Classi

- Le istruzioni sono raggruppate per ottenere una medesima funzionalità (metodo). Le diverse funzionalità (metodi) relative a un oggetto sono raggruppate nella classe.
- La classe assegna un nome comune a dati e metodi che essa riunisce. Tali dati e metodi sono detti **membri** della classe.
- Ogni cosa scritta in Java dev'essere membro di qualche classe: non è possibile dichiarare "variabili globali", "funzioni", "procedure".

Definizione di una classe

```
class MyClass
{
    //dichiarazioni dei campi
    //dichiarazioni dei costruttori
    //dichiarazioni dei metodi
}
```

parola riservata

- campi: descrivono i dati disponibili
- costruttori: permettono che ogni oggetto sia correttamente inizializzato quando è creato
- metodi: realizzano il comportamento

Campi e metodi

- Tutti gli oggetti della stessa classe hanno gli stessi campi e gli stessi metodi.
- Questo perché campi e metodi sono definiti a livello di classe, non di oggetto.
- Per i campi questo non è un problema: in effetti questa omogeneità è desiderabile.
- Comunque, anche se i campi sono gli stessi, i valori contenuti sono in generale diversi.

Campi e metodi

- Invece i metodi, che sono gli stessi per tutti gli oggetti della stessa classe, sono però invocati su oggetti specifici (eccetto, come vedremo, i metodi statici).
  - Utile per la portabilità, ma talvolta sconveniente.
- Tuttavia, gli oggetti possono anche **creare** altri oggetti e **sfruttarne** i metodi.
- Questo è proprio quanto viene fatto quando un programma viene eseguito. Si parte da un unico oggetto, il quale crea a seguire gli altri.

Il metodo `main`

- In Java non è possibile scrivere codice al di fuori di una classe.
- Quando scriveremo il codice di un programma scriveremo classi.
- La soluzione di un problema si ottiene facendo interagire oggetti costruiti come istanze di classi.
- Serve però un**punto di partenza**. Da dove cominciano le interazioni?

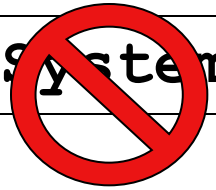
Il metodo `main`

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

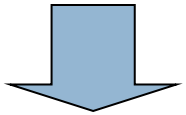
oggetto *metodo*

- È un'istruzione, non un'applicazione eseguibile.
- È necessario scrivere una classe che contenga un metodo che contenga l'istruzione.
- In particolare, ogni programma deve avere almeno una classe che contiene un metodo di nome main, usato dalla JVM come punto di partenza dell'esecuzione.

Il metodo `main`



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



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

- Questo testo va salvato in una classe chiamata `PrimoProgrammino.java`

Descrizione e commenti

- È bene scrivere commenti, ad esempio per spiegare cosa fa la classe o strutturare il codice

```
class PrimoProgrammino
{ /* una classe di esempio banale
     per illustrare l'uso del main */
  // campi della classe

  // costruttori della classe

  // metodi della classe
  public static void main(String[] args)
  { System.out.println("Hello");
  }
}
```

Il metodo `main`

- Il file `PrimoProgrammino.java` può essere compilato e, dato che contiene il `main`, eseguito
- Il nome del sorgente **deve** avere l'estensione **`.java`**, altrimenti il compilatore segnalerà che gli è impossibile trovare il file.
- Per compilare si può scrivere a riga di comando

  `javac PrimoProgrammino.java`

  (scrivendo esplicitamente `.java`)

Il metodo `main`

- La compilazione genererà nella medesima directory il file `PrimoProgrammino.class` che contiene il bytecode.
- Questo può essere eseguito dalla JVM

  `java PrimoProgrammino`

  (stavolta si omette `.class`)
- Riassumendo: (o si fa insieme con un IDE)
  - editing di **`PrimoProgrammino.java`**
  - **`javac PrimoProgrammino.java`**
  - **`java PrimoProgrammino`**

Il metodo `main`

- Il metodo **`main`** deve essere **`public`** e **`static`** e avere come tipo di ritorno **`void`**.
- Significato più generale che per il solo **`main`**:
- **`public`** significa che è accedibile da tutti. È l'opposto di **`private`**, che significa che vi può accedere solo quella classe.
- **`static`** significa che è un metodo che non viene usato per un oggetto, ma per una classe (quindi è più generale)

Metodi statici

- Può essere necessario dichiarare un metodo come static se questo metodo è pensato per essere invocato non in relazione a uno specifico oggetto di una classe.
- Abbiamo già visto un caso: la classe `Math`.
- Mentre `System.out` è un oggetto, `Math` è una classe. I metodi della classe `Math` (ad esempio `Math.sqrt`) sono quindi statici e invocano `Math`, non uno specifico oggetto.

