

Java SE

Java Foundation

Presented by

Valerio Cammarota

IBM Client Innovation Center - Italy

13/11/2023

IBM Client Innovation Center
Italy

Agenda



Introduzione a Java



Ambiente di sviluppo



Il primo programma



Componenti fondamentali



Principi OOP



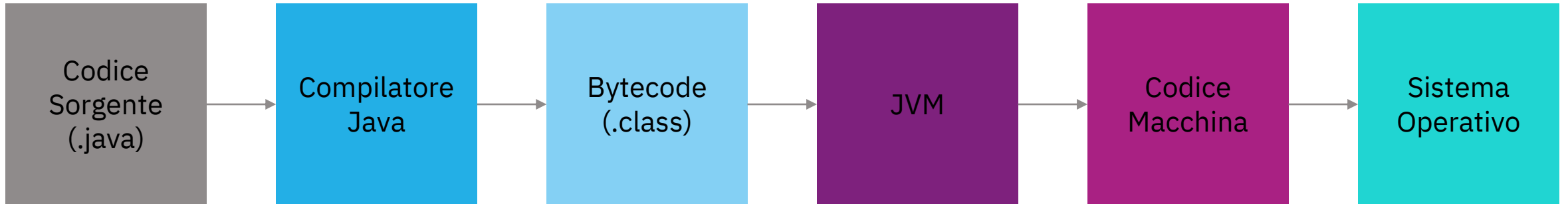
Caratteristiche avanzate

Introduzione a Java

- ❖ Java è un linguaggio di programmazione **orientato agli oggetti (OOP)**;
- ❖ È stato creato da **Sun Microsystems nel 1995** ed è disponibile gratuitamente;
- ❖ È un linguaggio di programmazione **fortemente tipizzato**, ogni variabile ha un suo tipo;
- ❖ I programmi sono costituiti da **oggetti**, ognuno con il proprio **stato**, che interagiscono tra loro;
- ❖ Il codice Java, previa compilazione, viene eseguito all'interno della **Java Virtual Machine (JVM)**;
- ❖ È **indipendente dalla piattaforma**, un programma è compilato in un formato intermedio (**bytecode**) offrendo un elevato grado di portabilità;

Fase di compilazione

Java è un linguaggio di programmazione **compilato**.



Vantaggi

- ❖ Java è facile da imparare, è stato progettato per essere facile da usare ed è facile da scrivere;
- ❖ È orientato agli oggetti (**OOP**), ciò consente di creare programmi modulari e codice riutilizzabile;
- ❖ È indipendente dalla piattaforma, è uno dei vantaggi più significativi;
- ❖ Ha un meccanismo di gestione della memoria (**Garbage Collector**);
- ❖ Meccanismi di sicurezza molto efficienti, eseguire un'applicazione Java significa isolare il codice nella JVM, senza un diretto accesso alla memoria;
- ❖ Ha un'ampia community di supporto.

Ambiente di sviluppo

Ambiente di sviluppo

- ❖ **Java Development Kit – JDK:** è un potente strumento che consente agli sviluppatori di creare, compilare ed eseguire applicazioni Java.
- ❖ Il JDK è costituito da diversi componenti come un compilatore, una JVM, un formattatore di documentazione, un generatore di file JAR, etc. utile allo sviluppo JAVA.
- ❖ È scaricabile gratuitamente dal sito della Oracle: <https://www.oracle.com/java/technologies/downloads>

Configurazione Ambiente di sviluppo: Windows

❖ Scaricare la versione del jdk per il proprio sistema operativo.

- Per windows:


Linux	macOS	Windows
Product/file description	File size	Download
x64 Compressed Archive	179.13 MB	https://download.oracle.com/java/19/latest/jdk-19_windows-x64_bin.zip (sha256)
x64 Installer	158.91 MB	https://download.oracle.com/java/19/latest/jdk-19_windows-x64_bin.exe (sha256)
x64 MSI Installer	157.76 MB	https://download.oracle.com/java/19/latest/jdk-19_windows-x64_bin.msi (sha256)

- Per *macOS*: Scegliere Arm 64 DMG Installer o x64 DMG Installer in base alla versione del processore

Linux	macOS	Windows
Product/file description	File size	Download
Arm 64 Compressed Archive	175.67 MB	https://download.oracle.com/java/19/latest/jdk-19_macos-aarch64_bin.tar.gz (sha256)
Arm 64 DMG Installer	175.07 MB	https://download.oracle.com/java/19/latest/jdk-19_macos-aarch64_bin.dmg (sha256)
x64 Compressed Archive	177.54 MB	https://download.oracle.com/java/19/latest/jdk-19_macos-x64_bin.tar.gz (sha256)
x64 DMG Installer	176.92 MB	https://download.oracle.com/java/19/latest/jdk-19_macos-x64_bin.dmg (sha256)

Configurazione Ambiente di sviluppo: Windows

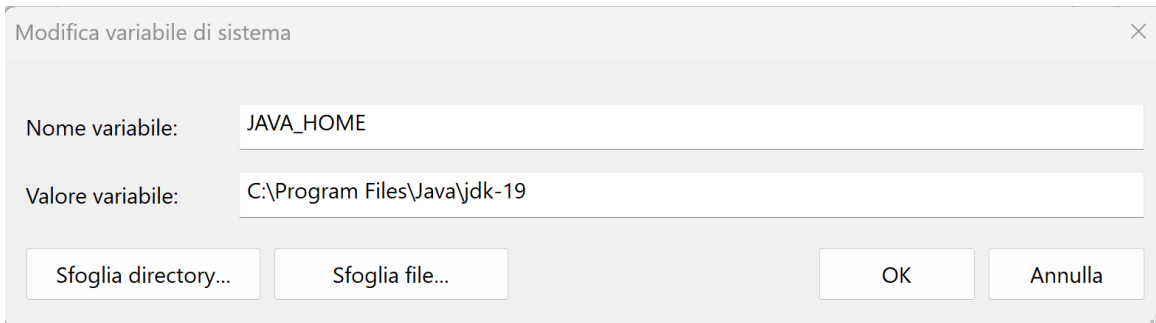
- ❖ Una volta completata l'installazione si crea una cartella al seguente path:

 > Questo PC > Windows (C:) > Programmi > Java > jdk-19

- ❖ Verificare l'installazione lanciando da terminale il comando `java -version`

```
C:\Users\065643758>java -version
java version "20.0.1" 2023-04-18
Java(TM) SE Runtime Environment (build 20.0.1+9-29)
Java HotSpot(TM) 64-Bit Server VM (build 20.0.1+9-29, mixed mode, sharing)
```

- ❖ Copiare il path in cui è stato installato Java e settare la seguente variabile d'ambiente:



