# Il linguaggio Java

- Nato nel maggio 95 (James Gosling & al.)
- Orientato ad oggetti, basato sulle classi, concorrente
- Fortemente tipato: distinzione chiara tra errori statici ed errori dinamici
- Ad alto livello: garbage collector, assenza di costrutti unsafe (es. array senza controllo sugli indici)
- La compilazione produce un codice binario indipendente dalla macchina (Java Virtual Machine)



# Applicazioni e applets

- Applicazioni = programmi “stand-alone”

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

- Applets = eseguibili da un browser Java-compatibile

```
import java.applet.Applet;
import java.awt.Graphics;

public class Ciao extends Applet {
    public void paint(Graphics g) {
            g.drawString("Ciao", 50, 25);
            }
}

<html>
<applet code=“Ciao.class” width=275 height=200> </applet>
</html>
```

# Analisi dell'applicazione

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

- **`class Ciao`**
  - dichiara il nome della classe. Quando questo codice viene compilato, viene generato un file chiamato `Ciao.class`
- **`public`**
  - il metodo `main` deve essere accessibile dappertutto, incluso l'interprete Java
- **`static`**
  - il metodo `main` è legato alla classe `Ciao`, non ad una sua istanza: questo ne permette l'esecuzione prima che il programma faccia qualsiasi altra cosa

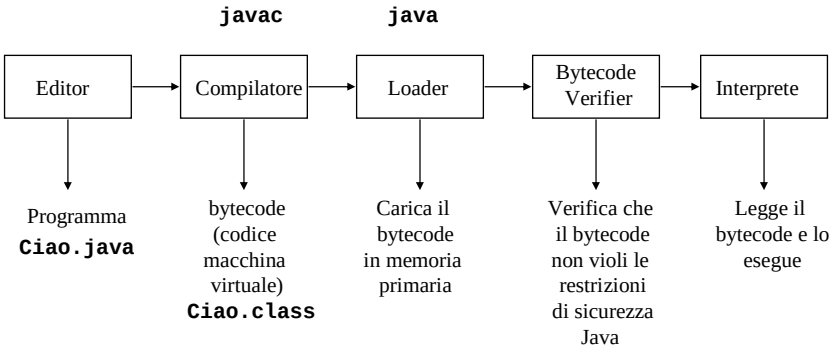


# Analisi dell'applicazione (ctd.)

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

- **`void`**
  - il metoto main non restituisce nessun valore
- **`String[] args`**
  - il metodo `main` ha come parametro un array di stringhe (di dimensione qualsiasi). Questo permette di chiamare il programma con un numero qualsiasi (anche nullo) di parametri.
- **`System.out.println`**
  - chiamata al *metodo* `println` dell'*oggetto* `out` che appartiene alla *classe* `System`.
  - La chiamata a questo metodo produce la stampa sullo standard output (video) della stringa passata come parametro attuale.

