

Programmazione Orientata agli Oggetti

Introduzione al Linguaggio di
Programmazione Java

Sommario

- Breve riassunto della storia del linguaggio Java
 - Motivazioni del linguaggio e scelte progettuali
 - Versione di Riferimento
- Macchina virtuale JVM
- La portabilità e la piattaforma Java
- Librerie Standard Java
- Java vs C
- Java vs C++
- Controllo del Flusso
- Tipi di dato primitivi
 - Un tipo molto particolare: **String**
- Errori a tempo di compilazione e di esecuzione
- Diagnostica in Java vs in C
- Limiti del Paradigma Procedurale

Breve Storia (1)

- Java è un linguaggio di programmazione sviluppato dalla Sun Microsystems e pubblicato nella sua versione 1.0 nel 1995
- Lo sviluppo di Java iniziò nei primi anni '90 sotto la guida di James Gosling
 - Inizialmente il nome del linguaggio era *Oak* (quercia)
 - ed era pensato per sistemi embedded (televisori, sportelli bancomat, palmari, ecc...)
- Verso il 1993 l'interesse verso i dispositivi embedded, per cui Java era stato pensato, crollò
 - Java aveva (ben presto!) perso il suo originale e principale dominio di applicazione!

Breve Storia (2)

- Tuttavia, sempre nel 1993, la NCSA rilasciò *Mosaic*
 - Uno dei primi browser con interfaccia grafica
 - Rese più accessibile Internet
- Il team di Java decise allora di dedicarsi allo sviluppo di un sistema embedded per i browser: le **applet**
 - Le applet sono/erano piccole applicazioni che potevano essere eseguite all'interno del browser
 - Lo scopo era quello di arricchire le pagine web tramite l'uso di contenuto interattivo
- Perché il concetto era rivoluzionario? il codice veniva scaricato da remoto ed eseguito (in maniera sicura) localmente. *Javascript* non c'era ancora...
- Ci fu un significativo effetto traino. Ma oggi, quanti sanno cosa sia un'applet?

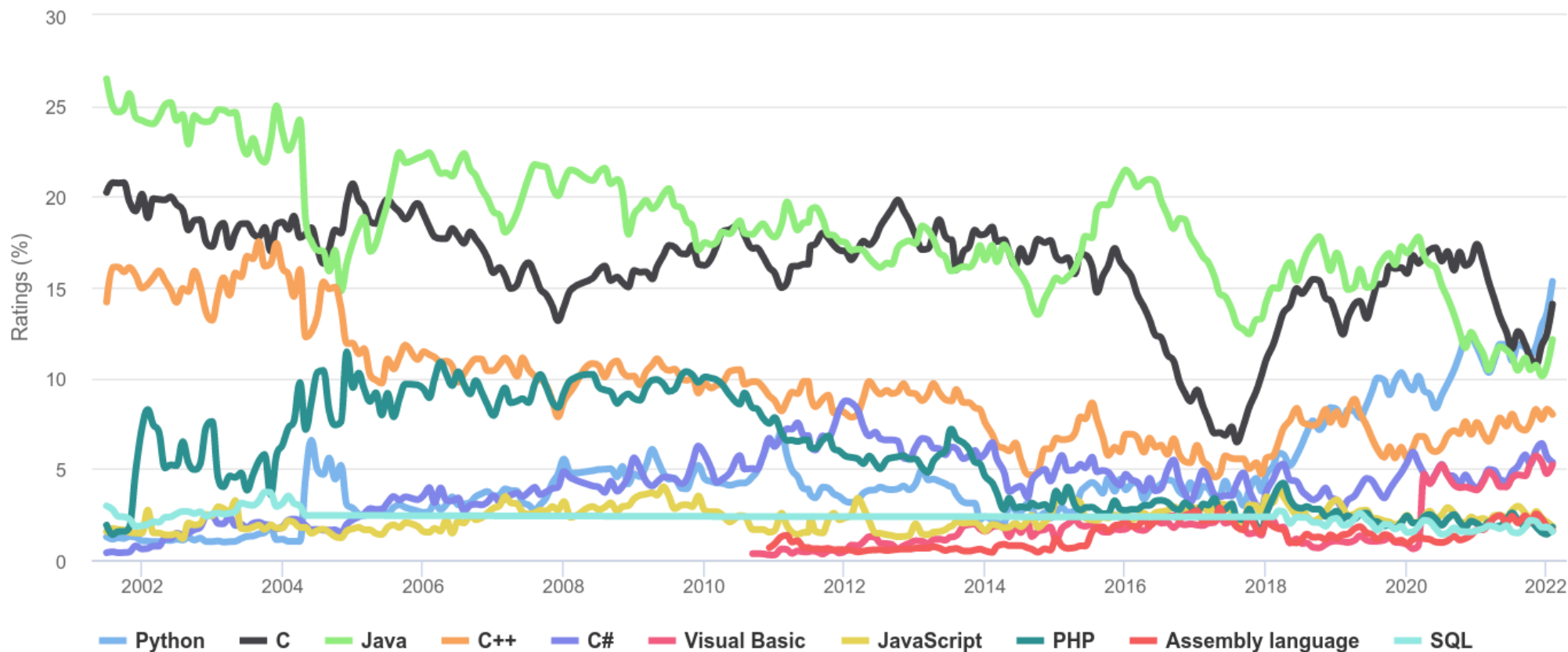
Breve Storia (3)