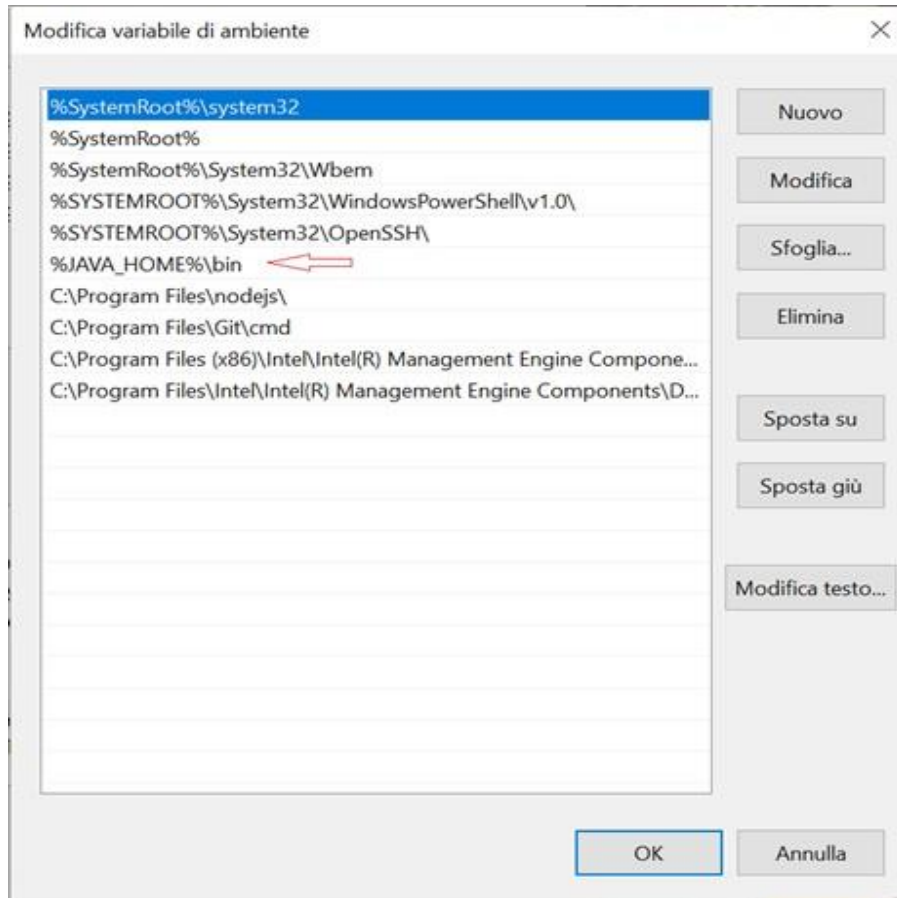
Modifica variabile di sistema

Nome variabile:

Valore variabile:

Configurazione Ambiente di sviluppo: Windows

❖ Doppio click sulla variabile di sistema Path e inserire **%JAVA_HOME%\bin**



Configurazione Ambiente di sviluppo: macOS

- ❖ Una volta completata l'installazione, lanciare da terminale il comando:
 - `/usr/libexec/java_home -V`
- ❖ Verificare il tipo di shell utilizzata con il seguente comando:
 - `echo $SHELL`
- ❖ Impostare la variabile d'ambiente JAVA_HOME con uno dei seguenti comandi (in base al tipo di shell) e riavviare il terminale:
 - `echo export "JAVA_HOME=$(/usr/libexec/java_home)" >> ~/.zshenv`
 - `echo export "JAVA_HOME=$(/usr/libexec/java_home)" >> ~/.zshrc`
 - `echo export "JAVA_HOME=$(/usr/libexec/java_home)" >> ~/.bash_profile`
 - `echo export "JAVA_HOME=$(/usr/libexec/java_home)" >> ~/.bashrc`
- ❖ Impostare la variabile PATH con il seguente comando (in base al tipo di shell):
 - `echo export "PATH=$JAVA_HOME/bin:$PATH" >> ~/.<tipo_shell>`
- ❖ Verificare che le variabili siano state impostate correttamente con i seguenti comandi:
 - `echo $JAVA_HOME`
 - `echo $PATH`

Ambiente di sviluppo

- ❖ Per scrivere un programma Java è necessario un editor;
- ❖ Troviamo due tipologie:
 - Editor di testo, come Notepad o VS Code;
 - Integrated Development Environment (**IDE**), come Eclipse, NetBeans, IntelliJ IDEA.

Ambiente di sviluppo: editor di testo

- ❖ Creiamo un file **HelloIBMCICAcademy** con estensione **.java** e aggiungiamo il seguente contenuto:

```
public class HelloIBMCICAcademy {  
    public static void main(String[] args) {  
        System.out.println("Hello IBM Client Innovation Center Java Academy!");  
    }  
}
```

- ❖ Apriamo un terminale e lanciamo il comando di compilazione:

- **javac HelloIBMCICAcademy.java**

```
C:\Progetti\EsempioClasseJavaEditor>javac HelloWorld.java  
C:\Progetti\EsempioClasseJavaEditor>
```

- ❖ Lanciamo da terminale il comando di esecuzione:

- **java HelloIBMCICAcademy**

```
C:\Progetti\EsempioClasseJavaEditor>java HelloWorld  
Hello World!
```

Ambiente di sviluppo: IDE Eclipse

- ❖ Download dell'IDE dal seguente link: <https://www.eclipse.org/downloads/packages/>
- ❖ Per **Windows** scegliere **x86_64**;
- ❖ Per **macOS** con processore **M1/M2/M3** scegliere **AArch64**.

The Eclipse Installer 2023-09 R now includes a JRE for macOS, Windows and Linux.

Try the Eclipse Installer 2023-09 R

The easiest way to install and update your Eclipse Development Environment.

[Find out more](#)


📦 1,382,210 Installer Downloads

📦 1,483,538 Package Downloads and Updates

Download

macOS [x86_64](#) | [AArch64](#)
Windows [x86_64](#)
Linux [x86_64](#) | [AArch64](#)

Eclipse IDE 2023-09 R Packages



Eclipse IDE for Java Developers

319 MB | 847,868 DOWNLOADS

The essential tools for any Java developer, including a Java IDE, a Git client, XML Editor, Maven and Gradle integration

Windows [x86_64](#)
macOS [x86_64](#) | [AArch64](#)
Linux [x86_64](#) | [AArch64](#)

Ambiente di sviluppo: IDE Eclipse

Eclipse è un ambiente di sviluppo integrato (**IDE**), multi-linguaggio e multiplatforma, ed è composto da:

- ❖ **Workspace:** uno spazio di lavoro che può contenere progetti, file e cartelle. Al lancio di Eclipse viene chiesto il workspace su cui lavorare;
 - **Per cambiare workspace:** File > Switch workspace > Other
- ❖ **Views:** consentono di visualizzare una rappresentazione grafica dei metadati del progetto.
 - **Per aprire una perspective:** Windows > Show View > Other
- ❖ **Perspective:** è il nome dato alla disposizione di un insieme di viste e ad un'area di editor, ognuna può essere modificata secondo le proprie esigenze;
 - **Per aprire una perspective:** Windows > Open Perspective > Other

Creiamo il primo programma

Aprire l'IDE Eclipse e procedere con i seguenti passi:

