Indice dei Contenuti

1. Introduzione al Corso di Paradigmi di Programmazione

Descrizione Generale del Corso

Il corso di "Paradigmi di Programmazione" (o "Altri Paradigmi di Programmazione") tenuto dal Prof. Michele Mauro si propone di esplorare i diversi approcci alla programmazione oltre il paradigma imperativo tradizionale. Il corso si concentra in particolare sui paradigmi di programmazione moderni utilizzati nello sviluppo di applicazioni complesse, con un'enfasi particolare su Java e le sue estensioni reattive.

L'obiettivo principale è fornire agli studenti una comprensione approfondita dei diversi modi di strutturare e concepire i programmi, analizzando vantaggi e svantaggi di ciascun approccio e le situazioni in cui risultano più appropriati.

Obiettivi Formativi

Al termine del corso, lo studente sarà in grado di:

1. **Comprendere e distinguere** i diversi paradigmi di programmazione (object-oriented, funzionale, reattivo)
2. **Padroneggiare** le caratteristiche avanzate del linguaggio Java, incluse le funzionalità introdotte nelle versioni recenti
3. **Applicare** i principi della programmazione concorrente e parallela
4. **Utilizzare** le API Stream di Java per l'elaborazione di dati
5. **Implementare** soluzioni basate su Reactive Extensions
6. **Progettare** applicazioni di rete e sistemi distribuiti
7. **Analizzare** problemi complessi e scegliere il paradigma più adatto alla loro risoluzione

Metodologia Didattica

Il corso combina lezioni teoriche con esercitazioni pratiche. Durante le lezioni vengono presentati i concetti fondamentali di ciascun paradigma, mentre le esercitazioni permettono agli studenti di applicare tali concetti attraverso la scrittura di codice e la risoluzione di problemi concreti.

Particolare attenzione viene dedicata all'analisi di casi di studio reali, per mostrare come i diversi paradigmi vengano applicati nello sviluppo di software complessi e in contesti industriali.

Modalità d'Esame

L'esame consiste in una prova scritta che valuta la comprensione teorica degli argomenti trattati e la capacità di applicare i concetti appresi. La prova è strutturata in:

1. **Domande a risposta multipla**: verificano la conoscenza dei concetti fondamentali
2. **Domande a risposta aperta**: valutano la capacità di argomentare e collegare diversi concetti
3. **Esercizi pratici**: richiedono l'analisi di frammenti di codice o la scrittura di brevi implementazioni

Dall'analisi degli appelli d'esame forniti, si nota che le domande coprono tutti gli argomenti del corso, con particolare enfasi su:

- Caratteristiche avanzate di Java
- Concorrenza e sincronizzazione
- Stream API
- Reactive Extensions
- Programmazione di rete
- Sistemi distribuiti

La preparazione all'esame richiede non solo la conoscenza teorica dei concetti, ma anche la capacità di riconoscere e applicare i pattern appropriati in diversi contesti applicativi.

2. Paradigmi di Programmazione

Definizione e Concetti Fondamentali

Un paradigma di programmazione rappresenta un modello concettuale che definisce come strutturare e organizzare il codice di un programma. Ogni paradigma offre un approccio diverso alla risoluzione dei problemi e influenza profondamente il modo in cui pensiamo e progettiamo il software.

I paradigmi di programmazione possono essere visti come "stili" o "filosofie" di programmazione che determinano:

- Come vengono rappresentati i dati

- Come viene controllato il flusso di esecuzione
- Come viene organizzata la logica del programma
- Come vengono gestite le astrazioni

La scelta del paradigma più appropriato dipende dalla natura del problema da risolvere, dalle caratteristiche del dominio applicativo e dalle esigenze non funzionali (come prestazioni, manutenibilità, scalabilità).

Classificazione dei Paradigmi

I paradigmi di programmazione possono essere classificati in diverse categorie, spesso non mutuamente esclusive:

1. Paradigma Imperativo

- Basato sull'idea di cambiamento di stato attraverso l'esecuzione sequenziale di istruzioni
- Il programma è visto come una sequenza di comandi che modificano lo stato del sistema
- Esempi: C, Pascal, FORTRAN

2. Paradigma Dichiarativo

- Descrive "cosa" deve essere calcolato piuttosto che "come" calcolarlo
- Il programmatore specifica il risultato desiderato, non i passaggi per ottenerlo
- Sottocategorie principali:
    - **Funzionale**: basato sul concetto di funzione matematica (Haskell, Lisp)
    - **Logico**: basato sulla logica formale (Prolog)

3. Paradigma Object-Oriented

- Organizza il codice attorno al concetto di "oggetto", che combina dati e comportamenti
- Principi fondamentali: incapsulamento, ereditarietà, polimorfismo
- Esempi: Java, C++, Python, C#

4. Paradigma Reattivo

- Focalizzato sulla gestione di flussi di dati asincroni e sulla propagazione dei cambiamenti
- Particolarmente adatto per applicazioni event-driven e sistemi distribuiti
- Implementazioni: Reactive Extensions (Rx), Akka

5. Paradigma Concorrente

- Gestisce l'esecuzione simultanea di più processi o thread
- Si concentra sulla sincronizzazione e comunicazione tra processi
- Modelli: attori, CSP (Communicating Sequential Processes)

6. Paradigma Event-Driven

- Il flusso del programma è determinato da eventi (input utente, messaggi da altri programmi)
- Ampiamente utilizzato nelle interfacce grafiche e nei sistemi distribuiti

Evoluzione Storica dei Paradigmi

L'evoluzione dei paradigmi di programmazione riflette sia l'avanzamento della teoria informatica sia le esigenze pratiche dello sviluppo software:

1940-1950: Programmazione Macchina e Assembly

- Programmazione a basso livello, direttamente in linguaggio macchina o assembly
- Forte accoppiamento con l'hardware specifico

1950-1960: Linguaggi Procedurali

- Nascita dei primi linguaggi di alto livello (FORTRAN, COBOL)
- Paradigma imperativo con focus sulle procedure

1960-1970: Programmazione Strutturata

- Enfasi su strutture di controllo ben definite (sequenza, selezione, iterazione)
- Eliminazione dei "goto" incontrollati (Dijkstra, "Go To Statement Considered Harmful")
- Linguaggi: Pascal, C

1970-1980: Programmazione Modulare e Astratta

- Introduzione di moduli e tipi di dati astratti
- Maggiore enfasi sull'incapsulamento
- Linguaggi: Modula-2, Ada

1980-1990: Programmazione Object-Oriented

- Diffusione dei concetti OOP: oggetti, classi, ereditarietà
- Linguaggi: Smalltalk, C++

1990-2000: Linguaggi Ibridi e Internet

- Combinazione di paradigmi diversi in singoli linguaggi
- Nascita di Java (1995) con focus su portabilità e sicurezza
- Crescita delle applicazioni web

2000-2010: Rinascita del Funzionale e Concorrenza

- Riscoperta della programmazione funzionale per gestire la concorrenza
- Introduzione di costrutti funzionali in linguaggi mainstream
- Linguaggi: Scala, F#, estensioni funzionali in Java

2010-Presente: Paradigma Reattivo e Cloud

- Focus su scalabilità, resilienza e reattività
- Pubblicazione del Reactive Manifesto (2013)
- Microservizi e architetture distribuite
- Programmazione asincrona e basata su eventi

Multiparadigma in Java

Java è nato come linguaggio principalmente object-oriented, ma nel corso degli anni ha incorporato elementi di altri paradigmi:

Java come Linguaggio OOP

- Tutto è un oggetto (eccetto i tipi primitivi)
- Forte tipizzazione
- Ereditarietà singola con interfacce multiple
- Polimorfismo

Elementi Funzionali in Java

- Lambda expressions (Java 8)
- Stream API per elaborazione dati (Java 8)
- Optional per gestione null-safety (Java 8)
- Method references (Java 8)

Elementi Reattivi in Java

- CompletableFuture per computazioni asincrone (Java 8)
- Flow API per reactive streams (Java 9)
- Librerie esterne come RxJava

Elementi Concorrenti in Java

- Thread e Runnable fin da Java 1
- Executor framework (Java 5)
- Fork/Join framework (Java 7)
- Parallel streams (Java 8)

La capacità di Java di incorporare diversi paradigmi lo rende un linguaggio versatile, in grado di adattarsi a diversi contesti applicativi e di evolvere con le esigenze dell'industria del software.

3. Programmazione Object-Oriented in Java

Concetti Fondamentali di OOP

La programmazione orientata agli oggetti (OOP) è un paradigma che organizza il software attorno a "oggetti", entità che combinano dati (attributi) e comportamenti (metodi). Java è stato progettato fin dall'inizio come linguaggio fortemente orientato agli oggetti, implementando tutti i principi fondamentali di questo paradigma.

Principi Fondamentali dell'OOP in Java

1. Incapsulamento

L'incapsulamento è il meccanismo che permette di nascondere i dettagli implementativi di una classe, esponendo solo le funzionalità necessarie attraverso un'interfaccia ben definita.

In Java, l'incapsulamento si realizza attraverso:

- Modificatori di accesso (`private` , `protected` , `public` , default/package)
- Metodi getter e setter per controllare l'accesso ai campi
- Classi interne per nascondere implementazioni ausiliarie

Esempio:

```
public class ContoBancario {
    private double saldo;  // Campo privato, non accessibile dall'esterno

    public double getSaldo() {
        return saldo;
    }

    public void deposita(double importo) {
        if (importo > 0) {
            saldo += importo;
        }
    }

    public boolean preleva(double importo) {
        if (importo > 0 && saldo >= importo) {
            saldo -= importo;
            return true;
        }
        return false;
    }
}
```

2. Astrazione

L'astrazione permette di modellare concetti complessi focalizzandosi sugli aspetti essenziali e ignorando i dettagli non rilevanti. In Java, l'astrazione si realizza principalmente attraverso:

- Classi astratte
- Interfacce
- Metodi astratti

Esempio:

```
public abstract class Veicolo {
    private String targa;

    public abstract void accelera();
    public abstract void frena();

    public String getTarga() {
        return targa;
    }
}

public class Automobile extends Veicolo {
    @Override
    public void accelera() {
        // Implementazione specifica per automobile
    }

    @Override
    public void frena() {
        // Implementazione specifica per automobile
    }
}
```

3. Ereditarietà

L'ereditarietà permette di definire una nuova classe basata su una classe esistente, ereditandone attributi e metodi. In Java:

- L'ereditarietà si realizza con la keyword `extends`
- Java supporta solo l'ereditarietà singola (una classe può estendere una sola classe)
- Tutte le classi ereditano implicitamente da `Object`

Esempio:

```
public class Persona {
    protected String nome;
    protected String cognome;

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

    public String getNomeCompleto() {
        return nome + " " + cognome;
    }
}

public class Studente extends Persona {
    private String matricola;

    public Studente(String nome, String cognome, String matricola) {
        super(nome, cognome);  // Chiamata al costruttore della classe padre
        this.matricola = matricola;
    }

    public String getMatricola() {
        return matricola;
    }
}
```

4. Polimorfismo

Il polimorfismo permette a oggetti di classi diverse di rispondere allo stesso messaggio in modi diversi. In Java, il polimorfismo si manifesta principalmente in due forme:

Polimorfismo in fase di compilazione (overloading):

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

    public double somma(double a, double b) {
        return a + b;
    }

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

Polimorfismo in fase di esecuzione (override):

```
public class Forma {
    public double calcolaArea() {
        return 0;
    }
}

public class Cerchio extends Forma {
    private double raggio;

    public Cerchio(double raggio) {
        this.raggio = raggio;
    }

    @Override
    public double calcolaArea() {
        return Math.PI * raggio * raggio;
    }
}

public class Rettangolo extends Forma {
    private double base;
    private double altezza;

    public Rettangolo(double base, double altezza) {
        this.base = base;
        this.altezza = altezza;
    }

    @Override
    public double calcolaArea() {
        return base * altezza;
    }
}
```

Ereditarietà e Polimorfismo in Java

Ereditarietà in Java

Java implementa l'ereditarietà con alcune caratteristiche specifiche:

1. **Ereditarietà singola**: Una classe può estendere una sola classe padre
2. **Ereditarietà multipla di interfacce**: Una classe può implementare più interfacce
3. **Gerarchia di classi**: Tutte le classi derivano da `Object`
4. **Modificatore `final`** : Impedisce l'estensione di una classe

Ordine di Inizializzazione nell'Ereditarietà

Quando si istanzia una classe derivata, l'ordine di esecuzione è il seguente:

1. Inizializzatori statici della classe padre (eseguiti al caricamento della classe)
2. Inizializzatori statici della classe figlia
3. Inizializzatori di istanza della classe padre
4. Costruttore della classe padre
5. Inizializzatori di istanza della classe figlia
6. Costruttore della classe figlia

Esempio tratto dagli appelli d'esame:

```
class Foo {
    Foo(int a) {
        // Costruttore Foo
    }
}

class Bar extends Foo {
    static {
        // Inizializzatore statico
    }
    {
        // Inizializzatore di istanza
    }
    Bar(int a, String b) {
        super(a);
        // Costruttore Bar
    }
}
```

Ordine di esecuzione:

1. Inizializzatore statico di Bar
2. Costruttore Foo
3. Inizializzatore di istanza di Bar
4. Costruttore Bar

Diamond Problem

Il Diamond Problem (o problema del diamante) si verifica quando una classe eredita da due classi che a loro volta ereditano da una classe comune, creando un'ambiguità su quale implementazione ereditare.

