Specifiche di Java

Programmazione

Java è un linguaggio di programmazione ad oggetti, fortemente tipizzato, compilato e multipiattaforma, progettato per garantire portabilità, sicurezza e robustezza.

È stato sviluppato da Sun Microsystems nel 1995 (poi acquisita da Oracle Corporation) con l'obiettivo di creare applicazioni eseguibili su qualunque sistema tramite il principio:

"Write Once, Run Anywhere" (WORA)

Architettura tecnica

Java non compila direttamente in codice macchina nativo, ma segue questo flusso:

1. **Codice sorgente (.java)**
2. **Compilazione tramite javac**
3. **Generazione di bytecode (.class)**
4. **Esecuzione sulla Java Virtual Machine (JVM)**

La JVM interpreta o compila Just-In-Time (JIT) il bytecode in codice macchina specifico per il sistema operativo.

Object-Oriented (Orientato agli Oggetti)

Java è un linguaggio fortemente orientato agli oggetti. Tutto ruota attorno a classi e oggetti.
Supporta pienamente i quattro pilastri OOP: incapsulamento, ereditarietà, polimorfismo e astrazione.
Questo consente modularità, riusabilità e manutenibilità del codice.

Esempio

```
1. class Persona {
2.     String nome;
3.
4.     void saluta() {
5.         System.out.println("Ciao, sono " + nome);
6.     }
7. }
8.
9. public class Main {
10.    public static void main(String[] args) {
11.        Persona p = new Persona();
12.        p.nome = "Mirko";
13.        p.saluta();
14.    }
15. }
```

Platform Independent (Write Once, Run Anywhere)

Il codice Java non viene compilato direttamente in codice macchina, ma in bytecode, eseguito dalla JVM (Java Virtual Machine).
Questo permette di eseguire lo stesso programma su qualsiasi sistema operativo che abbia una JVM.

Esempio

.java → compilazione (javac) → .class (bytecode) → JVM → esecuzione

```
1. public class Test {
2.     public static void main(String[] args) {
3.         System.out.println("Funziona su qualsiasi OS con JVM!");
4.     }
5. }
```

Strongly Typed (Tipizzazione Forte)

Java è un linguaggio fortemente tipizzato: ogni variabile deve avere un tipo dichiarato e non può cambiare tipo durante l'esecuzione.
Questo riduce errori a runtime e migliora la sicurezza del codice.

Esempio

```
1. int numero = 10;
2. // numero = "ciao"; // ERRORE di compilazione
3.
4. String testo = "Hello";
```

Garbage Collection Automatica

Java gestisce automaticamente la memoria tramite il Garbage Collector, che libera gli oggetti non più referenziati.
Questo evita memory leak tipici di linguaggi come C/C++.

Esempio

```
1. public class Esempio {
2.     public static void main(String[] args) {
3.         String s = new String("Java");
4.         s = null; // Oggetto pronto per il GC
5.     }
6. }
```

Multithreading Nativo

Java supporta il multithreading a livello di linguaggio.
È possibile creare thread in modo semplice, permettendo esecuzione concorrente e migliore utilizzo della CPU.

Esempio

```
class MioThread extends Thread {
    public void run() {
        System.out.println("Thread in esecuzione");
    }
}

public class Main {
    public static void main(String[] args) {
        MioThread t = new MioThread();
        t.start();
    }
}
```

Sicurezza (Security Model)

Java è progettato con un forte modello di sicurezza:

- Nessun accesso diretto alla memoria
- Verifica del bytecode
- Class loader
- Security Manager (storicamente)

Esempio

1. **private int saldo;**

Standard Library Molto Ampia

Java possiede una libreria standard ricca: Collections, Stream API, Networking, I/O, Concurrency, JDBC, ecc.
Questo riduce la necessità di librerie esterne per molti casi d'uso.

