LABORATORIO DI PROGRAMMAZIONE
CORSO DI LAUREA IN SICUREZZA DEI SISTEMI E DELLE RETI INFORMATICHE
UNIVERSITÀ DEGLI STUDI DI MILANO
2025–2026

Indice

Parte 1. Programmazione orientata agli oggetti 3
Esercizio 1 3
Classe `Matrice`*.* 3
Tempo: 10 min. 3
Esercizio 2 3
Rimozione doppioni. 3
Tempo: 10 min. 3
Esercizio 3 3
Rimozione doppioni – estensioni (1). 3
Tempo: 10 min. 3
Esercizio 4 3
Rimozione doppioni – estensioni (2). 3
Tempo: 20 min. 4
Esercizio 5 4
Classe `Password`*.* 4
Tempo: 20 min. 4
Esercizio 6 4
Classe `Dado`*.* 4
Tempo: 20 min. 4
Esercizio 7 5
Classi `Carta` *e* `MazzoDiCarte`*.* 5
Tempo: 30 min. 5

Parte 2. Eccezioni 5
Esercizio 8 5
Gestire eccezioni semplici (1). 5
Tempo: 10 min. 5
Esercizio 9 5
Gestire eccezioni semplici (2). 5
Tempo: 10 min. 6
Esercizio 10 6
Gestire eccezioni semplici (3). 6
Tempo: 10 min. 6
Esercizio 11 6

Ultima revisione: 14 dicembre 2025.

Migliorare un codice esistente aggiungendo le eccezioni (1). 6
Tempo: 10 min. 6
Esercizio 12 6
Migliorare un codice esistente aggiungendo le eccezioni (2). 6
Tempo: 10 min. 6
Esercizio 13 6
Migliorare un codice esistente aggiungendo le eccezioni (3). 6
Tempo: 20 min. 6
Esercizio 14 6
Definire eccezioni personalizzate (1). 6
Tempo: 20 min. 7
Esercizio 15 7
Definire eccezioni personalizzate (2). 7
Tempo: 20 min. 7

Iniziate completando gli esercizi della volta scorsa.

Parte 1. **Programmazione orientata agli oggetti**

Esercizio 1

Classe `Matrice`*.*

Tempo: 10 min.

Implementate la classe `Matrice` che rappresenta la classe di matrici di numeri interi.

- Definite attributi e metodi per gestire questa entità
- Tra i metodi, deve permettere di fare la *somma*, *moltiplicazione* (moltiplicazione matriciale riga per colonna, moltiplicazione per uno scalare).

Le operazioni di somma e moltiplicazione devono restituire una nuova matrice, senza modificare le matrici originali. Verificate che le operazioni siano possibili (ad esempio, per la somma le matrici devono essere della stessa dimensione). In caso contrario, restituite `null`. Scrivete un `main` per testare i metodi.

Esercizio 2

Rimozione doppioni.

Tempo: 10 min.

Scrivete la classe `UniqueListArray` che, a partire da un `Array` di `char`, ne elimini i doppioni. Specificate al momento della creazione dell'oggetto la grandezza massima `size`.

Per aggiungere un nuovo elemento ad un `UniqueListArray` utilizzate un metodo `add` che aggiunge un nuovo valore per volta, se non duplicato.

Esercizio 3

Rimozione doppioni – estensioni (1).

Tempo: 10 min.

Estendete l'oggetto `UniqueListArray` permettendo che venga creato anche a partire da un `Array` di `char` contenente dei doppioni. Cosa succederebbe se venisse creato ricevendo invece una `String`? Implementate anche questa variante.

Esercizio 4

Rimozione doppioni – estensioni (2).

Tempo: 20 min.

Estendete l'esercizio precedente in maniera tale che, oltre ai caratteri, possa avere in ingresso oggetti di altro tipo come ad esempio `Vett`, `Frazioni`, eccetera. Per farlo, utilizzare l'oggetto generico `Object`.

Che tipo di ADT rappresenta questa lista? Quale metodo devono avere questi oggetti al fine di poter eliminare i doppi? Pensate prima alla struttura del programma e delle classi prima di iniziare l'implementazione.

Esercizio 5

Classe `Password`.

Tempo: 20 min.

Definite una classe `Password` che rispetti le seguenti caratteristiche:

- Una password valida deve avere almeno 8 caratteri.
- Una password valida deve avere solo lettere e numeri (non caratteri speciali)
- Una password valida deve avere almeno 2 numeri.

Quali attributi deve avere la classe password? Con che visibilità? E quali metodi deve esporre?

Ad esempio, deve permettere di verifiare che la password inserita dall'utente sia uguale a quella contenuta dall'oggetto `Password`. Implementatate il metodo per farlo.

Nota: per svolgere l'esercizio implementate delle funzioni statiche che verifichino le caratteristiche di un carattere (se un carattere è un numero, o se è una lettera maiuscola o minuscola).

Testate la classe con un apposito `main`.

Come si potrebbe modificare la classe nel caso le specifiche cambiassero (ad esempio, se fosse richiesta la presenza di almeno un carattere maiuscolo?). E se dovessi definire più tipi di `Password` con caratteristiche diverse, come posso fare? Implementate le varianti alla classe `Password` seguendo questa specifica.

Esercizio 6

Classe `Dado`.

Tempo: 20 min.