Java evita questo problema con l'ereditarietà singola, ma l'introduzione dei default methods nelle interfacce (Java 8) ha reintrodotto una forma di Diamond Problem:

```
interface A {
    default void metodo() {
        System.out.println("A");
    }
}

interface B extends A {
    default void metodo() {
        System.out.println("B");
    }
}

interface C extends A {
    default void metodo() {
        System.out.println("C");
    }
}

// Errore di compilazione: metodo() è definito in entrambe le interfacce B e C
class D implements B, C {
    // Soluzione: sovrascrivere il metodo
    @Override
    public void metodo() {
        B.super.metodo(); // Scelta esplicita dell'implementazione
    }
}
```

Interfacce e Classi Astratte

Interfacce

Un'interfaccia in Java definisce un contratto che le classi implementanti devono rispettare. Caratteristiche principali:

- Può contenere solo costanti (`public static final`) e dichiarazioni di metodi (implicitamente `public abstract`)

- Da Java 8: può contenere metodi default e metodi statici
- Da Java 9: può contenere metodi privati
- Una classe può implementare più interfacce
- Un'interfaccia può estendere più interfacce

```
public interface Volante {
    void decolla();
    void atterra();

    // Metodo default (Java 8+)
    default void vola() {
        System.out.println("Sto volando");
    }

    // Metodo statico (Java 8+)
    static boolean puoVolare(Object oggetto) {
        return oggetto instanceof Volante;
    }

    // Metodo privato (Java 9+)
    private void controlliPreVolo() {
        System.out.println("Controlli pre-volo completati");
    }
}
```

Default Methods

I metodi default nelle interfacce, introdotti in Java 8, permettono di aggiungere nuove funzionalità alle interfacce senza rompere la compatibilità con il codice esistente.

Come evidenziato in uno degli appelli d'esame:

> "Perché nel linguaggio Java si è deciso di introdurre i metodi di default nelle Interfacce?"
>
> Risposta: "Per poter estendere delle interfacce consolidate senza richiedere l'aggiornamento del codice esistente."

Classi Astratte

Una classe astratta è una classe che non può essere istanziata direttamente e che può contenere metodi astratti (senza implementazione). Caratteristiche principali:

- Dichiarata con la keyword `abstract`
- Può contenere metodi astratti e metodi concreti
- Può contenere campi di istanza e costruttori
- Una classe può estendere una sola classe astratta

```
public abstract class Animale {
    protected String nome;

    public Animale(String nome) {
        this.nome = nome;
    }

    public abstract void emettiSuono();

    public void mangia() {
        System.out.println(nome + " sta mangiando");
    }
}

public class Cane extends Animale {
    public Cane(String nome) {
        super(nome);
    }

    @Override
    public void emettiSuono() {
        System.out.println("Bau!");
    }
}
```

Differenze tra Interfacce e Classi Astratte

Caratteristica	Interfaccia	Classe Astratta
Istanziazione	Non istanziabile	Non istanziabile
Ereditarietà multipla	Supportata	Non supportata
Campi	Solo costanti	Qualsiasi tipo di campo
Costruttori	Non ammessi	Ammessi
Metodi	Astratti, default, statici, privati	Qualsiasi tipo di metodo
Modificatori di accesso	Implicitamente public	Qualsiasi modificatore
Scopo principale	Definire un contratto	Fornire una base comune

Gestione delle Eccezioni

Java utilizza un sistema di gestione delle eccezioni basato su oggetti, dove le eccezioni sono istanze di classi che ereditano da `Throwable`.

Gerarchia delle Eccezioni

- `Throwable`: classe base per tutte le eccezioni
  - `Error`: errori gravi, tipicamente non recuperabili (es. `OutOfMemoryError`)
  - `Exception`: eccezioni recuperabili
    - `RuntimeException`: eccezioni non controllate (es. `NullPointerException`)
    - Altre eccezioni: eccezioni controllate (es. `IOException`)

Eccezioni Controllate vs Non Controllate

- **Eccezioni controllate**: devono essere dichiarate o gestite (con `throws` o `try-catch`)
- **Eccezioni non controllate**: non richiedono dichiarazione o gestione esplicita

Blocchi try-catch-finally

```
try {
    // Codice che potrebbe generare un'eccezione
    File file = new File("documento.txt");
    FileReader fr = new FileReader(file);
} catch (FileNotFoundException e) {
    // Gestione dell'eccezione
    System.err.println("File non trovato: " + e.getMessage());
} finally {
    // Codice eseguito sempre, indipendentemente dalle eccezioni
    System.out.println("Operazione completata");
}
```

Try-with-resources (Java 7+)

```
try (FileReader fr = new FileReader("documento.txt");
     BufferedReader br = new BufferedReader(fr)) {
    String line = br.readLine();
    // Elaborazione del file
} catch (IOException e) {
    System.err.println("Errore di I/O: " + e.getMessage());
}
// Le risorse vengono chiuse automaticamente
```

Multi-catch (Java 7+)

```
try {
    // Codice che potrebbe generare diverse eccezioni
} catch (IOException | SQLException e) {
    // Gestione unificata di eccezioni diverse
    System.err.println("Errore: " + e.getMessage());
}
```

Generics

I generics, introdotti in Java 5, permettono di creare classi, interfacce e metodi che operano su tipi parametrizzati, migliorando la type-safety e riducendo la necessità di cast espliciti.

Vantaggi dei Generics

1. **Type safety**: errori rilevati in fase di compilazione anziché a runtime
2. **Eliminazione dei cast**: riduzione del codice boilerplate
3. **Algoritmi generici**: implementazioni riutilizzabili per tipi diversi

Dichiarazione di Classi Generiche

```
public class Box<T> {
    private T contenuto;

    public void inserisci(T oggetto) {
        this.contenuto = oggetto;
    }

    public T estrai() {
        return contenuto;
    }
}

// Utilizzo
Box<String> scatolaDiStringhe = new Box<>();
scatolaDiStringhe.inserisci("Hello");
String s = scatolaDiStringhe.estrai(); // Nessun cast necessario
```

Wildcard

Le wildcard permettono di gestire la varianza dei tipi generici:

- `<?>` : wildcard non limitata, rappresenta "qualsiasi tipo"
- `<? extends T>` : wildcard limitata superiormente, rappresenta "T o un sottotipo di T"
- `<? super T>` : wildcard limitata inferiormente, rappresenta "T o un supertipo di T"

```
// Metodo che accetta liste di qualsiasi tipo
public void stampaLista(List<?> lista) {
    for (Object elemento : lista) {
        System.out.println(elemento);
    }
}

// Metodo che accetta liste di numeri o sottotipi di numeri
public double somma(List<? extends Number> numeri) {
    double somma = 0;
    for (Number n : numeri) {
        somma += n.doubleValue();
    }
    return somma;
}

// Metodo che accetta liste di Integer o supertipi di Integer
public void aggiungiInteri(List<? super Integer> lista) {
    lista.add(1);
    lista.add(2);
}
```

Type Erasure

In Java, le informazioni sui tipi generici sono disponibili solo in fase di compilazione e vengono "cancellate" durante la compilazione (type erasure). Questo significa che a runtime, una `List<String>` e una `List<Integer>` sono rappresentate allo stesso modo.

Questo comportamento è stato adottato per garantire la compatibilità con il codice pre-generics, ma introduce alcune limitazioni:

1. Non è possibile creare istanze di tipi generici: `new T()` non è consentito
2. Non è possibile creare array di tipi generici: `new T[10]` non è consentito
3. Non è possibile usare `instanceof` con tipi generici: `obj instanceof List<String>` non è consentito

Lambda Expressions

Le lambda expressions, introdotte in Java 8, permettono di trattare funzionalità come argomenti di metodo, implementando il paradigma della programmazione funzionale in Java.

Sintassi delle Lambda Expressions

```
// Sintassi generale
(parametri) -> espressione
(parametri) -> { istruzioni; }

// Esempi
Runnable r = () -> System.out.println("Hello");
Comparator<String> c = (s1, s2) -> s1.length() - s2.length();
Consumer<String> consumer = s -> {
    System.out.println("Elaborazione di: " + s);
    System.out.println("Lunghezza: " + s.length());
};
```

Interfacce Funzionali

Un'interfaccia funzionale è un'interfaccia con un solo metodo astratto. Le lambda expressions possono essere utilizzate per implementare interfacce funzionali.

Java 8 ha introdotto l'annotazione `@FunctionalInterface` per marcare e verificare le interfacce funzionali:

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}

// Utilizzo con lambda
Predicate<String> isEmpty = s -> s.isEmpty();
boolean risultato = isEmpty.test("Hello"); // false
```

Interfacce Funzionali Standard

Java 8 ha introdotto diverse interfacce funzionali standard nel package `java.util.function`:

- `Function<T, R>`: trasforma un input di tipo T in un output di tipo R
- `Predicate<T>`: valuta una condizione su un input di tipo T, restituendo un boolean
- `Consumer<T>`: consuma un input di tipo T senza restituire nulla
- `Supplier<T>`: fornisce un risultato di tipo T senza prendere input
- `BinaryOperator<T>`: combina due input di tipo T in un output di tipo T

```
// Function
Function<String, Integer> lunghezza = s -> s.length();
Integer len = lunghezza.apply("Hello"); // 5

// Predicate
Predicate<String> isEmpty = String::isEmpty;
boolean vuota = isEmpty.test(""); // true

// Consumer
Consumer<String> printer = System.out::println;
printer.accept("Hello"); // Stampa "Hello"

// Supplier
Supplier<Double> random = Math::random;
Double valore = random.get(); // Numero casuale tra 0 e 1

// BinaryOperator
BinaryOperator<Integer> somma = (a, b) -> a + b;
Integer risultato = somma.apply(5, 3); // 8
```

Method References

I method references sono una forma abbreviata di lambda expression che fanno riferimento a metodi esistenti:

```
// Riferimento a metodo statico
Function<String, Integer> parser = Integer::parseInt;

// Riferimento a metodo di istanza su un oggetto specifico
Consumer<String> printer = System.out::println;

// Riferimento a metodo di istanza su un oggetto arbitrario del tipo specificato
Function<String, Integer> lunghezza = String::length;

// Riferimento a costruttore
Supplier<List<String>> factory = ArrayList::new;
```

Type Inference

La type inference è la capacità del compilatore di dedurre il tipo di un'espressione senza che sia necessario indicarlo esplicitamente.

In Java, la type inference è stata introdotta gradualmente:

- Java 5: inferenza dei tipi generici in chiamate di metodo
- Java 7: operatore diamond (`<>`) per la creazione di istanze generiche
- Java 8: inferenza dei tipi per i parametri delle lambda expressions
- Java 10: inferenza dei tipi per le variabili locali con `var`

Esempi di Type Inference

Java 5: Inferenza in chiamate di metodo

```
List<String> lista = Collections.emptyList(); // Il tipo è dedotto dal contesto
```

Java 7: Operatore Diamond

```
// Prima di Java 7
Map<String, List<String>> mappa = new HashMap<String, List<String>>();

// Con Java 7+
Map<String, List<String>> mappa = new HashMap<>(); // Il tipo è dedotto dalla dichiarazione
```

Java 8: Inferenza nei parametri lambda

```
Comparator<String> comp = (s1, s2) -> s1.length() - s2.length();
// I tipi di s1 e s2 sono dedotti come String
```

Java 10: Variabili locali con var

```
var lista = new ArrayList<String>(); // Il tipo di lista è dedotto come ArrayList<String>
var numero = 42; // Il tipo di numero è dedotto come int
var stringa = "Hello"; // Il tipo di stringa è dedotto come String
```

Come evidenziato in uno degli appelli d'esame:

> "Quando si dice che il compilatore Java ha delle capacità di Type Inference si intende che:"
>
> Risposta: "È in grado di dedurre il tipo di alcune espressioni senza che sia necessario indicarlo esplicitamente."

Default Methods

I default methods, introdotti in Java 8, permettono di aggiungere metodi con implementazione alle interfacce, senza rompere la compatibilità con il codice esistente.

Caratteristiche dei Default Methods

- Dichiarati con la keyword `default`
- Forniscono un'implementazione predefinita
- Possono essere sovrascritti dalle classi implementanti
- Permettono l'evoluzione delle interfacce senza rompere il codice esistente

```
public interface Collection<E> {
    // Metodi esistenti...

