SAPIENZA
UNIVERSITÀ DI ROMA

Appunti di Metodologie di Programmazione

Colacel Alexandru Andrei

Disclaimer

Indice

1 Concetti introdutivi

Metodo **main** per eseguire il programma.

Algortmo: non ambiguo, eseguibile, finito.

Identificatore: nome di una variabile di un metodo o di una classe; non possono iniziare con una cifra, non possono avere simboli (tranne $ e _); Case sensitive. I nomi delle variabili dovrebbero iniziare con una lettera minuscola (camelCase), i nomi delle **classi** con la maiuscola (PascalCase).

Gli **oggetti** sono entità di un programma che si possono manipolare invocando **metodi**. Gli oggetti appartengono a diverse classi (l'oggetto *System.out* appartiene alla classe PrintStream).

Metodo: sequenza di istruzioni che accede ai dati di un oggetto. Esempio:

```
public static void main(String[] args) {

}
```

public = specificatore di accesso
(String[] args) = parametri
{} = corpo

In Java, i **numeri** sono **tipi primitivi**, non classi.

Classe: insieme di oggetti con lo stesso comportamento. Una classe specifica i metodi che possono essere applicati ai suoi oggetti.

L'**interfaccia pubblica** di una classe specifica cosa si può fare con i suoi oggetti; L'implementazione nascosta descrive come si svolgono tali azioni.

Parametro: dati in ingresso a un metodo; non tutti i metodi hanno parametri. Il tipo del parametro implicito è la classe m cui è definito il metodo (non viene menzionato → implicito).

Overloading: più metodi con lo stesso nome ma numero e/o tipo di parametri diversi; l'overloading può essere fatto nella stessa classe o nelle classi derivate. Il tipo di ritorno deve essere diverso.

Overriding: sovrascrittura di metodi ereditati da una superclasse.

Costruzione: processo che crea un nuovo oggetto.

Metodo d'Accesso: metodo che accede ad un oggetto e restituisce informazioni senza modificare l'oggetto (metodi `get`)

Metodo Modificatore: modifica lo stato di un oggetto.

Programma di collaudo:

1. Definisce una nuova classe
2. Definisce in essa il metodo `main()`
3. Costruisce uno o più oggetti all'interno del metodo `main()`
4. Applica metodi agli oggetti
5. Visualizza i risultati delle invocazioni dei metodi
6. Visualizza i valori previsti

Astrazione: processo di eliminazione delle caratteristiche inesenziali, finchè non rimane soltanto l'essenza del concetto;

- Fu usato il processo di astrazione per inventare tipi di dati a un livello superiore rispetto a numeri e caratteri
- Nella programmazione ad oggetti, gli oggetti sono *Scatole Nere* (non si conoscono i meccanismi interni)
- **Incapsulamento:** la struttura interna di un oggetto è nascosta al programmatore, che però ne conosce il comportamento

Costrutture: ha il nome della classe; contiene istruzioni per l'inizializzazione degli oggetti. Assegna un valore iniziale alle variabili di istanza di un oggetto.

Iterfaccia pubblica = costruttori + metodi pubblici

Variabili di istanza: un oggetto memorizza i propri dati all'interno di CAMPI(variabili, identifica una posizione all'interno di un blocco di memoria) di ESEMPLARE (istanza, è un oggetto creato dalla classe). La dichiarazione della classe specifica le sue variabili di istanza. Specificatore di **accesso**(private) + **tipo**(double, string, ...) + **nome**.

Ciascun oggetto di una classe ha il proprio insieme di variabili di istanza.

Se le variabili di istanza sono private, ogni accesso ai dati deve avvenire tramite metodi pubblici. L'incapsulamento prevede l'occultamento dei dati degli oggetti, fornendo metodi per accedervi.

this: permette di accedere al parametro implicito. Ogni metodo ha sempre un parametro implicito(this), ad eccezione dei metodi dichiarati `static`

Tipi primitivi: int (4 bytes), byte(1 byte), short (2 bytes), long (8 bytes), double (8 bytes), float (4 bytes), char (2 bytes), boolean (1 bit).

Variabile Final: una volta settato il valore non può essere più cambiato, usato per assegnare valori costanti. Se usato su una classe questa non potrà avere sottoclassi; se usato su un metodo, le sottoclassi non possono effettuare un `@override`; se usato su un attributo, non potrà essere modificato. Inomi delle costanti sono scritti in MAIUSCOLO.

Static Final: per consentire l'accesso ad altre classi.

I metodi *static* se usati sul metodo è accessibile sena dover instanziare l'oggetto (es. `Math.sqrt(n)`). Se usato su un attributo, questo sarà condiviso tra tutte le istanze della classe.

Cast (forzatura): converte in numero intero il valore in virgola mobile. Forzare un valore di un tipo ad avere un altro tipo. Qualche esempio:

```
int x = 10;
double y = x; // Casting implicito da int a double
System.out.println(y); // Output: 10.0

double a = 10.5;
int b = (int) a; // Casting esplicito da double a int
System.out.println(b); // Output: 10

class Veicolo {
  // ...
}

class Auto extends Veicolo {
    // ...
}

Veicolo veicolo = new Auto();
Auto auto = (Auto) veicolo; // Casting da Veicolo ad Auto
```

2 Utilizzare oggetti

2.1 Tipi e variabili

Ogni valore è di un determinato tipo.

```
String greeting = "Hello, World!";
PrintStream printer = System.out;
int luckyNumber = 13;
```

Per definire una variabile:

```
nomeTipo nomeVariabile = valore;
  \\oppure
nomeTipo nomeVariabile;
```

2.2 Identificatori

Identificatore è il nome di una variabile, di un metodo o di una classe. Regole per gli identificatori in Java:

- Possono essere composti di lettere, cifre, caratteri "dollaro" ($) e segni di sottolineatura (_)
- non possono iniziare con una cifra
- non si possono usare altri simboli, come ? o %.
- gli spazi non sono ammessi all'interno degli identificatori
- le parole riservate non possono essere usate come identificatori
- sono sensibili alla differenza tra lettere maiuscole e minuscole

Per convenzione, i nomi delle **variabili** dovrebbero iniziare con una lettera minuscola mentre i nomi delle **classi** dovrebbero iniziare con una lettera maiuscola. Scrivere un'inizializzazione così è un **errore**:

```
int luckyNumber;
System.out.println(luckyNumber);
// ERRORE - variabile priva di valore
```

Infatti bisogna assegnarla prima di stamparla:

```
nomeVariabile = valore;
luckyNumber = 12;
```

2.2.1 Oggetti

Gli **Oggetti** sono entità di un programma che si possono manipolare invocando metodi. Tali oggetti appartengono a diverse classi. Per esempio l'oggetto `System.out` appartiene alla classe `PrintStream`.
Solitamente l'oggetto creato dall'operatore `new` viene memorizzato in una variabile, in questo modo:

```
REctangle box = new Rectangle(5, 10, 20, 30)
```

2.2.2 Metodi

I **Metodi** è una sequenza di istruzioni che accede ai dati di un oggetto. Gli oggetti possono essere manipolati invocando metodi. Quando in una classe si definisce un metodo, vengono specificati i tipi dei parametri espliciti e del valore restituito. Il tipo del parametro implicito è la classe in cui è definito il metodo: ciò non viene menzionato nella definizione del metodo, e proprio per questo si parla di parametro"implicito".

Se il metodo non restituisce un valore, il tipo di valore restituito viene dichiarato come `void`. Il nome di un metodo è **overloading** se una classe definisce più metodi con lo stesso nome (ma con parametri di tipi diversi).

2.2.3 Classe

Una **Classe** è un insieme di oggetti con lo stesso comportamento. Una classe specifica i metodi che possono essere applicati ai suoi oggetti.

2.2.4 Interfaccia pubblica

L'**interfaccia pubblica** di una classe specifica cosa si può fare con i suoi oggetti mentre l'implementazione nascosta descrive come si svolgono tali azioni.

2.3 Parametri impliciti ed espliciti

I parametri **espliciti** sono quelli che vengono dichiarati nella firma del metodo e devono essere passati in modo specifico quando il metodo viene chiamato.

```
int sum = addNumbers(a, b);
System.out.println("La somma è: " + sum);
```

I parametri **impliciti** sono quelli che vengono passati a un metodo senza essere dichiarati nella sua firma. Questo si verifica quando il metodo ha accesso alle variabili di istanza o di classe della classe in cui è definito.

```
// Chiamata al metodo senza parametri espliciti
int sum = example.addNumbers();
System.out.println("La somma è: " + sum);
```

2.4 Costrutture

I costruttori contengono istruzioni per inizializzare gli oggetti. Il nome di un costruttore è sempre uguale al nome della classe.

```
public BankAccount()
  {
    // corpo, che verrà riempito più tardi
  }
```

Il corpo del costruttore è una sequenza di enunciati che viene eseguita quando viene costruito un nuovo oggetto. Gli enunciati presenti nel corpo del costruttore imposteranno i valori dei dati interni dell'oggetto che è in fase di costruzione. Tutti i costruttori di una classe hanno lo stesso nome, che è il nome della classe. Il compilatore è in grado di distinguere i costruttori, perché richiedono parametri diversi.
Un costruttore assegna un valore iniziale alle variabili di istanza di un oggetto.

```
public BankAccount()
{
balance = 0;
}
public BankAccount(double initialBalance)
{
balance = initialBalance;
}
```

2.5 Esempio di invocazione di un costruttore

```
BankAccount harrysChecking = new BankAccount(1000);
```

Creazione di un nuovo oggetto di tipo BankAccount:

- Invocazione del secondo costruttore (perché è stato fornito un parametro di costruzione).
- Assegnazione del valore 1000 alla variabile parametro initialBalance.
- Assegnazione del valore di initialBalance al campo di esemplare balance dell'oggetto appena creato.
- Restituzione, come valore dell'espressione new, di un riferimento a un oggetto, che è la posizione in memoria dell'oggetto appena creato.
- Memorizzazione del riferimento all'oggetto nella variabile harrysChecking.

Alcuni metodi non restituiscono alcun valore mentre altri possono restituirlo. Si usa `void` nel primo caso mentre si specifica il tipo nel secondo caso.

2.6 Esempio di una invocazione di metodo

```
harrysChecking.deposit(500);
```

- Assegnazione del valore 500 alla variabile parametro amount.
- Lettura del campo balance dell'oggetto che si trova nella posizione memorizzata nella variabile harrysChecking.
- Addizione tra il valore di amount e il valore di balance, memorizzando il risultato nella variabile newBalance.
- Memorizzazione del valore di newBalance nel campo di esemplare balance, sovrascrivendo il vecchio valore.