Tipi di dato

- È necessario che i metodi della classe `Math` siano statici perchè devono poter operare su numeri, ad es.: `radiceDue = Math.sqrt(2);`
- In Java, **i numeri non sono oggetti.**
- Tipi di dato numerico:
  - `int` (intero)
  - `double` (virgola mobile doppia precisione)
  - poi `float`, `long`, `short`, `byte`, `char`, `boolean`
- Sono tutti (e soli) tipi di dato **primitivo**

Tipi di dato

- Java prevede due casi di tipi, nettamente distinti tra loro:
  - Tipi primitivi (predefiniti dal linguaggio)
  - Tipi riferimento
    - sono: **classi**, interfacce, array
    - sono virtualmente infiniti, definiti dal programmatore
- L'esistenza del tipo riferimento/classe rende possibile ad esempio avere un valore di ritorno di un metodo che è un oggetto.

Espressioni vs. Istruzioni

- Una porzione di codice Java a cui sono associati un **tipo** e un **valore** è un'**espressione**.
- Ad esempio sono espressioni:
  - il **nome** di una variabile (tipo e valore sono quelli della variabile)
  - l'invocazione di un **metodo** (il tipo è il tipo di ritorno, il valore è il risultato del metodo)
  - un'espressione che risulta da più espressioni unite da operatori. Intervengono regole per deciderne tipo e valore.

Espressioni vs. Istruzioni

- Un’espressione non è un’unità di calcolo completa, in quanto non ha effetto (non produce cambiamenti di valori in variabili).
- Un’istruzione è invece un’unità eseguibile completa, terminata dal carattere “;”
  - può essere un assegnamento, una valutazione, una chiamata ad oggetto, o combinazioni di essi

Tipi di dato

- Il concetto di variabile è un'astrazione del concetto di locazione di memoria.
- In realtà tutte le variabili sono sequenze di bit della memoria. Tuttavia, in base ai tipi, possono essere interpretate in modo diverso.
- La nozione di tipo consente di astrarre rispetto all'effettiva memorizzazione dei dati.
  - Ovvero: in realtà ci sono solo i dati, ma l'interfaccia col programma avviene con i tipi.

Tipi di dato

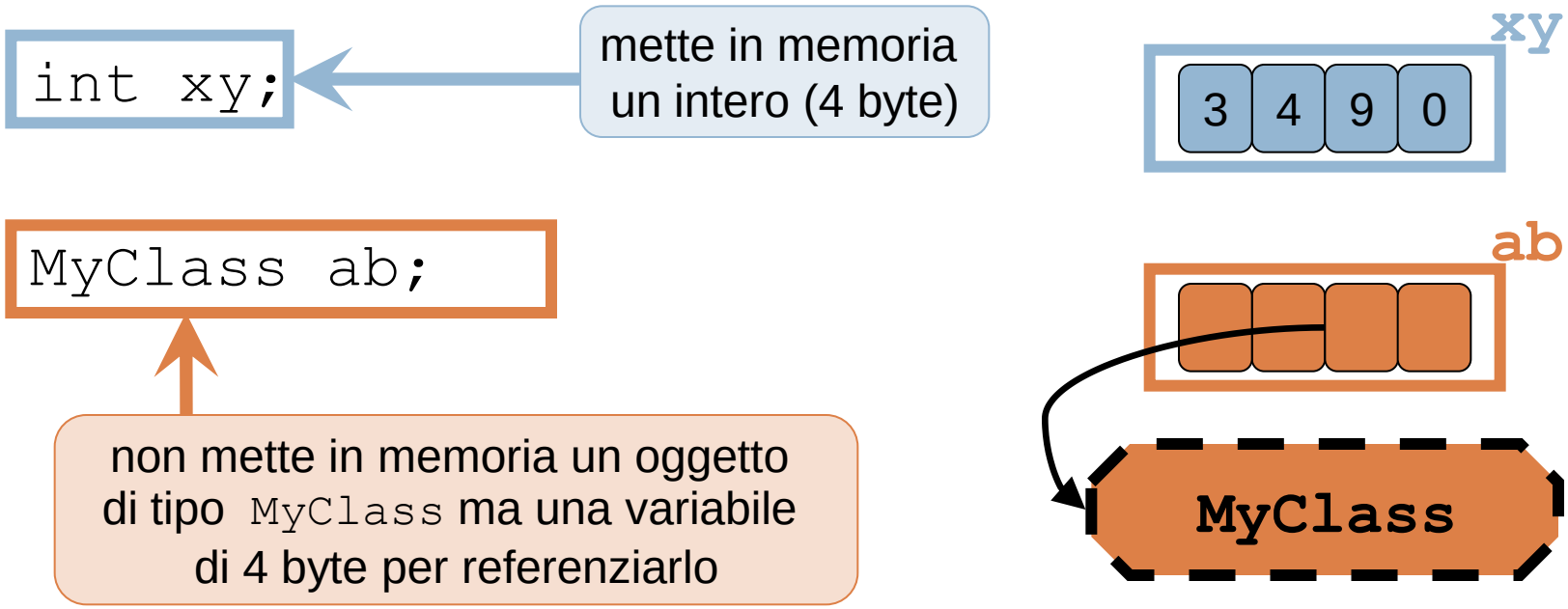
- Java è un linguaggio fortemente tipizzato.
- Ciò significa che è possibile determinare il tipo di ogni espressione che appare in un programma semplicemente leggendone il testo (quindi ciò è noto al compilatore)
- Quindi, durante la compilazione è possibile effettuare tutti i controlli di compatibilità dei tipi (e stabilire se è necessario effettuare cast, ovvero conversioni, implicite)