❖ **Creare il progetto: IBMClCAcademyProject**

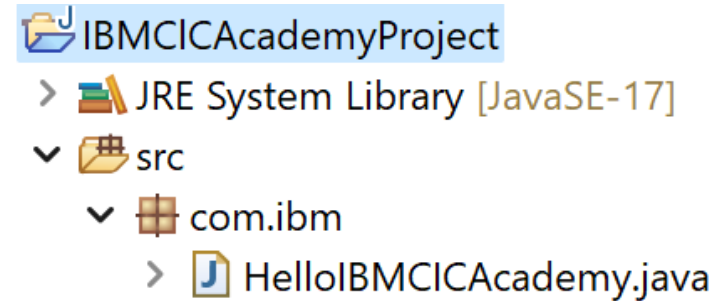
- File > New > Other > Java Project

❖ **Aggiungere il package: com.ibm**

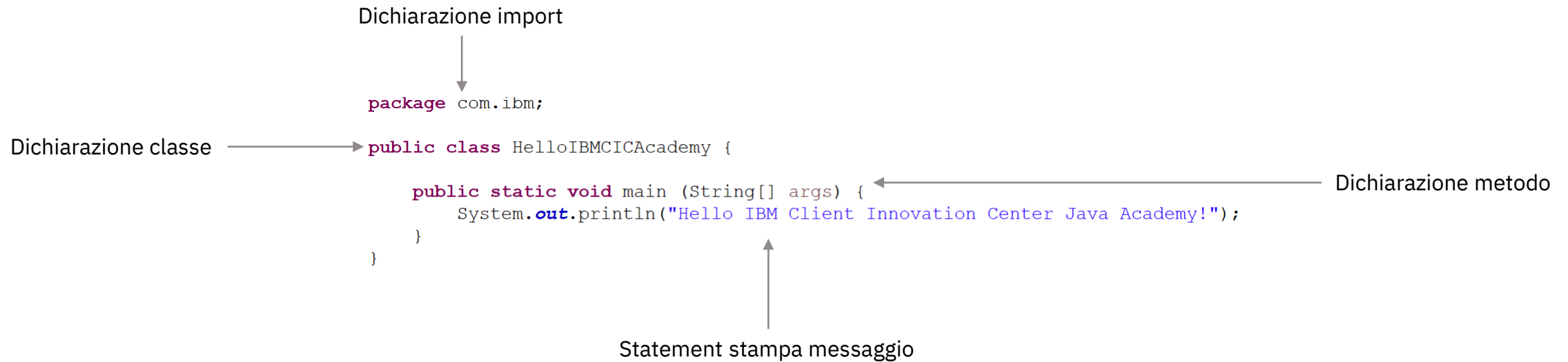
- Tasto dx su src > New > Package

❖ **Aggiungere la classe: HelloIBMClCAcademy**

- Tasto dx sul package > New > Class



Struttura di una classe



Esecuzione: Tasto dx > Run As > Java Application

Esecuzione in debug: Tasto dx > Debug As > Java Application

Componenti fondamentali

Componenti fondamentali: COMMENTI

❖ Non sono elaborati dal compilatore per la generazione del bytecode;

❖ Commenti a riga singola:

```
// Invio le notifiche a tutti gli studenti
```

❖ Commenti a più righe:

```
/*  
 * Metodo per l'invio delle notifiche a tutti  
 * gli studenti, tramite il server SMTP IBM.  
 */
```

Componenti fondamentali: TIPI DI DATI

In Java troviamo due tipologie di tipi di dati:

- ❖ **Primitivi**: non sono oggetti, non hanno una classe associata, non possono essere estesi e non possiedono metodi;
- ❖ **Non primitivi** e **Wrapper**: sono le classi custom e i wrapper dei tipi primitivi ;

Tipologia	Primitivo	Memoria utilizzata	Wrapper
Numeri interi	byte	8bit	Byte
	short	16bit	Short
	int	32bit	Integer
	long	64bit	Long
Numeri floating point	double	64bit	Double
	float	32bit	Float
Caratteri	char	16bit	Character
Booleani	boolean	1bit	Boolean

Componenti fondamentali: VARIABILI

- ❖ Una variabile è un'area di memoria in cui un certo **tipo** di dato viene immagazzinato;
- ❖ Distinguiamo due fasi per l'utilizzo delle variabili:
 - **Dichiarazione**: definizione della variabile (**identificatore**);
 - **Assegnazione**: assegnazione di un valore alla variabile (**valore**);
- ❖ L'identificatore della variabile può essere qualsiasi stringa, ad eccezione delle seguenti.

try	private	enum	while	float	interface
catch	default	volatile	for	int	class
finally	protected	new	break	long	abstract
throws	public	this	continue	char	extends
throw	static	synchronized	goto	short	implements
import	final	void	if	double	return
transient	package	super	else	boolean	
native	strictfp	assert	do	byte	
instanceof	switch	const	case		

```
// Dichiarazione
String name;
Integer age;
Double weight;
```

```
// Assegnazione
name = "Mattia";
age = 21;
weight = 67.8;
```

```
// Ri-assegnazione
name = "Federica";
age = 23;
weight = 54.8;
```

Componenti fondamentali: OPERATORI

Tipologia Operatore	Descrizione	Operatore	Esempio
Assegnazione	Assegnazione valore	=	a = 21; b = 20;
Aritmetico	Somma	+	a + b
	Sottrazione	-	a - b
	Moltiplicazione	*	a * b
	Divisione	/	a / b
	Modulo	%	a % 2
Pre/Post incremento	Pre-incremento	++	++a
	Post-incremento	--	a++
Pre/Post decremento	Pre-decremento	--	--a
	Post-decremento	--	a--

Componenti fondamentali: OPERATORI

Tipologia Operatore	Descrizione	Operatore	Esempio
Confronto	Uguale	==	int a = 20; int b = 20; a == b
	Diverso	!=	a != b
	Maggiore	>	a > b
	Minore	<	a < b
	Maggiore o Uguale	>=	a >= b
	Minore o Uguale	<=	a <= b
Logico	NOT	!	!(a == b)
	XOR (almeno una è vera)	^	((a > 0) ^ (b < 30))
	Short circuit AND	&&	(a >= 10 && b <= 30)
	Short circuit OR		(a > 10 b < 30)

Componenti fondamentali: STRINGHE

- ❖ Sono **oggetti** e possono essere **istanziati** anche con la parola chiave **new**:

```
String academy = new String("IBM CIC Java Academy");
```

- ❖ Java ne semplifica l'utilizzo consentendo l'uso come se fossero tipi di dati primitivi:

```
String academy = "IBM CIC Java Academy";
```

- ❖ Essendo una classe, String mette a disposizione una serie di **metodi**;
- ❖ È bene ricordare che in Java, le stringhe sono **oggetti immutabili**, il loro stato non cambia.
 - Un'istanza della classe String assume un valore all'atto della creazione e non può più essere cambiato.