2.7 Variabili di istanza

La dichiarazione di una variabile di istanza è così composta:

- Uno specificatore d'accesso (solitamente private)
- Il tipo del campo di esemplare (come double)
- Il nome del campo di esemplare (come balance)

Ciascun oggetto di una classe ha il proprio insieme di variabili di istanza. Le variabili di istanza sono generalmente dichiarate con lo specificatore di accesso private.

```
specificatoreDiAccesso class NomeClasse
{
. . .
specificatoreDiAccesso tipoVariabile nomeVariabile;
. . .
}

Esempio:
public class BankAccount
{
. . .
private double balance;
. . .
}
```

Serve a definire un campo che sia presente in ciascun oggetto di una classe.

Le variabili di istanza sono generalmente dichiarati con lo specificatore di accesso private: a essi si può accedere soltanto da metodi della medesima classe e da nessun altro metodo. Se i campi di esemplare (o variabili di istanza) vengono dichiarati privati, ogni accesso ai dati deve avvenire tramite metodi pubblici. L'incapsulamento prevede l'occultamento dei dati degli oggetti, fornendo metodi per accedervi.

2.8 Parametri impliciti ed espliciti nei metodi

Il parametro implicito di un metodo è l'oggetto con cui il metodo viene invocato ed è rappresentato dal riferimento `this`. Il riferimento `this` permette di accedere al parametro implicito. All'interno di un metodo, il nome di un campo di esemplare rappresenta il campo di esemplare del parametro implicito.

```
public void deposit(double amount){
double newBalance = balance + amount;
this.balance = newBalance;
}
```

3 Tipi di Dati

Type	Description	Size
`int`	The integer type, with range -2,147,483,648 (`Integer.MIN_VALUE`) . . . 2,147,483,647 (`Integer.MAX_VALUE`)	4 bytes
`byte`	The type describing a single byte, with range -128 . . . 127	1 byte
`short`	The short integer type, with range -32768 . . . 32767	2 bytes
`long`	The long integer type, with range -9,223,372,036,854,775,808 . . . 9,223,372,036,854,775,807	8 bytes
`double`	The double-precision floating-point type, with a range of about $\pm 10^{308}$ and about 15 significant decimal digits	8 bytes
`float`	The single-precision floating-point type, with a range of about $\pm 10^{38}$ and about 7 significant decimal digits	4 bytes
`char`	The character type, representing code units in the Unicode encoding scheme	2 Bytes
`boolean`	The type with the two truth values `false` and `true`	1 bit

Figura 1

3.1 final

Utilizza nomi simbolici per tutti i valori, anche quelli che sembrano ovvi. Una variabile `final` è una costante. Una volta che il suo valore è stato impostato, non può essere modificato. Le costanti rendono i programmi più facili da leggere e mantenere. La convenzione ci consiglia di utilizzare nomi completamente in maiuscolo per le costanti.

```
final double QUARTER_VALUE = 0.25;
final double DIME_VALUE = 0.1;
final double NICKEL_VALUE = 0.05;
final double PENNY_VALUE = 0.01;

payment = dollars + quarters * QUARTER_VALUE + dimes * DIME_VALUE + nickels * NICKEL_VALUE +
    pennies * PENNY_VALUE;
```

3.2 static final

Il modificatore static, in combinazione con il modificatore final, viene utilizzato anche per definire le costanti. Il modificatore final indica che il valore di questo campo non può cambiare.

Ad esempio, la seguente dichiarazione di variabile definisce una costante denominata PI, il cui valore è un'approssimazione di pi greco (il rapporto tra la circonferenza di un cerchio e il suo diametro):

```
static final double PI = 3.141592653589793;
```

Le costanti definite in questo modo non possono essere riassegnate ed è un errore in fase di compilazione se il tuo programma tenta di farlo. Per convenzione, i nomi dei valori costanti sono scritti in lettere maiuscole. Se il nome è composto da più parole, le parole sono separate da un carattere di sottolineatura (_).

3.3 Casting

Il casting del tipo si verifica quando si assegna un valore di un tipo di dati primitivo a un altro tipo. In Java, ci sono due tipi di casting:

- Widening Casting (automatico): conversione di un carattere più piccolo in uno più grande.

```
byte -> short -> char -> int -> long -> float -> double


\\Esempio:
public class Main {
public static void main(String[] args) {
int myInt = 9;
double myDouble = myInt; // Automatic casting: int to double

System.out.println(myInt);      // Outputs 9
System.out.println(myDouble);   // Outputs 9.0
  }
}
```

- Narrowing Casting (manualmente): conversione di un tipo più grande in un tipo più piccolo.

```
double -> float -> long -> int -> char -> short -> byte


\\Esempio:
public class Main {
public static void main(String[] args) {
double myDouble = 9.78d;
int myInt = (int) myDouble; // Manual casting: double to int

System.out.println(myDouble);   // Outputs 9.78
System.out.println(myInt);      // Outputs 9
  }
}
```

3.4 Formattare l'output

Possiamo usare il metodo `printf` per decidere come formattare un valore.
Condideriamo:

```
price = 1.215962441314554;
System.out.printf("%.2f", price);
\\output
>> 1.22
```

4 Prendere in input: Scanner

Se vogliamo ottenre un oggetto di tipo `Scanner` bisogna importare la classe:

```
import java.util.Scanner;
```

e successivamente:

```
Scanner in = new Scanner(System.in);
```

Per leggere un valore di tipo `int` possiamo scrivere:

```
System.out.print("Please enter the number of bottles: ");
int bottles = in.nextInt();
```

Quando il metodo `nextInt` è chiamato, il programma attende che l'utente inserisce da tastiera il valore e preme Invio. Successivamente il valore viene inserito nella variabile.

4.1 Leggere, scrivere e creare file

```
import java.io.*;

public class FileIOExample {
    public static void main(String[] args) {
        String filePath = "path/to/your/file.txt";

        // Creazione di un nuovo file
        try {
            File file = new File(filePath);
            if (file.createNewFile()) {
                System.out.println("Il file è stato creato con successo.");
            } else {
                System.out.println("Il file esiste già.");
            }
        } catch (IOException e) {
            System.out.println("Si è verificato un errore durante la creazione del file.");
            e.printStackTrace();
        }

        // Scrittura su un file
        try {
            FileWriter writer = new FileWriter(filePath);
            writer.write("Questo è un esempio di testo scritto nel file.");
            writer.close();
            System.out.println("Il testo è stato scritto nel file con successo.");
        } catch (IOException e) {
            System.out.println("Si è verificato un errore durante la scrittura del file.");
            e.printStackTrace();
        }

        // Lettura da un file
        try {
            FileReader reader = new FileReader(filePath);
            BufferedReader bufferedReader = new BufferedReader(reader);
            String line;
            System.out.println("Contenuto del file:");
            while ((line = bufferedReader.readLine()) != null) {
                System.out.println(line);
            }
            bufferedReader.close();
        } catch (IOException e) {
            System.out.println("Si è verificato un errore durante la lettura del file.");
```

```
            e.printStackTrace();
        }
    }
}
```

5 I cicli while, for, do

Il ciclo `while` si scrive:

```
while (balance < targetBalance)
{
year++;
double interest = balance * RATE /
100; balance = balance + interest;
}
```

Ciclo `for`:

```
for (int counter = 1; counter <= 10; counter++)
{
System.out.println(counter);
}
```

Table 2 for Loop Examples

Loop	Values of i	Comment
`for (i = 0; i <= 5; i++)`	0 1 2 3 4 5	Note that the loop is executed 6 times. (See Programming Tip 6.3 on page 260.)
`for (i = 5; i >= 0; i--)`	5 4 3 2 1 0	Use `i--` for decreasing values.
`for (i = 0; i < 9; i = i + 2)`	0 2 4 6 8	Use `i = i + 2` for a step size of 2.
`for (i = 0; i != 9; i = i + 2)`	0 2 4 6 8 10 12 14 ... (infinite loop)	You can use `<` or `<=` instead of `!=` to avoid this problem.
`for (i = 1; i <= 20; i = i * 2)`	1 2 4 8 16	You can specify any rule for modifying `i`, such as doubling it in every step.
`for (i = 0; i < str.length(); i++)`	0 1 2 ... until the last valid index of the string `str`	In the loop body, use the expression `str.charAt(i)` to get the `i`th character.

Figura 2: Esempi di clicli `for`

Ciclo `do` I cicli `do` Esegue il corpo del ciclo di test una volta e dopo esegue il resto del corpo.

```
int value;
do
{
System.out.print("Enter an integer < 100: ");
value = in.nextInt();
}
while (value >= 100);
```

5.1 Valori sentinella

Denotano la fine di un Dataset ma non fa parte dei dati. Ovvero denotato la fine di una sequenza di inputer o il bordo tra sequenze di input.

6 Array e ArrayList

6.1 Array

Gli **Array** sono una sequenza di valori omogenei. Memorizzare in una variabile il riferimento all'array. Il tipo di una variabile che fa riferimento a un array è il tipo dell'elemento. Nel momento in cui viene creato l'array, tutti i suoi valori sono inizializzati al valore:

- 0 (per un array di numeri come int[], double[] o String[]),
- false (per un array boolean[]),
- null (per un array di riferimenti a oggetti).