# Ambiente Java

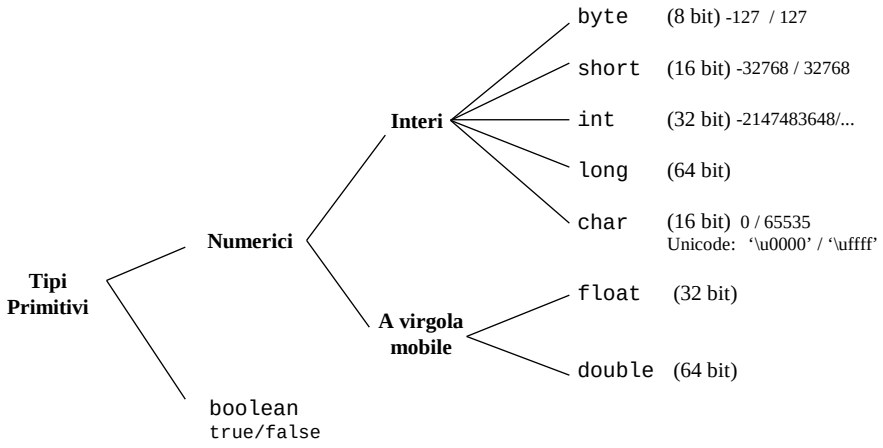





# Tipi e Valori

- Ogni variabile ed espressione ha un tipo conosciuto in compilazione, che ne limita i valori e l'uso
- Ci sono due categorie di tipi, in corrispondenza delle quali ci sono due categorie di valori:
  - tipi primitivi & valori primitivi
  - tipi riferimento & valori riferimento = riferimenti ad oggetti
  - oltre a questi c'e' un tipo speciale, il tipo nullo, che non ha nome ed ha come unico valore l'espressione **`null`**
- Tipi Primitivi:
  - devono essere definiti allo stesso modo in ogni macchina e in ogni implementazione
  - una variabile di tipo primitivo ha sempre un valore primitivo dello stesso tipo.
  - il valore di una variabile di tipo primitivo può essere modificato solo mediante assegnamento a quella variabile

# Tipi Primitivi






# Operatori sui tipi primitivi

- operatori di confronto (valori di tipo boolean)
  - `<, <=, >=, >, ==, !=`
- operatori numerici (valori di tipo int o long)
  - unari: +, -
  - additivi e moltiplicativi: `+, -, *, /, %` (modulo)
  - di incremento e decremento `++, --` (prefissi e suffissi)
  - operatori di shift: `<<, >>, >>>`
  - complemento bit a bit: ~
  - operatori interi bit a bit: & (and), | (or), ^ (or esclusivo)
- operatore condizionale `(b ? e1 : e2)`
- casting

# Operatori

- Operatori di shift
  - `a << b` dà come risultato $a*2^b$
  - `a >> b` dà come risultato $a/2^b$ (preserva il segno)
  - `>>>` "lavora" sulla sequenza di bit, mettendo zeri sui bit più significativi (non preserva il segno)
- Casting
  - Le variabili sono automaticamente "promosse" al tipo di dimensioni maggiori (ad esempio `int` viene promosso automaticamente a `long`)
  - l'assegnamento tra `short` e `char` richiede un casting esplicito

**Esempio**

```
long valoreLungo = 99L;
int ristretto = (int) valoreLungo; // casting esplicito
long valoreGrande=6;               // casting automatico
```



# Underflow e overflow

Attenzione! gli operatori sugli interi non segnalano in nessun modo problemi underflow o di overflow

```
class Test {
        public static void main(String[] args) {
                int i = 1000000;
                System.out.println(i>10 ? ">10" : "<=10");
                System.out.println(i * i);
                long l = i;
                System.out.println(l * l);
        }
}
```

produce il seguente output

```
>10
-727379968
1000000000000
```

# Numeri a virgola mobile

- Lo standard IEEE 754-1985
  - numeri positivi e negativi $s*m*2^e$
  - zero negativo e zero positivo,
  - infinito positivo e negativo, es. `Float.POSITIVE_INFINITY`
  - un valore speciale NaN (not a number), che rappresenta il risultato di certe operazioni come la divisone per 0
    ci sono quindi le costanti `Float.NaN` e `Double.NaN`
- Un overflow, sui numeri a virgola mobile, produce un infinito; un underflow produce `0`.
- Ogni operazione con un valore `NaN` produce `NaN`, e ogni confronto con `NaN` produce `false`



# Esempio

```
class Test {
        public static void main(String[] args) {
                double d = 1e308;
                System.out.println("overflow: " + d*10);

                System.out.println("meno infinito: " + Float.NEGATIVE_INFINITY);

                d = 0.0/0.0;
                System.out.println("0.0/0.0: " + d);

                System.out.print("risultati inesatti con numeri float:");
                for (int i = 0; i < 100; i++) {
                        float z = 1.0f / i;
                        if (z * i != 1.0f)
                                System.out.print(" " + i);
                }

                d = 12345.6;
                System.out.print("\n casting: " + (int)d );
        }
}
```

**produce come risultato:**

```
overflow: INF
 meno infinito: -INF
 0.0/0.0: NAN(000)
 risultati inesatti con numeri float: 0 41 47 55 61 82 83 94 97
 casting: 12345
```