Componenti fondamentali: STRINGHE

Esempio pratico, metodo **replace()** senza ri-assegnazione:

```
String academy = "IBM CIC Java Academy";  
academy.replace("CIC", "Client Innovation Center");
```

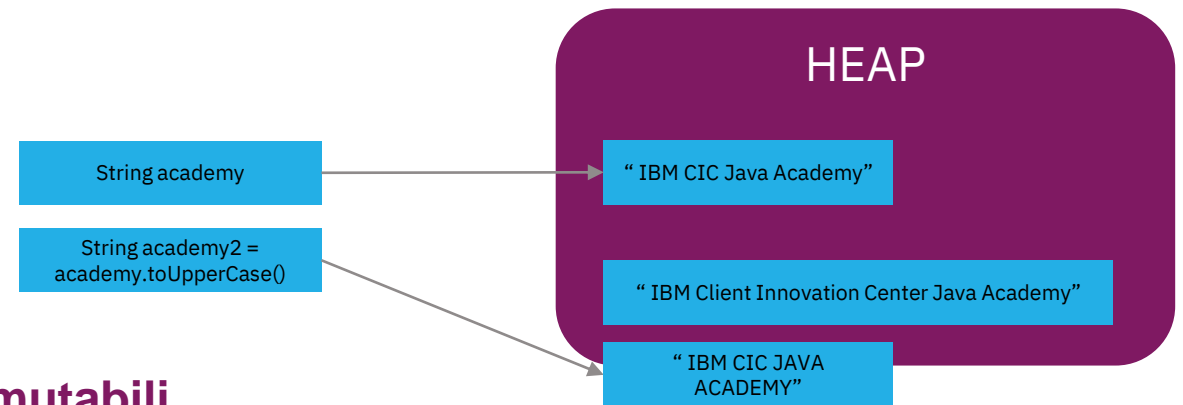
È corretto questo utilizzo?

No!

È bene ricordare che in Java, le stringhe sono **oggetti immutabili**.

- Un'istanza della classe String assume un valore all'atto della creazione e non può più essere cambiato.

```
academy = academy.replace('CIC', 'Client Innovation Center');
```



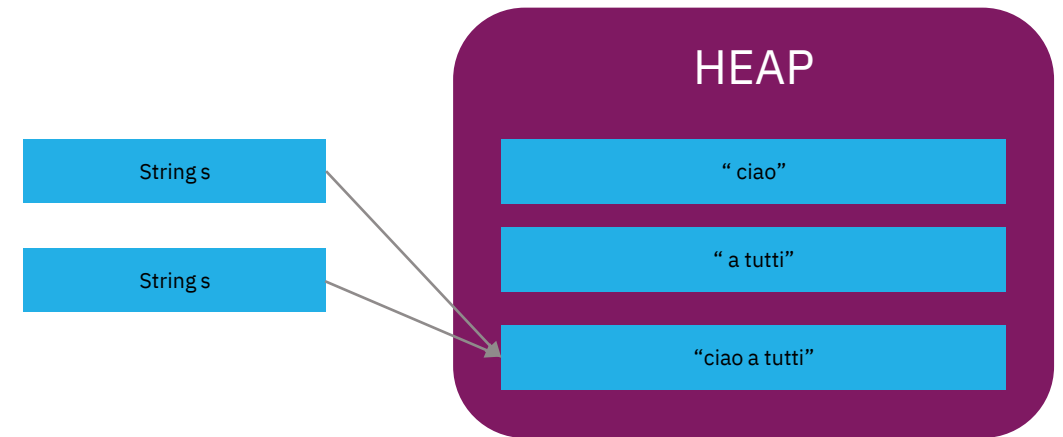
Componenti fondamentali: STRINGHE

```
String s = "ciao";  
s = s + " a tutti";
```

All'inizio il valore della stringa sia "ciao" e che poi venga cambiata in "ciao a tutti".

In realtà viene creata:

- una prima istanza di String con valore "ciao";
- una seconda istanza con valore " a tutti";
- una terza istanza con valore "ciao a tutti".



L'assegnazione fa sì che **s** punti alla terza istanza. La prima e la seconda istanza non hanno più riferimenti e le loro aree di memoria sono quindi recuperate dal **garbage collector**.

Nessuna stringa ha cambiato il proprio valore!

Componenti fondamentali: STRINGHE

Metodi di uso comune della classe String

Metodo	Descrizione
substring(start, stop)	Ritorna la sotto-stringa partendo dalla posizione start alla posizione stop compresi.
trim()	Ritorna una nuova stringa con senza gli spazi iniziali e finali.
replace(x, y)	Ritorna una nuova stringa dove tutte le occorrenze di x sono sostituite con y .
toUpperCase()	Ritorna una nuova stringa con tutti i caratteri in maiuscolo.
toLowerCase()	Ritorna una nuova stringa con tutti i caratteri in minuscolo.
charAt(index)	Ritorna il carattere con indice 'index'.
s1.equals(s2)	Confronta la stringa s1 con la stringa s2 e restituisce true se s1 e s2 sono uguali, altrimenti false . Attenzione all'utilizzo dell'operatore == <ul style="list-style-type: none">▪ Il metodo equals() effettua un confronto sul contenuto;▪ L'operatore == verifica se entrambe le stringhe puntano alla stessa locazione di memoria.

Componenti fondamentali: STRINGHE

L'oggetto String è immutabile ovvero una volta creato non può essere modificato.

Per rendere le stringhe mutabili si utilizza **StringBuilder** o **StringBuffer** che non creano nuovi oggetti inutilizzati.

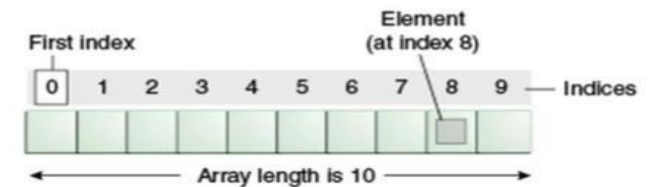
Classe	Descrizione
StringBuilder	È thread-safe ovvero si si unsa nell'ambito di processi concorrenti, viene garantita la mutua esclusione.
StringBuffer	Non + thread-save, non avendo questa caratteristica, risulta essere più performate.

```
StringBuilder sb = new StringBuilder();  
sb.append("Hello ");  
sb.append("World");
```

```
StringBuffer sbuff = new StringBuffer();  
sbuff.append("Hello ");  
sbuff.append("World");
```

Componenti fondamentali: ARRAY

- ❖ È una **collezione di dati** omogenei, dello stesso tipo, in Java gli array, hanno una lunghezza fissa.
- ❖ Gli elementi sono **accessibili tramite indici** interi;
- ❖ L'indice di un array **inizia sempre da zero**;
- ❖ Utilizzo di un array:



- **Dichiarazione:** posporre (o anteporre) le parentesi quadre all'identificatore.

```
String[] programmingLanguages; oppure String programmingLanguages[];
```

- **Creazione:** è un oggetto speciale e va istanziato con la parola chiave **new**, specificando la dimensione.

```
programmingLanguages = new String[n];
```

- **Inizializzazione:** assegnare un valore agli elementi dell'array tramite l'indice.

```
programmingLanguages[0] = "Java";
```

Componenti fondamentali: ENUM

- ❖ Rappresenta un gruppo di costanti (variabili il cui valore è immutabile).
- ❖ Per la creazione si usa la parola chiave **enum** e al suo interno si definisce l'elenco di costanti in lettere maiuscole, separate da virgola;
- ❖ Possono contenere un valore, che se non specificato è pari ad un intero (**ordinal**) inizializzato con la posizione della costante nell'elenco.

```
public enum Level {  
    LOW,  
    MEDIUM,  
    HIGH;  
}  
  
public enum LevelWithValue {  
    LOW(1),  
    MEDIUM(2),  
    HIGH(3);  
  
    private int value;  
  
    private LevelWithValue(int value) {  
        this.value = value;  
    }  
  
    public int getValue() {  
        return value;  
    }  
}
```