Si accede agli elementi di un array a tramite un indice di tipo intero, usando la notazione a[i].

```
System.out.println("The value of this data item is " + data[2]);
```

- I valori per gli indici di un array vanno da 0 a length - 1. L'accesso a un elemento non esistente provoca il lancio di un'eccezione per errori di limiti.
- Per conoscere il numero di elementi di un array usare il campo length.
- Gli array hanno un limite pesante: la loro lunghezza è fissa.

```
//Inizializzazione Array
  double[] persone = new double[3];

//Inizializzazione Array di oggetti
public class Persona {
  private String nome;
  private int eta;

  public Persona(String nome, int eta) {
      this.nome = nome;
      this.eta = eta;
  }

  public String getNome() {
      return nome;
  }

  public int getEta() {
      return eta;
  }

}

  public class Main {
    public static void main(String[] args) {
        Persona[] persone = new Persona[3];

        persone[0] = new Persona("Mario", 30);
        persone[1] = new Persona("Luigi", 35);
        persone[2] = new Persona("Peach", 25);

        for (int i = 0; i < persone.length; i++) {
            System.out.println("Nome: " + persone[i].getNome() + ", Età: " + persone[i].getEta());
        }
    }
}
```

6.1.1 Array a due dimensioni

Gli array bidimensionali rappresentano una tabella, una disposizione di elementi a due dimensioni. Si accede agli elementi di un array bidimensionale usando una coppia di indici, a[i][j]. Quando si costruisce un array bidimensionale, si deve specificare quante righe e quante colonne servono.

```
final int ROWS = 3;
final int COLUMNS = 3;
String[][] board = new String[ROWS][COLUMNS];
```

Per accedere a un particolare elemento della matrice, si usano due indici tra parentesi quadre separatea[i][j].

```
board[i][j] = "x";
```

Quando si inseriscono o si cercano dati in un array bidimensionale, di solito si usano due cicli annidati. Per esempio, questa coppia di cicli assegna a tutti gli elementi dell'array una stringa contenente il solo carattere di spaziatura.

```
for (int i = 0; i < ROWS; i++)
  for (int j = 0; j < COLUMNS; j++)
    board[i][j] = " ";
```

6.1.2 Clonare un Array

Per copiare gli elementi di un array usate il metodo clone.

```
double[] prices = (double[]) data.clone();
```

6.1.3 Copiare un Array

Si usa il metodo `System.arraycopy` per copiare elementi da un array a un altro.

```
System.arraycopy (from, fromStart, to, toStart, count);
```

6.1.4 Inserire un elemento in un Array

```
System.arraycopy(data, i, data, i + 1, data.length - i - 1); data[i] = x;
```

6.1.5 Rimuovere un elemento da un array

```
System.arraycopy(data, i + 1, data, i, data.length - i - 1);
```

6.1.6 Far crescere un Array

Il metodo `System.arraycopy` viene anche utilizzato per far crescere di dimensione un array che non ha più spazio, seguendo queste fasi operative:

1. Creare un nuovo array, di dimensione maggiore

   ```
   double[] newData = new double[2 * data.length];
   ```

2. Copiare tutti gli elementi nel nuovo array

   ```
   System.arraycopy(data, 0, newData, 0, data.length);
   ```

3. Memorizzare nella variabile array il riferimento al nuovo array

   ```
   data = newData;
   ```

6.1.7 Trasformare array paralleli in array di oggetti

```
// non fate così
int[] accountNumbers;
double[] balances;
```

Evitare di usare array paralleli trasformandoli in array di oggetti. Usare un unico array di oggetti. Definiamo come `data.length` come la capacità dell'array data, mentre `dataSize` è la dimensione reale dell'array. Continuando ad aggiungere elementi all'array, bisogna incrementare di pari passo la variabile dimensione.

6.2 ArrayList

La classe ArrayList (vettore o lista sequenziale) gestisce oggetti disposti in sequenza.

- Un vettore può crescere e calare di dimensione in base alle necessità
- La classe ArrayList fornisce metodi per svolgere le operazioni più comuni, come l'inserimento e la rimozione di elementi
- La classe ArrayList è una classe generica: ArrayList$< T >$ contiene oggetti di tipo T.

```
ArrayList<BankAccount> accounts = new ArrayList<BankAccount>();
accounts.add(new BankAccount(1001));
accounts.add(new BankAccount(1015));
accounts.add(new BankAccount(1022));
```

- Il metodo size restituisce la dimensione attuale del vettore

6.2.1 Ispezionare gli elementi negli ArrayList

Per ispezionare gli oggetti contenuti nel vettore si usa il metodo `get` e come con gli array, i valori degli indici iniziano da 0 a i-1. Ad esempio, `accounts.get(2)` restituisce il conto bancario avente indice 2, cioè il terzo elemento del vettore:

```
BankAccount anAccount = accounts.get(2);
// fornisce il terzo elemento del vettore
```

6.2.2 Aggiungere elementi negli ArrayList

Per assegnare un nuovo valore a un elemento di un vettore già esistente si usa il metodo set:

```
BankAccount anAccount = new BankAccount(1729);
accounts.set(2, anAccount);
```

È possibile anche inserire un oggetto in una posizione intermedia all'interno di un vettore.

```
accounts.add(i, a)
```

L'invocazione `accounts.add(i, a)` aggiunge l'oggetto `c` nella posizione `i` e sposta tutti gli elementi di una posizione, a partire dall'elemento attualmente in posizione `i` fino all'ultimo elemento presente nel vettore.

6.2.3 Rimuovere elementi negli ArrayList

L'invocazione `accounts.remove(i)` elimina l'elemento che si trova in posizione `i`, sposta di una posizione all'indietro tutti gli elementi che si trovano dopo l'elemento rimosso e diminuisce di uno la dimensione del vettore.

6.2.4 Involucri negli ArrayList

Non si possono inserire valori di tipo primitivo direttamente nei vettori. Per poter manipolare valori di tipo primitivo come se fossero oggetti si usano le classi involucro[1].

```
ArrayList<Double> data = new ArrayList<Double>();
data.add(29.95);
double x = data.get(0);
Double d = data.get(0);
```

Esistono classi involucro per tutti gli otto tipi di dati primitivi.

Tipo primitivo	Classe involucro
byte	Byte
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

Figura 3

6.2.5 Ciclo `for` generalizzato

```
  int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  for (int item : numbers) {
    System.out.println("Count is: " + item);
}
```

In questo esempio, abbiamo un array di interi chiamato numbers contenente i numeri da 1 a 10. Utilizziamo il ciclo for-each per iterare attraverso ogni elemento dell'array senza dover gestire manualmente l'indice o la lunghezza dell'array.

Nella riga del ciclo for, int item : numbers, la variabile item è dichiarata come il tipo degli elementi nell'array numbers, che in questo caso è int. Durante ogni iterazione, item assume il valore dell'elemento corrente nell'array. Quindi, all'interno del blocco del ciclo for, stampiamo il valore dell'elemento utilizzando System.out.println().

6.2.6 Sintassi ciclo `for` generalizzato

```
for (Tipo variabile : aggregato)
  istruzioneInterna
    \\Esempio:
  for (double e : data)
  sum = sum + e;
```

6.3 Redirezione del flusso di ingresso

Memorizzare i valori in ingresso in un file

```
File input1.txt
15000
2
```

[1]Double è scritto con la "D" maiuscola

```
1015
10000
```

Scrivete questo comando in una finestra di shell:

```
java BankTester < input1.txt
```

Visualizza:

```
Count: 2
Expected: 2
Balance of matching account: 10000.0
Expected: 10000
```

E' possibile redigere anche il flusso in uscita

```
java BankTester < input1.txt > output1.txt
```

Il risultato di questo programma sarà la stampa dei numeri da 1 a 10, con la frase "Count is: " che precede ciascun numero. Ad esempio:

```
Count is: 1
Count is: 2
Count is: 3
...
Count is: 10
```

Questo tipo di ciclo è particolarmente utile quando si desidera solo accedere agli elementi dell'array uno alla volta senza dover gestire esplicitamente l'indice.

6.3.1 Cancellazione e Inserimento

Se vogliamo cancellare un elemento in Java presente in un Array possiamo usare due tecniche:

1. Se l'ordine non importa inseriamo nella posizione `pos` che dobbiamo cancellare l'ultimo elemento presente nell'array.
2. Altrimenti dobbiamo decrementare per ciascun elemento successivo `pos` di uno come segue:

```
for (int i = pos + 1; i < currentSize; i++){
  values[i - 1] = values[i];
}
currentSize--;
```

Nell'inserimento possiamo usare lo stesso approccio ma nel secondo caso bisogna incrementare di uno ciascun valore in questo modo: `values[i] = values[i - 1]`

6.3.2 Copiatura

Se vogliamo copiare un array possiamo invocail il metodo `Array.copyOf`:

```
double[] prices = Array.copyOf(values, values.length);
```

Nota: values.length è la nuova lunghezza che vogliamo assegnare al nuovo array. Quindi siamo noi a decidere.

7 Ereditarietà

L'ereditarietà è un metodo per estendere classi esistenti aggiungendovi metodi e campi. Per esempio, immaginate di dover definire una classe `SavingsAccount` ("conto di risparmio") per descrivere un conto bancario che garantisce un tasso di interesse fisso sui depositi.

```
class SavingsAccount extends BankAccount
{
nuovi metodi
nuovi campi di esemplare
}
```

Tutti i metodi e tutti i campi di esemplare della classe BankAccount vengono ereditati automaticamente dalla classe SavingsAccount. La classe più generica, che costituisce le basi dell'ereditarietà, viene chiamata **superclasse** (BankAccount), mentre la classe più specifica, che eredita dalla superclasse, è detta sottoclasse (SavingAccount). Ereditare da una classe è diverso da realizzare un'interfaccia[2]: la sottoclasse eredita il comportamento e lo stato della propria superclasse. Uno dei vantaggi dell'ereditarietà è il riutilizzo del codice.

Tipo	**Come si realizza**	**A cosa serve**
Ereditarietà	`extends`	Creare sottoclassi
Interfaccia	`implements`	Definiscono comportamenti di una classe

Una sottoclasse non ha accesso ai campi privati della sua superclasse. Una sottoclasse deve usare un metodo pubblico della superclasse stessa.

7.1 Ereditare metodi e campi di esemplare

Definire nuovi metodi:

- Potete sovrascrivere (o ridefinire, "override") metodi della superclasse. Se specificate un metodo con la stessa firma, il metodo prevale su quello avente lo stesso nome e definito nella superclasse, ovvero la sovrascrive. Ogni volta che si applica il metodo a un oggetto della sottoclasse, viene eseguito il metodo ridefinito, invece di quello originale.
- Potete ereditare metodi dalla superclasse. Se non sovrascrivete esplicitamente un metodo della superclasse, lo ereditate automaticamente e lo potete applicare agli oggetti della sottoclasse.
- Potete definire nuovi metodi. Se si definisce un metodo che non esiste nella superclasse, il nuovo metodo si può applicare solo agli oggetti della sottoclasse.[3]

La situazione dei campi di esemplare è piuttosto diversa: non potete sovrascrivere campi di esemplare e, per i campi di una sottoclasse, esistono solo due possibilità:

1. Tutti i campi di esemplare della superclasse sono ereditati automaticamente dalla sottoclasse.
2. Qualsiasi nuovo campo di esemplare che definite nella sottoclasse esiste solo negli oggetti della sottoclasse.

La parola chiave `super` in Java viene utilizzata per fare riferimento alla classe genitore o superclasse di una classe.