Tipi di dato

- Nota: mentre il tipo di un'espressione è determinabile dal compilatore, il suo valore viene calcolato solo durante l'esecuzione del programma.
- Semantica statica (i tipi) vs. dinamica (i valori, perché sono noti solo durante l'esecuzione).
- Il controllo di correttezza semantica dei tipi è importante, ma non esaustivo.

Tipi primitivi vs. riferimento

- Differenza principale:
  - Una variabile di tipo primitivo è **già pronta** a contenere un valore. Una di tipo riferimento contiene invece solo il riferimento all'oggetto.

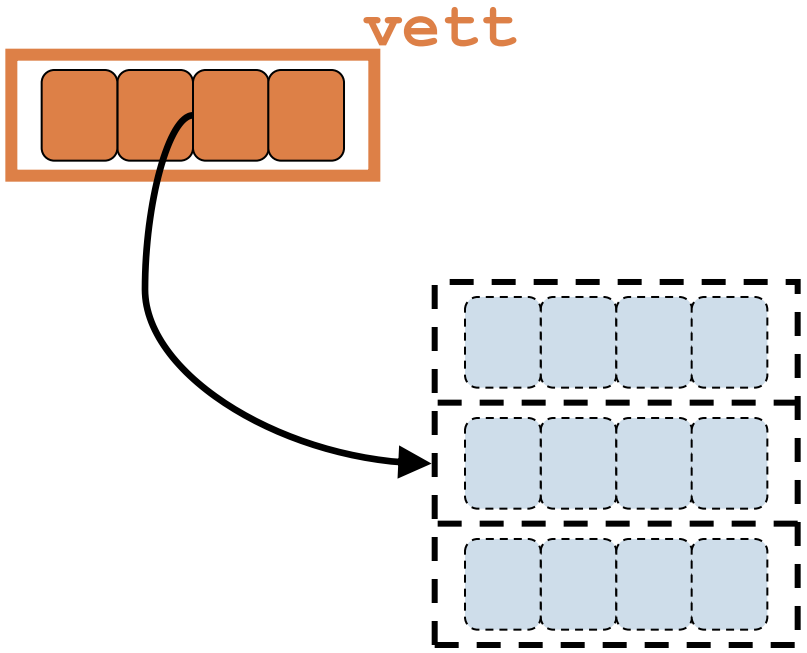


Tipi primitivi vs. riferimento

- Anche per gli array

```
int vett[];
```

alloca 4 byte per riferire un (futuro) array di interi



Tipi primitivi vs. riferimento

- Nonostante la somiglianza tra le variabili riferimento e i puntatori del C, Java non consente la conversione tra numeri interi e indirizzi o le operazioni aritmetiche (ambiente maggiormente protetto).