Costrutti condizionali

Costrutti condizionali: COSTRUTTO IF

- ❖ Permette di prendere semplice **decisioni sulla base del valore di una condizione**.
- ❖ La condizione è rappresentata da **un'espressione booleana** che può essere vera o falsa.
- ❖ Sintassi if/else:

```
if(espressione booleana){  
    istruzione;  
}
```

```
if(espressione booleana){  
    istruzione;  
} else {  
    istruzione;  
}
```

```
if (espressione booleana) istruzione1;  
    istruzione2;  
  
else  
    istruzione3;
```

- ❖ Operatore ternario:

```
(espressione booleana) ? Istruzione_true : istruzione_false;
```


Costrutti condizionali: COSTRUTTO SWITCH

- ❖ Può essere un'alternativa al costrutto IF;
- ❖ I blocchi di codice vengono in base al valore che assume **la variabile di test**;
- ❖ La variabile di test può assumere valori: `byte`, `short`, `char`, `int`;
- ❖ Dalle versione 7 di Java è possibile usare una stringa come variabile di test;
- ❖ Parola chiave **break**: provoca l'immediata uscita dal costrutto:
 - Se non è presente verranno eseguite tutte le istruzioni appartenenti ai case;
- ❖ Parola chiave **default**: è opzionale e serve a determinare una porzione di codice che sarà comunque eseguita quando non viene verificata nessuna clausola `case`.

```
switch(variabile di test){  
    case valore_1: {  
        blocco_1;  
    }  
    break;  
    ...  
    case valore_n: {  
        blocco_n;  
    }  
    break;  
    [default: {  
        blocco;  
    }]  
}
```

Costrutti iterativi

Costrutti iterativi: COSTRUTTO WHILE

- ❖ Permette di **iterare un'istruzione o un blocco di istruzioni per un numero di volte**, sulla base del valore di una condizione booleana, potrebbe non essere mai eseguito;
- ❖ Sintassi:

```
[inizializzazione];  
while(espressione booleana) {  
    istruzioni;  
    [aggiornamento iterazione;]  
}
```

Costrutti iterativi: COSTRUTTO DO-WHILE

- ❖ Simile al while, solo che la condizione viene verificata alla fine del ciclo.
- ❖ **Il blocco di istruzioni viene eseguito almeno una volta.**
- ❖ Sintassi:

```
[inizializzazione;]  
do{  
    istruzioni;  
    [aggiornamento iterazione;]  
} while(espressione booleana);
```

Costrutti iterativi : COSTRUTTO FOR

- ❖ Viene utilizzato per iterare array e liste, ed eseguire **un numero fissato di istruzioni**;
- ❖ Sintassi:

```
for(inizializzazione; espressione booleana; aggiornamento){  
    istruzioni;  
}
```

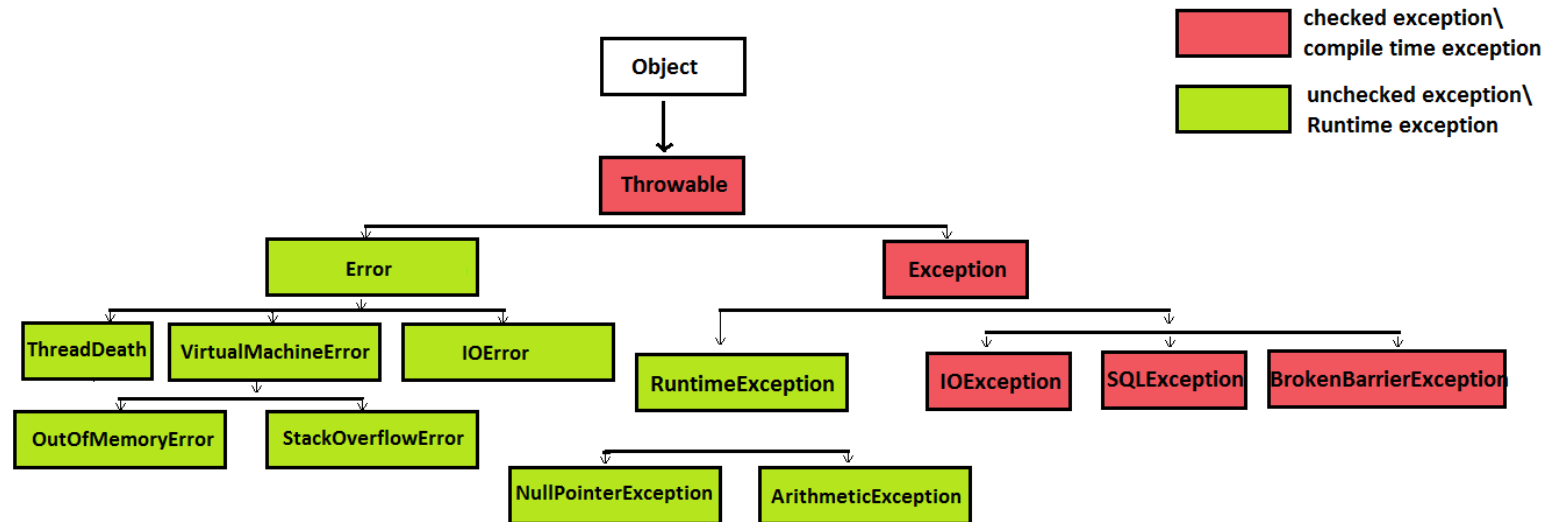
- ❖ Dalla versione Java 1.5 è stata introdotta una versione migliorata definita **foreach**;
- ❖ Utilizzato per **iterare su tutti gli elementi di una collezione di dati** (iterator);
- ❖ Sintassi:

```
for(variabile_temporanea : oggetto iterabile){  
    istruzioni;  
}
```

Gestione delle eccezioni

Gestione delle eccezioni

- ❖ In Java le eccezioni sono strutturate in modo gerarchico;
- ❖ Troviamo due categorie di eccezioni:
 - ❖ **Checked Exception**: errori che possono verificarsi a compile-time, sono segnalati dall'IDE;
 - ❖ **Unchecked Exception**: errori che possono verificarsi a run-time, non sono segnalati dall'IDE.