[2]L'ereditarietà in Java permette a una classe di ereditare le caratteristiche di un'altra classe, creando una gerarchia di classi. L'interfaccia definisce un contratto di comportamento che una classe deve implementare. Mentre l'ereditarietà permette la condivisione di campi e metodi tra classi, l'interfaccia fornisce solo la definizione dei metodi senza implementazione. L'ereditarietà crea una relazione "è un" tra le classi, mentre l'interfaccia crea una relazione "può fare" tra le classi che la implementano. Entrambi i meccanismi favoriscono la modularità e il riutilizzo del codice.

[3]I campi di istanza (o campi di esemplare) sono variabili definite all'interno di una classe e sono associate ad ogni istanza (oggetto) creata da quella classe. Ogni istanza della classe ha una copia separata dei campi di istanza, che rappresentano lo stato specifico dell'oggetto.

7.2 Controllo di accesso

Java fornisce quattro livelli per il controllo di accesso a variabili, metodi e classi:

- accesso `public`, si può accedere alle caratteristiche pubbliche tramite i metodi di tutte le classi
- accesso `private`, si può accedere alle caratteristiche private solamente mediante i metodi della loro classe
- accesso `protected`
- accesso di pacchetto, se non fornite uno specificatore del controllo di accesso. Tutti i metodi delle classi contenute nello stesso pacchetto possono accedere alla caratteristica in esame. E' una buona impostazione predefinita per le classi, ma è una scelta estremamente infelice per le variabili

I campi di esemplare e le variabili statiche delle classi dovrebbero essere sempre private. Ecco un ristretto numero di eccezioni:

- Le costanti pubbliche (variabili public static final) sono utili e sicure.
- È necessario che alcuni oggetti, come `System.out`, siano accessibili a tutti i programmi, pertanto dovrebbero essere pubblici.
- In rare occasioni alcune classi in un pacchetto devono cooperare molto strettamente: può essere utile fornire ad alcune variabili l'accesso di pacchetto. Le classi interne sono una soluzione migliore.

I metodi dovrebbero essere `public` o `private`.

7.3 Vantaggi dell'ereditarietà

- Evitare la duplicazione di codice
- Permettere il riuso di funzionalità
- Semplificare la costruzione di nuove classi
- Facilitare la manutenzione
- Garantire la consistenza delle interfacce

8 Interfaccia

I tipi interfaccia vengono utilizzati per rendere il codice maggiormente riutilizzabile. Una classe può realizzare più di una interfaccia la classe deve definire tutti i metodi richiesti da tutte le interfacce che realizza. Per convertire un riferimento a interfaccia in un riferimento a classe serve un `cast`.

8.1 Interfacce in Java

Nel linguaggio di programmazione Java, un'interfaccia è un tipo di riferimento, simile a una classe, che può contenere solo costanti, firme di metodi, metodi predefiniti, metodi statici e tipi nidificati. I corpi dei metodi esistono solo per metodi predefiniti e metodi statici. Le interfacce non possono essere istanziate: possono essere implementate solo da classi o estese da altre interfacce.

La definizione di un'interfaccia è simile alla creazione di una nuova classe:

```
public class BankAccount implements Measurable
{
public double getMeasure()
{
return balance;
}
// altri metodi e campi
}
```

Si noti che le firme del metodo non hanno parentesi graffe e terminano con un punto e virgola.

Per usare un'interfaccia, scrivi una classe che implementa l'interfaccia. Quando una classe istanziabile implementa un'interfaccia, fornisce un corpo del metodo per ciascuno dei metodi dichiarati nell'interfaccia. Per esempio:

```
public class OperateBMW760i implements OperateCar {

    // the OperateCar method signatures, with implementation --
    // for example:
    public int signalTurn(Direction direction, boolean signalOn) {
       // code to turn BMW's LEFT turn indicator lights on
       // code to turn BMW's LEFT turn indicator lights off
       // code to turn BMW's RIGHT turn indicator lights on
       // code to turn BMW's RIGHT turn indicator lights off
    }

    // other members, as needed -- for example, helper classes not
    // visible to clients of the interface
}
```

Nell'esempio dell'auto robotica sopra, sono le case automobilistiche che implementeranno l'interfaccia. L'implementazione di Chevrolet sarà sostanzialmente diversa da quella di Toyota, ovviamente, ma entrambi i produttori aderiranno alla stessa interfaccia. I produttori di sistemi di guida, che sono i clienti dell'interfaccia, costruiranno sistemi che utilizzano i dati GPS sulla posizione di un'auto, mappe stradali digitali e dati sul traffico per guidare l'auto. Così facendo, i sistemi di guida invocheranno i metodi di interfaccia: svolta, cambio di corsia, frenata, accelerazione e così via.

8.2 Differenze fra tipi numerici e tipi di classe

- Quando convertite tipi numerici, c'è una potenziale perdita di informazioni, e usate il cast per dire al compilatore che ne siete al corrente.
- Quando convertite tipi di oggetto, affrontate il rischio di provocare il lancio di un'eccezione, e dite al compilatore che siete disposti a correre questo rischio.

8.3 Esempio di cast di un oggetto

```
Coin maxCoin = (Coin) max;
String name = maxCoin.getName();
```

9 Polimorfismo

Il principio secondo cui il tipo effettivo di un oggetto determina il metodo da chiamare è detto polimorfismo. Il polimorfismo è un principio secondo cui il comportamento di un programma può variare in relazione al tipo effettivo di un oggetto. [4] La **selezione posticipata** (late binding) si ha quando la scelta del metodo avviene al momento dell'esecuzione del programma.

Polimorfismo è la capacità di comportarsi in modo diverso con oggetti di tipo diverso. Come funziona il polimorfismo:

```
public void transfer(double amount, BankAccount other)
{
withdraw(amount);
other.deposit(amount);
}
```

9.1 Classi interne

Si può definire interna la classe contenuta in un'altra, ma al di fuori dei metodi di quest'ultima: in questo modo la classe interna sarà visibile a tutti i metodi della classe che la contiene.

```
\\ Dopo la compilazione

public class Tester
{
public static void main(String[] args)
{
class RectangleMeasurer implements Measurer
{
. . .
}
. . .
}
}
>> DataSetTester$1$RectangleMeasurer.class
```

9.2 Overloading

Il linguaggio di programmazione Java supporta i metodi di sovraccarico e Java può distinguere tra metodi con diverse firme di metodo . Ciò significa che i metodi all'interno di una classe possono avere lo stesso nome se hanno elenchi di parametri diversi (ci sono alcune precisazioni a questo che verranno discusse nella lezione intitolata "Interfacce ed ereditarietà").

Supponiamo di avere una classe in grado di utilizzare la calligrafia per disegnare vari tipi di dati (stringhe, numeri interi e così via) e che contenga un metodo per disegnare ciascun tipo di dati. È complicato utilizzare un nuovo nome per ciascun metodo, ad esempio , drawString, drawIntegere drawFloatcosì via. Nel linguaggio di programmazione Java è possibile utilizzare lo stesso nome per tutti i metodi di disegno ma passare a ciascun metodo un elenco di argomenti diverso. Pertanto, la classe di disegno dei dati potrebbe dichiarare quattro metodi denominati draw, ognuno dei quali ha un elenco di parametri diverso.

```
public class DataArtist {
  // ...
public void draw(String s) {
  // ...
}
public void draw(int i) {
  // ...
}
public void pareggio(double f) {
```

[4] Esiste una differenza importante fra polimorfismo e sovraccarico. Il compilatore sceglie un metodo sovraccarico quando traduce il programma, prima che il programma venga eseguito.Questa selezione del metodo é detta **selezione anticipata** (early binding).

```
    // ...
  }
  public void draw(int i, double f) {
    // ...
  }
}
```

I metodi sottoposti a overload sono differenziati dal numero e dal tipo di argomenti passati al metodo. Nell'esempio di codice, draw(String s)e draw(int i)sono metodi distinti e univoci perché richiedono tipi di argomenti diversi.

Non puoi dichiarare più di un metodo con lo stesso nome e lo stesso numero e tipo di argomenti, perché il compilatore non può distinguerli.

Il compilatore non considera il tipo restituito durante la differenziazione dei metodi, pertanto non è possibile dichiarare due metodi con la stessa firma anche se hanno un tipo restituito diverso.

10 Modificatori di visibilità

- **public:** Indica che il membro è accessibile da qualsiasi parte del codice, sia all'interno della stessa classe, che da altre classi o pacchetti.
- **protected:** Indica che il membro è accessibile all'interno della stessa classe, da classi derivate (sotto-classi) nello stesso pacchetto e da classi derivate in un pacchetto diverso.
- **private:** Indica che il membro è accessibile solo all'interno della stessa classe. Non è visibile dalle classi esterne, nemmeno se fanno parte dello stesso pacchetto.
- **Nessun modificatore** (conosciuto anche come default o "package-private"): Indica che il membro è accessibile all'interno della stessa classe e da tutte le classi nello stesso pacchetto. Non è accessibile da classi esterne al pacchetto.
- **Final**:
  - Per le variabili, final indica che il valore della variabile non può essere modificato una volta assegnato. Ad esempio, final int MAX_NUM = 100; dichiara una costante MAX_NUM con il valore 100 che non può essere modificato successivamente.
  - Per le classi, final indica che la classe non può essere estesa da altre classi. Ad esempio, final class MyClass ... dichiara una classe MyClass che non può essere estesa da altre classi.
  - Per i metodi, final indica che il metodo non può essere sovrascritto da classi derivate. Ad esempio, public final void print() ... dichiara un metodo print() che non può essere sovrascritto dalle classi derivate.
- **Static**:
  - Se usato su un metodo allora sarà accessibile senza dover istanziare un oggetto di tale classe
  - Se usato su un attributo sarà condiviso tra tutte le istanze di tale classe

11 Principi Solid

- **Single Responsibility Principle (Principio di Singola Responsabilità)**: Una classe dovrebbe avere una sola ragione per cambiare. In altre parole, una classe dovrebbe essere responsabile di una singola funzionalità o aspetto del sistema. Ciò favorisce la coesione e riduce la dipendenza tra le classi.

  less Copy code

- **Open-Closed Principle (Principio di Estensione/Chiusura)**: Le entità del software (classi, moduli, metodi, ecc.) dovrebbero essere aperte all'estensione, ma chiuse alla modifica. Il comportamento di una classe dovrebbe essere estendibile senza modificare il codice esistente. Questo principio favorisce il riuso del codice e la stabilità del sistema.