    // Nuovo metodo default aggiunto in Java 8
    default boolean removeIf(Predicate<? super E> filter) {
        boolean removed = false;
        Iterator<E> it = iterator();
        while (it.hasNext()) {
            if (filter.test(it.next())) {
                it.remove();
                removed = true;
            }
        }
        return removed;
    }
}
```

Risoluzione dei Conflitti

Quando una classe implementa più interfacce con default methods che hanno la stessa firma, si verifica un conflitto che deve essere risolto esplicitamente:

```
interface A {
    default void metodo() {
        System.out.println("A");
    }
}

interface B {
    default void metodo() {
        System.out.println("B");
    }
}

class C implements A, B {
    // Errore di compilazione se non si sovrascrive il metodo

    @Override
    public void metodo() {
        // Soluzione 1: chiamare l'implementazione di A
        A.super.metodo();

        // Soluzione 2: chiamare l'implementazione di B
        // B.super.metodo();

        // Soluzione 3: fornire una nuova implementazione
        // System.out.println("C");
    }
}
```

Scopo dei Default Methods

Come evidenziato negli appelli d'esame, i default methods sono stati introdotti principalmente per:

> "Per poter estendere delle interfacce consolidate senza richiedere l'aggiornamento del codice esistente."

Questo ha permesso l'evoluzione delle interfacce della libreria standard di Java (come `Collection`, `List`, `Map`) con nuove funzionalità, senza rompere la compatibilità con il codice esistente.

Ad esempio, l'interfaccia `List` è stata arricchita con metodi come `sort`, `replaceAll` e `spliterator` in Java 8, senza richiedere modifiche alle classi che già implementavano `List`.

4. Concorrenza in Java

Thread e Processi

La concorrenza in Java si basa principalmente sul concetto di thread, che rappresenta un flusso di esecuzione all'interno di un processo. A differenza dei processi, che sono istanze indipendenti di programmi con spazi di memoria separati, i thread condividono lo stesso spazio di memoria all'interno di un processo.

Differenze tra Processi e Thread

Come evidenziato negli appelli d'esame:

> "Quale delle seguenti affermazioni riguardo ai rapporti fra Processi, Thread e Fiber è vera:"
>
> Risposta: "Le risorse dei Processi sono controllate dal Sistema Operativo, mentre all'interno dei Processi i Thread devono direttamente controllare il loro accesso. Le Fiber rendono esplicita la concorrenza con lo scopo di essere ancora più leggere dei Thread."

Caratteristica	Processo	Thread
Spazio di memoria	Indipendente	Condiviso con altri thread dello stesso processo
Comunicazione	Complessa (IPC)	Semplice (memoria condivisa)
Creazione	Costosa	Relativamente leggera
Cambio di contesto	Costoso	Più efficiente

Caratteristica	Processo	Thread

Ciclo di Vita di un Thread in Java

Un thread in Java può trovarsi in uno dei seguenti stati:

1. **NEW**: il thread è stato creato ma non ancora avviato
2. **RUNNABLE**: il thread è in esecuzione o pronto per essere eseguito
3. **BLOCKED**: il thread è bloccato in attesa di un monitor lock
4. **WAITING**: il thread è in attesa indefinita che un altro thread esegua una particolare azione
5. **TIMED_WAITING**: il thread è in attesa per un tempo specificato
6. **TERMINATED**: il thread ha completato la sua esecuzione

Come evidenziato negli appelli d'esame:

> "Un Thread esce dallo stato blocked quando:"
>
> Risposta: "Ottiene la risorsa di sistema che aveva richiesto."

> "Un Thread esce dallo stato timed waiting quando:"
>
> Risposta: "Ottiene il lock che stava aspettando, oppure viene interrotto o trascorre il timeout impostato."

Creazione e Avvio di Thread in Java

In Java, esistono due modi principali per creare un thread:

1. Estendendo la classe Thread

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

    public static void main(String[] args) {
        MioThread thread = new MioThread();
        thread.start(); // Avvia il thread
    }
}
```

2. Implementando l'interfaccia Runnable

```
public class MioRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Thread in esecuzione");
    }

    public static void main(String[] args) {
        Thread thread = new Thread(new MioRunnable());
        thread.start(); // Avvia il thread
    }
}
```

L'approccio con `Runnable` è generalmente preferito perché:

- Permette di estendere altre classi
- Separa il task (cosa fare) dal thread (come farlo)
- Facilita il riutilizzo del codice

Executor Framework

Introdotto in Java 5, l'Executor Framework fornisce un'astrazione di alto livello per la gestione dei thread, separando la creazione e la gestione dei thread dall'esecuzione dei task.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class EsempioExecutor {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(5);

        for (int i = 0; i < 10; i++) {
            final int taskId = i;
            executor.execute(() -> {
                System.out.println("Esecuzione task " + taskId + " su thread " + Thread.currentThread().getName());
            });
        }

        executor.shutdown(); // Termina l'executor dopo l'esecuzione dei task
    }
}
```

Future e CompletableFuture

La classe `Future` rappresenta il risultato di un'operazione asincrona:

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

public class EsempioFuture {
    public static void main(String[] args) throws Exception {
        ExecutorService executor = Executors.newSingleThreadExecutor();

        Future<Integer> future = executor.submit(() -> {
            Thread.sleep(1000);
            return 42;
        });

        System.out.println("Attendo il risultato...");
        Integer risultato = future.get(); // Blocca fino al completamento del task
        System.out.println("Risultato: " + risultato);

        executor.shutdown();
    }
}
```

Come evidenziato negli appelli d'esame:

> "Un oggetto Future rappresenta:"
>
> Risposta: "Un calcolo che potrebbe produrre un risultato dopo un certo tempo."

Java 8 ha introdotto `CompletableFuture`, che estende `Future` con funzionalità per la composizione di operazioni asincrone:

```
import java.util.concurrent.CompletableFuture;

public class EsempioCompletableFuture {
    public static void main(String[] args) {
        CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            return "Risultato";
        }).thenApply(s -> s + " elaborato")
          .thenApply(String::toUpperCase);

        future.thenAccept(System.out::println); // RISULTATO ELABORATO

        // Attendi il completamento per evitare la terminazione prematura del programma
        future.join();
    }
}
```

Sincronizzazione e Lock

La sincronizzazione è necessaria quando più thread accedono a risorse condivise, per evitare condizioni di race e garantire la consistenza dei dati.

Keyword synchronized

La keyword `synchronized` in Java può essere utilizzata in due modi:

1. Metodi sincronizzati

```
public class Contatore {
    private int valore = 0;

    public synchronized void incrementa() {
        valore++;
    }

    public synchronized int getValore() {
        return valore;
    }
}
```

2. Blocchi sincronizzati

```
public class Contatore {
    private int valore = 0;
    private final Object lock = new Object();

    public void incrementa() {
        synchronized(lock) {
            valore++;
        }
    }

    public int getValore() {
        synchronized(lock) {
            return valore;
        }
    }
}
```

Relazione happens-before

La keyword `synchronized` introduce una relazione di happens-before nel codice, che garantisce un ordinamento preciso delle operazioni tra thread.

Come evidenziato negli appelli d'esame:

> "Quando si dice che la parola chiave synchronized introduce una relazione di happens-before nel codice, si intende che:"
>
> Risposta: "Il compilatore viene istruito a garantire che il codice sorvegliato dalla parola chiave synchronized venga effettivamente eseguito prima del codice che lo segue, e da un solo Thread alla volta."

Questa relazione garantisce che:

1. Tutte le modifiche fatte da un thread prima di rilasciare un lock siano visibili a qualsiasi thread che acquisisce lo stesso lock successivamente
2. Solo un thread alla volta possa eseguire il codice protetto dal lock

Lock Espliciti

Java 5 ha introdotto l'interfaccia `Lock` nel package `java.util.concurrent.locks`, che offre funzionalità più flessibili rispetto alla keyword `synchronized`:

```
 import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class ContatoreConLock {
    private int valore = 0;
    private final Lock lock = new ReentrantLock();

    public void incrementa() {
        lock.lock();
        try {
            valore++;
        } finally {
            lock.unlock(); // Importante: rilasciare sempre il lock in un blocco finally
        }
    }

    public int getValore() {
        lock.lock();
        try {
            return valore;
        } finally {
            lock.unlock();
        }
    }
}
```

Vantaggi dei lock espliciti:

- Possibilità di tentare l'acquisizione senza bloccarsi (`tryLock()`)
- Supporto per l'interruzione durante l'attesa (`lockInterruptibly()`)
- Possibilità di specificare timeout per l'acquisizione
- Supporto per condizioni multiple

ReadWriteLock

L'interfaccia `ReadWriteLock` fornisce un lock che distingue tra operazioni di lettura e scrittura:

```
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class Cache {
    private final ReadWriteLock rwLock = new ReentrantReadWriteLock();
    private final Map<String, Object> data = new HashMap<>();

    public Object get(String key) {
        rwLock.readLock().lock(); // Più thread possono leggere contemporaneamente
        try {
            return data.get(key);
        } finally {
            rwLock.readLock().unlock();
        }
    }

    public void put(String key, Object value) {
        rwLock.writeLock().lock(); // Solo un thread può scrivere alla volta
        try {
            data.put(key, value);
        } finally {
            rwLock.writeLock().unlock();
        }
    }
}
```

Variabili Volatili

La keyword `volatile` garantisce che le letture e le scritture di una variabile avvengano direttamente nella memoria principale, bypassando le cache dei processori:

```
public class Flag {
    private volatile boolean running = true;

    public void stop() {
        running = false;
    }

    public void run() {
        while (running) {
            // Esegui operazioni
        }
    }
}
```

La keyword `volatile` garantisce:

1. Visibilità: le modifiche fatte da un thread sono immediatamente visibili agli altri thread
2. Ordinamento: le operazioni su variabili volatili non vengono riordinate dal compilatore o dalla JVM

Tuttavia, `volatile` non garantisce l'atomicità delle operazioni composte (es. `count++`).

Atomic Classes

Le classi del package `java.util.concurrent.atomic` forniscono operazioni atomiche su singoli valori, senza necessità di sincronizzazione esplicita.

Come evidenziato negli appelli d'esame:

> "Le classi del package java.concurrent.atomic:"
>
> Risposta: "Sono particolarmente efficienti in caso di modifica concorrente del dato che rappresentano perché usano (se disponibili) delle funzionalità fornite direttamente dall'hardware."

