che ha come proprietà un'istanza di un'altra classe
 - p. es.: Tail in Cat e Dog

Ereditarietà in Java

una classe può avere una sola superclasse.
Se vuole avere più caratteristiche deve rifarsi a un'interfaccia (tutto quello che è taggato come public, != da interface. Mi interfaccio con le altre classi usando questi metodi).
Se non estende altre classi, implicitamente estende Object, quindi i metodi qui contenuti sono accessibili a tutti.
è un'ereditarietà transitiva, transita tra le varie relazioni.

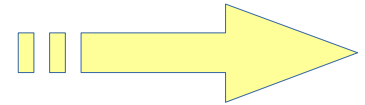
- Single inheritance: una sola superclass
- Implicita derivazione da Object (che non ha superclass) by default il metodo toString è qui dentro
- Una subclass può essere usata al posto della sua superclass (is-a)
- Una subclass può aggiungere proprietà e metodi a quelli ereditati dalla superclass (attenzione 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

entrambi ci fanno riferire ad un dettaglio della classe stessa, quando parliamo di metodi di istanza. Di metodo/proprietà statiche ce n'è uno per classe, mentre qui parliamo dell'oggetto corrente: visto con effetto cipolla, quindi completo. Uso this. e super. per chiamare. Quando dico super, mi riferisco alla sola classe che sta sopra rispetto a quella che mi interessa, non a tutte le altre che la mia classe estende a catena.

per accedere alla proprietà dell'oggetto o a un costruttore nella stessa classe.

- **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 si riferisce a un altro costruttore, che fa l'inizializzazione al posto del costruttore di default (che non ha nessun parametro)
 - **super()** – nella superclass



Esempio di ereditarietà

quando non trovo qualcosa in Dog passo tutte le classi sopra in ordine finché non trovo. In questo caso, Pet.

```
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();
```

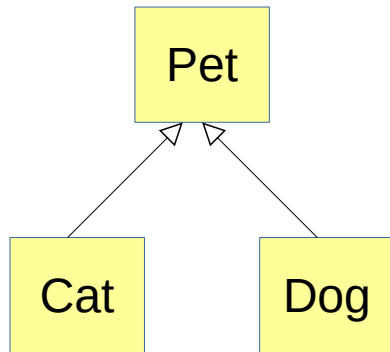
posso chiamarlo in due modi, o chiamo solo il nome o nome e velocità.

```
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;  
    }  
}
```

costruttore
sto chiamando un costruttore perché non c'è il punto ma la tonda dopo this
costruttore
anche se 0 è un int, lo posso convertire a double: 0.0
speed è la proprietà dell'oggetto corrente
metodo

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"); //lo posso fare, è un upcasting.
```

```
Dog dog = (Dog) pet; // OK le parentesi sono il cast pet (reference) è un dog(oggetto), lo faccio tornare un reference a dog con dog
```

```
Cat cat = (Cat) pet; // trouble at runtime
```

```
if(pet instanceof Cat) { // OK "solo se pet è un'istanza di gatto" e siccome non è verificato, è impossibile da fare. quindi il downcasting non si può fare.
```

```
    Cat tom = (Cat) pet;
```

```
}
```

Il contrario non è sicuro, con il downcast lo proteggo inserendo instanceof: operatore, alla sua sinistra metto il reference che mi interessa controllare, a destra la classe con cui la voglio confrontare. Con if, prima di fare il cast mi accerto che pet sia davvero un gatto.

Eccezioni

è successo qualcosa che non mi aspettavo

- 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

aiuta a trovare anche gli errori
- Possono chiarire il motivo scatenante dell'errore

ad esempio se non posso per qualche motivo leggere un file che mi serviva per il mio codice
- 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 { qui c'è il codice che tengo sotto controllo, il flusso normale di  
           esecuzione  
        g();  
    } catch (Exception ex) { lo eseguo se trovo  
                               un'eccezione  
        // ...  
    } finally { non è obbligatorio, ma quando c'è va eseguito sempre  
        cleanup();  
    }  
}  
  
// ...  
  
public void g() throws Exception {  
    // ...  
    if (somethingUnexpected()) {  
        throw new Exception(); è un oggetto di tipo Exception  
    }  
}
```