- **Liskov Substitution Principle (Principio di Sostituzione di Liskov)**: Gli oggetti di una classe derivata devono essere sostituibili senza influire sulla correttezza del programma. Una classe derivata deve rispettare il contratto (interfaccia, comportamento, precondizioni e postcondizioni) della classe base. Questo principio favorisce l'interoperabilità e l'estendibilità delle classi.

- **Interface Segregation Principle (Principio di Separazione delle Interfacce)**: I clienti non dovrebbero essere costretti a dipendere da interfacce che non utilizzano. Le interfacce dovrebbero essere specifiche per i clienti e non dovrebbero contenere metodi non necessari. Ciò favorisce la modularità e la manutenibilità del codice.

- **Dependency Inversion Principle (Principio di Inversione delle Dipendenze)**: I moduli di alto livello non dovrebbero dipendere direttamente dai moduli di basso livello. Entrambi i tipi di moduli dovrebbero dipendere da astrazioni (interfacce o classi astratte). Ciò favorisce la riduzione delle dipendenze e l'inversione del controllo.

12 Eccezioni in Java

In Java le eccezioni ricadono entro due categorie:

- Controllate
  - Il compilatore verifica che l'eccezione non venga ignorata.
  - Le eccezioni controllate sono dovute a circostanze esterne che il programmatore non può evitare
  - La maggior parte delle eccezioni controllate vengono utilizzate nella gestione dei dati in ingresso o in uscita, che è un fertile terreno per guasti esterni che non sono sotto il vostro controllo
  - Tutte le sottoclassi di IOException sono eccezioni controllate
- Non controllate
  - Le eccezioni non controllate estendono la classe RuntimeException o Error
  - Le eccezioni non controllate rappresentano un vostro errore
  - Le sottoclassi di RuntimeException sono non controllate. Esempio:

    ```
    NumberFormatException
    IllegalArgumentException
    NullPointerException
    ```

  - Esempio di errore: OutOfMemoryError

Un gestore di eccezioni si installa con l'enunciato `try/catch`. Ciascun blocco try contiene una o più invocazioni di metodi che possono provocare il lancio di un'eccezione. Ciascun blocco catch (IOException exception) contiene codice che viene eseguito quando viene lanciata un'eccezione di tipo IOException

12.0.1 Esempio di cattura eccezione

```
try
{
  String filename = . . .;
  FileReader reader = new FileReader(filename);
  Scanner in = new Scanner(reader);
  String input = in.next();
  int value = Integer.parseInt(input);
  . . .
  }
  catch (IOException exception)
  {
    exception.printStackTrace();
    }
    catch (NumberFormatException exception)
    {
      System.out.println("Input was not a number");
      }
```

Per avere la che un blocco venga eseguito sempre bisogna aggiungere la clausola `finally`[5]. Il codice della clausola finally viene eseguito al termine del blocco try, in una delle seguenti situazioni:

- Dopo aver portato a termine l'ultimo enunciato del blocco try
- Dopo aver portato a termine l'ultimo enunciato di una clausola catch che abbia catturato un'eccezione lanciata nel blocco try
- Quando nel blocco try è stata lanciata un'eccezione che non viene catturata

[5]Non usare clausole catch e finally nel medesimo blocco try

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class FileExample {

    public static void main(String[] args) {
        try {
            // Lettura di un file
            BufferedReader reader = new BufferedReader(new FileReader("input.txt"));
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
            reader.close();

            // Scrittura su un file
            FileWriter writer = new FileWriter("output.txt");
            writer.write("Questo è un esempio di scrittura su file.");
            writer.close();
        } catch (IOException e) {
            // Gestione dell'eccezione
            System.err.println("Si è verificato un errore durante la lettura
            o la scrittura del file: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Nell'esempio sopra, abbiamo utilizzato le classi BufferedReader per la lettura di file e FileWriter per la scrittura su file. All'interno del blocco try, leggiamo il file linea per linea e stampiamo ogni riga a schermo. Successivamente, scriviamo una stringa su un file utilizzando FileWriter.

Se si verifica un'eccezione durante la lettura o la scrittura del file, viene catturata l'eccezione di tipo IOException nel blocco catch. In questo caso, viene stampato un messaggio di errore e viene tracciata la pila delle chiamate per ottenere informazioni aggiuntive sull'eccezione.

È importante gestire correttamente le eccezioni durante la lettura e la scrittura di file per garantire un'elaborazione affidabile degli errori e una corretta chiusura delle risorse.

13 Metodi utili Classe Scanner e PrintWriter

La classe Scanner e la classe PrintWriter sono utili per la lettura e la scrittura di dati da e verso file o altri flussi di dati in Java. Per leggere dati da un file presente sul disco costruite un oggetto di tipo FileReader. Poi usate FileReader per costruire un oggetto Scanner.

```
FileReader reader = new FileReader("input.txt");
Scanner in = new Scanner(reader);
```

Ecco alcuni metodi comuni di queste classi che possono essere utili:

13.1 Classe Scanner

- Scanner(File source): Crea un oggetto Scanner per leggere dati da un file specificato.
- Scanner(InputStream source): Crea un oggetto Scanner per leggere dati da un'origine di input specificata, ad esempio System.in per l'input da tastiera.
- hasNext(): Restituisce true se c'è ancora un altro elemento nel flusso di input.
- next(): Restituisce la prossima stringa dal flusso di input.
- nextInt(): Restituisce il prossimo intero dal flusso di input.
- nextDouble(): Restituisce il prossimo valore di tipo double dal flusso di input.
- close(): Chiude il Scanner e rilascia le risorse associate.

Ecco un esempio di utilizzo della classe Scanner per leggere interi da un file:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class ScannerExample {
    public static void main(String[] args) {
        String filePath = "path/to/your/file.txt";

        try {
            File file = new File(filePath);
            Scanner scanner = new Scanner(file);

            while (scanner.hasNext()) {
                int number = scanner.nextInt();
                System.out.println("Numero letto: " + number);
            }

            scanner.close();
        } catch (FileNotFoundException e) {
            System.out.println("File non trovato.");
            e.printStackTrace();
        }
    }
}
```

13.2 Classe PrintWriter

- PrintWriter(File file): Crea un oggetto PrintWriter per scrivere dati su un file specificato.
- PrintWriter(OutputStream out): Crea un oggetto PrintWriter per scrivere dati su un flusso di output specificato, ad esempio System.out per l'output a schermo.
- print(String s): Scrive una stringa sul flusso di output.

- println(String s): Scrive una stringa sul flusso di output, seguita da un carattere di nuova riga.
- print(int i): Scrive un intero sul flusso di output.
- println(int i): Scrive un intero sul flusso di output, seguito da un carattere di nuova riga.
- close(): Chiude il PrintWriter e rilascia le risorse associate.

Ecco un esempio di utilizzo della classe PrintWriter per scrivere dati su un file:

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class PrintWriterExample {
    public static void main(String[] args) {
        String filePath = "path/to/your/file.txt";

        try {
            FileWriter fileWriter = new FileWriter(filePath);
            PrintWriter printWriter = new PrintWriter(fileWriter);

            printWriter.println("Prima riga");
            printWriter.println("Seconda riga");
            printWriter.print("Terza riga");

            printWriter.close();
        } catch (IOException e) {
            System.out.println("Si è verificato un errore durante la scrittura del file.");
            e.printStackTrace();
        }
    }
}
```

Bisogna sempre chiudere i file quando avete terminato di usarli; altrimenti può darsi che non tutti i dati siano stati realmente scritti nel file su disco `out.close()`;

Quando un file usato per leggere dati non esiste, è possibile che si verifichi l'eccezione `FileNotFoundExeption`. Per gestire l'eccezione, etichettate il metodo stesso in questo modo:

```
public static void main(String[] args)) throws FileNotFoundException
```

13.2.1 Esempio: IllegalArgumentException

Per segnalare una condizione eccezionale, usate l'enunciato throw per lanciare un oggetto eccezione.

```
IllegalArgumentException exception = new IllegalArgumentException("Amount exceeds balance");
  throw exception;
```

Non c'è bisogno di memorizzare l'oggetto eccezione in una variabile

```
throw new IllegalArgumentException("Amount exceeds balance");
```

14 Java Collections

Problema: raggruppare un insieme di oggetti insieme e accedere ad essi secondo regole particolari (per esempio una Coda). Spesso l'utilizzo degli array non è sufficiente. [6]

- Soluzione 1: Realizzare una propria classe che, utilizzando internamente gli array, fornisce i metodi di accesso opportuni
- Soluzione 2: Utilizzare classi già pronte fornite da Java, scegliendo quella più opportuna ai propri bisogni
- Collection
  - List
    - ArrayList
    - LinkedList
    - Vector
  - Set
    - SortedSet
      - TreeSet
    - HashSet
    - LinkedHashSet
- Altre interfacce disponibili
  - Queue,Dequeue,Stack, Map, SortedMap ...

Table 1 The Methods of the `Collection` Interface

`Collection<String> coll = new ArrayList<>();`	The `ArrayList` class implements the `Collection` interface.
`coll = new TreeSet<>();`	The `TreeSet` class (Section 15.3) also implements the `Collection` interface.
`int n = coll.size();`	Gets the size of the collection. `n` is now `0`.
`coll.add("Harry");` `coll.add("Sally");`	Adds elements to the collection.
`String s = coll.toString();`	Returns a string with all elements in the collection. `s` is now `[Harry, Sally]`.
`System.out.println(coll);`	Invokes the `toString` method and prints `[Harry, Sally]`.
`coll.remove("Harry");` `boolean b = coll.remove("Tom");`	Removes an element from the collection, returning `false` if the element is not present. `b` is false.
`b = coll.contains("Sally");`	Checks whether this collection contains a given element. `b` is now `true`.

`for (String s : coll)`
`{`
`  System.out.println(s);`
`}`
	You can use the "for each" loop with any collection. This loop prints the elements on separate lines.

`Iterator<String> iter = coll.iterator();`	You use an iterator for visiting the elements in the collection (see Section 15.2.3).