Esempio

```
1. import java.util.ArrayList;
2.
3. public class Main {
4.     public static void main(String[] args) {
5.         ArrayList<String> lista = new ArrayList<>();
6.         lista.add("Java");
7.         lista.add("OOP");
8.         System.out.println(lista);
9.     }
10. }
```

Caratteristica	Perché è importante
OOP	Modularità e riusabilità
JVM	Portabilità
Tipizzazione forte	Sicurezza
Garbage Collector	Gestione memoria automatica
Multithreading	Concorrenza
Security model	Robustezza
Librerie standard	Produttività

Generics (Tipizzazione Parametrica)

I Generics permettono di scrivere codice riusabile e type-safe evitando cast espliciti. Introdotti in Java 5, abilitano polimorfismo parametrico e riducono errori a runtime.

Consentono:

- **Type safety a compile time**
- **Eliminazione dei cast**
- **Riutilizzo di strutture dati**

Esempio

```
1. import java.util.List;
2. import java.util.ArrayList;
3.
4. public class Main {
5.     public static void main(String[] args) {
6.         List<String> lista = new ArrayList<>();
7.         lista.add("Java");
8.         // lista.add(10); // Errore di compilazione
9.     }
10. }
```

Stream API (Programmazione Funzionale)

Introdotta in Java 8, la Stream API consente elaborazioni dichiarative su collezioni usando concetti funzionali:

- **map**
- **filter**
- **reduce**
- **lambda expressions**

Esempio

```
1. import java.util.List;
2.
3. public class Main {
4.     public static void main(String[] args) {
5.         List<Integer> numeri = List.of(1,2,3,4,5);
6.
7.         numeri.stream()
8.                 .filter(n -> n % 2 == 0)
9.                 .map(n -> n * 2)
10.                .forEach(System.out::println);
11.    }
12. }
```

Lambda Expressions

Le lambda implementano funzioni anonime compatibili con interfacce funzionali (Single Abstract Method).
Abilitano programmazione funzionale e riducono boilerplate.

Esempio

```
1. Runnable r = () -> System.out.println("Esecuzione lambda");
2. new Thread(r).start();
```

Reflection API

La Reflection consente di analizzare e manipolare classi, metodi e campi a runtime.
È alla base di framework come Spring e Hibernate.

Permette:

- **Ispezione metadati**
- **Dependency Injection**
- **Dynamic proxy**

Esempio

```
1. import java.lang.reflect.Method;
2.
3. public class Main {
4.     public static void main(String[] args) throws Exception {
5.         Class<?> c = String.class;
6.         Method[] methods = c.getMethods();
7.         System.out.println(methods.length);
8.     }
9. }
```

Concurrency Framework (java.util.concurrent)

Oltre ai thread base, Java offre un framework avanzato per:

- ExecutorService
- Future
- Callable
- Lock
- Concurrent collections

Esempio

```
1. import java.util.concurrent.*;
2.
3. public class Main {
4.     public static void main(String[] args) throws Exception {
5.         ExecutorService executor = Executors.newFixedThreadPool(2);
6.
7.         Future<Integer> result = executor.submit(() -> 10 + 20);
8.
9.         System.out.println(result.get());
10.        executor.shutdown();
11.    }
12. }
```

NIO (New I/O)

Il package java.nio introduce:

- Buffer
- Channel
- Selectors
- I/O non bloccante

Fondamentale per sistemi ad alta scalabilità (server, networking).

Esempio

```
1. import java.nio.file.*;
2. import java.io.IOException;
3.
4. public class Main {
5.     public static void main(String[] args) throws IOException {
6.         String content = Files.readString(Path.of("file.txt"));
7.         System.out.println(content);
8.     }
9. }
```

Records (Java 16+)

I record sono classi immutabili concise pensate per Data Carrier. Generano automaticamente:

- costruttore
- getter
- equals()
- hashCode()
- toString()

Esempio

```
1. public record Persona(String nome, int eta) {}
2.
3. public class Main {
4.     public static void main(String[] args) {
5.         Persona p = new Persona("Mirko", 35);
6.         System.out.println(p.nome());
7.     }
8. }
```

Area	Feature
Type System	Generics, Records
Functional	Lambda, Stream API
Runtime	Reflection
Concurrency	Executor, Virtual Threads
Architettura	Modularity
I/O	NIO