- Le applet sono ormai obsolete
 - Ma cavalcando gli anni della diffusione di massa di internet, hanno sicuramente contribuito a rendere Java uno dei linguaggi più popolari tra gli sviluppatori dell'epoca
- La popolarità e diffusione di Java sono dovuti a molti fattori
- Tra i principali sicuramente l'incremento di produttività dei programmatori che lo adottarono
- Java semplifica alcuni aspetti rispetto al C/C++
 - possiede un gran numero di strumenti facenti parte della *piattaforma* che semplificano grandemente lo sviluppo
 - riduce significativamente il livello di *expertise* necessario ad un programmatore per essere produttivo
 - In sintesi, Java ampliò la disponibilità di manodopera (>>)

Breve Storia (4)

- Oltre 20 anni dopo la nascita e la sua diffusione di massa, rimane molto adottato sia nell'industria che nell'accademia
- Talvolta forse addirittura con eccessivo entusiasmo (RM3!)
- Ha conosciuto nuovi ed importanti ambiti di utilizzo
 - Applicazioni su Web
 - Sistemi embedded
 - App Android
 - E molti altri ancora (*"3 Billion Devices Run Java"*)
- Vari motivi dietro la *durata* di questo successo (>>)
 - pur avendo accumulato molti difetti, ha saputo innovarsi senza rompere la retro-compatibilità, che rimane un obiettivo dogmatico nonostante il succedersi di numerose nuove versioni, talune anche molto innovative

Quanto è Popolare Java?



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

Da Java 1 a Java 17 in ~25 Anni

- Java 1.0: 1995, la prima versione di Java
 - Java 1.1: 1997, Inner Classes e Reflection
 - Java 1.2: 1998, Just In Time Compiler introdotto nella JVM di Sun,
Java Collections Framework
 - Java 1.3: 2000
 - Java 1.4: 2002
 - Java 1.5 / 5: 2004, Generics, Auto boxing/unboxing, Enum, Varargs, for each loop
 - Java 1.6 / 6: 2006
 - Java 1.7 / 7: 2011
-
- Java 1.8 / 8: 2014, “Deriva funzionale”, lambda expressions
 - Java 9: 2017, “Moduli”
 - Java 10: 2018
 - Java 11: 2019
 - ... (Cambio politica dei rilasci, ora accelerati)
 - Java 17!: 2021
 - ... e presto seguiranno altre: rilasci semestrali