Figura 4

[6] Tutti gli oggetti in java estendono da Object. E' corretto scrivere: `Object o = new Integer(10)`. Gli oggetti vengono ritornati come Object e **non** del loro tipo specifico. Per ottenere il tipo originario è necessario il cast esplicito `Integer i=(Integer) a.get(0);//cast esplicito`

14.1 Possibili problemi con le collection

- Necessità di ricorrere al cast degli elementi anche quando il tipo di elementi è noto
- Possibili cast degli elementi a tipi non corretti
- Nessun controllo sui tipi di dati inseriti all'interno di un vettore
- Poca chiarezza sul tipo di dati trattati[7]

14.2 L'interfaccia `set`

Un `set` è una raccolta non ordinata di elementi unici. Dispone i suoi elementi in modo da trovare, aggiungere e rimuovere elementi è più efficiente. Due meccanismi per farlo:

- Tabelle hash
- Alberi binari di ricerca

Un set organizza i suoi valori in un ordine ottimizzato per efficienza. Potrebbe non essere l'ordine in cui aggiungi gli elementi. L'inserimento e la rimozione di elementi è più efficiente con un set rispetto a con un elenco. L'interfaccia Set ha gli stessi metodi della Collection interfaccia. Un set non ammette duplicati. Due classi di implementazione:

- HashSet basato sulla tabella hash
- TreeSet basato sull'albero di ricerca binario

Un'implementazione Set dispone gli elementi in modo che possano essere individuati rapidamente.

14.2.1 List<>

Una raccolta ordinata List (nota anche come sequenza). L'utente di questa interfaccia ha un controllo preciso su dove viene inserito ogni elemento nell'elenco. L'utente può accedere agli elementi in base al loro indice intero (posizione nell'elenco) e cercare gli elementi nell'elenco. A differenza degli insiemi, gli elenchi in genere consentono elementi duplicati. Più formalmente, gli elenchi in genere consentono coppie di elementi e1 ed e2 tali che e1.equals(e2) e in genere consentono più elementi null se consentono elementi null. Non è inconcepibile che qualcuno possa voler implementare un elenco che proibisca i duplicati, generando eccezioni di runtime quando l'utente tenta di inserirle, ma ci aspettiamo che questo utilizzo sia raro.

L' interfaccia List inserisce clausole aggiuntive, oltre a quelle specificate nell'interfaccia Collection , sui contratti dei metodi iterator , add , remove , equals e hashCode . Dichiarazioni per altri metodi ereditati sono incluse anche qui per comodità.

14.2.2 LinkedList

Implementazione dell'elenco doppiamente collegato delle interfacce Liste Deque. Implementa tutte le operazioni di elenco facoltative e consente tutti gli elementi (incluso null). Tutte le operazioni si comportano come ci si potrebbe aspettare da un elenco doppiamente collegato. Le operazioni che indicizzano nell'elenco attraverseranno l'elenco dall'inizio o dalla fine, qualunque sia più vicino all'indice specificato.

Si noti che questa implementazione non è sincronizzata. Se più thread accedono contemporaneamente a un elenco collegato e almeno uno dei thread modifica strutturalmente l'elenco, è necessario sincronizzarlo esternamente. (Una modifica strutturale è qualsiasi operazione che aggiunge o elimina uno o più elementi; la semplice impostazione del valore di un elemento non è una modifica strutturale.) Ciò viene in genere eseguito sincronizzando su un oggetto che incapsula naturalmente l'elenco.

[7] In tutti i casi, il codice risulta sintatticamente corretto e non viene segnalato alcun errore dal compilatore. L'errore viene scoperto solo a Runtime!

14.2.3 Vector

In Java, la classe Vector fa parte del pacchetto java.util ed è una struttura dati che rappresenta un array dinamico, cioè un insieme di elementi che può espandersi o restringersi dinamicamente a seconda delle necessità. La classe Vector è stata introdotta in Java molto tempo fa e, sebbene sia ancora supportata per motivi di retrocompatibilità, si consiglia di utilizzare l'implementazione più moderna dell'interfaccia List, come ArrayList, in scenari di sviluppo più recenti.

```
  import java.util.Vector;

  public class VectorExample {
    public static void main(String[] args) {
        // Create a Vector
        Vector<String> vector = new Vector<>();

        // Add elements to the Vector
        vector.add("Apple");
        vector.add("Banana");
        vector.add("Orange");

        // Get the size of the Vector
        int size = vector.size();
        System.out.println("Size: " + size);

        // Access elements in the Vector
        String firstElement = vector.get(0);
        System.out.println("First Element: " + firstElement);

        // Remove an element from the Vector
        vector.remove(1);

        // Iterate over the Vector
        for (String element : vector) {
            System.out.println(element);
        }
    }
}
```

14.2.4 Tabella hash

Gli elementi del set sono raggruppati in raccolte più piccole di elementi che condividono la stessa caratteristica. Gli elementi in una tabella hash devono implementare il metodo hashCode. Puoi formare set di hash contenenti oggetti di tipo String, Integer, Double, Point, Rectangle o Color. HashSet¡String¿ , HashSet¡Rectangle¿ o a HashSet¡HashSet¡Intero¿¿ In una tabella hash, gli oggetti con lo stesso codice hash sono inseriti nello stesso gruppo.

14.2.5 TreeSet

Gli elementi sono tenuti in ordine ordinato Gli elementi sono memorizzati nei nodi. I nodi sono disposti a forma di albero, non in una sequenza lineare È possibile formare insiemi di alberi per qualsiasi classe che implementa il Interfaccia comparabile: Esempio: String o Integer.

14.2.6 Map

Mantiene le associazioni tra oggetti chiave e valore. Ogni chiave nella mappa ha un valore associato. La mappa memorizza le chiavi, i valori e le associazioni tra di essi. A volte si vuole enumerare tutte le chiavi in una mappa. Il metodo keySet restituisce il set di chiavi. Modo "universale" per scorrere collezioni di elementi, Chiedi al set di chiavi un iteratore e ottieni tutte le chiavi. Per ogni chiave, puoi trovare il valore associato con il get metodo. Per stampare tutte le coppie chiave/valore in una mappa m:

```
Set<String> keySet = m.keySet();
for (String key : keySet)
{
Color value = m.get(key);
System.out.println(key + "->" + value);
}
```

14.3 Stack

- Ricorda l'ordine degli elementi
- Si possono aggiungere e rimuovere elementi solo da sopra

```
Stack<String> s = new Stack<>();
s.push("A"); s.push("B"); s.push("C");
while (s.size() > 0)
{
System.out.print(s.pop() + " "); // Prints C B A
}
```

14.3.1 Queue

L'interfaccia Queue nella libreria Java standard ha:

- un metodo `add` per aggiungere un elemento alla coda della coda
- un metodo `remove` per rimuovere l'inizio della coda, e
- un metodo `peek` per ottenere l'elemento head della coda senza rimuoverlo.

La classe LinkedList implementa l'interfaccia Queue. Quando hai bisogno di una coda, inizializza una variabile Queue con a Oggetto LinkedList:

14.3.2 Linked Lists: Queue

- Una struttura dati utilizzata per raccogliere una sequenza di oggetti
- Consente un'efficiente aggiunta e rimozione di elementi nel mezzo del sequenza.
- Un elenco collegato è costituito da un numero di nodi;
- Ogni nodo ha un riferimento al nodo successivo.
- Un nodo è un oggetto che memorizza un elemento e fa riferimento a esso nodi vicini.
- Ogni nodo in un elenco collegato è connesso ai nodi vicini.

L'**aggiunta e la rimozione** di elementi nel mezzo di un elenco collegato è **efficiente**. **Visitare** gli elementi di un elenco collegato in ordine sequenziale è **efficiente**. L'**accesso casuale** è **inefficiente**. Modo "universale" per scorrere collezioni di elementi, indipendentemente dalla particolare disposizione degli elementi.

```
\\ Iterator
iterator.add("qualcosa");
```

Table 2 Working with Linked Lists

`LinkedList<String> list = new LinkedList<>();`	An empty list.
`list.addLast("Harry");`	Adds an element to the end of the list. Same as `add`.
`list.addFirst("Sally");`	Adds an element to the beginning of the list. `list` is now `[Sally, Harry]`.
`list.getFirst();`	Gets the element stored at the beginning of the list; here `"Sally"`.
`list.getLast();`	Gets the element stored at the end of the list; here `"Harry"`.
`String removed = list.removeFirst();`	Removes the first element of the list and returns it. `removed` is `"Sally"` and `list` is `[Harry]`. Use `removeLast` to remove the last element.
`ListIterator<String> iter = list.listIterator()`	Provides an iterator for visiting all list elements (see Table 3 on page 684).

Figura 5

15 Strutture dati

15.1 HashMap

HashMap è una struttura dati basata su una tabella hash che permette l'accesso, l'inserimento e la rimozione degli elementi in tempo costante (O(1)) in media. Utilizza una funzione hash per mappare le chiavi agli elementi corrispondenti. È una scelta comune quando la velocità di accesso è prioritaria rispetto all'ordinamento. Alcuni metodi utili di HashMap includono:

- put(key, value): Inserisce una coppia chiave-valore nella mappa.
- get(key): Restituisce il valore associato alla chiave specificata.
- remove(key): Rimuove l'elemento associato alla chiave specificata.
- containsKey(key): Verifica se la chiave è presente nella mappa.
- keySet(): Restituisce un set di tutte le chiavi presenti nella mappa.

Ecco un esempio di utilizzo di HashMap:

```
  import java.util.HashMap;

  public class HashMapExample {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();

        // Inserimento di elementi nella mappa
        map.put("Alice", 25);
        map.put("Bob", 30);
        map.put("Charlie", 35);

        // Accesso ai valori
        System.out.println(map.get("Bob")); // Output: 30

        // Rimozione di un elemento
        map.remove("Alice");

        // Iterazione su tutte le chiavi
        for (String key : map.keySet()) {
            System.out.println(key + ": " + map.get(key));
        }
    }
}
```

15.2 TreeMap

TreeMap è una struttura dati basata su un albero di ricerca bilanciato (di solito un albero rosso-nero). Mantiene gli elementi ordinati in base alle chiavi. L'accesso, l'inserimento e la rimozione richiedono un tempo di esecuzione logaritmico (O(log n)). È una scelta appropriata quando si necessita di un ordinamento naturale degli elementi. Oltre ai metodi di HashMap, TreeMap fornisce anche:

- firstKey(): Restituisce la prima (più piccola) chiave nell'ordine.
- lastKey(): Restituisce l'ultima (più grande) chiave nell'ordine.
- higherKey(key): Restituisce la chiave successiva alla chiave specificata.
- lowerKey(key): Restituisce la chiave precedente alla chiave specificata.
- keySet(): Restituisce un set di tutte le chiavi presenti nella mappa nell'ordine.