Queste classi utilizzano operazioni atomiche a livello hardware (come Compare-And-Swap) per garantire l'atomicità senza bloccare i thread.

```
import java.util.concurrent.atomic.AtomicInteger;

public class ContatoreAtomico {
    private final AtomicInteger valore = new AtomicInteger(0);

    public void incrementa() {
        valore.incrementAndGet(); // Operazione atomica
    }

    public int getValore() {
        return valore.get();
    }
}
```

Principali classi atomiche:

- `AtomicBoolean`, `AtomicInteger`, `AtomicLong`: per tipi primitivi
- `AtomicReference<V>`: per riferimenti a oggetti
- `AtomicIntegerArray`, `AtomicLongArray`, `AtomicReferenceArray<V>`: per array
- `AtomicMarkableReference<V>`, `AtomicStampedReference<V>`: per riferimenti con metadati

Operazioni Atomiche Composte

Le classi atomiche supportano anche operazioni composte atomiche:

```
AtomicInteger counter = new AtomicInteger(0);

// Incremento atomico
counter.incrementAndGet(); // equivalente a counter++

// Aggiornamento atomico condizionale
counter.updateAndGet(x -> x < 10 ? x + 1 : x);

// Aggiornamento atomico con calcolo del nuovo valore
counter.accumulateAndGet(5, (x, y) -> x + y); // equivalente a counter += 5
```

Problemi di Concorrenza

Race Condition

Una race condition si verifica quando il comportamento di un programma dipende dall'ordine di esecuzione dei thread, che non è prevedibile.

```
// Esempio di race condition
public class Contatore {
    private int valore = 0;

    public void incrementa() {
        valore++; // Non atomico: leggi, incrementa, scrivi
    }

    public int getValore() {
        return valore;
    }
}
```

Se due thread chiamano `incrementa()` contemporaneamente, il valore finale potrebbe essere 1 invece di 2.

Deadlock

Un deadlock si verifica quando due o più thread sono bloccati indefinitamente, ciascuno in attesa di risorse detenute dagli altri.

```
// Esempio di potenziale deadlock
public class RisorsaCondivisa {
    private final Object lock1 = new Object();
    private final Object lock2 = new Object();

    public void metodo1() {
        synchronized(lock1) {
            System.out.println("Metodo1: lock1 acquisito");
            try { Thread.sleep(100); } catch (InterruptedException e) {}

            synchronized(lock2) {
                System.out.println("Metodo1: lock2 acquisito");
            }
        }
    }

    public void metodo2() {
        synchronized(lock2) {
            System.out.println("Metodo2: lock2 acquisito");
            try { Thread.sleep(100); } catch (InterruptedException e) {}

            synchronized(lock1) {
                System.out.println("Metodo2: lock1 acquisito");
            }
        }
    }
}
```

Se un thread chiama `metodo1()` e un altro thread chiama `metodo2()` contemporaneamente, può verificarsi un deadlock.

Livelock

Un livelock è simile a un deadlock, ma i thread coinvolti cambiano continuamente stato senza fare progressi.

```
// Esempio di potenziale livelock
public class Persona {
    private String nome;
    private boolean devePassare = true;

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

    public boolean devePassare() {
        return devePassare;
    }

    public void setDevePassare(boolean devePassare) {
        this.devePassare = devePassare;
    }

    public void passaPersona(Persona altra) {
        while (devePassare) {
            System.out.println(nome + ": Per favore, passa tu prima");
            setDevePassare(false);
            try { Thread.sleep(1000); } catch (InterruptedException e) {}

            if (altra.devePassare()) {
                System.out.println(nome + ": Ok, passo io");
                setDevePassare(true);
            }
        }
    }
}
```

Starvation

La starvation si verifica quando un thread non riceve mai accesso alle risorse di cui ha bisogno, perché altri thread con priorità più alta le occupano continuamente.

```
// Esempio di potenziale starvation
public class Risorsa {
    private final Object lock = new Object();

    public void usaRisorsa(int priorita) {
        synchronized(lock) {
            System.out.println("Thread con priorità " + priorita + " sta usando la risorsa");
            try { Thread.sleep(1000); } catch (InterruptedException e) {}
        }
    }

    public static void main(String[] args) {
        Risorsa risorsa = new Risorsa();

        // Thread a bassa priorità
        Thread threadBasso = new Thread(() -> {
            while (true) {
                risorsa.usaRisorsa(1);
            }
        });
        threadBasso.setPriority(Thread.MIN_PRIORITY);

        // 10 thread ad alta priorità
        for (int i = 0; i < 10; i++) {
            Thread threadAlto = new Thread(() -> {
                while (true) {
                    risorsa.usaRisorsa(10);
                }
            });
            threadAlto.setPriority(Thread.MAX_PRIORITY);
            threadAlto.start();
        }

        threadBasso.start(); // Potrebbe non ottenere mai la risorsa
    }
}
```

Condizioni di Coffman

Le condizioni di Coffman sono un insieme di condizioni necessarie per l'instaurarsi di un deadlock in un sistema concorrente.

Come evidenziato negli appelli d'esame:

> "Selezionare quali delle seguenti sono condizioni di Coffman necessarie per l'instaurarsi di un deadlock."
>
> Risposta: "Possesso e attesa, Mutua Esclusione, Risorse non riassegnabili, Attesa circolare"

1. Mutua Esclusione

Le risorse coinvolte devono essere non condivisibili, cioè possono essere utilizzate da un solo thread alla volta.

2. Possesso e Attesa

Un thread deve mantenere almeno una risorsa mentre è in attesa di acquisirne altre.

3. Risorse non Riassegnabili

Le risorse non possono essere forzatamente sottratte ai thread che le detengono.

4. Attesa Circolare

Deve esistere una catena circolare di thread, ciascuno in attesa di una risorsa detenuta dal thread successivo nella catena.

Strategie per Prevenire i Deadlock

Per ogni condizione di Coffman, esiste una strategia per prevenire i deadlock:

1. **Mutua Esclusione**: Utilizzare algoritmi lock-free o wait-free
2. **Possesso e Attesa**: Acquisire tutte le risorse necessarie in un'unica operazione atomica
3. **Risorse non Riassegnabili**: Implementare meccanismi di preemption per le risorse
4. **Attesa Circolare**: Imporre un ordinamento globale nell'acquisizione delle risorse

Come evidenziato negli appelli d'esame:

> "Associare ad ogni condizione di Coffmann una strategia utile per rimuoverla."
>
> Risposta: "Attesa circolare → Ordinamento dell'acquisizione, Mutua Esclusione → Algoritmi lock-free, Risorse non riassegnabili → Pre-emption, Possesso e attesa → Assegnazione delle risorse transazionale"

Thread Safety

Una classe è thread-safe quando può essere utilizzata in modo sicuro in un ambiente multi-thread, senza necessità di sincronizzazione esterna.

Caratteristiche di una Classe Thread-Safe

1. **Stato interno protetto**: Tutte le variabili di istanza sono private e accessibili solo attraverso metodi sincronizzati
2. **Operazioni atomiche**: Le operazioni che modificano lo stato sono atomiche
3. **Visibilità garantita**: Le modifiche fatte da un thread sono visibili agli altri thread
4. **Invarianti preservati**: Le invarianti della classe sono mantenuti anche in presenza di accessi concorrenti

Classi Thread-Safe nella Libreria Standard

Java fornisce diverse classi thread-safe nella libreria standard:

- `Vector`, `Hashtable`: versioni thread-safe di `ArrayList` e `HashMap` (legacy)
- `ConcurrentHashMap`, `CopyOnWriteArrayList`: implementazioni moderne thread-safe
- `BlockingQueue`: interfaccia per code thread-safe con operazioni bloccanti
- `AtomicInteger`, `AtomicReference`: classi per operazioni atomiche su singoli valori

Classi Non Thread-Safe

Come evidenziato negli appelli d'esame:

> "Quale delle seguenti frasi è falsa per una struttura dati non Thread-Safe in caso di accesso concorrente:"
>
> Risposta: "È più performante nel caso generale"

Le strutture dati non thread-safe in caso di accesso concorrente:

- Possono dare risultati errati
- Possono trovarsi in uno stato inconsistente
- Possono lanciare eccezioni
- Sono generalmente meno costose in termini di cicli CPU rispetto alle versioni thread-safe, ma solo in caso di accesso non concorrente

Strategie per la Thread Safety

1. **Immutabilità**: Le classi immutabili sono intrinsecamente thread-safe
2. **Confinamento**: Limitare l'accesso a un oggetto a un singolo thread
3. **Sincronizzazione**: Utilizzare `synchronized`, `Lock` o altre primitive di sincronizzazione
4. **Delegazione**: Delegare la thread safety a classi thread-safe esistenti

```
// Esempio di classe immutabile (thread-safe)
public final class Punto {
    private final int x;
    private final int y;

    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public Punto trasla(int dx, int dy) {
        return new Punto(x + dx, y + dy); // Crea un nuovo oggetto
    }
}
```

Thread Pools e Fork/Join

Thread Pools

Un thread pool è un gruppo di thread worker riutilizzabili, che eseguono task sottomessi dall'applicazione.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class EsempioThreadPool {
    public static void main(String[] args) {
        // Crea un pool con un numero fisso di thread
        ExecutorService executor = Executors.newFixedThreadPool(5);

        for (int i = 0; i < 10; i++) {
            final int taskId = i;
            executor.execute(() -> {
                System.out.println("Task " + taskId + " eseguito da " + Thread.currentThread().getName());
            });
        }

        executor.shutdown();
    }
}
```

Tipi di thread pool in Java:

- `newFixedThreadPool(n)` : pool con un numero fisso di thread
- `newCachedThreadPool()` : pool che crea nuovi thread secondo necessità e riutilizza quelli esistenti
- `newSingleThreadExecutor()` : pool con un singolo thread
- `newScheduledThreadPool(n)` : pool che può eseguire task dopo un ritardo o periodicamente

Fork/Join Framework

Il Fork/Join Framework, introdotto in Java 7, è un'implementazione del pattern divide-et-impera per il calcolo parallelo.

```
 import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;

public class SommaArray extends RecursiveTask<Long> {
    private static final int SOGLIA = 1000;
    private final int[] array;
    private final int inizio;
    private final int fine;

    public SommaArray(int[] array, int inizio, int fine) {
        this.array = array;
        this.inizio = inizio;
        this.fine = fine;
    }

    @Override
    protected Long compute() {
        int lunghezza = fine - inizio;
        if (lunghezza <= SOGLIA) {
            // Calcolo diretto per array piccoli
            long somma = 0;
            for (int i = inizio; i < fine; i++) {
                somma += array[i];
            }
            return somma;
        } else {
            // Divide et impera per array grandi
            int medio = inizio + lunghezza / 2;
            SommaArray taskSinistra = new SommaArray(array, inizio, medio);
            SommaArray taskDestra = new SommaArray(array, medio, fine);

            taskDestra.fork(); // Esegui taskDestra in un altro thread
            long sommaSinistra = taskSinistra.compute(); // Esegui taskSinistra in questo thread
            long sommaDestra = taskDestra.join(); // Attendi il risultato di taskDestra

            return sommaSinistra + sommaDestra;
        }
    }

    public static void main(String[] args) {
        int[] array = new int[100000];
        for (int i = 0; i < array.length; i++) {
            array[i] = i + 1;
        }

        ForkJoinPool pool = new ForkJoinPool();
        long risultato = pool.invoke(new SommaArray(array, 0, array.length));
        System.out.println("Somma: " + risultato);
    }
}
```

Il Fork/Join Framework è particolarmente adatto per problemi che possono essere suddivisi ricorsivamente in sottoproblemi più piccoli, come l'elaborazione di array o alberi.

5. Stream API

Concetti Fondamentali

La Stream API, introdotta in Java 8, fornisce un modo dichiarativo per elaborare sequenze di elementi. Uno stream rappresenta una sequenza di elementi che supporta operazioni aggregate sequenziali o parallele.

Caratteristiche Principali degli Stream

1. **Non memorizzano dati**: Uno stream non è una struttura dati, ma piuttosto una vista sui dati di una fonte (come una collezione, un array, un generatore o un

canale I/O)
2. **Funzionali per natura**: Le operazioni su stream producono risultati senza modificare la fonte
3. **Lazy evaluation**: Molte operazioni su stream sono eseguite solo quando necessario
4. **Possibilmente illimitati**: Gli stream possono rappresentare sequenze infinite di elementi
5. **Consumabili**: Gli elementi di uno stream possono essere attraversati una sola volta

Creazione di Stream

Esistono diversi modi per creare uno stream:

Da Collezioni