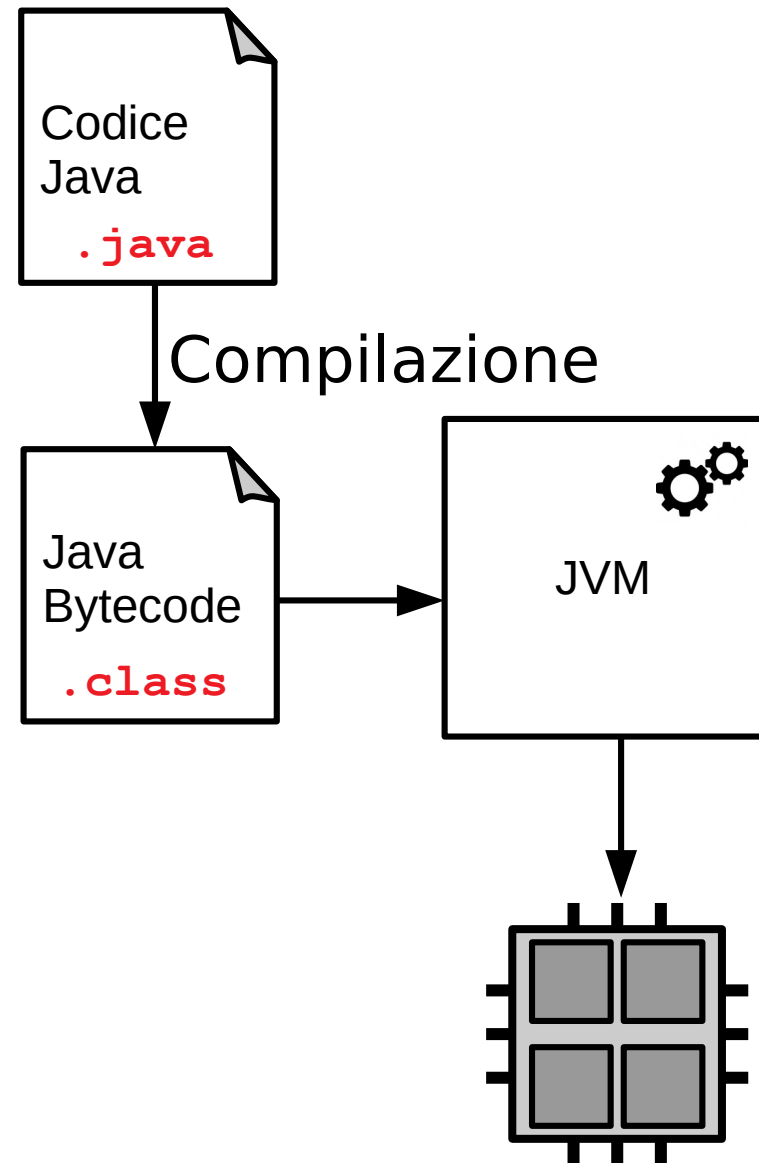
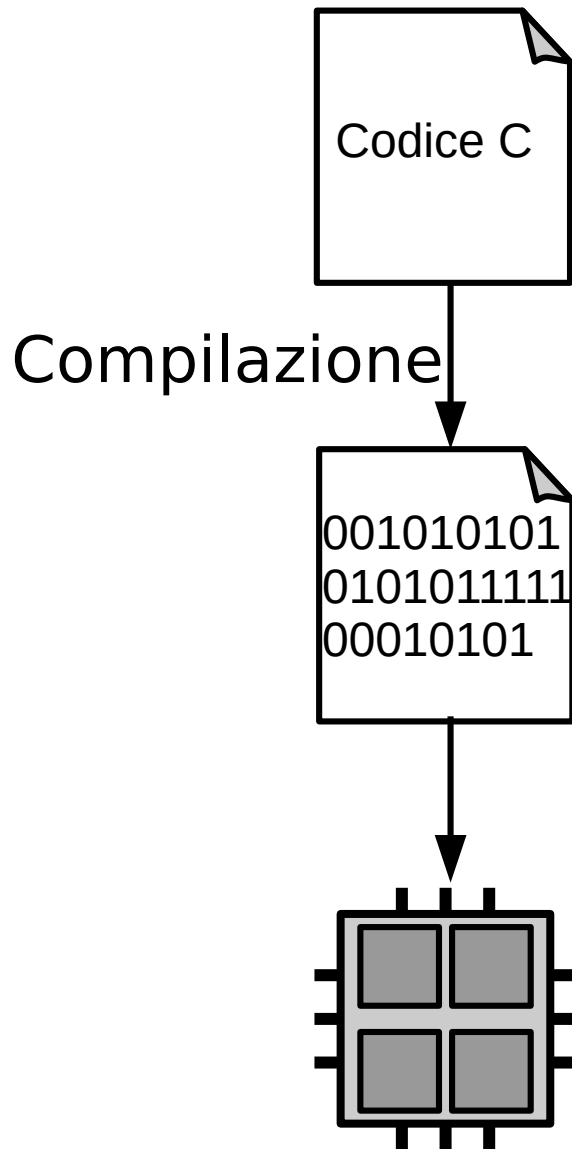
La Portabilità di Java