Gerarchia delle eccezioni

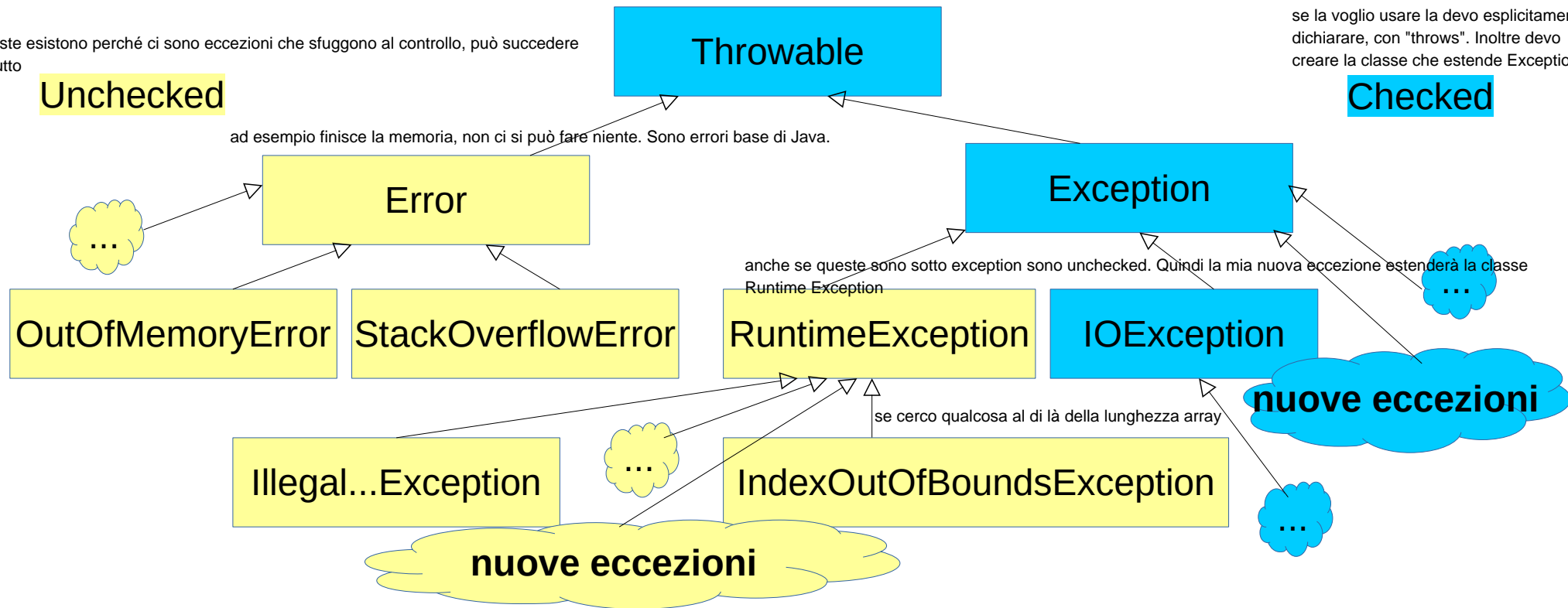
l'oggetto Eccezione appartiene a questa gerarchia, per usarla la devo sapere

se la voglio usare la devo esplicitamente dichiarare, con "throws". Inoltre devo creare la classe che estende Exception

Checked

Unchecked

ad esempio finisce la memoria, non ci si può fare niente. Sono errori base di Java.