# Esempio

**Esempi o di dichiarazioni, inizializzazioni ed assegnamenti di variabili di tipi primitivi**

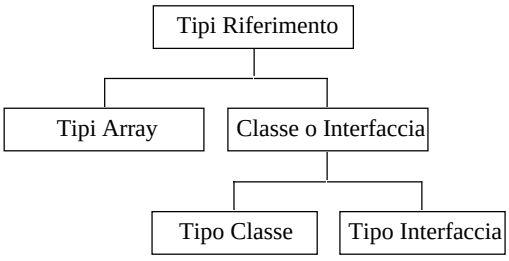
```
public class Assign{
        public static void main(string args[]){
         int x,y;         // dichiara due variabili intere
         float z=3.414f;  // dichiara e inizializza una variabile float
         double w=3.1415; // dichiara e inizializza una variabile double
         boolean vero=false; // dichiara e inizializza una variabile boolean
         char c;          // dichiara una variabile char
         c='A';           // assegna un valore alla variabile di tipo char
         x=6;
         y=1000;          // assegna valori alle variabili di tipo int
           …
         }
}
```

**Esempio di assegnamenti non validi**

```
y=3.14156        // 3.1415926 non è di tipo int (è necessario un casting!)
w=175,000        // la virgola non può apparire!
vero=1           // errore comune a chi è abituato a programmare in C o C++
z=3.14156        // un double non può essere assegnato a un float (è necessario
                 // il casting!)
```



# Tipi Riferimento




```
class Punto { int[] coordinate; }
interface Traslazione { void sposta(int deltax, int deltay);}
```

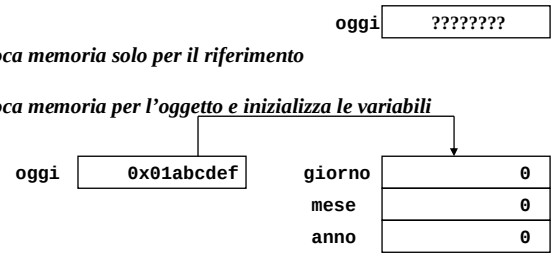
# Creazione di Riferimenti

- Quando viene dichiarata una variabile di tipo primitivo, l'allocazione di memoria fa parte della stessa operazione. Quando viene dichiarata una variabile di tipo riferimento (`Array`, `classi`, `interfacce`) *non* viene allocata memoria per l'oggetto.

Esempio

```
class Data{
    int giorno;
    int mese;
    int anno;
}
Data oggi;            // alloca memoria solo per il riferimento

oggi = new Data();    // alloca memoria per l'oggetto e inizializza le variabili
```






# Classi, Oggetti, Array

- **Classe** è il modo in Java per creare nuovi tipi
- **Oggetto** è o un'istanza di una classe o un array.
- Un'**istanza** di classe (oggetto) è creata esplicitamente con `new + creatore` oppure invocando il metodo `newinstance` della classe `Class`
- Un **array** è creato esplicitamente con un'espressione di creazione di array. L'indice di un array inizia sempre da `0`.
- Anche le **stringhe** sono oggetti. Quando si usa l'operatore + di concatenazione di stringhe, viene creato automaticamente un nuovo oggetto della classe `String`

# Classe = variabili + metodi

```
Class Impiegato {
    String nome;         // variabili o campi della classe
    String cognome;
    String reparto;

    void print(){        // metodo della classe Impiegato
        System.out.println(nome + cognome + " del reparto " + reparto);
    }
}
```

Per creare un'istanza di questa classe, e inizializzare l'oggetto:

```
impiegato = new Impiegato();
impiegato.nome = "Giovanni";
impiegato.cognome = "Bianchi";
impiegato.reparto = "officina";
```

La chiamata al metodo `impiegato.print()` stamperà la stringa

```
Giovanni Bianchi del reparto officina
```



# Overloading dei nomi dei metodi

- Nella stessa classe è possibile definire diversi metodi con lo stesso nome, purché differiscano nella lista degli argomenti
- Ad esempio,

```
public void println(int i);
public void println(float f);
public void println(String s);
```

- Il tipo restituito può essere diverso, ma in ogni caso la lista degli argomenti deve essere diversa in modo tale da permettere una determinazione non ambigua del metodo che effettivamente viene chiamato.