- Java ha rivoluzionato il concetto di portabilità
- Uno dei principi fondamentali di Java
 - *Write Once Run Anywhere*
 - Un programma Java può essere eseguito su tutte le piattaforme che supportano Java *senza dover ricompilare il codice* per quella specifica piattaforma
- Java è infatti nato come un linguaggio interpretato
 - Il risultato della compilazione è il cosiddetto *Bytecode*: codice oggetto che può essere eseguito da un processore virtuale noto come *Java Virtual Machine (JVM)*

JVM vs Codice Nativo

- Una JVM è, banalizzando, un processore “software”
- Programma in grado di interpretare ed eseguire codice oggetto Java (bytecode) letto da file di estensione **.class**
 - I dispositivi che supportano Java forniscono una implementazione della JVM per il sottostante hardware
- Il Bytecode può essere eseguito su qualunque computer che disponga di una implementazione della JVM
 - Al contrario, i compilatori C generano codice *nativo* eseguibile *solo* su un particolare hardware
 - Un programma C deve essere *ricompilato* per ogni piattaforma su cui si desidera eseguirlo

Compilazione Nativa vs JVM



Java e Portabilità

- Java è supportato praticamente da tutti i sistemi operativi più diffusi (Mac OS, Windows, Linux)
- E da altrettante piattaforme hardware
 - Lo stesso programma Java può essere eseguito su tutte le piattaforme che dispongono di una JVM
 - Senza bisogno di ricompilarlo!
 - Basta spostare i file .class
- Java permette di scrivere programmi altamente portabili
 - Senza preoccuparsi di quali operazioni siano supportate da quali piattaforme
 - N.B. Tuttavia le piattaforme sottostanti sono diverse, e talvolta rimane possibile osservarne le differenze anche programmando in Java
 - Ad es. nomi dei file (>>)

Java e Portabilità: Dov'Eravamo Rimasti? Il Linguaggio C...

- Ad esempio, il seguente codice in C ha un comportamento indefinito

```
int i = 127;
```

```
i = i++;
```

- In C la rappresentazione dei tipi predefiniti non possiede una dimensione fissata da uno standard
 - ✓ Un `int` è grande quanto la parola del microprocessore usato. Rappresentazione degli interi a 8, 16 o 32 bit? Dipende dalla piattaforma: possibile overflow!
 - Se `++` viene eseguito prima dell'assegnazione allora il valore in `i` sarà 1 (incrementa `i` a 2 ma il risultato, assegnato a `i`, è 1)
 - Se `++` è eseguito dopo l'assegnazione allora il valore in `i` sarà 2.
- Il comportamento dipende *anche* dalla precedenza data dal compilatore alle diverse operazioni