```
List<String> lista = Arrays.asList("a", "b", "c");
Stream<String> stream = lista.stream();
```

Da Array

```
String[] array = {"a", "b", "c"};
Stream<String> stream = Arrays.stream(array);
```

Da Valori

```
Stream<String> stream = Stream.of("a", "b", "c");
```

Stream Infiniti

```
// Stream infinito di numeri casuali
Stream<Double> randomStream = Stream.generate(Math::random);

// Stream infinito di numeri interi consecutivi
Stream<Integer> intStream = Stream.iterate(0, n -> n + 1);
```

Operazioni Intermedie e Terminali

Le operazioni su stream si dividono in due categorie:

Operazioni Intermedie

Le operazioni intermedie restituiscono un nuovo stream e sono lazy (non eseguite finché non viene invocata un'operazione terminale).

Principali operazioni intermedie:

filter

Filtra gli elementi in base a un predicato:

```
Stream<String> stream = Stream.of("Java", "Python", "C++", "JavaScript");
Stream<String> filtered = stream.filter(s -> s.startsWith("J"));
// Risultato: "Java", "JavaScript"
```

map

Trasforma ogni elemento applicando una funzione:

```
Stream<String> stream = Stream.of("a", "b", "c");
Stream<String> mapped = stream.map(String::toUpperCase);
// Risultato: "A", "B", "C"
```

flatMap

Trasforma ogni elemento in uno stream e poi appiattisce i risultati:

```
List<List<Integer>> listOfLists = Arrays.asList(
    Arrays.asList(1, 2, 3),
    Arrays.asList(4, 5, 6),
    Arrays.asList(7, 8, 9)
);

Stream<Integer> flattened = listOfLists.stream()
    .flatMap(Collection::stream);
// Risultato: 1, 2, 3, 4, 5, 6, 7, 8, 9
```

distinct

Elimina gli elementi duplicati:

```
Stream<String> stream = Stream.of("a", "b", "a", "c", "b");
Stream<String> distinct = stream.distinct();
// Risultato: "a", "b", "c"
```

sorted

Ordina gli elementi:

```
Stream<String> stream = Stream.of("c", "a", "b");
Stream<String> sorted = stream.sorted();
// Risultato: "a", "b", "c"

// Con comparatore personalizzato
Stream<String> sortedByLength = stream.sorted(Comparator.comparing(String::length));
```

peek

Esegue un'azione su ogni elemento senza modificare lo stream:

```
Stream<String> stream = Stream.of("a", "b", "c");
Stream<String> peeked = stream.peek(System.out::println);
// Stampa ogni elemento ma restituisce lo stream originale
```

limit

Limita il numero di elementi:

```
Stream<Integer> stream = Stream.iterate(1, n -> n + 1);
Stream<Integer> limited = stream.limit(5);
// Risultato: 1, 2, 3, 4, 5
```

skip

Salta i primi n elementi:

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
Stream<Integer> skipped = stream.skip(2);
// Risultato: 3, 4, 5
```

Operazioni Terminali

Le operazioni terminali producono un risultato o un effetto collaterale e consumano lo stream.

Principali operazioni terminali:

forEach

Esegue un'azione per ogni elemento:

```
Stream<String> stream = Stream.of("a", "b", "c");
stream.forEach(System.out::println);
// Stampa: a, b, c
```

collect

Accumula gli elementi in una collezione o altro contenitore:

```
Stream<String> stream = Stream.of("a", "b", "c");
List<String> list = stream.collect(Collectors.toList());
// Risultato: lista contenente "a", "b", "c"

Map<Boolean, List<String>> map = stream.collect(
    Collectors.partitioningBy(s -> s.length() > 1)
);
// Raggruppa gli elementi in base alla lunghezza
```

reduce

Combina gli elementi per produrre un singolo risultato:

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
Optional<Integer> sum = stream.reduce(Integer::sum);
// Risultato: 15

Integer sumWithIdentity = stream.reduce(0, Integer::sum);
// Risultato: 15 (con valore iniziale 0)
```

count

Conta gli elementi:

```
Stream<String> stream = Stream.of("a", "b", "c");
long count = stream.count();
// Risultato: 3
```

anyMatch, allMatch, noneMatch

Verificano se gli elementi soddisfano una condizione:

```
Stream<String> stream = Stream.of("Java", "JavaScript", "Python");
boolean anyJ = stream.anyMatch(s -> s.startsWith("J"));
// Risultato: true

boolean allJ = stream.allMatch(s -> s.startsWith("J"));
// Risultato: false

boolean noneC = stream.noneMatch(s -> s.startsWith("C"));
// Risultato: true
```

findFirst, findAny

Restituiscono un elemento dello stream:

```
Stream<String> stream = Stream.of("a", "b", "c");
Optional<String> first = stream.findFirst();
// Risultato: Optional["a"]

Optional<String> any = stream.findAny();
// Risultato: Optional contenente un elemento qualsiasi
```

Stream Paralleli

Gli stream possono essere elaborati in parallelo per sfruttare i sistemi multi-core:

```
List<Integer> numeri = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

// Creazione di uno stream parallelo da una collezione
Stream<Integer> parallelStream = numeri.parallelStream();

// Conversione di uno stream sequenziale in parallelo
Stream<Integer> anotherParallelStream = Stream.of(1, 2, 3, 4, 5).parallel();

// Esempio di elaborazione parallela
int sum = parallelStream.reduce(0, Integer::sum);
```

Come evidenziato negli appelli d'esame:

> "Usando i Reactive Stream, la gestione più granulare della composizione della pipeline di elaborazione dello stream permette di:"
>
> Risposta: "Isolare la parte di pipeline che si desidera sia resa parallela; gli Stream della libreria standard sono o completamente paralleli, o completamente seriali."

Considerazioni sugli Stream Paralleli

1. **Non sempre più veloci**: L'overhead di parallelizzazione può superare i benefici per stream piccoli
2. **Stateless e associativi**: Le operazioni devono essere stateless e preferibilmente associative
3. **Interferenza**: Modificare la fonte durante l'elaborazione può causare risultati imprevedibili
4. **Ordinamento**: Alcune operazioni possono non preservare l'ordinamento in modalità parallela

Collectors

I collectors sono implementazioni dell'interfaccia `Collector` che specificano come raccogliere gli elementi di uno stream in un risultato.

Collectors Predefiniti

La classe `Collectors` fornisce numerosi collectors predefiniti:

toList, toSet, toMap

```
List<String> list = stream.collect(Collectors.toList());
Set<String> set = stream.collect(Collectors.toSet());
Map<String, Integer> map = stream.collect(Collectors.toMap(
    Function.identity(),
    String::length
));
```

joining

```
String joined = Stream.of("a", "b", "c")
    .collect(Collectors.joining(", ", "[", "]"));
// Risultato: "[a, b, c]"
```

counting

```
long count = stream.collect(Collectors.counting());
```

summingInt, averagingInt, summarizingInt

```
int sum = Stream.of("a", "bb", "ccc")
    .collect(Collectors.summingInt(String::length));
// Risultato: 6

double avg = Stream.of("a", "bb", "ccc")
    .collect(Collectors.averagingInt(String::length));
// Risultato: 2.0

IntSummaryStatistics stats = Stream.of("a", "bb", "ccc")
    .collect(Collectors.summarizingInt(String::length));
// Contiene count, sum, min, average, max
```

groupingBy, partitioningBy

```
Map<Integer, List<String>> grouped = Stream.of("a", "bb", "ccc", "dd")
    .collect(Collectors.groupingBy(String::length));
// Risultato: {1=["a"], 2=["bb", "dd"], 3=["ccc"]}

Map<Boolean, List<String>> partitioned = Stream.of("a", "bb", "ccc")
    .collect(Collectors.partitioningBy(s -> s.length() > 1));
// Risultato: {false=["a"], true=["bb", "ccc"]}
```

Collectors Personalizzati

È possibile creare collectors personalizzati implementando l'interfaccia `Collector`:

```
public class CustomCollector<T> implements Collector<T, List<T>, List<T>> {
    @Override
    public Supplier<List<T>> supplier() {
        return ArrayList::new;
    }

    @Override
    public BiConsumer<List<T>, T> accumulator() {
        return List::add;
    }

    @Override
    public BinaryOperator<List<T>> combiner() {
        return (list1, list2) -> {
            list1.addAll(list2);
            return list1;
        };
    }

    @Override
    public Function<List<T>, List<T>> finisher() {
        return Function.identity();
    }

    @Override
    public Set<Characteristics> characteristics() {
        return EnumSet.of(Characteristics.IDENTITY_FINISH);
    }
}
```

Come evidenziato negli appelli d'esame:

> "L'interfaccia Collector permette di eseguire la riduzione ad un risultato di uno Stream parallelo in modo più efficiente perché:"
>
> Risposta: "Perchè gestisce un accumulatore mutabile, che riduce la pressione sulla Garbage Collection."

Short-Circuiting

Le operazioni short-circuiting sono quelle che possono terminare l'elaborazione di uno stream prima che tutti gli elementi siano stati processati.

Come evidenziato negli appelli d'esame:

> "Una operatore short-circuiting all'interno di una catena di elaborazione di uno Stream può:"
>
> Risposta: "Produrre il risultato prima che lo Stream sia stato interamente consumato."

Operazioni Short-Circuiting Intermedie

- `limit(n)` : limita lo stream ai primi n elementi
- `skip(n)` : salta i primi n elementi
- `takeWhile(predicate)` (Java 9+): prende elementi finché il predicato è vero
- `dropWhile(predicate)` (Java 9+): scarta elementi finché il predicato è vero

Operazioni Short-Circuiting Terminali

- `findFirst()` : trova il primo elemento
- `findAny()` : trova un elemento qualsiasi
- `anyMatch(predicate)` : verifica se almeno un elemento soddisfa il predicato
- `allMatch(predicate)` : verifica se tutti gli elementi soddisfano il predicato
- `noneMatch(predicate)` : verifica se nessun elemento soddisfa il predicato

```
boolean found = Stream.iterate(1, n -> n + 1)
    .anyMatch(n -> n > 100);
// Termina dopo aver trovato il primo numero > 100
```

Stream Flags

Gli Stream Flags sono caratteristiche che uno stream può dichiarare e che gli operatori possono utilizzare per ottimizzare l'esecuzione.

Come evidenziato negli appelli d'esame:

> "Quale dei seguenti non è uno Stream Flag, cioè una caratteristica che uno Stream può dichiarare ed uno operatore (intermedio o terminale) utilizzare per organizzare l'esecuzione:"
>
> Risposta: "UNTYPED - non è noto a priori il tipo degli elementi."

I principali Stream Flags sono:

1. **SIZED**: lo stream ha una dimensione nota
2. **DISTINCT**: gli elementi sono tutti distinti
3. **SORTED**: gli elementi sono ordinati secondo un criterio
4. **ORDERED**: lo stream ha un ordine incontrato significativo
5. **SUBSIZED**: lo stream può essere diviso in partizioni di dimensione nota
6. **CONCURRENT**: lo stream supporta l'elaborazione parallela
7. **IMMUTABLE**: gli elementi non possono essere modificati
8. **NONNULL**: tutti gli elementi sono diversi da null

Vantaggi della Separazione tra Costruzione ed Esecuzione

Una caratteristica fondamentale della Stream API è la separazione tra la costruzione della pipeline di elaborazione e la sua esecuzione.

Come evidenziato negli appelli d'esame:

> "Nella implementazione degli Stream della libreria standard, che vantaggio si ottiene dal fatto che la API consente di costruire la catena di elaborazione separatamente dalla sua esecuzione?"
>
> Risposta: "L'implementazione può analizzare le operazioni della catena, e prendere decisioni su come applicarle in funzione delle loro caratteristiche."

Questa separazione permette:

1. **Ottimizzazioni**: l'implementazione può riordinare o fondere operazioni per migliorare l'efficienza
2. **Lazy evaluation**: le operazioni intermedie sono eseguite solo quando necessario
3. **Short-circuiting**: l'elaborazione può terminare prima di processare tutti gli elementi
4. **Parallelizzazione**: l'implementazione può decidere come suddividere il lavoro tra i thread

Esempio di Ottimizzazione

