UNIVERSITÀ DEGLI STUDI DI BRESCIA · unibs.it

AA 2026/27

PAJC · PROGRAMMAZIONE AVANZATA IN JAVA E C

LO1

Warm-up: il bimondo

Oggetti, classi e i due mondi del corso

prof. Massimiliano Redolfi

Università degli Studi di Brescia · Ingegneria Informatica

Il corso in una slide

Docente	prof. Massimiliano Redolfi, massimiliano.redolfi@unibs.it
Assistente	dott. Carlo Santagostino, carlo.santagostino@unibs.it
Orario	mercoledì · lezione — venerdì pomeriggio · laboratorio
Ore	88 totali: 40 di lezione, 48 di laboratorio
Materiale	sito del corso + repository git del corso
Piattaforma	Java 21 o superiore · C (C17) · Eclipse · VS Code

> Le slide sono l'indice del corso: non sostituiscono le dispense né, soprattutto, il codice scritto insieme.

Come lavoreremo

- **Lezione** — i concetti, con il codice sempre in primo piano; Java e C a confronto
- **Laboratorio a step** — un problema in passi da 5–10 minuti: prima lo affrontate voi, poi lo risolviamo insieme
- **Repository del corso** — le tracce sono disponibili *prima* del laboratorio, le soluzioni *dopo*
- Ogni scheda di laboratorio dichiara il suo **punto di atterraggio minimo**: la parte da completare; il resto è estensione

> Il laboratorio è efficace solo se il codice lo scrivete voi.

L’esame

Due modalità (regolamento completo sul sito del corso):

1. **Prova scritta** — test a risposta multipla *(soglia 5/10)* seguito da uno scritto con domande di teoria e un esercizio Java
2. **Progetto** — test a risposta multipla *(soglia 7/10)* e progetto di gruppo (al più 3 persone): applicazione **client/server** con GUI **Swing**, pattern **MVC** e **thread**

L’idea di progetto va sottoposta al docente e approvata **prima** di iniziare lo sviluppo.

L’obiettivo del corso

Al termine saprete progettare e realizzare **un’applicazione grafica distribuita** — nella pratica del corso: un gioco in rete.

- una **GUI** a componenti, con eventi e animazioni
- un **modello** separato dalla vista (pattern MVC)
- **thread** per calcolare senza bloccare l’interfaccia
- un **server** che coordina più client via socket

Il **C** ci accompagna per tutto il semestre con un ruolo preciso: capire come Java funziona a livello di macchina — memoria, puntatori, stack, heap.

01

I due mondi

Java e C, a confronto dal primo giorno

Hello, world — due volte

JAVA

```java
// Hello.java — Java 21
public class Hello {
    public static void main(String[] args) {
        System.out.println("Ciao PAJC");
    }
}
```

In Java il `main` è un **metodo**: sta in una classe, ed è `static` perché nessun oggetto esiste ancora.

C

```c
/* hello.c — C17 */
#include <stdio.h>

int main(void) {
    printf("Ciao PAJC\n");
    return 0;
}
```

In C il `main` è una **funzione**: nessuna classe, nessun oggetto.

Dal sorgente all’esecuzione

JAVA

```
Hello.java
   │  javac
   ▼
Hello.class      bytecode portabile
   │  java (JVM + JIT)
   ▼
esecuzione, qualunque piattaforma
```

Un solo compilato; una JVM per piattaforma.

C

```
hello.c
   │  gcc / clang
   ▼
hello            binario nativo
   │  caricato dal sistema operativo
   ▼
esecuzione, su questa piattaforma
```

Un binario per ogni piattaforma di destinazione.

02

Il bimondo

Un mondo a due dimensioni, i cui abitanti devono classificare i propri oggetti. Il nostro compito: aiutarli.

Un rettangolo: dati e comportamento

JAVA

```java
public class Rectangle {
    private int width, height;

    public Rectangle(int w, int h) {
        this.width = w;
        this.height = h;
    }

    public int perimeter() {
        return 2 * (width + height);
    }
}
```

La classe unisce dati e comportamento; `this` è l'oggetto corrente.

C

```c
typedef struct {
    int width;
    int height;
} rectangle_t;

int perimeter(const rectangle_t *r) {
    return 2 * (r->width + r->height);
}
```

La struct contiene solo i dati; il comportamento è una funzione a parte, e l'oggetto le va passato.

`new`, costruttori, riferimenti

```
Rectangle r = new Rectangle(5, 7);   // allocazione + costruzione
Rectangle s = r;                     // copia del riferimento, non dell'oggetto

System.out.println(r.perimeter());           // 24
```

- `new` alloca l'oggetto nello **heap** e restituisce un *riferimento*
- il **costruttore** garantisce che l'oggetto nasca in uno stato valido
- `r.perimeter()` equivale a: *esegui* `perimeter` *con* `this = r`

> Stack, heap e garbage collector saranno trattati in dettaglio nella parte C del corso.