Portabilità del Codice Java

- In Java la sintassi e la semantica del linguaggio è ben definita e coerente su tutte le implementazioni della JVM
- Sono descritte in un documento noto come *Java Language Specification*
 - Eventuali ambiguità residuali sono progressivamente identificate, quindi rimosse e/o accettate
- Le istruzioni non dipendono da una particolare piattaforma
 - rimangono le stesse per ogni piattaforma hardware

Librerie di Java

- Altro principale motivo del successo di Java: distribuzione congiunta linguaggio e librerie
Piattaforma Java = Linguaggio + Compilatore + Librerie +...
- In C la libreria standard permette di gestire
 - la memoria (**malloc**, **free**, ...)
 - l'I/O (**FILE**, **printf**, **scanf**, ...)
 - le stringhe (**strcpy**, **strcat**, ...)
 - ecc. ecc....
- Ma non esiste il supporto per liste, insiemi, array associativi ecc. ecc.
- Tutto si può aggiungere ma...

Librerie Standard di Java

- In Java quasi tutto quello di cui si ha bisogno è già presente nella libreria del linguaggio distribuita assieme al compilatore ed all'ambiente di esecuzione
 - Le collezioni sono presenti in Java da anni
 - Librerie spesso sviluppate da esperti del settore
 - Usate e testate da migliaia di utenti
- Rarissimo che si trovino ancora bug di significativo impatto pratico
- Queste librerie sono *>sempre<*, *invariabilmente*, meglio delle nostre implementazioni "artigianali"

Filosofia Java vs C (1)

- Al giorno d'oggi alcune differenze tra il linguaggio C ed il linguaggio Java (che pure deriva dal primo), si possono interpretare correttamente solo considerando le differenze nella filosofia di fondo da cui scaturirono i due linguaggi
- A sua volta queste derivano dalle differenze negli obiettivi
- A sua volta queste differenze negli obiettivi nascono dalle differenze nel contesto in cui i due linguaggi sono nati
 - C: nato in un contesto in cui le risorse di calcolo erano scarse
 - Java: era già chiaro che le risorse di calcolo sarebbero state sempre meno un problema per gli applicativi ordinari
 - Il vero problema (per l'industria) erano i programmatori!
- C: linguaggio di *medio* livello di astrazione
 - Ideato per scrivere parti dei S.O.
- Java: linguaggio di *alto* livello di astrazione

Filosofia Java vs C (2)

- Java nasce cercando di correggere in tal senso i più evidenti difetti dei linguaggi C/C++ rispetto agli obiettivi all'epoca sentiti come più nuovi ed urgenti
- C: prestazioni e controllo anzitutto
 - meglio veloce/compatto che semplice e portabile
 - non fare nulla se non è richiesto esplicitamente dal programmatore a cui si lascia il controllo diretto e quasi totale dell'esecuzione
- Java: semplice e portabile anzitutto
 - evitiamo gli errori più comuni anche se farlo costa, in generale, e non serve sempre: semplicità piuttosto che prestazioni e portabilità piuttosto che controllo totale
 - togliamo al programmatore alcune responsabilità, soprattutto quelle più faticose da gestire correttamente (es. gestione memoria)

Java vs C++

- Alcune volte Java sembra veramente limitarsi a *rimuovere* alcune possibilità aggiuntive offerte dal linguaggio di programmazione C++
 - Il vero competitor iniziale di Java, più del linguaggio C...
 - Perché nasce per supportare lo stesso paradigma (>>)
 - Java rimuove (e non rimpiazza) i costrutti del linguaggio C++ più discutibili rispetto gli obiettivi della piattaforma Java stessa
 - I costrutti rimossi non sono strettamente indispensabili e rendono il linguaggio più utile solo per pochissimi esperti
 - Contemporaneamente, però, finiscono per renderlo più difficile da utilizzare correttamente da parte di tutti gli altri
- Ad es. in Java non esiste
 - Ereditarietà multipla delle implementazioni (sino Java 7)
 - Operator overloading
 - Costruttore di copia
 - Passaggio di parametri per variabile/riferimento
 - Puntatori/Aritmetica dei puntatori...