Test eccezioni in JUnit 3

Math.abs() di
Integer.MIN_VALUE
è
Integer.MIN_VALUE!

quando mi aspetto che il mio metodo tiri un
eccezione, devo fare in modo che il mio test mi dia
un' eccezione.

```
public int negate(int value) { è unchecked, non gli scrivo throws  
    if(value == Integer.MIN_VALUE) {  
        throw new IllegalArgumentException("Can't negate MIN_VALUE");  
    } se cerco di cambiargli il segno rimane uguale, perché non esiste il corrispettivo positivo del minvalue  
    return -value;  
}
```

```
@Test  
void negateException() {  
    Simple simple = new Simple();  
  
    try {  
        simple.negate(Integer.MIN_VALUE);  
    } catch (IllegalArgumentException iae) {  
        String message = iae.getMessage();  
        assertTrue(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 attenzione con java.util.date



implementazioni
più chiare,
immutabili e
thread-safe

LocalDate e LocalTime

- Non hanno costruttori pubblici
- Factory methods: `now()`, `of()` metodi statici che creano nuovi oggetti
- 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); aggregazione, has a.
System.out.println(ldt);
```

java.sql Date, Time, Timestamp

- Supporto JDBC a date/time SQL
 - Date, Time, Timestamp
- Conversioni bisogna sempre convertire da java a sql e viceversa
 - *.valueOf(Local*)
 - Date.toLocalDate()
 - Time.toLocalTime()
 - Timestamp.toLocalDateTime()
 - Timestamp.toInstant()

La libreria java.io

- Supporto a operazioni di **input** e **output** dati che vengono copiati dal mio processo per andare da un'altra parte
- In un programma solitamente i dati sono
 - Letti da sorgenti di input connessione ad un altro computer, file, ecc
 - Scritti su destinazioni di output schermo, file, destinazione altro processo
- Basata sul concetto di **stream** generalizzazione di tutti questi concetti
 - Flusso sequenziale di dati
 - binari (byte)
 - testuali (char)
 - Aperto in lettura o scrittura prima dell'uso, va esplicitamente chiuso al termine dalla tastiera posso solo leggere, sulla console posso solo scrivere
 - Astrazione di sorgenti/destinazioni (connessioni di rete, buffer in memoria, file su disco ...) se non chiudo il SO spreca risorse

File

- Accesso a file e directory su memoria di massa

- I suoi quattro costruttori

crea una nuova directory

deve già esistere

- `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"));` assoluto

Forward slash anche per Windows

Metodi per il test di File

- `exists()` se la directory esiste davvero, true or false
- `isFile()`
- `isDirectory()`
- `isHidden()`
ad esempio .git è un file nascosto
- `canRead()`
- `canWrite()`
- `canExecute()`
- `isAbsolute()`
se è completo, cioè se parte dalla radice, vedi slide prima

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

la data in formato unix

non torna un int ma un long. num di byte

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();
```

sono due wrapper, per dare funzionalità a file che su file e basta non posso avere metodi

scriviamo hello sul file e non sulla console

porta giù i dati, il buffer che memorizza non mi serve più perché ho finito, se non faccio il flush non memorizzo definitivamente sul disco. Porta a termine le scritture che sono in corso

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();
```

mi permette di leggere una riga per volta

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 il file da cui vogliamo leggere (potrebbe essere anche la tastiera) allo stream in lettura
 - Loop su `hasNext...()` per determinare se c'è un se c'è qualcosa da leggere token in lettura del tipo atteso
 - Con `next...()` si legge il token pezzo ben delimitato
 - 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); // la tastiera (input standard)
        while (scanner.hasNext()) { // finché scanner ha qualcosa da darmi looppa
            if (scanner.hasNextDouble()) {
                result += scanner.nextDouble();
            } else {
                System.out.println("Bad input, discarded: " + scanner.next()); // stampa tutto ciò che è bad input
            }
        }
        scanner.close(); // see try-with-resources
        System.out.println("Total is " + result);
    }
}
```

try-with-resources

per evitare di scrivere il close. Interfaccia che ha la definizione del metodo close, solo quello

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());
        }
    }
}

a questo punto la close viene fatta comunque, perché col try ho capito che sto implementando autocloseable, anche se ci fosse un'eccezione

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

Java Util Logging

scopo del log = mi permette di rintracciare dove sono i problemi, stampandolo nel file e non solo sulla console