Tipi primitivi vs. riferimento

- In realtà durante l'esecuzione la JVM alloca per ogni dato primitivo il massimo disponibile in fatto di rappresentazione dei dati
  - Se ho una macchina a 32 bit, la JVM riserverà 32 bit sia per gli interi che per i byte
- Contro: maggior consumo (spreco) di risorse
- Pro: maggiore portabilità del bytecode, maggiore sfruttamento delle capacità di calcolo della piattaforma

Dichiarazione e definizione

- In Java le variabili devono essere dichiarate prima del loro utilizzo, per specificarne il tipo.
- Si possono dichiarare in una classe (variabili d'istanza) o in un metodo (variabili locali), tipicamente all'inizio. Il loro scope è ivi limitato.

definizione = dichiarazione + assegnamento

Assegnamenti

- Occorre quindi partire dagli assegnamenti.
- Questi identica al C, tramite l'operatore =

```
int valore;
valore = 5;
```

dichiarazione + assegnamento

```
int valore = 5;
```

definizione

Inizializzazione

- Anche se è possibile definire una variabile senza inizializzarla, il compilatore Java si rifiuta di compilare il codice se viene acceduta una variabile non inizializzata.
- Pertanto è bene controllare che tutte le variabili siano assegnate.
  - Per i tipi primitivi: basta dare un valore di default
  - Per i tipi riferimento: bisogna scrivere un costruttore

Operazioni dei tipi primitivi

- Analogo al C: operazioni + - * /
- Attenzione alla divisione tra `int`: corrisponde alla divisione intera (c'è anche % per il resto)
- Le assegnazioni possono essere combinate con le operazioni: += -= *= e anche ++ -- etc
- Come in C esistono poi gli operatori >, <, >=, ==, !=, etc.

Operazioni dei tipi primitivi

- Quando le operazioni coinvolgono tipi diversi, si effettua un **cast**.
- Operatore di cast: le parentesi tonde `()`
- Tipicamente automatico quando non comporta perdita di informazioni (es.: da `int` a `double`), mentre va fatto esplicitamente quando potenzialmente può far perdere informazioni (da `double` a `int`)

```
int a;
a = (int) 5 / 2.5;
```

Costanti

- La parola chiave per le costanti è `final`.

```
final double CAN_SIZE = 0.33;
int numLattine = 2;
double quantaBirra = numLattine * CAN_SIZE;
```

- Usando costanti non si deve riscrivere tutto (negli USA, basta mettere `CAN_SIZE = 0.355`)
- Tipicamente il nome è tutto maiuscolo.

Costanti

- È lecito scrivere variabili blank final. Per queste si può fare un solo assegnamento.

```
final double CAN_SIZE;
CAN_SIZE = 0.33; //ammesso
int numLattine = 2;
double quantaBirra = numLattine * CAN_SIZE;
CAN_SIZE = 0.355; //errore! era final
```

Costanti

- Esistono costanti già definite nelle librerie, ad esempio `Math.PI` in `java.lang`.
- Quando una costante viene usata molto di frequente, risulta scomodo ripetere ogni volta il nome della classe.
- Java fornisce quindi un meccanismo di importazione statica simile a quello usato per importare le classi.

Costanti

- Per utilizzare la costante PI della classe Math senza dover indicare ogni volta il nome qualificato Math.PI, possiamo scrivere:

```
import static java.lang.Math.PI;
```

  - Così possiamo chiamare Math.PI soltanto “PI”.
- In realtà possiamo importare qualunque membro statico di un’altra classe, compresi campi statici e metodi statici. Bisogna fare attenzione a non compromettere la leggibilità.

Stringhe

- Le stringhe in Java sono un tipo già pronto: ma non sono primitive, sono date da `java.lang`.
- Si scrivono racchiuse tra virgolette (ma le virgolette non sono comprese nella stringa). Si possono usare inoltre tutte le sequenze di escape del C (`\n`, `\t` eccetera).
- La “parola chiave” del tipo stringa è `String`.
- Attenzione, `String` è una classe (diversamente da `int` e `double`).

Stringhe

- Il fatto che `String` sia una classe comporta che se creo una variabile di tipo `String`...

  ```
  String saluto = "Hello!";
  ```

  ...questa è un oggetto!!
- Di conseguenza, con le stringhe si possono chiamare metodi.
- Un metodo utile è length, il cui prototipo è:

  ```
  int String.length()
  ```

  il quale ritorna la lunghezza della stringa -1

Stringhe

- Poiché `length` è un metodo, si può scrivere:

```
String saluto = "Hello!";
int n = saluto.length();
```

...mentre è sbagliato scrivere

```
int n = String.length(saluto); !
```

- La stringa "" ha lunghezza 0.