Gestione delle eccezioni

In Java le eccezioni vengono gestite grazie a speciali blocchi di gestione, definiti da tre istruzioni:

- ❖ L'istruzione **try** consente di definire un blocco di codice;
- ❖ L'istruzione **catch** consente di catturare eventuali eccezioni;
- ❖ L'istruzione **finally** viene eseguito indipendentemente del try/catch, è sempre eseguito per ultimo.

```
try {  
    System.out.println("Blocco try...");  
    int x = 1, y = 0;  
    int result = (x / y);  
    System.out.println(result);  
} catch (Exception e) {  
    System.out.println("Blocco catch...");  
} finally {  
    System.out.println("Blocco finally...");  
}
```


Componenti fondamentali: LIBRERIA STANDARD

❖ Java possiede un'enorme libreria di componenti standard, chiamate classi, organizzate in package;

- **java.io**: che contiene le classi per realizzare l'input e l'output in Java;
- **java.util**: contiene classi di utilità (es. `java.util.Date`);
- **java.lang**: contiene le classi core del linguaggio (es. `System` e `String`);
 - Viene importato automaticamente dal compilatore;

❖ Per usare una classe di una libreria, è necessario importarla usando la parola chiave **import**:

```
import java.util.Date;
```

❖ Per importare tutte le classi del package, si usa la notazione:

```
import java.util.*;
```

<https://docs.oracle.com/javase/10/core/java-core-libraries1.htm>

<https://docs.oracle.com/javase/8/docs/api/>

Componenti fondamentali: CLASSI E OGGETTI

Object Oriented Programming (OOP) è paradigma di programmazione in cui i programmi sono organizzati attraverso un insieme di **oggetti**, ognuno dei quali è un'**istanza** di una **classe**.

Le classi sono parte di una **gerarchia** di entità collegate tra loro mediante una relazione di **ereditarietà**.

Troviamo tre concetti fondamentali:

- ❖ OOP **utilizza un insieme di oggetti**;
- ❖ Ogni **oggetto** è **istanza** di una **classe**;
- ❖ Ogni **classe** è legata alle altre attraverso una **relazione detta di ereditarietà**.

Se in un programma manca anche solo una di queste caratteristiche, non lo si può definire object-oriented.

Componenti fondamentali: CLASSI E OGGETTI

- ❖ Una classe Java è la definizione di un tipo di oggetto;
- ❖ Una classe specifica il **tipo** che assumerà l'oggetto della classe;
- ❖ In una classe si definiscono:
 - ❖ le variabili (**variabili d'istanza**) o attributi di una classe;
 - ❖ i **metodi** utilizzati per manipolare lo stato dell'oggetto.

```
public class Persona {  
    private String nome;  
    private String cognome;  
    private Integer eta;  
  
    public String getNome(){  
        return this.nome;  
    }  
}
```

Componenti fondamentali: ISTANZA DI UNA CLASSE

- ❖ Per istanziare una classe Java si utilizza la parola chiave **new**;

```
Persona p = new Persona(); //crea un oggetto p istanza della classe Persona;
```

- ❖ Vengono predisposte tutte le variabili di istanza all'interno dell'oggetto creato;
- ❖ Viene restituito un riferimento all'indirizzo di memoria dell'oggetto creato;
- ❖ L'istruzione **=** assegna il riferimento dell'oggetto alla variabile **p** (un puntatore).

Componenti fondamentali: VARIABILI DI UNA CLASSE

❖ In una classe Java le variabili sono costituite da:

- **Modificatori d'accesso:** private, protected, public;
- **Tipo di dato:** String, Integer, int, ecc..;
- Nome della variabile: nome con cui utilizzare la variabile;
- Inizializzazione: valore con il quale inizializzare la variabile;
 - L'inizializzazione per i tipi primitivi (int, double, boolean, char, short...) è obbligatoria
 - L'inizializzazione per i tipo wrapper o custom è facoltativa.

Diagram illustrating the components of a Java variable declaration:

```
public String academy = "IBM CIC Java Academy";
```

The diagram shows the following components and their labels:

- public**: Labeled as **Modificatore** (Access Modifier).
- String**: Labeled as **Tipo** (Data Type).
- academy**: Labeled as **Nome** (Variable Name).
- = "IBM CIC Java Academy";**: Labeled as **Valore** (Value).

Componenti fondamentali: METODI DI UNA CLASSE

- ❖ Consentono di manipolare lo stato di un oggetto;
- ❖ Gli oggetti utilizzano i metodi per comunicare tra loro;
- ❖ Evitano la duplicazione del codice e ne favoriscono il riuso.

Tipo di ritorno

Modificatore

Parametro

```
public String say(String messaggio){  
    String messaggioCompleto = "Hello, " + messaggio; // variabile locale al metodo  
    return messaggioCompleto;  
}
```



Componenti fondamentali: PASSAGGIO PARAMETRI

- ❖ Troviamo due modalità per il passaggio dei parametri in Java;
- ❖ Passaggio **per valore**, al metodo viene passata una copia dell'argomento;
- ❖ Passaggio **per riferimento** (o reference), al metodo viene passato l'indirizzo di memoria.

```
/**
 * Passaggio per valore.
 * @param lato
 */
public int calcolaAreaQuadrato(int lato){
    int area = (lato * lato);
    return area;
}
```

```
/**
 * Passaggio per riferimento.
 * @param quadrato
 */
public void calcolaAreaQuadrato(Quadrato quadrato){
    int area = (quadrato.getLato() * quadrato.getLato());
    quadrato.area = area; //area è un attributo public
}
```

Componenti fondamentali: COSTRUTTORI

- ❖ È un metodo speciale di una classe Java che ha lo stesso nome della classe e non ha un tipo di ritorno;
- ❖ Il compilatore Java inserisce automaticamente un costruttore di default se non definito esplicitamente.
- ❖ Tutti gli oggetti hanno almeno uno, ma possono esserci più costruttori;
- ❖ Viene utilizzato per l'inizializzazione le variabili d'istanza;
- ❖ Può avere una lista di parametri;

```
public class Quadrato {  
  
    public int lato;  
    public int area;  
  
    public Quadrato() {  
        // Costruttore vuoto - default  
    }  
  
    public Quadrato(int lato) {  
        this.lato = lato;  
    }  
}
```

this si usa davanti per evitare problemi
di ambiguità con le variabili locali;
scope differenti.

Componenti fondamentali: MODIFICATORI

Il modificatore (detto anche scope, visibilità) è una parola chiave che può cambiare il significato di una componente Java. Si antepone alla dichiarazione di una componente.

Modificatore	Classe	Attributo	Metodo	Costruttore	Visibilità
private	NO	SI	SI	SI	Visibile solo all'interno della stessa classe. È il modificatore d'accesso più restrittivo.
protected	NO	SI	SI	SI	Visibile solo dalle classi dello stesso package e dalle sottoclassi.
default	SI	SI	SI	SI	Visibile dallo stesso package e dalle sottoclassi se sono nello stesso package. È la visibilità assegnata di default se non viene specificato nulla.
public	SI	SI	SI	SI	Visibile da qualsiasi classe del programma. È il modificatore d'accesso più permissivo.
abstract	SI	NO	SI	NO	-
final	SI	SI	SI	NO	-
static	NO	SI	SI	NO	-

Principi OOP

Principi OOP

Il paradigma di programmazione orientato agli oggetti, in Java si basa su tre concetti chiave.

- ❖ **Incapsulamento**: consente di conservare lo stato di un oggetto, consentendone l'aggiornamento solo attraverso i metodi public messi a disposizione;
- ❖ **Ereditarietà**: proprietà di una classe di estendere un'altra classe, ereditandone alcune proprietà e comportamenti;
- ❖ **Polimorfismo**: proprietà di un'oggetto, in particolare dei suoi metodi, di assumere comportamenti differenti in base ai parametri con il quale viene invocato.