Un Programma Java

- La sintassi è simile al C perché ideata per esserlo
 - con lo scopo di attrarre il parco programmatori del linguaggio Java, che non furono “sorpresi” dalla nuova sintassi
 - Java è il più famoso linguaggio *C-like* ma ne esistono altri (C++)
- Ogni programma inizia con l'esecuzione del cosiddetto *metodo* statico `main()` esattamente come in C l'esecuzione comincia dalla *funzione* statica `main()`
- Curiosamente i nuovi linguaggi (tra tutti Scala) che cercano di “scalzare” Java, usano tattiche simili, se non addirittura più efficaci
 - Compatibilità con la JVM

Programma in C vs Java (1)

- Un programma che stampa tutti gli argomenti passati da linea di comando
- In C:

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    int i = 0;
    for (i=0; i<argc; i++)
        printf("%s\n", argv[i]);

    return 0;
}
```

Programma in C vs Java (2)

- In Java (scriviamo un file `JavaApp.java`):

```
public class JavaApp {  
    public static void main(String[] args) {  
        for (int i=0; i<args.length; i++)  
            System.out.println(args[i]);  
    }  
}
```

Per eseguirlo (da linea di comando):

```
$ javac JavaApp.java
```

```
$ java JavaApp ciao mondo
```

```
$ ciao mondo
```

Compilazione
in ByteCode

Esecuzione
tramite la JVM

Metodo `main()`

- Ogni programma Java inizia con l'esecuzione del `main()`
 - La dichiarazione del metodo *deve* essere preceduta dalla parola chiave `static`
 - N.B. usiamola per questo scopo ma dimentichiamoci della sua esistenza per le lezioni a venire (>>)
- Il `main()`
 - Non restituisce alcun valore
 - Il suo tipo di ritorno è `void`
 - Prende in ingresso un array di stringhe
 - `String[] args`
 - argomento passato da linea di comando a sostituire la coppia di parametri in C
 - `char *argv[]`
 - `int argc`

Java: Produrre Stampe

- Per stampare sullo schermo è possibile usare `System.out.println(<valore>)`
- `println` stampa `<valore>` e si posiziona su una nuova riga
- Il tipo del parametro attuale di `println()` può essere qualsiasi:
 - `int, boolean, float, double, String, ...`

Tipi di Dato Primitivi Java

- N.B. a differenza che nel linguaggio C, la dimensione della rappresentazione di questi tipi di dato primitivi non dipendono dalla piattaforma, ma sono fissati (per sempre) nella JLS
- Java supporta i seguenti tipi primitivi
 - **int**
interi a 32 bit (da -2^{31} a $2^{31} - 1$)
 - **long**
interi a 64 bit (da -2^{63} a $2^{63} - 1$)
 - **char**
caratteri in codifica UTF-16 (alfabeto latino, arabo, greco, cinese, giapponese, ecc....)
 - **float**
numeri in virgola mobile a 32 bit
 - **double**
numeri in virgola mobile a 64 bit
 - Altri tipi di dato primitivi meno usati ...

Dichiarazione di Variabili

- Per dichiarare una variabile si usa la seguente sintassi: `<tipo> <nome>;`

```
int eta = 18;
```

- Java è case sensitive quindi `eta != Eta != eTa != ETA != ...`
- Per evitare ambiguità (differentemente dal linguaggio C), ad *ogni* variabile deve essere assegnato un valore prima del suo utilizzo
- ✓ In caso contrario la compilazione non andrà a buon fine

Tipo Booleano in Java

- A differenza di quanto accade nel linguaggio C, Java ha un tipo dedicato per le variabili booleane:
 - **boolean**
variabili di tale tipo possono avere solo due valori: **true** e **false**
- Java offre i gli operatori logici *and* e *or* con la seguente sintassi:
 - And: **&&**
 - Or: **||**

Sistema dei Tipi Java

- Ogni variabile ha un certo tipo e può assumere solo valori appartenenti al *dominio* di quel tipo
 - Il tipo di una variabile deve essere specificato al momento della sua dichiarazione