```
List<String> result = strings.stream()
    .filter(s -> s.startsWith("a"))
    .sorted()
    .map(String::toUpperCase)
    .collect(Collectors.toList());
```

In questo esempio, l'implementazione potrebbe:

1. Applicare prima `filter` per ridurre il numero di elementi da ordinare
2. Applicare `sorted` solo agli elementi filtrati
3. Applicare `map` solo agli elementi filtrati e ordinati

Confronto con Reactive Extensions

Mentre la Stream API è progettata principalmente per l'elaborazione di sequenze finite di dati, le Reactive Extensions (Rx) sono progettate per l'elaborazione asincrona di sequenze potenzialmente infinite di eventi.

Principali differenze:

1. **Push vs Pull**: Rx utilizza un modello push (l'Observable emette eventi), mentre Stream utilizza un modello pull (il consumatore richiede il prossimo elemento)
2. **Asincronia**: Rx è intrinsecamente asincrono, mentre Stream è principalmente sincrono (anche se può essere parallelo)
3. **Gestione degli errori**: Rx ha meccanismi integrati per la gestione degli errori, mentre Stream richiede try-catch espliciti
4. **Backpressure**: Rx ha meccanismi per gestire la backpressure (quando il produttore è più veloce del consumatore), mentre Stream non ne ha bisogno grazie al modello pull
5. **Composizione**: Rx offre operatori più ricchi per la composizione di flussi asincroni

Come evidenziato negli appelli d'esame, la gestione più granulare della composizione della pipeline in Reactive Stream permette di isolare le parti che devono essere parallele, mentre gli Stream della libreria standard sono o completamente paralleli o completamente seriali.

6. Reactive Extensions

Concetti Fondamentali

Le Reactive Extensions (Rx) rappresentano un paradigma di programmazione che si concentra sulla gestione di flussi di dati asincroni e sulla propagazione dei cambiamenti. Questo approccio è particolarmente adatto per applicazioni event-driven, sistemi distribuiti e interfacce utente reattive.

Definizione e Scopo

Come evidenziato negli appelli d'esame:

> "Lo scopo delle Reactive Extensions è:"
>
> Risposta: "Fornire una semantica per definire elaborazioni asincrone di sequenze di oggetti."

Le Reactive Extensions combinano i migliori aspetti di:

- Pattern Observer (per la notifica di cambiamenti)
- Pattern Iterator (per l'accesso sequenziale agli elementi)
- Programmazione funzionale (per la manipolazione dei dati)

Principi Fondamentali

1. **Asincronia**: Le operazioni non bloccano il thread chiamante
2. **Basato su eventi**: Il flusso di dati è guidato dagli eventi
3. **Composizione funzionale**: Le operazioni possono essere composte in modo dichiarativo
4. **Gestione degli errori**: Meccanismi integrati per la propagazione e la gestione degli errori

Reactive Manifesto

Il Reactive Manifesto, pubblicato nel 2013, definisce le caratteristiche dei sistemi reattivi:

1. **Responsive**: I sistemi rispondono in modo tempestivo
2. **Resilient**: I sistemi rimangono reattivi anche in caso di guasti
3. **Elastic**: I sistemi rimangono reattivi sotto carichi variabili
4. **Message-Driven**: I sistemi si basano sullo scambio di messaggi asincroni

Come evidenziato negli appelli d'esame:

> "Quando di un sistema reattivo si indicano le sue qualità come Responsive, Resilient, Elastic, Message-Oriented, con la qualità 'Message-Oriented' si intende:"
>
> Risposta: "L'unica primitiva di comunicazione fra i componenti è il messaggio asincrono."

Observable e Observer

Il modello Rx si basa su due concetti principali:

1. **Observable**: Una fonte che emette una sequenza di elementi nel tempo
2. **Observer**: Un consumatore che reagisce agli elementi emessi dall'Observable

Observable

Un Observable rappresenta una sorgente di dati che può emettere:

- Zero o più elementi
- Un errore
- Un segnale di completamento

```
// Creazione di un Observable che emette una sequenza di stringhe
Observable<String> observable = Observable.just("Hello", "World", "Rx");

// Observable da una collezione
List<String> lista = Arrays.asList("a", "b", "c");
Observable<String> fromList = Observable.fromIterable(lista);

// Observable che emette valori periodicamente
Observable<Long> interval = Observable.interval(1, TimeUnit.SECONDS);
```

Observer

Un Observer si sottoscrive a un Observable per ricevere notifiche:

```
Observer<String> observer = new Observer<String>() {
    @Override
    public void onSubscribe(Disposable d) {
        System.out.println("Sottoscrizione avvenuta");
    }

    @Override
    public void onNext(String s) {
        System.out.println("Elemento ricevuto: " + s);
    }

    @Override
    public void onError(Throwable e) {
        System.err.println("Errore: " + e.getMessage());
    }

    @Override
    public void onComplete() {
        System.out.println("Sequenza completata");
    }
};

// Sottoscrizione dell'observer all'observable
observable.subscribe(observer);
```

Sottoscrizione Semplificata

È possibile sottoscriversi a un Observable in modo più conciso:

```
// Solo onNext
observable.subscribe(s -> System.out.println("Elemento: " + s));

// onNext e onError
observable.subscribe(
    s -> System.out.println("Elemento: " + s),
    e -> System.err.println("Errore: " + e.getMessage())
);

// onNext, onError e onComplete
observable.subscribe(
    s -> System.out.println("Elemento: " + s),
    e -> System.err.println("Errore: " + e.getMessage()),
    () -> System.out.println("Completato")
);
```

Come evidenziato negli appelli d'esame:

> "Nelle Reactive Extensions, quali di queste operazioni non è necessario (o possibile) specificare per elaborare gli oggetti emessi da un observable:"
>
> Risposta: "Il numero di oggetti che l'observable è autorizzato ad inviare. Il comportamento alla richiesta di separazione di uno stream parallelo."

Operatori

Gli operatori sono funzioni che trasformano un Observable in un altro Observable, permettendo di costruire pipeline di elaborazione complesse.

Operatori di Creazione

just

Crea un Observable che emette gli elementi specificati:

```
Observable<String> observable = Observable.just("a", "b", "c");
```

fromIterable

Crea un Observable da una collezione:

```
List<String> lista = Arrays.asList("a", "b", "c");
Observable<String> observable = Observable.fromIterable(lista);
```

interval

Crea un Observable che emette numeri interi crescenti a intervalli regolari:

```
Observable<Long> observable = Observable.interval(1, TimeUnit.SECONDS);
```

range

Crea un Observable che emette una sequenza di numeri interi:

```
Observable<Integer> observable = Observable.range(1, 5); // 1, 2, 3, 4, 5
```

empty, never, error

Creano Observable con comportamenti speciali:

```
Observable<String> empty = Observable.empty(); // Non emette elementi e completa immediatamente
Observable<String> never = Observable.never(); // Non emette elementi e non completa mai
Observable<String> error = Observable.error(new RuntimeException("Errore")); // Emette solo un errore
```

Operatori di Trasformazione

map

Trasforma ogni elemento emesso applicando una funzione:

```
Observable<String> observable = Observable.just("a", "b", "c");
Observable<String> mapped = observable.map(String::toUpperCase);
// Risultato: "A", "B", "C"
```

flatMap

Trasforma ogni elemento in un Observable e appiattisce i risultati:

```
Observable<String> observable = Observable.just("a", "b");
Observable<String> flatMapped = observable.flatMap(s ->
    Observable.just(s + "1", s + "2")
);
// Risultato: "a1", "a2", "b1", "b2"
```

scan

Applica una funzione all'elemento corrente e al risultato precedente:

```
Observable<Integer> observable = Observable.just(1, 2, 3, 4, 5);
Observable<Integer> scanned = observable.scan((acc, val) -> acc + val);
// Risultato: 1, 3, 6, 10, 15
```

Operatori di Filtraggio

filter

Emette solo gli elementi che soddisfano un predicato:

```
Observable<Integer> observable = Observable.range(1, 10);
Observable<Integer> filtered = observable.filter(n -> n % 2 == 0);
// Risultato: 2, 4, 6, 8, 10
```

take

Emette solo i primi n elementi:

```
Observable<Integer> observable = Observable.range(1, 10);
Observable<Integer> taken = observable.take(3);
// Risultato: 1, 2, 3
```

skip

Salta i primi n elementi:

```
Observable<Integer> observable = Observable.range(1, 10);
Observable<Integer> skipped = observable.skip(7);
// Risultato: 8, 9, 10
```

distinct

Emette solo elementi distinti:

```
Observable<Integer> observable = Observable.just(1, 2, 1, 3, 2, 4);
Observable<Integer> distinct = observable.distinct();
// Risultato: 1, 2, 3, 4
```

Operatori di Combinazione

merge

Combina più Observable emettendo tutti gli elementi man mano che arrivano:

```
Observable<String> obs1 = Observable.just("a", "b");
Observable<String> obs2 = Observable.just("c", "d");
Observable<String> merged = Observable.merge(obs1, obs2);
// Possibile risultato: "a", "b", "c", "d" o "a", "c", "b", "d" ecc.
```

concat

Combina più Observable emettendo gli elementi in sequenza:

```
Observable<String> obs1 = Observable.just("a", "b");
Observable<String> obs2 = Observable.just("c", "d");
Observable<String> concatenated = Observable.concat(obs1, obs2);
// Risultato: "a", "b", "c", "d"
```

zip

Combina gli elementi di più Observable applicando una funzione:

```
Observable<String> obs1 = Observable.just("a", "b", "c");
Observable<Integer> obs2 = Observable.just(1, 2, 3);
Observable<String> zipped = Observable.zip(obs1, obs2,
    (s, i) -> s + i
);
// Risultato: "a1", "b2", "c3"
```

Operatori di Gestione degli Errori

onErrorReturn

Sostituisce un errore con un valore:

```
Observable<String> observable = Observable.error(new RuntimeException("Errore"));
Observable<String> recovered = observable.onErrorReturn(e -> "Valore di fallback");
```

onErrorResumeNext

Sostituisce un errore con un altro Observable:

```
Observable<String> observable = Observable.error(new RuntimeException("Errore"));
Observable<String> recovered = observable.onErrorResumeNext(
    Observable.just("a", "b", "c")
);
```

retry

Riprova l'Observable in caso di errore:

```
Observable<String> observable = Observable.error(new RuntimeException("Errore"));
Observable<String> retried = observable.retry(3); // Riprova fino a 3 volte
```

Subject

Un Subject è sia un Observable che un Observer, permettendo sia di emettere elementi sia di sottoscriversi ad altri Observable.

Come evidenziato negli appelli d'esame:

> "Nell'astrazione delle Reactive Extensions, un subject può:"
>
> Risposta: "Osservare diversi observable, e comportarsi da observable esso stesso, modificando la struttura dell'elaborazione dello stream."

Tipi di Subject

PublishSubject

Emette agli observer sottoscritti tutti gli elementi emessi dopo la loro sottoscrizione:

```
PublishSubject<String> subject = PublishSubject.create();
subject.subscribe(s -> System.out.println("Observer 1: " + s));

subject.onNext("a"); // Observer 1 riceve "a"

subject.subscribe(s -> System.out.println("Observer 2: " + s));

subject.onNext("b"); // Entrambi gli observer ricevono "b"
subject.onNext("c"); // Entrambi gli observer ricevono "c"

subject.onComplete(); // Entrambi gli observer ricevono onComplete
```

BehaviorSubject

Emette l'elemento più recente e tutti gli elementi successivi:

```
BehaviorSubject<String> subject = BehaviorSubject.createDefault("default");
subject.subscribe(s -> System.out.println("Observer 1: " + s)); // Riceve "default"

subject.onNext("a"); // Observer 1 riceve "a"

subject.subscribe(s -> System.out.println("Observer 2: " + s)); // Riceve "a"

subject.onNext("b"); // Entrambi gli observer ricevono "b"
```

ReplaySubject

Emette tutti gli elementi emessi, indipendentemente da quando avviene la sottoscrizione:

```
ReplaySubject<String> subject = ReplaySubject.create();
subject.onNext("a");
subject.onNext("b");

subject.subscribe(s -> System.out.println("Observer 1: " + s)); // Riceve "a", "b"

subject.onNext("c"); // Observer 1 riceve "c"

subject.subscribe(s -> System.out.println("Observer 2: " + s)); // Riceve "a", "b", "c"
```

AsyncSubject

Emette solo l'ultimo elemento prima del completamento:

```
AsyncSubject<String> subject = AsyncSubject.create();
subject.subscribe(s -> System.out.println("Observer 1: " + s));

subject.onNext("a");
subject.onNext("b");
subject.onNext("c");

subject.subscribe(s -> System.out.println("Observer 2: " + s));

subject.onComplete(); // Entrambi gli observer ricevono "c"
```

Schedulers

Gli Scheduler controllano su quali thread vengono eseguiti gli Observable e gli Observer.