Principi OOP: INCAPSULAMENTO

- ❖ L'incapsulamento si ha quando l'oggetto **mantiene il suo stato privato** all'interno della classe. Altri oggetti non hanno accesso diretto a questo stato, ma possono accedervi, o modificarlo utilizzando le funzioni pubbliche, chiamate anche metodi;
- ❖ L'incapsulamento si basa sul concetto che **le variabili** di una classe dovrebbero essere **accessibili solo attraverso l'utilizzo dei suoi metodi**;
- ❖ Questo principio, in java, si traduce nel rendere i dati delle classi non visibili dall'esterno, quindi **private**, e creare dei metodi **pubblici** delegati all'accesso ad essi: **getter** e **setter**.

```
public class Persona {  
  
    private String cognome;  
    private String nome;  
    private int eta;  
  
    public String getCognome() {  
        return cognome;  
    }  
    public void setCognome(String cognome) {  
        this.cognome = cognome;  
    }  
}
```

Principi OOP: EREDITARIETA'

- ❖ L'ereditarietà consente di creare classi (**classe derivata**) che **riutilizzano**, **estendono** e **modificano** il comportamento della classe estesa (**classe base**).
- ❖ Si traduce nell'utilizzo della parola chiave **extends**;
- ❖ In Java è consentito **estendere una sola classe**, per realizzare l'ereditarietà multipla si utilizzano le interfacce;
- ❖ Se una classe estende un'altra, eredita solo i suoi membri non privati, per ovviare a questo si può usare il modificatore **protected**;
- ❖ Concetto di **HAS** and **IS**: utilizzo ed estensione.

```
// Superclasse
public class FormaGeometrica {

    protected String nome;
    protected int area;
}
```

```
// Triangolo IS FormaGeometrica
public class Triangolo extends FormaGeometrica {

    private int base;
    private int altezza;
}
```

```
// Collazione HAS Quadrato, Triangolo
public class Collezione {

    private Quadrato quadrato;
    private Triangolo triangolo;
}
```

Principi OOP: POLIMORFISMO

Il polimorfismo indica l'attitudine di un oggetto a mostrare più implementazioni per una singola funzionalità.

❖ Viene realizzato in due modi distinti:

- ❖ **Override**: è la possibilità che hanno le **sottoclassi** di **ridefinire** un metodo della superclasse, la firma del metodo non cambia, cambia solo l'implementazione;
- ❖ **Overloading**: nella stessa classe possono esserci metodi con lo stesso nome, ma con parametri differenti la firma del metodo cambia.

Tipologia	Compile Time	Run time	Descrizione
overriding	NO	SI	Utilizzato per implementare il polimorfismo dinamico a Runtime, è principalmente utilizzato per ridefinire uno specifico comportamento che altrimenti è fornito dalla sua superclasse.
overloading	SI	NO	Utilizzato per implementare il polimorfismo statico a Compile Time, è principalmente utilizzato per estendere la leggibilità del programma.

Principi OOP: OVERLOAD

- ❖ Nella stessa classe possono esserci metodi con lo stesso nome, ma con parametri differenti;
 - Tecnicamente si dice che la firma del metodo cambia.

- ❖ L'overload si distingue in due tipi:

- Basato su tipo

`somma(int a, int b)` - `somma(int a, double b)`

- Numerico

`somma(int a, int b)` - `somma(int a, int b, int c)`

```
public class Calcolatrice {  
  
    public int somma(int a, int b) {  
        return (a + b);  
    }  
  
    public int somma(int a, int b, int c) {  
        return (a + b + c);  
    }  
}
```

- ❖ Il tipo di ritorno non partecipa alla definizione della firma del metodo.

Principi OOP: OVERRIDE

- ❖ È la possibilità che hanno le **sottoclassi** di **ridefinire** un metodo della loro superclasse;
 - La firma del metodo non cambia; cambia solo l'implementazione.
- ❖ Regole per l'override:
 - Nella sottoclasse la firma deve essere la stessa;
 - Il tipo di ritorno del metodo di cui si fa l'override non deve cambiare;
 - Il metodo ridefinito non deve avere un'accessibilità minore di quello che ridefinisce;
- ❖ Non è possibile effettuare l'override dei costrutti, questi non sono ereditati dalla sottoclassi.

```
public class Dipendente {  
    private int oreLavorate;  
    private int pagaOraria;  
  
    public int stipendio(){  
        return oreLavorate * pagaOraria;  
    }  
}
```

```
public class Manager extends Dipendente{  
    private int bonus;  
  
    @Override  
    public int stipendio(){  
        return ((getOreLavorate * getPagaOraria)+ bonus);  
    }  
}
```


Caratteristiche avanzate

Caratteristiche avanzate: CLASSI ASTRATTE

- ❖ Una classe astratta è utilizzata per definire caratteristiche comuni fra classi di una determinata gerarchia;
- ❖ Una classe astratta non può essere istanziata, è progettata per svolgere la funzione di **classe base** e da cui le **classi derivate** possono ereditare i metodi;
- ❖ Può avere metodi non pubblici, un costruttore, come una classe a tutti gli effetti ma non istanziabile;
- ❖ La sua dichiarazione è caratterizzata dall'utilizzo della keyword **abstract**.

```
public abstract class Figura {  
    private String nome;  
  
    public Figura(nome) {  
        this.nome = nome;  
    }  
  
    protected String getNome() {  
        return nome;  
    }  
}
```

Caratteristiche avanzate: CLASSI ASTRATTE

- ❖ Una classe astratta può contenere o meno metodi astratti, ma una classe che contiene metodi astratti deve necessariamente essere dichiarata come astratta;
- ❖ I metodi astratti non hanno un'implementazione e necessariamente la sottoclasse dovrà effettuarne l'**override**.

```
public abstract class Figura {  
    public String getColore(){...};  
  
    public abstract int getArea();  
}
```

```
public class Triangolo extends Figura {  
    public int getArea(){...};  
}
```

```
public class Quadrato extends Figura {  
    public int getArea(){...};  
}
```

Caratteristiche avanzate: INTERFACCE

- ❖ Un'interfaccia è un tipo astratto usato per **descrivere il comportamento** che una classe deve implementare;
- ❖ Al suo interno tutti i metodi non hanno implementazione;
 - La definizione di quest'ultimi è lasciata alle classi che **implementano** l'interfaccia;
- ❖ È definita con la parola chiave **interface** e viene implementata da una classe;
- ❖ Una classe **può estendere solo una classe, ma implementare più interfacce**;
- ❖ Java non consente l'ereditarietà multipla ma tramite l'utilizzo delle interfacce possiamo gestirla.