```
int eta = 18;
```
 - Se ad una variabile si assegna un valore non appartenente al dominio del suo tipo si genera un errore durante la compilazione

```
eta = false;
```
- Il comportamento di Java e C sono simili
- Sono entrambi linguaggi *staticamente* tipati
 - ✓ già durante la compilazione deve essere noto il tipo di ogni dato utilizzato

Il Tipo di Dato `String`

- Java introduce il tipo di dato `String` per rappresentare stringhe
 - `String` non è un tipo di dato primitivo (come ad es. `int`, `double`, ...)
 - Le stringhe, per diffusione ed importanza, hanno da subito meritato un supporto molto particolare dal linguaggio (e dal compilatore)
 - si volevano attirare programmatori non esperti facilitando l'utilizzo delle stringhe
 - È possibile dichiarare una variabile di tipo `String` ed assegnargli un valore come segue:

```
String nome = "Bob";
```

```
System.out.println(nome); // stampa Bob
```

String

- I letterali di tipo stringa sono compresi tra doppi apici
 - `"una stringa"`
- A differenza di C, è possibile assegnare più volte ad una variabile di tipo `String` un letterale di stringa

```
String nome = "Bob";
```

```
nome = "Alice"; // ok
```

```
System.out.println(nome); // Stampa Alice
```

- Molte operazioni sulle stringhe offerte direttamente dalle librerie Java distribuite con la piattaforma (>>)

Controllo del Flusso

- Le istruzioni per il controllo del flusso sono le stesse del C
 - La sintassi è esattamente la stessa
- Sono quindi presenti
 - `if e else`
 - `switch, break`
 - `for, while, do while`
 - `break e continue`

Controllo del Flusso

- Essendo presente il tipo `boolean`, le condizioni sono sempre espresse su espressioni di tipo `boolean`

```
int i = 0;
```

```
if (i) { ... }
```

- In Java NON compila
- In C è prassi comune eseguire controlli direttamente su espressioni di tipo `int`

```
char stringa[] = "null terminated";
```

```
int c = 0;
```

```
while (stringa[c]) {
```

```
    printf("%c\n", stringa[c]);
```

```
    c++;
```

```
}
```


Errori: Tempo di Compilazione vs Tempo di Esecuzione

- Due momenti ben distinti in cui si può riscontrare un errore in un programma
 - *A tempo di compilazione*
 - *A tempo di esecuzione*
- La rapida risoluzione degli errori in tempi brevi, è importante per la produttività
- La diagnostica dovrebbe sempre riportare il motivo dell'errore e la posizione esatta in cui si è verificato
- La diagnostica del C talvolta risulta “vaga”
 - Soprattutto per gli errori a tempo di esecuzione

Segmentation fault

Diagnostica

- La diagnostica è decisamente uno dei principali punti forti della piattaforma Java
- Permette di trovare gli errori di entrambi i tipi per risolverli tempestivamente, così riducendo:
 - i tempi di ricerca degli errori (debugging)
 - i costi
- Buona parte di questi vantaggi (soprattutto a tempo dinamico) derivano dall'adozione di un processore virtuale (JVM)
- ✓ Ottenere le stesse funzionalità con un processore fisico è decisamente più ostico
- Consiglio:

Abituarsi fin da subito ad un uso ossessivo della diagnostica

Errori a Tempo di Compilazione