```
Observable.just("a", "b", "c")
    .subscribeOn(Schedulers.io()) // Esegue l'Observable su un thread I/O
    .observeOn(AndroidSchedulers.mainThread()) // Notifica l'Observer sul thread principale
    .subscribe(s -> System.out.println("Elemento: " + s));
```

Tipi di Scheduler

- **Schedulers.io()**: Per operazioni di I/O non bloccanti
- **Schedulers.computation()**: Per lavoro computazionale
- **Schedulers.newThread()**: Crea un nuovo thread per ogni task
- **Schedulers.single()**: Esegue su un singolo thread
- **Schedulers.trampoline()**: Esegue sul thread corrente, ma in coda
- **AndroidSchedulers.mainThread()**: Esegue sul thread principale di Android (solo in RxAndroid)

Backpressure

La backpressure è un meccanismo per gestire situazioni in cui un Observable produce elementi più velocemente di quanto un Observer possa consumarli.

Strategie di Backpressure

Buffer

Memorizza gli elementi in eccesso in un buffer:

```
Flowable.interval(1, TimeUnit.MILLISECONDS)
    .onBackpressureBuffer(1000) // Buffer fino a 1000 elementi
    .observeOn(Schedulers.computation())
    .subscribe(/* ... */);
```

Drop

Scarta gli elementi in eccesso:

```
Flowable.interval(1, TimeUnit.MILLISECONDS)
    .onBackpressureDrop() // Scarta gli elementi che non possono essere processati
    .observeOn(Schedulers.computation())
    .subscribe(/* ... */);
```

Latest

Mantiene solo l'elemento più recente:

```
Flowable.interval(1, TimeUnit.MILLISECONDS)
    .onBackpressureLatest() // Mantiene solo l'ultimo elemento
    .observeOn(Schedulers.computation())
    .subscribe(/* ... */);
```

Reactive Streams

Reactive Streams è una specifica per l'elaborazione asincrona di stream con backpressure, che definisce un'interfaccia standard per l'interoperabilità tra diverse implementazioni.

Componenti di Reactive Streams

1. **Publisher**: Fonte di dati che emette elementi
2. **Subscriber**: Consumatore che riceve elementi
3. **Subscription**: Rappresenta la relazione tra Publisher e Subscriber
4. **Processor**: Sia Publisher che Subscriber

```
Publisher<Integer> publisher = /* ... */;
publisher.subscribe(new Subscriber<Integer>() {
    @Override
    public void onSubscribe(Subscription s) {
        s.request(Long.MAX_VALUE); // Richiede tutti gli elementi
    }

    @Override
    public void onNext(Integer i) {
        System.out.println("Elemento: " + i);
    }

    @Override
    public void onError(Throwable t) {
        t.printStackTrace();
    }

    @Override
    public void onComplete() {
        System.out.println("Completato");
    }
});
```

Vantaggi di Reactive Streams

Come evidenziato negli appelli d'esame:

> "Usando i Reactive Stream, la gestione più granulare della composizione della pipeline di elaborazione dello stream permette di:"

> Risposta: "Isolare la parte di pipeline che si desidera sia resa parallela; gli Stream della libreria standard sono o completamente paralleli, o completamente seriali."

Reactive Streams offre:

1. **Controllo del flusso**: Backpressure per gestire produttori veloci e consumatori lenti
2. **Composizione granulare**: Controllo preciso su quali parti della pipeline sono parallele
3. **Interoperabilità**: Standard comune per diverse implementazioni
4. **Asincronia**: Elaborazione non bloccante

Confronto con Java Stream API

Caratteristica	Reactive Extensions	Java Stream API
Modello	Push (Observable emette)	Pull (Consumer richiede)
Asincronia	Intrinsecamente asincrono	Principalmente sincrono
Errori	Gestione integrata	Try-catch espliciti
Backpressure	Supportata	Non necessaria (pull)
Completezza	Emette elementi, errori, completamento	Emette solo elementi
Parallelismo	Granulare	Tutto o niente
Uso tipico	Eventi UI, I/O asincrono	Elaborazione dati in memoria

RxJava

RxJava è l'implementazione Java delle Reactive Extensions, che fornisce un'API per la programmazione reattiva.

Dipendenza Maven

```
<dependency>
    <groupId>io.reactivex.rxjava3</groupId>
    <artifactId>rxjava</artifactId>
    <version>3.1.5</version>
</dependency>
```

Esempio Completo

```
import io.reactivex.rxjava3.core.Observable;
import io.reactivex.rxjava3.schedulers.Schedulers;
import java.util.concurrent.TimeUnit;

public class RxJavaExample {
    public static void main(String[] args) throws InterruptedException {
        // Crea un Observable che emette numeri ogni secondo
        Observable<Long> observable = Observable.interval(1, TimeUnit.SECONDS)
            .take(5); // Limita a 5 elementi

        // Sottoscrizione con gestione completa
        observable
            .subscribeOn(Schedulers.io()) // Esegue su thread I/O
            .map(n -> "Numero: " + n) // Trasforma in stringa
            .subscribe(
                s -> System.out.println("Elemento: " + s), // onNext
                e -> System.err.println("Errore: " + e.getMessage()), // onError
                () -> System.out.println("Completato") // onComplete
            );

        // Attendi per vedere i risultati
        Thread.sleep(6000);
    }
}
```

Applicazioni Pratiche

Gestione di Eventi UI

```
// Android (con RxAndroid)
RxView.clicks(button)
    .throttleFirst(500, TimeUnit.MILLISECONDS) // Previene doppi click
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(event -> handleButtonClick());
```

Chiamate di Rete

```
// Retrofit con RxJava
api.getUserData(userId)
    .subscribeOn(Schedulers.io()) // Esegue la chiamata su thread I/O
    .observeOn(AndroidSchedulers.mainThread()) // Gestisce il risultato sul thread principale
    .subscribe(
        userData -> updateUI(userData),
        error -> showError(error)
    );
```

Composizione di Operazioni Asincrone

```
// Esegui operazioni in sequenza
Observable.just(userId)
    .flatMap(id -> api.getUserProfile(id)) // Prima chiamata
    .flatMap(profile -> api.getFriends(profile.getId())) // Seconda chiamata
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(
        friends -> showFriends(friends),
        error -> showError(error)
    );
```

Polling Periodico

```
// Controlla aggiornamenti ogni minuto
Observable.interval(1, TimeUnit.MINUTES)
    .flatMap(tick -> api.checkForUpdates())
    .filter(updates -> !updates.isEmpty())
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(
        updates -> showUpdates(updates),
        error -> logError(error)
    );
```

7. Programmazione di Rete

Socket e ServerSocket

La programmazione di rete in Java si basa principalmente sulle classi `Socket` e `ServerSocket`, che forniscono un'astrazione per la comunicazione tra processi su una rete.

Socket

Un `Socket` rappresenta un endpoint per la comunicazione tra due macchine. In Java, la classe `Socket` implementa il lato client di una connessione.

```
import java.io.*;
import java.net.*;

public class ClientSocket {
    public static void main(String[] args) {
        try {
            // Crea un socket e si connette al server
            Socket socket = new Socket("localhost", 8080);

            // Ottiene gli stream di input e output
            OutputStream output = socket.getOutputStream();
            PrintWriter writer = new PrintWriter(output, true);

            InputStream input = socket.getInputStream();
            BufferedReader reader = new BufferedReader(new InputStreamReader(input));

            // Invia un messaggio al server
            writer.println("Hello, Server!");

            // Legge la risposta dal server
            String response = reader.readLine();
            System.out.println("Server response: " + response);

            // Chiude la connessione
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

ServerSocket

Un `ServerSocket` attende le connessioni in entrata e crea un `Socket` per ogni connessione accettata.

```
import java.io.*;
import java.net.*;

public class ServerSocketExample {
    public static void main(String[] args) {
        try {
            // Crea un server socket sulla porta 8080
            ServerSocket serverSocket = new ServerSocket(8080);
            System.out.println("Server in ascolto sulla porta 8080...");

            while (true) {
                // Accetta una connessione in entrata
                Socket clientSocket = serverSocket.accept();
                System.out.println("Client connesso: " + clientSocket.getInetAddress());

                // Gestisce la connessione in un nuovo thread
                new Thread(() -> handleClient(clientSocket)).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private static void handleClient(Socket clientSocket) {
        try {
            // Ottiene gli stream di input e output
            InputStream input = clientSocket.getInputStream();
            BufferedReader reader = new BufferedReader(new InputStreamReader(input));

            OutputStream output = clientSocket.getOutputStream();
            PrintWriter writer = new PrintWriter(output, true);

            // Legge il messaggio dal client
            String message = reader.readLine();
            System.out.println("Messaggio ricevuto: " + message);

            // Invia una risposta al client
            writer.println("Echo: " + message);

            // Chiude la connessione
            clientSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Gestione delle Connessioni

Come evidenziato negli appelli d'esame:

> "È importante gestire velocemente l'accettazione di una nuova connessione su di un ServerSocket perché:"
>
> Risposta: "Finché non c'è un thread che attende in ServerSocket::accept(), le nuove richieste di connessione si accodano su di un buffer del sistema operativo che può avere una lunghezza molto limitata."

Se non si gestiscono rapidamente le connessioni in entrata, il buffer del sistema operativo potrebbe riempirsi, causando il rifiuto di nuove connessioni.

Comunicazione su Socket

La comunicazione su Socket avviene tramite stream di byte, il che comporta alcune sfide:

Come evidenziato negli appelli d'esame:

> "La comunicazione su Socket ha diversi vantaggi, ma il fatto che i dati vengano presentati come uno stream di byte ha i seguenti svantaggi:"
>
> Risposta: "Deve essere definito un protocollo con cui i due lati della comunicazione riconoscono l'inizio e la fine dei messaggi. Le due parti devono concordare in qualche modo l'encoding delle stringhe all'interno del protocollo di comunicazione."

Per gestire questi svantaggi, è necessario definire un protocollo di comunicazione che specifichi:

1. Come delimitare i messaggi (ad esempio, con caratteri speciali o indicando la lunghezza)
2. Come codificare i dati (ad esempio, UTF-8 per le stringhe)
3. Come strutturare i messaggi (formato, campi, ecc.)

Datagram

I datagram sono pacchetti di dati indipendenti che vengono inviati attraverso la rete senza stabilire una connessione persistente. In Java, i datagram sono implementati tramite le classi `DatagramSocket` e `DatagramPacket`.

DatagramSocket e DatagramPacket

```
 import java.io.IOException;
import java.net.*;

public class UDPClient {
    public static void main(String[] args) {
        try {
            // Crea un socket UDP
            DatagramSocket socket = new DatagramSocket();

            // Prepara il messaggio
            String message = "Hello, UDP Server!";
            byte[] buffer = message.getBytes();

            // Crea un pacchetto con il messaggio
            InetAddress address = InetAddress.getByName("localhost");
            DatagramPacket packet = new DatagramPacket(buffer, buffer.length, address, 9876);

            // Invia il pacchetto
            socket.send(packet);

            // Prepara un buffer per la risposta
            byte[] receiveBuffer = new byte[1024];
            DatagramPacket receivePacket = new DatagramPacket(receiveBuffer, receiveBuffer.length);

            // Riceve la risposta
            socket.receive(receivePacket);

            // Elabora la risposta
            String response = new String(receivePacket.getData(), 0, receivePacket.getLength());
            System.out.println("Server response: " + response);

            // Chiude il socket
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
import java.io.IOException;
import java.net.*;

public class UDPServer {
    public static void main(String[] args) {
        try {
            // Crea un socket UDP sulla porta 9876
            DatagramSocket socket = new DatagramSocket(9876);
            System.out.println("Server UDP in ascolto sulla porta 9876...");

            while (true) {
                // Prepara un buffer per ricevere i dati
                byte[] buffer = new byte[1024];
                DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

                // Riceve un pacchetto
                socket.receive(packet);

                // Elabora il messaggio ricevuto
                String message = new String(packet.getData(), 0, packet.getLength());
                System.out.println("Messaggio ricevuto: " + message);

                // Prepara la risposta
                String response = "Echo: " + message;
                byte[] responseBuffer = response.getBytes();

                // Crea un pacchetto con la risposta
                InetAddress address = packet.getAddress();
                int port = packet.getPort();
                DatagramPacket responsePacket = new DatagramPacket(responseBuffer, responseBuffer.length, address, port);

                // Invia la risposta
                socket.send(responsePacket);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Vantaggi dei Datagram

Come evidenziato negli appelli d'esame:

> "Quali dei seguenti possono essere indicati come vantaggi dell'uso dei Datagram:"
>
> Risposta: "Un singolo Datagram può essere inviato a molti indirizzi con una sola istruzione. Il singolo Datagram è isolato, quindi non è necessario introdurre nel protocollo dei separatori fra messaggi diversi."