Caratteristiche avanzate: INTERFACCE

Classe Astratta

- ❖ Si definisce con la parola chiave **abstract** *class*;
- ❖ I metodi con modificatori `public`, `protected`, `private`;
- ❖ Può dichiarare campi che non sono costanti;
- ❖ Si utilizza la parola chiave **extends**;
- ❖ Può contenere metodi non astratti;
- ❖ Ha un costruttore;

Interfaccia

- ❖ Si definisce con la parola chiave **interface**;
- ❖ I metodi hanno tutti modificatore **public**;
- ❖ Non può contenere metodi non astratti;
- ❖ Si utilizza la parola chiave **implements**;
- ❖ Si possono definire solo costanti;
- ❖ Non ha costruttore;

Caratteristiche avanzate: INTERFACCE

```
public interface DatabaseRepository {

    /**
     * Metodo per la ricerca di tutte le righe dal DB.
     * @return
     */
    public List<Row> findAll();

    /**
     * Metodo per la ricerca di una riga per ID.
     * @param id
     * @return
     */
    public String findById(int id);

    /**
     * Metodo per la creazione di una nuova riga.
     * @param id
     * @param row
     * @return
     */
    public String insert(Row row);

    /**
     * Metodo per l'aggiornamento di una riga.
     * @param id
     * @param row
     * @return
     */
    public String update(Row row);

    /**
     * Metodo per la cancellazione di una riga.
     * @param id
     * @return
     */
    public boolean deleteById(int id);

}
```

```
public class OracleDatabaseRepository implements DatabaseRepository {

    @Override
    public List<Row> findAll() {
        // TODO Auto-generated method stub
        return null;
    }

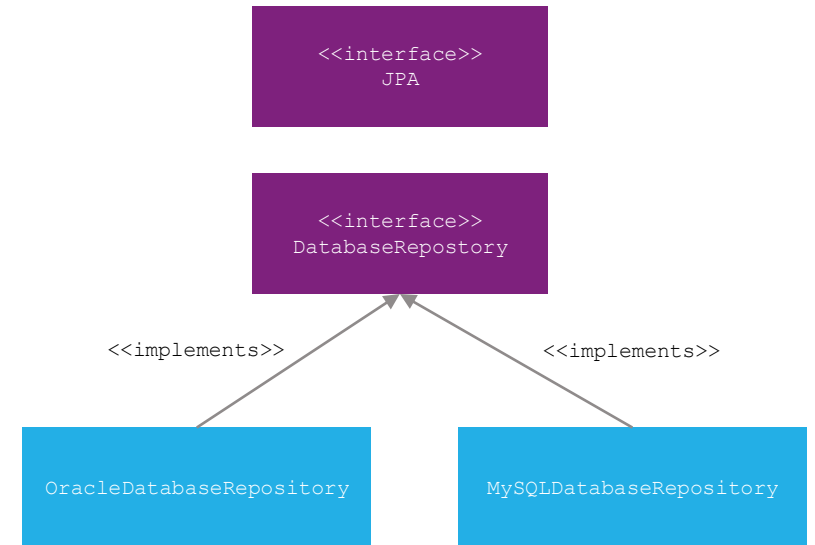
    @Override
    public String findById(int id) {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public String insert(Row row) {
        // TODO Auto-generated method stub
        return null;
    }

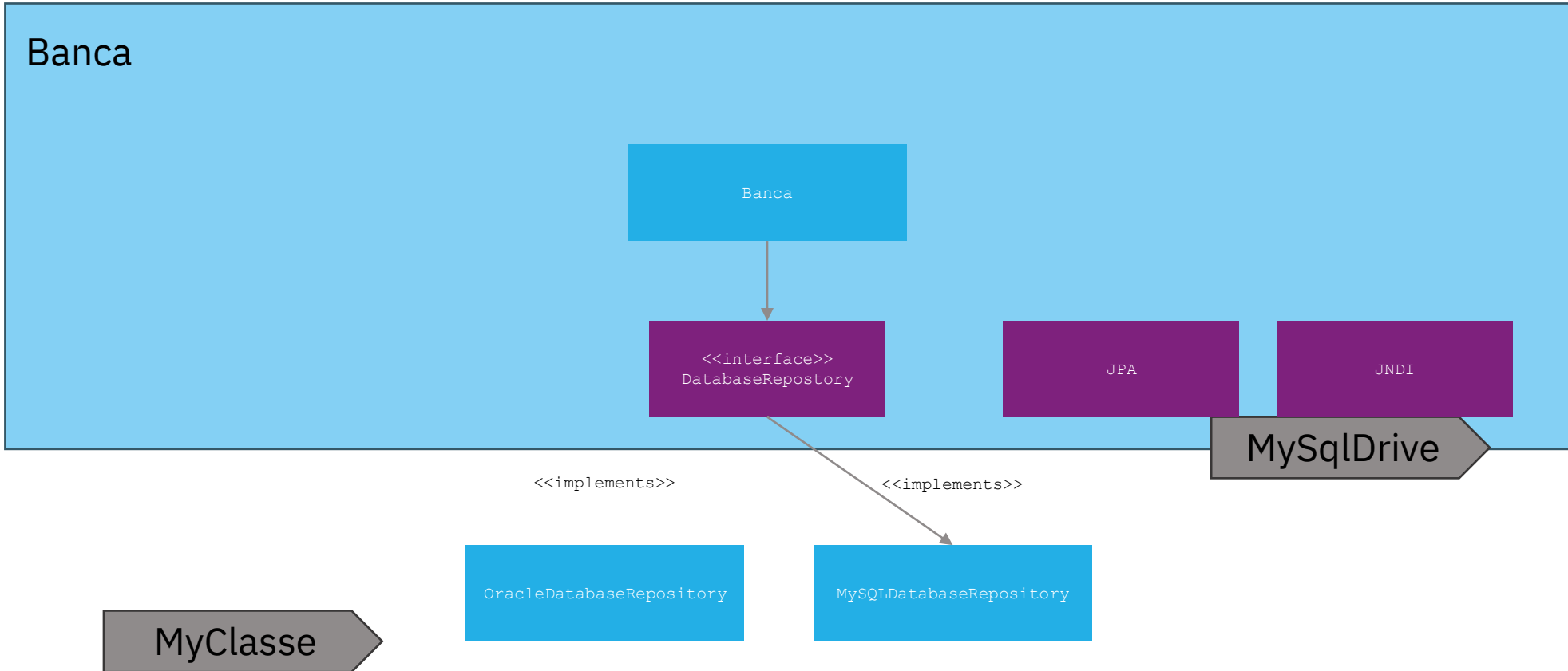
    @Override
    public String update(Row row) {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public boolean deleteById(int id) {
        // TODO Auto-generated method stub
        return false;
    }

}
```



Caratteristiche avanzate: INTERFACCE



Esercitazione Java

Creare un piccolo progetto Java (**JavaAcademyFirstProject**) che consente di effettuare due calcoli:

- La somma degli elementi di un array di interi;
- La media degli elementi di un array di interi;
- Di trasformare tutti gli elementi di un array in uppercase;

Le operazioni matematiche dovranno essere messe a disposizione da una classe **MathUtils** che espone i seguenti metodi:

- `public int computeSum(Integer[] numbers);`
- `public int computeAvg(Integer[] numbers);`

Le operazioni sulle stringhe dovranno essere messe a disposizione da una classe **StringUtils** che espone il seguente metodo:

- `public List<String> toUpper(List<String> stringList).`

Testare la logica del programma con una classe **Test.java** che contiene un solo metodo, il main.