Definite una classe `Dado`, che simuli il comportamento di un dado da gioco. Ogni dado viene inizializzato con un numero di facce, che poi è fisso. Gestite accuratamente come definire gli attributi della classe `Dado`. Definire il metodo `lancia` che restituisce un valore casuale simulando il lancio del dado. Definire infine dei metodi statici che permettano di ottenere un valore lanciato da un dado, senza aver bisogno di istanziare un oggetto di classe `Dado`.

> Per ottenere un numero casuale intero utilizzate la funzione `nextInt(int)` della classe `SecureRandom`, definita in `java.security.SecureRandom`.

Testate la classe così creata in un programma `main` che svolge le seguenti operazioni. Simulate il 200 lanci di un dado da 20 e contate quante volte ogni occorrenza viene estratta. Per farlo utilizzare un `Array`. Visualizzate il risultato a schermo.

Esercizio 7

Classi `Carta` e `MazzoDiCarte`.

Tempo: 30 min.

Definite una classe `Carta`, che definisca una carta da gioco. Ogni classe deve avere un seme ed un valore. Definire semi e valori con delle `enum`. Il metodo `toString` della carta deve restituire il valore della stringa, seguito dal seme, come ad esempio `Asso di cuori`.

Definite una classe `MazzoDiCarte` che rappresenti un mazzo di carte di 52 elementi. Tale valore deve essere definito con una costante `static final`. Al momento della creazione di un nuovo oggetto `MazzoDiCarte`, vengono creati gli oggetti di tipo `Carta`, ordinati (prima i cuori, poi i quadri, ...).

Definire un metodo `mescola`, che riordini in maniera casuale il mazzo. Definire un metodo `distribuisci` che restituisca una carta del mazzo, in ordine. L'oggetto `MazzoDiCarte` deve tenere conto di quante carte sono state distribuite.

Per rimescolare il mazzo, utilizzate il metodo `nextInt` usata al punto precedente per scambiare due carte di posizione.

Come esercizio aggiuntivo, implementate l'algoritmo di Fisher-Yates[a] per rimescolare il mazzo di carte.

[a] `https://en.wikipedia.org/wiki/Fisher-Yates_shuffle`

Testare il programma con un `main` che istanzi un mazzo di carte, lo rimescoli, e stampi in sequenza tutte le carte presenti.

Parte 2. Eccezioni

Esercizio 8

Gestire eccezioni semplici (1).

Tempo: 10 min.

Scrivete un programma che, inserito un numero intero da tastiera, stampi a video il risultato della divisione di tale numero per un altro numero generato casualmente, compreso tra 0 e 3. Gestite l'eccezione relativa.

Esercizio 9

Gestire eccezioni semplici (2).

Tempo: 10 min.

Scrivete un programma che, istanziato un vettore di `n` elementi, provi ad accedere all'elemento `n+1`-esimo. Gestite l'eccezione. Cosa succede se non viene gestita?

Esercizio 10

Gestire eccezioni semplici (3).

Tempo: 10 min.

Scrivete un programma che prenda in ingresso un numero reale, oppure un numero immaginario, sfruttando il costrutto `try-catch`. Assumete che un numero immaginario sia rappresentato come una stringa che termina con la lettera `j`, ad esempio `3.5j`. Usate il metodo statico `Double.parseDouble` per convertire un stringa in un numero reale, il quale solleva una eccezione `NumberFormatException` se la stringa non rappresenta un numero reale valido.

Esercizio 11

Migliorare un codice esistente aggiungendo le eccezioni (1).

Tempo: 10 min.

Partite dall'esercizio 1 della lezione 16 (classe `Matrice`). Modificate i metodi di somma e moltiplicazione in modo che, in caso di dimensioni incompatibili, venga sollevata una eccezione di tipo `IllegalArgumentException`. Usate il `main` per testare adeguatamente la nuova funzionalità.

Esercizio 12

Migliorare un codice esistente aggiungendo le eccezioni (2).

Tempo: 10 min.

Partite dall'esercizio 9 della lezione 14 (classe `Frazione`). Scrivete un costruttore che sollevi una eccezione di tipo `IllegalArgumentException` se il denominatore è uguale a zero. Usate il `main` per testare adeguatamente la nuova funzionalità.

Nel `main` dovete per forza usare `try/catch`. Perché?

Esercizio 13

Migliorare un codice esistente aggiungendo le eccezioni (3).

Tempo: 20 min.

Partite dall'esercizio 5 (classe `Password`). Migliorate la classe `Password` in modo che, se la password non rispetta i vincoli richiesti, venga sollevata una eccezione di tipo `IllegalArgumentException`. Testate la nuova funzionalità con un apposito `main`.

Esercizio 14

Definire eccezioni personalizzate (1).

Tempo: 20 min.

Modificate l'esercizio precedente in modo che venga definita una eccezione personalizzata `InvalidPasswordException` che estende la classe `Exception`. La nuova eccezione deve essere sollevata quando la password non rispetta i vincoli richiesti. Testate la nuova funzionalità con un apposito `main`.

Esercizio 15

Definire eccezioni personalizzate (2).

Tempo: 20 min.

Modificate l'esercizio precedente definendo una serie di eccezioni personalizzate che estendono la classe `InvalidPasswordException`, una per ogni vincolo non rispettato (ad esempio, `TooShortPasswordException`, `NoDigitPasswordException`, `SpecialCharacterPasswordException`, eccetera). Testate la nuova funzionalità con un apposito `main`. Ragionate su vantaggi e svantaggi di questa soluzione rispetto alla precedente.