# Costruttori

- Il nome del metodo deve coincidere esattamente con il nome della classe, e non deve restituire nessun valore
- Possono esserci diversi costruttori (che differiscono per la lista di parametri), che possono invocarsi a vicenda

```
public class Impiegato{
    private String nome;
    private int salario;

    public Impiegato(String n, int s){
            nome = n; salary = s
    }
    public Impiegato(String n){
            this(n,0)
    }
    public Impiegato(){
            this("Sconosciuto")
    }
}
```



# Assegnamento su Riferimenti

```
class Value { int val; }

class Test {
        public static void main(String[] args) {
                int i1 = 3;
                int i2 = i1;
                i2 = 4;
                System.out.print("i1=" + i1);
                System.out.println(" mentre i2=" + i2)

                Value v1 = new Value();
                v1.val = 5;
                Value v2 = v1;
                v2.val = 6;
                System.out.print("v1.val=" + v1.val);
                System.out.println(" e v2.val=" + v2.val);
        }
}
```


**due variabili distinte**


**stessa istanza**

**produce come risultato:**

```
i1=3 mentre i2=4
v1.val=6 e v2.val=6
```

# Passaggio dei parametri

- In un metodo o costruttore il passaggio dei parametri è sempre per *valore*.
- Se si passa come parametro un oggetto, questo può essere modificato accedendo ai suoi campi e metodi: il contenuto dell'oggetto (i valori delle sue variabili) potrà essere cambiato ma il riferimento all'oggetto stesso no.



# Array

- Oggetti che raggruppano dati dello stesso tipo
- Si possono dichiarare array i cui elementi sono di tipo qualsiasi (primitivo o riferimento)

```
char s[];
char [] s;        //forma alternativa

Data giorni[];
Data [] giorni;   //forma alternativa
```

- La dichiarazione alloca memoria solo per il riferimento
- Per creare un array bisogna usare l'operatore `new`

```
s = new char[20];
giorni = new Data[2];
giorni[0] = new Data();
giorni[1] = new Data();
```

# Inizializzazione di un array

- Quando si crea un array, tutti gli elementi sono inizializzati automaticamente ai valori di default (`null` nel caso di classi)
- Per fare un'inizializzazione esplicita:

```
String nomi[] = {            // inizializza un array di 3 stringhe
    "Chiara",
    "Paola",
    "Silvia",
};

Data giorni[] = {            // inizializza un array di due
    new Data(),              // oggetti di tipo Data
    new Data(),
};
```



# Array multidimensionali

- Tabelle di tabelle

```
int dueDim [][]= new int[2][];
dueDim[0] = new int[5];
dueDim[1] = new int[3];
```

L'oggetto creato dalla prima chiamata di new è un array che contiene 2 elementi, ciascuno di tipo array di interi

- Tabelle rettangolari

```
int rett [][]= new int[2][3];
```

oppure

```
dueDim[0] = new int[3];
dueDim[1] = new int[3];
```

L'oggetto creato dalla prima chiamata di new è un array che contiene 2 elementi, ciascuno di tipo array di 3 interi

# Copiare il contenuto di un array

- Un array non è ridimensionabile
- Si può usare la stessa variabile riferimento per un riferirsi ad un array completamente diverso

```
int elementi[] = new int[6];
elementi = new int[10];
```

- Per copiare il contenuto di un array in un altro, si usa il metodo `System.arraycopy()`

```
int piccolo[] = { 1,2,3,4,5,6 };
int grande[]  = { 11,12,13,14,15,16,17,18,19,20 };
System.arraycopy( piccolo,0,grande,0,piccolo.length );
```

a questo punto l'array `grande` ha il seguente contenuto: `{ 1,2,3,4,5,6,17,18,19,20 }`