```
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");
}
```

livello di interesse del messaggio

```
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); il livello è sulla scala del riquadro di sinistra
    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

un esempio è la collezione, o il concetto di Map ed Entry: l'ultima è inner class della prima

Generic

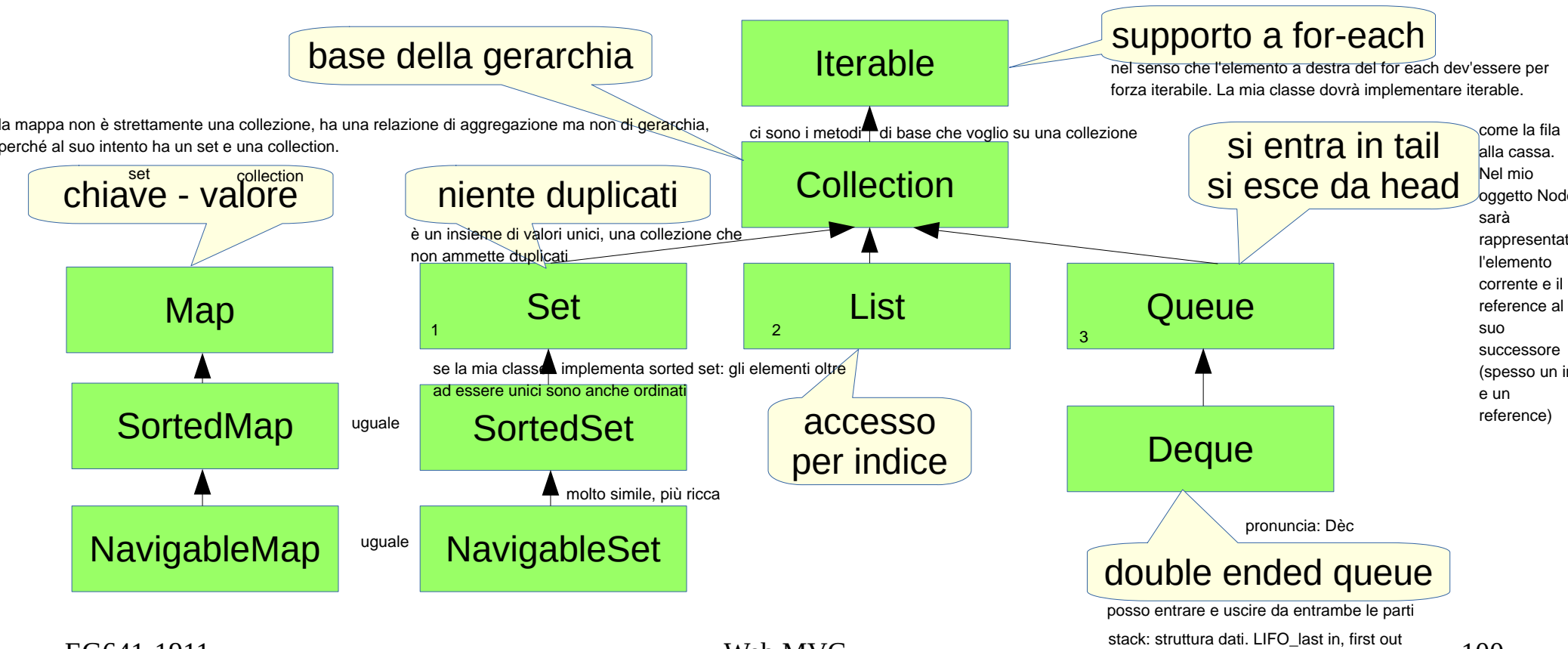
Espressa con l'uso delle parentesi angolari < >. E' una classe in cui devo specificare i parametri su cui lavoro, e Map ne è un esempio. Le collezioni sono fatte per contenere oggetti, quindi int non è accettabile come parametro, Integer sì perché è un reference. Vantaggio fondamentale: miglioro la sicurezza del codice, poiché indicando nei parametri cosa voglio nessuno può inserire un altro oggetto (es. voglio come key una stringa e come value un integer, nessuno può mettere dentro un Dog)