Ecco un esempio di utilizzo di TreeMap:

```
import java.util.TreeMap;

public class TreeMapExample {
    public static void main(String[] args) {
        TreeMap<String, Integer> map = new TreeMap<>();

        // Inserimento di elementi nella mappa
        map.put("Alice", 25);
        map.put("Bob", 30);
        map.put("Charlie", 35);

        // Accesso ai valori
        System.out.println(map.get("Bob")); // Output: 30

        // Rimozione di un elemento
        map.remove("Alice");

        // Iterazione su tutte le chiavi
        for (String key : map.keySet()) {
            System.out.println(key + ": " + map.get(key));
        }
    }
}
```

16 Generic Classes

Programmazione generica è la creazione di costrutti di programmazione che può essere utilizzato con molti tipi diversi.

Generic class ha uno o più parametri di tipo. Non si può usare come sostituto uno degli otto tipi di dati primitivi

16.1 Implementing Generic Types

- E Element type in a collection
- K Key type in a map
- V Value type in a map
- T General type
- S, U Additional general types

16.2 Il tipo generico eredita da una superclasse

```
/**
 A class for executing binary searches through an array.
*/
public class BinarySearcher<T extends Comparable>
{
 private T[] a;

 /**
    Constructs a BinarySearcher.
    @param anArray a sorted array
 */
 public BinarySearcher(T[] anArray)
 {
    a = anArray;
 }

 /**
    Finds a value in a sorted array, using the binary
    search algorithm.
    @param v the value to search
    @return the index at which the value occurs, or -1
          if it does not occur in the array
 */
 public int search(T v)
 {
    int low = 0;
    int high = a.length - 1;
    while (low <= high)
    {
       int mid = (low + high) / 2;
       int diff = a[mid].compareTo(v);

       if (diff == 0) // a[mid] == v
          return mid;
       else if (diff < 0) // a[mid] < v
          low = mid + 1;
       else
          high = mid - 1;
    }
    return -1;
 }
}
```

17 Domande teoria

Quali sono i vantaggi e le motivazioni dietro all'utilizzo dei metodi astratti in Java? Fornire un esempio d'utilizzo.

Un'interfaccia è simile a una classe, ma ci sono parecchie differenze una di queste sono proprio i metodi astratti: Tutti i metodi di un'interfaccia sono astratti, cioè hanno un nome, un elenco di parametri,un tipo di valore restituito, ma non hanno un'implementazione.

Qualè la differenza tra overloading e overriding? con esempio scritto in Java

Overloading e Overriding sono concetti legati al polimorfismo il quale, nella programmazione ad oggetti, si riferisce alla capacità di una classe A di assumere i valori di un qualunque tipo descritto dalla classe B, con B sottoclasse di A. Si parla di overloading quando sono presenti più metodi all'interno di una classe o di una sua superclasse con lo stesso nome, tipo di ritorno ma con tipo e/o numero di parametri diverso. Si parla invece di overriding quando un metodo della classe derivata B sovrascrive il metodo della classe A mantenendo quindi la stessa firma ma cambiando il corpo della funzione.

Spiegare le principali differenze tra ArrayList e Set in Java.

ArrayList implementa l'interfaccia List appartenente a sua volta all'interfaccia Collection di Java. Un ArrayList è una collezione di oggetti memorizzati sequenzialmente secondo una politica FIFO. Esso può contenere valori il cui tipo può essere primitivo o meno, ammettendo duplicati. La dimensione dell'ArrayList è dinamica e l'accesso ai dati è randomico (posizionale). I Set, appartenenti anch'essi all'interfaccia Collection, rappresentano un insieme di valori senza duplicati. Tale interfaccia Set è a sua volta implementata da HashSet e TreeSet: la prima utilizza una tabella hash, ofrendo quindi un tempo di esecuzione costante per le operazioni di add, remove, size, contains. TreeSet, al contrario, utilizza un albero interno per le operazioni di memorizzazione offrendo quindi la possibilità di avere un ordinamento. HashSet permette la presenza di un valore nullo, mentre TreeSet no.

Eccezioni controllate e non controllate

In Java le eccezioni ricadono entro due categorie:

- Controllate
  - Il compilatore verifica che l'eccezione non venga ignorata.
  - Le eccezioni controllate sono dovute a circostanze esterne che il programmatore non può evitare
  - La maggior parte delle eccezioni controllate vengono utilizzate nella gestione dei dati in ingresso o in uscita, che è un fertile terreno per guasti esterni che non sono sotto il vostro controllo
  - Tutte le sottoclassi di IOException sono eccezioni controllate
- Non controllate
  - Le eccezioni non controllate estendono la classe RuntimeException o Error
  - Le eccezioni non controllate rappresentano un vostro errore
  - Le sottoclassi di RuntimeException sono non controllate. Esempio:

    ```
    NumberFormatException
    IllegalArgumentException
    NullPointerException
    ```

  - Esempio di errore: OutOfMemoryError

Conversioni di tipo

Il casting del tipo si verifica quando si converte un valore di un tipo primitivo a un altro tipo. In Java, ci sono due tipi di casting:

- Automatico: conversione di un carattere più piccolo in uno più grande.

```
int myInt = 9;
double myDouble = myInt;
```

- Manuale: conversione di un tipo più grande in un tipo più piccolo.

```
double myDouble = 9.78d;
int myInt = (int) myDouble;
```

Incapsulamento

L'incapsulamento consiste nella visibilità di uno stato di un dato oggetto che non viene liberamente incapsulato, in modo che sia raggiungibile e modificabile solo secondo alcune regole In Java si nota attraverso l'esistenza di gerarchie di accessibilità di metodi e campi e anche attraverso il concetto di proprietà.

Heap & Stack

Lo **stack** è l'area di memoria dove sono memorizzate lo stato di esecuzione del programma, gli alias delle variabili locali di ogni metodo, il loro indirizzo di memoria e il loro valore. Nota. Lo stack ha una dimensione limitata di pochi megabyte. Se viene superata si genera un errore di StackOverflowError. E' una pila in modalità LIFO (Last Input First Output). Contiene i record di attivazione, ossia le informazioni necessarie per invocare i metodi. Ogni record di attiazione contiene i parametri, l'oggetto invocato (this), le variabili locali, il valore di ritorno, il punto di ritorno dell'invocazione di un metodo, l'istruzione successiva alla chiamata.
L'**heap** è l'area di memoria dove sono memorizzati gli oggetti e il loro stato. E' un'area di memoria dinamica allocata al programma. Può essere modificata dalla macchina virtuale. Qui sono registrati gli oggetti creati con l'operatore new. Esempio. Al momento della creazione di un oggetto viene allocata una parte della memoria, restituito il riferimento dell'oggetto e invocato il costruttore della classe per inizializzare l'area di memoria.

Metodi Statici

I metodi static se usati sul metodo è accessibile sena dover instanziare l'oggetto (es. Math.sqrt(n)). Se usato su un attributo, questo sarà condiviso tra tutte le istanze della classe.
Dalla docs: un metodo statico è un metodo associato alla classe in cui è definito piuttosto che a qualsiasi oggetto. Ogni istanza della classe condivide i suoi metodi statici.

Strutture di controllo

- Strutture selettive
  - if
  - switch: l'istruzione switch può avere un numero di possibili percorsi di esecuzione. Uno switch funziona con i tipi di dati primitivi byte, short, char e int. Funziona anche con i tipi enumerati (discussi in Enum Types), la classe String e alcune classi speciali che racchiudono determinati tipi primitivi: Character, Byte, Short e Integer

```
public class SwitchDemo {
public static void main(String[] args) {

  int month = 8;
  String monthString;
  switch (month) {
      case 1:  monthString = "January";
               break;
      case 2:  monthString = "February";
               break;
      case 3:  monthString = "March";
               break;
      case 4:  monthString = "April";
               break;
      case 5:  monthString = "May";
               break;
      case 6:  monthString = "June";
               break;
      case 7:  monthString = "July";
               break;
      case 8:  monthString = "August";
               break;
      case 9:  monthString = "September";
               break;
      case 10: monthString = "October";
               break;
      case 11: monthString = "November";
               break;
      case 12: monthString = "December";
               break;
      default: monthString = "Invalid month";
               break;
  }
  System.out.println(monthString);
    }
}
```

- Strutture iterative
  - while
  - do ... while
  - for
  - For each (for generalizzato)

Enumerazioni

Un tipo enum è un tipo di dati speciale che consente a una variabile di essere un insieme di costanti predefinite. La variabile deve essere uguale a uno dei valori predefiniti per essa. Esempi comuni includono le direzioni della bussola (valori di NORD, SUD, EST e OVEST) ei giorni della settimana.
Poiché sono costanti, i nomi dei campi di un tipo enum sono in lettere maiuscole.
Nel linguaggio di programmazione Java, si definisce un tipo enum utilizzando la enumparola chiave. Ad esempio, specificare un tipo di enumerazione dei giorni della settimana come:

```
public enum Giorno {
DOMENICA LUNEDÌ MARTEDÌ MERCOLEDÌ,
GIOVEDÌ, VENERDÌ, SABATO};
```

È necessario utilizzare i tipi enum ogni volta che è necessario rappresentare un insieme fisso di costanti.

```
/** EnumTest.java*/
public class EnumTest {
  public enum Giorno { LUNEDI, MARTEDI, MERCOLEDI, GIOVEDI, VENERDI, SABATO, DOMENICA };
  public static void main(String[] args) {
    // scegliamo un valore
    Giorno giornoDellaSettimana = Giorno.GIOVEDI;
    // definiamo una logica
    switch(giornoDellaSettimana){
      case LUNEDI:
      System.out.println("Oggi è Lunedì");
      break;
      case MARTEDI:
      System.out.println("Oggi è Martedì");
      break;
      case MERCOLEDI:
      System.out.println("Oggi è Mercoledì");
      break;
      case GIOVEDI:
      System.out.println("Oggi è Giovedì");
      break;
      case VENERDI:
      System.out.println("Oggi è Venerdì");
      break;
      case SABATO:
      System.out.println("Oggi è Sabato");
      break;
      case DOMENICA:
      System.out.println("Oggi è Domenica");
      break;
      }
      }
      }
```

Ereditarietà

L'ereditarietà è un metodo per estendere classi esistenti aggiungendovi metodi e campi. Si utilizza `extends` per creare sottoclassi. Una sottoclasse non ha accesso ai campi privati della sua superclasse. Una sottoclasse deve usare un metodo pubblico della superclasse stessa. Si può sovrascrvere un metodo della superclasse con `@Override`. I vantaggi:

- No duplicazione codice
- Riuso di funzionalità
- Facilitare la manutenzione
- Costruzione semplificata di nuove classi

Classi e metodi astratti

Un metodo astratto[a] è un metodo privo di implementazione (definito in una classe astratta), seguito sempre da un punto e virgola, che rappresenta un'operazione generale. Spesso, i metodi astratti hanno una o più implementazioni fornite separatamente, per esempio, sotto forma di sottoclassi concrete, che possono includere metodi astratti o proprietà astratte che sono condivise dai suoi sottotipi. Un metodo non astratto invece, è chiamato concreto (o classe concreta). Una classe che definisce un metodo astratto o che eredita un metodo astratto senza sovrascriverlo deve essere dichiarata `abstract`.