# Array: "oggetti" particolari

- I membri di un tipo array sono:
  - i membri ereditati dalla sua superclasse (implicita) `Object`, ad esempio il metodo `getClass`.
  - il campo `length`, che è una costante (`final`) di ogni array, il cui tipo è `int` e che contiene il numero di elementi dell'array

```
class Test {
        public static void main(String[] args) {
                int[] a = new int[3];
                int[] b = new int[6];
                System.out.println(a.getClass() == b.getClass());
                System.out.println("num elementi = " + a.length);
        }
}
```

produce in output

```
true
num elementi = 3
```

# Esempio: oggetti e array

```
class Point {
        int x, y;
        Point() { System.out.println("default"); }          // creatore di default
        Point(int x, int y) { this.x = x; this.y = y; }     // creatore a 2 valori
        public String toString() {                          // overriding del metodo toString
                return " (" + x + "," + y + ") ";           //            della classe Object
        }
}

class Test {
        public static void main(String[] args) {
                Point p1 = null;
                Point p2 = new Point();
                Point a[] = { new Point(1,1), new Point(2,3) };

                System.out.println("p1, p2: " + p1 + p2);
                System.out.println("a: { " + a[0] + ", " + a[1] + " }");

                String sa[] = new String[2];
                sa[0] = "Ci"; sa[1] = "ao";
                System.out.println(sa[0] + sa[1]);
        }
}
```

produce come risultato:

```
default
p1,p2: null (0,0)
a:{ (1,1) , (2,3) }
Ciao
```



# Uso dei tipi

- Nelle dichiarazioni
  - un tipo può essere importato da un altro package
  - ogni campo (variabile) di una classe ha un tipo
  - ogni parametro nei metodi e nei costruttori di una classe ha un tipo
  - il risultato di un metodo (se esiste) ha un tipo
  - le variabili locali e il parametro del gestore delle eccezioni hanno un tipo
- Nelle espressioni
  - nella creazione di una istanza di una classe
  - nella creazione di un array
  - nel casting
  - usando l'operatore `instanceof`

# Esempio

```
import java.util.Random;
class MiscMath {
        int divisore;

        MiscMath(int divisore) {
                this.divisore = divisore;
        }

        float ratio(long l) {
                try {
                        l /= divisor;
                } catch (Exception e) {
                        if (e instanceof ArithmeticException)
                                l = Long.MAX_VALUE;
                        else
                                l = 0;
                }
                return (float)l;
        }

        double gausser() {
                Random r = new Random();
                double[] val = new double[2];
                val[0] = r.nextGaussian();
                val[1] = r.nextGaussian();
                return (val[0] + val[1]) / 2;
        }
}
```



# Variabili

- Una variabile è una locazione di memoria cui è associato un tipo fissato
- Il valore di una variabile può essere modificato solo da assegnamenti o da operatori di incremento e decremento
- Le variabili di tipo primitivo contengono sempre valori esattamente di quel tipo
- Le variabili di tipo riferimento contengono valori che possono essere il valore `null` oppure un riferimento ad un qualsiasi oggetto la cui classe sia compatibile rispetto all'assegnamento con il tipo della variabile

# Variabili di tipo..., Oggetti di classe...

```
public interface Colorable {
        void setColor(byte r, byte g, byte b);
}
class Point { int x, y; }

class ColoredPoint extends Point implements Colorable {
        byte r, g, b;

        public void setColor(byte rv, byte gv, byte bv) {
                r = rv; g = gv; b = bv;
        }
}

class Test {
        public static void main(String[] args) {
                Point p = new Point();
                ColoredPoint cp = new ColoredPoint();
                p = cp;
                Colorable c = cp;
        }
}
```



# Variabili di tipo..., Oggetti di classe...

- “Tipo” è un concetto legato alla compilazione. Il tipo di un’espressione può essere dedotto in compilazione
- Ogni oggetto appartiene ad una particolare classe. Un oggetto è detto istanza della sua classe e di tutte le sue superclassi
- Una variabile o un’espressione può avere un tipo interfaccia, ma *non ci sono istanze di interfaccia*: una variabile o espressione il cui tipo è un’interfaccia può essere un riferimento a qualsiasi oggetto la cui classe implementa quell’interfaccia