```
public class Prova {  
    public static void main(String[] args) {  
        System.out.println("hi")  
    }  
}
```

Exception in thread "main" java.lang.Error:
Unresolved compilation problem:

Syntax error, insert ";" to complete
BlockStatements

Dove si è
verificato

Il problema

at Prova.main(Prova.java:3)

Errori a Tempo di Esecuzione (1)

- Gli errori a tempo di esecuzione sono decisamente più temibili
 - Si verificano durante l'esecuzione del codice, non sono né riconoscibili né prevedibili dal compilatore
 - In Java, gli errori a tempo di esecuzione sono spesso anche detti *runtime-exception*
 - prendendo in prestito il nome del costrutto del linguaggio pensato per gestirli (>>)
- Anche in questo caso la diagnostica è molto precisa

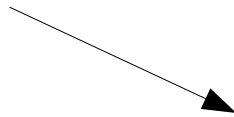
Errori a Tempo di Esecuzione (2)

```
public class Prova {  
    public static void main(String[] args) {  
        int infinito = 1/0;  
        System.out.println(infinito);  
    }  
}
```

La compilazione va a buon fine ma durante l'esecuzione si ha

Exception in thread "main"
java.lang.ArithmeticException: / by zero

Dove si è
verificato



Il problema

at Prova.main(Prova.java:3)

Paradigma di Programmazione

- Il *paradigma di programmazione* influenza decisamente il modo di programmare
- Non esiste una definizione comunemente accettata di *paradigma di programmazione* (e forse neanche serve...)
- Con un pò di esperienza inter-paradigma risulta facile apprezzare come un linguaggio di programmazione rende un paradigma più semplice da esprimere rispetto ad altri linguaggi
- Il vostro corso di studi ne tiene conto!
 - Procedurale: FdI - I anno (Linguaggio C)
 - Oggetti: POO - II anno (Linguaggio Java)
 - Funzionale: PF - III anno (Linguaggio OCaml)
 - Paradigma Ibrido Oggetti/Funzionale (Scala)

Il Paradigma Procedurale

- Il paradigma procedurale è stato il primo a diffondersi
- La divisione tra dati ed operazioni è tipica dell'hardware moderno (memoria/CPU) e dell'architettura di Von Neumann
 - Tale divisione impone che i dati siano sempre separati dalle operazioni
 - La correlazione tra dati ed il codice che li lavora non è *esplicita*
- Nel paradigma procedurale un programma è specificato come una sequenza di comandi che cambiano lo stato dell'esecuzione, passo dopo passo
- Il C è un linguaggio di programmazione che rende naturale esprimere il paradigma procedurale
- In effetti è un linguaggio “vicino alla macchina”

Limiti del Paradigma Procedurale (1)

- Uno dei maggiori limiti del paradigma procedurale è proprio la *netta* separazione tra dati ed operazioni sugli stessi
 - lo stato è ben distinto e separato dalle operazioni
- Dati e funzioni che li lavorano, sono comunque separati nel codice

```
struct Persona {  
    const char* nome;  
    int eta;  
};
```

```
int isMaggiorenne(struct Persona persona) {  
    return ( persona.eta >= 18 );  
}
```


Limiti del Paradigma Procedurale (2)

- È del tutto legittimo avere operazioni sugli stessi dati in luoghi anche *molto distanti* del codice
- Addirittura incentivata(!) la ripartizione dei due aspetti in file distinti
- La definizione di un tipo di dato in C specifica solo i dati componenti ma non le operazioni che vi si possono applicare
 - queste sono definite altrove e non fanno parte della definizione del tipo di dato

Limiti del Paradigma Procedurale (3)

File: Persona.h

```
.....  
  
struct Persona {  
    const char* nome;  
    int eta;  
};
```

Dati

File: Persona.c

```
.....  
  
void invecchia(struct Persona *p) {  
    p->eta = p->eta + 1;  
}  
  
int isMaggiorenne(struct Persona p)  
{  
    return ( p.eta >= 18 );  
}
```

Operazioni

- Il tipo di dato **Persona** non definisce le operazioni che possono essere eseguite sul tipo stesso