Ereditarietà

```java
public class Square extends Rectangle {
    public Square(int side) {
        super(side, side);        // prima si costruisce la parte Rectangle
    }
}
```

```java
Square q = new Square(9);
System.out.println(q.perimeter());        // 36 — metodo ereditato
```

- `Square` **è un** `Rectangle`: ne eredita campi e metodi
- `super(...)`: costruzione a catena, dal padre verso il figlio
- alla radice di ogni gerarchia c'è `Object`

Tipo statico e tipo dinamico

```
Object obj = new Square(9);   // compilatore: Object — runtime: Square
```

La forma classica — test e cast separati:

```
if (obj instanceof Rectangle) {
    Rectangle r = (Rectangle) obj;
    System.out.println(r.perimeter());
}
```

Pattern matching JAVA 16 — test e variabile in un passo:

```
if (obj instanceof Rectangle r) {
    System.out.println(r.perimeter());
}
```

`equals` e `hashCode`: perché contano

```
var visited = new HashSet<Point>();       // Point: una classe con i campi x e y
visited.add(new Point(1, 2));
visited.contains(new Point(1, 2));        // false!
```

`HashSet` e `HashMap` confrontano *per contenuto* solo se la classe definisce `equals` e `hashCode`. Altrimenti vale l'identità del riferimento, come `==`.

```
@Override public boolean equals(Object o) {
    return o instanceof Point p && x == p.x && y == p.y;
}
@Override public int hashCode() { return Objects.hash(x, y); }
```

> Il `contains` che non trova un elemento presente è tra i bug più frequenti nei progetti d'esame. La causa è quasi sempre questa.

Interfacce: contratti di comportamento

Nel bimondo arrivano le astronavi: non sono forme, ma anche loro si disegnano.

```
public interface Drawable {
    void draw();
}

public class Circle    extends Shape2d implements Drawable { ... }
public class SpaceShip implements Drawable { ... }   // nessuna parentela
```

```
List<Drawable> scene = List.of(circle, ship);
scene.forEach(Drawable::draw);
```

L’ereditarietà dice **che cosa è** un oggetto; l’interfaccia dice **che cosa sa fare**.

`enum` e macchine a stati

```
public enum State { IDLE, RUNNING, PAUSED, DONE }
```

Con le **switch expression** JAVA 14 la tabella di transizione è compatta e completa:

```
state = switch (state) {
    case IDLE    -> State.RUNNING;
    case RUNNING -> State.PAUSED;
    case PAUSED  -> State.RUNNING;
    case DONE    -> State.DONE;
};
```

- nessun `break`, nessun *fall-through* accidentale; il compilatore richiede tutti i casi
- gli enum sono classi: possono avere campi, costruttori e metodi

Le collezioni del corso

Interfaccia	Implementazione	Quando
`List<T>`	`ArrayList`	ordine e accesso per indice
`Set<T>`	`HashSet`, `TreeSet`	unicità
`Map<K,V>`	`HashMap`	ricerca per chiave
`Deque<T>`	`ArrayDeque`	pila e coda

```
List<String> nomi = List.of("Rectangle", "Circle");   // immutabile
Map<String, Integer> conta = new HashMap<>();
```

> `Vector` e `Stack` sono classi legacy: nel codice nuovo si usano `ArrayList` e `ArrayDeque`.

Il censimento del bimondo

Contare le forme per tipo, con gli strumenti visti finora:

```
Map<Class<?>, Integer> census = new HashMap<>();

for (Shape2d s : shapes) {
    Integer c = census.get(s.getClass());
    census.put(s.getClass(), c == null ? 1 : c + 1);
}
// {class Rectangle=3, class Circle=5, class Diamond=1}
```

- `getClass()` restituisce il tipo a runtime
- lo schema *leggi–modifica–scrivi* sulla mappa è corretto ma verboso

> Alla settimana 3, con gli stream, questa operazione diventerà una sola riga.

Risorse e chiusura: try-with-resources

La gestione manuale, fragile e verbosa:

```
BufferedReader in = null;
try {
    in = new BufferedReader(new FileReader("dati.txt"));
} finally {
    if (in != null) in.close();
}
```

La forma corretta, da usare sempre nel corso:

```
try (var in = new BufferedReader(new FileReader("dati.txt"))) {
    String line;
    while ((line = in.readLine()) != null)
        System.out.println(line);
}   // chiusura garantita, anche in presenza di eccezioni
```

03

Verso il laboratorio

Per casa: `FileUtil`

Da completare prima del prossimo laboratorio:

1. `print(String path)` — stampa un file di testo numerando le righe
2. `dump(String path)` — stampa un file byte per byte, in esadecimale
3. *Estensione*: output a colori con le sequenze ANSI

> Venerdì in laboratorio: configurazione dell'ambiente, il bimondo con l'interfaccia Shape2d, StringMagic. Le tracce sono già nel repository del corso.

Riferimenti

- **JDK 21 o superiore** (Temurin) — adoptium.net, da installare prima di venerdì
- **Eclipse** 2023-12 o successivo
- **Repository del corso** — collegamento dal sito redolfi-unibs.github.io

Per approfondire

Effective Java, J. Bloch	progettazione delle classi
dev.java/learn	tutorial ufficiali aggiornati
Dispense del corso	in pubblicazione, per capitoli

Prossima lezione: **classi anonime, lambda, method reference** e le interfacce a fondo.