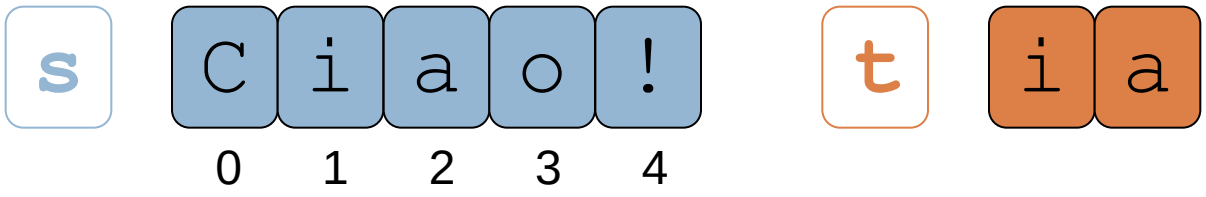
Sottostringhe

- Esiste il metodo `substring`:

```
String String.substring (int start,
  int pastEnd);
```

che riporta la sottostringa tra `start` (incluso) e `pastEnd` (escluso).

```
String s="Ciao!";
String t=s.substring(1,3);
```



- Per motivi storici, il conteggio parte da 0!

Concatenazione

- Le stringhe si possono concatenare con l'operatore +.

```
String nome = "Mario";
String cognome = "Rossi";
String fullname = nome + " " + cognome;
```

- L'operatore + se serve casta tutto a String:

```
int year = 84;
String bandName = "Equipe " + year;
```

- Così per convertire un numero a stringa si può

```
String name = "" + 883;
```

Conversione Stringa→Numero

- Per convertire una stringa a un numero bisognerebbe invocare un metodo (di un oggetto).
- Però i numeri in Java non sono oggetti.
- Soluzione: usare le **classi involucro**.
  - Sono classi associate a ogni tipo di dato primitivo, in particolare ai numeri, che consente di rappresentare quei dati come oggetti
  - Ad esempio `Integer` e `Double`.

Conversione Stringa→Numero

- I tipi di dati primitivi non sono implementati come oggetto per ragioni di efficienza (si possono benissimo creare oggetti `Integer` per rappresentare interi, ma non conviene).
- La classe Integer ha un metodo statico:

  ```
  public static int parseInt(String s)
  ```

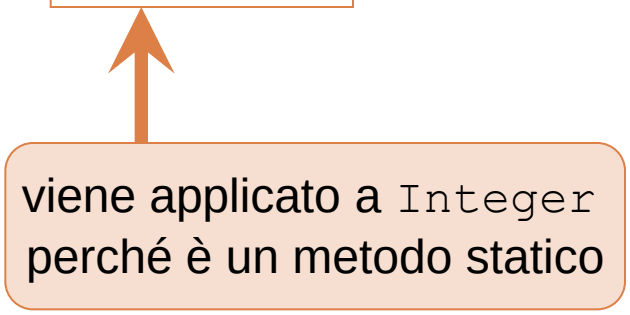
  che prende una stringa e ritorna un intero.
- C'è anche l'opposto (più chiaro di +"")

  ```
  public static String toString(int x)
  ```

Conversione Stringa→Numero

- Abbiamo una variabile stringa `anno`, vogliamo convertirla in una variabile `int` chiamata `y`.
- Dobbiamo applicare il metodo `parseInt`.
- A cosa lo applichiamo? Risposta:

```
int y = Integer.parseInt(anno);
```



Confronto stringhe

- Sebbene esista l'operatore ==, questo non può essere usato per confrontare stringhe.
- Infatti esso non ritorna TRUE se due oggetti coincidono per "valore", ma solo se sono **lo stesso oggetto**. Questo è vero per le stringhe, che sono oggetti, ma anche più in generale per ogni tipo di oggetto Java.

Confronto stringhe

- Ciò significa che

```
String saluto = “Ciao a tutti!”;
String s1 = “Ciao”;
String s2 = s1;
String s3 = saluto.substring(0,4);
boolean x = (s1 == s2); // x vale TRUE
boolean y = (s1 == s3); // y vale FALSE
```

nonostante `s1`, `s2`, `s3` siano tutti `“Ciao”`.

Confronto stringhe

- Esiste quindi un metodo chiamato `equals`, che rivedremo spesso per altre classi.
- Per vedere se la stringa `s1` “uguaglia” `s2` scriveremo quindi

  ```
  s1.equals(s2)
  ```

  che ritorna TRUE nel caso le stringhe siano identiche nel senso generale, FALSE altrimenti.

Confronto stringhe

- Altri metodi per confrontare stringhe sono:
  - `equalsIgnoreCase` che non distingue tra maiuscole e minuscole
  - `compareTo` che riporta una relazione d'ordine lessicografica: `s1.compareTo(s2)` è <0 se `s1` precede `s2` nel vocabolario (inteso come codici ASCII), >0 se segue, ==0 se sono uguali