- 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 passare da una collezione all'altra con facilità
- Basate su alcune interfacce standard gerarchia di interfacce usate per generare le collezioni vere e proprie. in più, Collections contiene metodi statici generici e l'interfaccia Iterator mi dice il modo in cui posso iterare sul mio metodo
- 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)` E= element, tipo generico. Voglio aggiungere ad esempio una stringa alla mia collezione: se si può fare torna true.
- `boolean addAll(Collection<? extends E>)`
- `void clear()` rimuovere gli elementi che ci sono nella collezione
- `boolean contains(Object);`
- `boolean equals(Object);`
- `boolean isEmpty();`
- `Iterator<E> iterator();` permette di iterare su tutti gli elementi della collezione
- `boolean remove(Object);`
- `boolean retainAll(Collection<?>);`
- `int size();` è come dire length. Ricorda: per le stringhe metodo length, per array proprietà length, per le collezioni size
- `Object[] toArray();` converte la collezione in un array
- `<T> T[] toArray(T[]);`

vedi foglio esercizi

Alcuni metodi in List<E>

- void add(int, E) aggiunge l'elemento nella posizione int
- E get(int) mi ritorna l'elemento
- int indexOf(Object) tornami l'indice dell'elemento
- E remove(int)
- E set(int, E) rimuovo l'elemento in una data posizione e ci metto il valore

Alcuni metodi in SortedSet<E>

- E first()
- E last()
- SortedSet<E> subSet(E, E) gli elementi compresi in un certo intervallo

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) elimina l'elemento in cima allo stack (quello più a destra in un array), aggiungi l'elemento in cima allo stack
- E removeFirst(), E removeLast()

Alcuni metodi in Map<K, V>

map entry: inner class che mi permette di gestire chiave e valore

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) metto dentro solo se non c'è già
- 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>

la parte di sinistra mi dice che abstract data type stiamo usando (cioè in memoria cosa stiamo utilizzando e come), a destra il tipo di interfaccia che viene implementata.

è la più usata, molto meglio di un array. quando non sai che collezione usare usa questa. La uso quando devo tenere insieme una serie di valori ed è comodo accedervi tramite indice.

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

capacità: memoria che occupa nello heap. E' inizializzata a 10 di default. Se ho una capacità specifica chiamo il secondo costruttore per risparmiare tempo. L'ultimo mi permette di passare da una qualunque collezione ad ArrayList. La mia collezione dovrà essere E o un tipo che la estende.

LinkedList<E>

- implements List<E>, Deque<E>
- Lista doppiamente linkata
- Accesso diretto solo a head e tail^{costo $O(1)$}
- 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

gli chiedo di trovarmi un elemento
anche tra 10.00000 e lui me lo ritorna
subito.

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, $O(\log(N))$ è proprio la struttura ad albero che mi permette di mantenere l'ordine.
- Gli elementi inseriti devono implementare Comparable ed essere tutti mutualmente comparabili se devo metterli in ordine devono essere comparabili
- Ctors:
 - TreeSet() // vuoto, ordine naturale
 - TreeSet(Collection<? extends E>) // copy
 - TreeSet(Comparator<? super E>) // sort by comparator impongo al costruttore di usare una comparazione che gli dico io
 - TreeSet(SortedSet<E>) // copy + comparator

TreeSet e Comparator

collezione

è un metodo sttico perché c'è la maiusc Arrays (classe che contiene tutti i metodi per lavorare con gli array)

```
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);  
    }  
}
```

creo il comparator

```
MyStringComparator msc = new MyStringComparator();
```

creo l'oggetto

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

il treeset usa questo comparator
tutti gli elementi in data vanno in ts2

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

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

ordine naturale

comparator

plain

reversed

Java 8 lambda

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()