[a] I metodi astratti possono essere chiamati anche esistenziali.

This e super

Il `this` la parola chiave punta a un riferimento della classe corrente, mentre la `super` la parola chiave punta a un riferimento della classe genitore. this può essere utilizzato per accedere a variabili e metodi della classe corrente, e super può essere utilizzato per accedere a variabili e metodi della classe genitore dalla sottoclasse.

Overloading e overriding

Override ha la capacità di una sottoclasse di sovrascrivere un metodo consente a una classe di ereditare da una superclasse il cui comportamento può modificare il comportamento secondo necessità. Il metodo di override ha lo stesso nome, numero e tipo di parametri e tipo restituito del metodo di cui esegue l'override. Il linguaggio di programmazione Java supporta i metodi di `overloading` e si può distinguere tra metodi con diverse firme di metodo. Ciò significa che i metodi all'interno di una classe possono avere lo stesso nome se hanno elenchi di parametri diversi.

Modificatori di visibilita

- Public: visibile da qualsiasi parte del programma
- Private: visibile solo all'interno della stessa classe
- Protected: visibile solo dalle classi dello stesso package e da tutte le sottoclassi
- Default: visibile dallo stesso package e dalle sottoclassi se sono nello stesso package. È la visibilità assegnata di default se non viene specificato nulla.

Polimorfismo

Il polimorfismo si basa sul concetto di ereditarietà, che consente a una classe di ereditare le caratteristiche e il comportamento da un'altra classe. Quando una classe eredita da un'altra, può ridefinire i metodi della classe padre per adattarli alle proprie esigenze.
Il polimorfismo viene sfruttato utilizzando una variabile di tipo più generico per fare riferimento a un oggetto di tipo più specifico. Ad esempio, supponiamo di avere una classe chiamata "Animal" e due classi derivate chiamate "Dog" e "Cat". Entrambe le classi derivate ereditano dalla classe "Animal". Possiamo creare un'istanza di "Dog" o "Cat" e assegnarla a una variabile di tipo "Animal".

Downcasting e upcasting

L'upcasting avviene quando si assegna un'istanza di una classe derivata a una variabile di tipo della classe base. In altre parole, si sta "salendo" nella gerarchia delle classi. Questo tipo di conversione è sicuro e non richiede una sintassi speciale.

```
  class Animal {
   public void makeSound() {
      System.out.println("Generi di suoni animali");
   }
}

class Dog extends Animal {
   public void makeSound() {
      System.out.println("Bau bau!");
   }
}

public class Main {
   public static void main(String[] args) {
      Animal animal = new Dog(); // Upcasting

      animal.makeSound(); // Stampa "Bau bau!"
   }
}
```

Il downcasting avviene quando si converte un'istanza di una classe base in una classe derivata. In altre parole, si sta "scendendo" nella gerarchia delle classi. Tuttavia, il downcasting è un'operazione potenzialmente rischiosa perché si sta cercando di trattare un oggetto di un tipo più generico come un tipo più specifico.

```
  class Animale{

  }
  public class Mucca extends Animale{
      public static void main(String[] args) {

          Animale m1 = new Mucca();
          Animale a1 = (Mucca) m1;

      }

  }
```

La classe Object

La classe Object è la root della gerarchia delle classi. Ogni classe ha Object come superclasse. Tutti gli oggetti, inclusi gli array, implementano i metodi di questa classe.

Final

Si può dichiarare alcuni o tutti i metodi di una classe final. Si utilizza la parola chiave final in una dichiarazione di metodo per indicare che il metodo non può essere sovrascritto dalle sottoclassi. La Object classe fa questo: alcuni dei suoi metodi sono final.

Interfaccia

Nel linguaggio di programmazione Java, un'interfaccia e un tipo di riferimento, simile a una classe, che può contenere solo costanti, firme di metodi, metodi predefiniti, metodi statici e tipi nidificati. Le interfacce non possono essere istanziate: possono essere implementate solo da classi o estese da altre interfacce. Si usa la parola `implements`:

```
    public c l a s s BankAccount implements Measurable
    {
    public double getMeasure ( )
    {
    return b a l a n c e ;
    }
    // altri metodi e campi
}
```

Differenza tra classe astratta e interfaccia

Le classi astratte sono classi parzialmente implementate che possono essere estese da altre classi e consentono l'ereditarietà di metodi e variabili di istanza. Le interfacce sono contratti che definiscono metodi che devono essere implementati da classi diverse, consentendo l'implementazione multipla. Le classi astratte forniscono un maggiore livello di flessibilità rispetto alle interfacce, ma limitano l'ereditarietà a una sola classe astratta, mentre le interfacce consentono l'implementazione multipla.

Eccezioni try-catch-finally

- Il try viene utilizzato per specificare un blocco di codice che potrebbe generare un'eccezione.
- Il catch viene utilizzato per gestire l'eccezione se viene generata.
- Il finally viene utilizzato per eseguire il codice dopo che i blocchi trye catch sono stati eseguiti.

```
  public class Main {
   public static void main(String[] args) {
       try {
          // Code that may throw an exception
       } catch (Exception e) {
          // Code to handle the exception
       } finally {
          // Code that is always executed
       }
   }
}
```

Politica catch-or-declare in java

La "catch-or-declare" (cattura-o-dichiara) è una regola associata alle eccezioni nel linguaggio di programmazione Java. In Java, le eccezioni possono verificarsi durante l'esecuzione di un programma e possono essere gestite utilizzando la gestione delle eccezioni. La regola "catch-or-declare" richiede che ogni metodo che può generare un'eccezione dichiari esplicitamente le eccezioni che può lanciare o che gestisca tali eccezioni utilizzando il blocco "try-catch". La regola "catch-or-declare" è stata introdotta per favorire la gestione e la consapevolezza delle eccezioni in un programma Java. Obbedendo a questa regola, i programmatori devono essere espliciti riguardo alle eccezioni che possono essere sollevate da un metodo, aiutando così a migliorare la comprensione e la manutenibilità del codice.

Parole chiave throw e throws

La parola chiave "throw" viene utilizzata per lanciare esplicitamente un'eccezione all'interno di un metodo. Quando si verifica una condizione che richiede la segnalazione di un'eccezione, si utilizza "throw" seguito dall'oggetto dell'eccezione che si desidera lanciare. Ad esempio:

```
public void metodoA() {
  if (condizione) {
    throw new RuntimeException("Messaggio di errore"); // Lancia un'eccezione
  }
}
```

La parola chiave "throws" viene utilizzata nella dichiarazione di un metodo per indicare che il metodo può generare un'eccezione controllata. Quando un metodo può sollevare un'eccezione controllata, è necessario dichiarare il tipo di eccezione utilizzando la parola chiave "throws". Ad esempio:

```
  public void metodoB() throws IOException {
// Implementazione del metodo che può generare IOException
}
```

Classi di errori ed eccezioni

In Java, le classi `Throwable`, `Exception` ed `Error` sono tutte parte della gerarchia delle eccezioni e delle situazioni anomale.
`Throwable` è la superclasse di tutte le eccezioni e le situazioni anomale in Java. È una classe astratta e può essere estesa da due sottoclassi principali: `Exception` e `Error`. La classe `Throwable` fornisce i metodi fondamentali per gestire le eccezioni, come `getMessage()`, `printStackTrace()`, ecc.
`Exception` è una sottoclasse di `Throwable` e rappresenta le eccezioni controllate in Java. Le eccezioni controllate sono quelle che un programma può prevedere e gestire. Le eccezioni di tipo `Exception` possono essere divise ulteriormente in due categorie:

- Eccezioni controllate (*checked exceptions*): Queste sono eccezioni che devono essere dichiarate nella firma del metodo o gestite utilizzando un blocco `try-catch`. Le eccezioni controllate estendono direttamente la classe `Exception` (ad esempio, `IOException`, `SQLException`, ecc.).
- Eccezioni non controllate (*unchecked exceptions*): Queste sono eccezioni che non richiedono una dichiarazione o una gestione esplicita. Sono solitamente causate da errori nel codice o da condizioni impreviste. Le eccezioni non controllate estendono la classe `RuntimeException` (ad esempio, `NullPointerException`, `ArrayIndexOutOfBoundsException`, ecc.).

`Error` è un'altra sottoclasse di `Throwable` e rappresenta situazioni anomale gravi che si verificano durante l'esecuzione del programma. Gli `Error` indicano problemi irrecuperabili che di solito sono legati all'ambiente di esecuzione o al sistema stesso. Al contrario delle eccezioni, gli `Error` non dovrebbero essere catturati o gestiti, poiché non è possibile recuperarsi da tali situazioni. Ad esempio, `OutOfMemoryError`, `StackOverflowError` sono esempi di `Error` comuni.

Implementazione Stack

```
import java.util.ArrayList;

public class Stack<T> {
    private ArrayList<T> items;

    public Stack() {
        this.items = new ArrayList<>();
    }

    public void push(T item) {
        this.items.add(item);
    }

    public T pop() {
        if (this.isEmpty()) {
            return null;
        }
        return this.items.remove(this.items.size() - 1);
    }

    public T peek() {
        if (this.isEmpty()) {
            return null;
        }
        return this.items.get(this.items.size() - 1);
    }

    public boolean isEmpty() {
        return this.items.isEmpty();
    }
}
```

Implementazione Queue

Queue:

```
  import java.util.LinkedList;

  public class Queue<T> {
    private LinkedList<T> elements;

    public Queue() {
        elements = new LinkedList<>();
    }

    public void enqueue(T item) {
        elements.addLast(item);
    }

    public T dequeue() {
        return elements.poll();
    }

    public boolean isEmpty() {
        return elements.isEmpty();
    }

    public int size() {
        return elements.size();
    }

    public T peek() {
        return elements.peek();
    }
}
```