I datagram offrono:

1. **Indipendenza**: Ogni pacchetto è autonomo e contiene tutte le informazioni necessarie
2. **Multicast/Broadcast**: Possibilità di inviare lo stesso pacchetto a più destinatari
3. **Semplicità**: Non è necessario stabilire o mantenere una connessione
4. **Efficienza**: Minor overhead per comunicazioni brevi e sporadiche

Gestione dei Datagram

Come evidenziato negli appelli d'esame:

> "È importante gestire velocemente l'accettazione di un pacchetto da una DatagramSocket perché:"
>
> Risposta: "I pacchetti ricevuti vengono conservati in buffer limitati; se si riempiono, i messaggi successivi sono scartati."

Se non si elaborano rapidamente i pacchetti in arrivo, i buffer del sistema operativo potrebbero riempirsi, causando la perdita di pacchetti successivi.

```
// Esempio di gestione efficiente dei datagram
public class EfficientUDPServer {
    public static void main(String[] args) {
        try {
            DatagramSocket socket = new DatagramSocket(9876);

            // Pool di thread per elaborare i pacchetti
            ExecutorService executor = Executors.newFixedThreadPool(10);

            while (true) {
                byte[] buffer = new byte[1024];
                DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

                // Blocca finché non arriva un pacchetto
                socket.receive(packet);

                // Crea una copia del pacchetto per l'elaborazione asincrona
                DatagramPacket finalPacket = new DatagramPacket(
                    packet.getData().clone(),
                    packet.getLength(),
                    packet.getAddress(),
                    packet.getPort()
                );

                // Elabora il pacchetto in un thread separato
                executor.execute(() -> processPacket(socket, finalPacket));
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private static void processPacket(DatagramSocket socket, DatagramPacket packet) {
        try {
            // Elaborazione del pacchetto...
            String message = new String(packet.getData(), 0, packet.getLength());

            // Invio della risposta...
            String response = "Processed: " + message;
            byte[] responseBuffer = response.getBytes();
            DatagramPacket responsePacket = new DatagramPacket(
                responseBuffer,
                responseBuffer.length,
                packet.getAddress(),
                packet.getPort()
            );
            socket.send(responsePacket);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Channels

I Channels, introdotti in Java NIO (New I/O), forniscono un'astrazione di più alto livello per le operazioni di I/O, inclusa la comunicazione di rete.

Vantaggi dei Channels

Come evidenziato negli appelli d'esame:

> "Implementare un programma di rete usando l'astrazione dei 'Channels' della libreria standard di Java permette di non occuparsi di molti dettagli riguardanti l'interazione con il mezzo di comunicazione, ma:"
>
> Risposta: "È necessario ristrutturare il nostro codice riorganizzando in metodi che vengono richiamati all'avvenire di specifici eventi di I/O."

I Channels offrono:

1. **I/O non bloccante**: Possibilità di gestire molte connessioni con pochi thread
2. **Selettori**: Meccanismo per monitorare più canali contemporaneamente
3. **Buffer diretti**: Accesso diretto alla memoria per operazioni di I/O più efficienti
4. **Scatter/Gather**: Lettura/scrittura da/a più buffer in un'unica operazione

SocketChannel e ServerSocketChannel

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;

public class NIOClient {
    public static void main(String[] args) {
        try {
            // Crea e connette un SocketChannel
            SocketChannel socketChannel = SocketChannel.open();
            socketChannel.connect(new InetSocketAddress("localhost", 8080));

            // Prepara il messaggio
            String message = "Hello, NIO Server!";
            ByteBuffer buffer = ByteBuffer.wrap(message.getBytes());

            // Invia il messaggio
            socketChannel.write(buffer);

            // Prepara un buffer per la risposta
            ByteBuffer responseBuffer = ByteBuffer.allocate(1024);

            // Legge la risposta
            int bytesRead = socketChannel.read(responseBuffer);

            if (bytesRead > 0) {
                responseBuffer.flip();
                byte[] bytes = new byte[responseBuffer.remaining()];
                responseBuffer.get(bytes);
                String response = new String(bytes);
                System.out.println("Server response: " + response);
            }

            // Chiude il canale
            socketChannel.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Iterator;
import java.util.Set;

public class NIOServer {
    public static void main(String[] args) {
        try {
            // Crea un selettore
            Selector selector = Selector.open();
```

```
            // Crea e configura un ServerSocketChannel
            ServerSocketChannel serverChannel = ServerSocketChannel.open();
            serverChannel.bind(new InetSocketAddress(8080));
            serverChannel.configureBlocking(false);

            // Registra il canale con il selettore per accettare connessioni
            serverChannel.register(selector, SelectionKey.OP_ACCEPT);
            System.out.println("Server NIO in ascolto sulla porta 8080...");

            while (true) {
                // Blocca finché non ci sono eventi
                selector.select();

                // Ottiene le chiavi degli eventi pronti
                Set<SelectionKey> selectedKeys = selector.selectedKeys();
                Iterator<SelectionKey> keyIterator = selectedKeys.iterator();

                while (keyIterator.hasNext()) {
                    SelectionKey key = keyIterator.next();

                    // Rimuove la chiave dal set per evitare di processarla di nuovo
                    keyIterator.remove();

                    if (!key.isValid()) {
                        continue;
                    }

                    if (key.isAcceptable()) {
                        // Accetta una nuova connessione
                        handleAccept(key, selector);
                    } else if (key.isReadable()) {
                        // Legge dati da una connessione
                        handleRead(key);
                    }
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private static void handleAccept(SelectionKey key, Selector selector) throws IOException {
        ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();
        SocketChannel clientChannel = serverChannel.accept();
        clientChannel.configureBlocking(false);

        // Registra il canale del client per leggere
        clientChannel.register(selector, SelectionKey.OP_READ);
        System.out.println("Client connesso: " + clientChannel.getRemoteAddress());
    }

    private static void handleRead(SelectionKey key) throws IOException {
        SocketChannel clientChannel = (SocketChannel) key.channel();
        ByteBuffer buffer = ByteBuffer.allocate(1024);

        int bytesRead;
        try {
            bytesRead = clientChannel.read(buffer);
        } catch (IOException e) {
            // Connessione chiusa dal client
            key.cancel();
            clientChannel.close();
            return;
        }
```

```
        if (bytesRead == -1) {
            // Connessione chiusa dal client
            key.cancel();
            clientChannel.close();
            return;
        }

        // Elabora il messaggio
        buffer.flip();
        byte[] bytes = new byte[buffer.remaining()];
        buffer.get(bytes);
        String message = new String(bytes);
        System.out.println("Messaggio ricevuto: " + message);

        // Prepara la risposta
        String response = "Echo: " + message;
        ByteBuffer responseBuffer = ByteBuffer.wrap(response.getBytes());

        // Invia la risposta
        clientChannel.write(responseBuffer);
    }
}
```

Selector

Il `Selector` è un componente chiave dell'I/O non bloccante, che permette di monitorare più canali con un singolo thread.

```
 // Registrazione di canali con un selettore
channel1.register(selector, SelectionKey.OP_READ);
channel2.register(selector, SelectionKey.OP_WRITE);
channel3.register(selector, SelectionKey.OP_READ | SelectionKey.OP_WRITE);

// Attesa di eventi
int readyChannels = selector.select();
if (readyChannels == 0) {
    // Nessun canale pronto
    continue;
}

// Iterazione sulle chiavi pronte
Set<SelectionKey> selectedKeys = selector.selectedKeys();
Iterator<SelectionKey> keyIterator = selectedKeys.iterator();

while (keyIterator.hasNext()) {
    SelectionKey key = keyIterator.next();
    keyIterator.remove();

    if (key.isAcceptable()) {
        // Nuova connessione in arrivo
    } else if (key.isReadable()) {
        // Dati pronti per essere letti
    } else if (key.isWritable()) {
        // Canale pronto per la scrittura
    }
}
```

Buffer

I `Buffer` sono contenitori per dati che vengono letti da o scritti su un canale.

```
// Creazione di un buffer
ByteBuffer buffer = ByteBuffer.allocate(1024);

// Scrittura nel buffer
buffer.put("Hello".getBytes());

// Preparazione per la lettura
buffer.flip();

// Lettura dal buffer
byte[] bytes = new byte[buffer.remaining()];
buffer.get(bytes);
String message = new String(bytes);

// Preparazione per la scrittura
buffer.clear();
```

Serializzazione

La serializzazione è il processo di conversione di un oggetto in un formato che può essere salvato su disco o trasmesso attraverso la rete.

Serializzazione in Java

In Java, un oggetto può essere serializzato se la sua classe implementa l'interfaccia `Serializable`:

```
import java.io.Serializable;

public class Persona implements Serializable {
    private static final long serialVersionUID = 1L;

    private String nome;
    private String cognome;
    private int eta;

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

    // Getter e setter...

    @Override
    public String toString() {
        return "Persona{nome='" + nome + "', cognome='" + cognome + "', eta=" + eta + "}";
    }
}
```

Scrittura e Lettura di Oggetti Serializzati

```
import java.io.*;

public class SerializationExample {
    public static void main(String[] args) {
        // Crea un oggetto
        Persona persona = new Persona("Mario", "Rossi", 30);

        try {
            // Serializza l'oggetto su file
            FileOutputStream fileOut = new FileOutputStream("persona.ser");
            ObjectOutputStream out = new ObjectOutputStream(fileOut);
            out.writeObject(persona);
            out.close();
            fileOut.close();
            System.out.println("Oggetto serializzato salvato in persona.ser");

            // Deserializza l'oggetto dal file
            FileInputStream fileIn = new FileInputStream("persona.ser");
            ObjectInputStream in = new ObjectInputStream(fileIn);
            Persona personaLetta = (Persona) in.readObject();
            in.close();
            fileIn.close();
            System.out.println("Oggetto deserializzato: " + personaLetta);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Serializzazione su Rete

```
// Server
ServerSocket serverSocket = new ServerSocket(8080);
Socket socket = serverSocket.accept();

ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
out.writeObject(new Persona("Mario", "Rossi", 30));

// Client
Socket socket = new Socket("localhost", 8080);
ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
Persona persona = (Persona) in.readObject();
```

Problematiche della Serializzazione

Come evidenziato negli appelli d'esame:

> "Indicare quali fra le seguenti sono problematiche che rendono la serializzazione un processo complesso, che richiede molte attenzioni:"
>
> Risposta: "Il processo di serializzazione deve essere efficiente nel tempo e nello spazio impiegati. Un oggetto può contenere altri oggetti, che potrebbbero non essere rappresentabili. È necessario disporre di un metodo per verificare integrità e affidabilità dei dati ricevuti. Mittente e ricevente possono avere versioni diverse dell'oggetto serializzato."

Le principali problematiche sono:

1. **Efficienza**: La serializzazione può essere costosa in termini di tempo e spazio
2. **Oggetti non serializzabili**: Non tutti gli oggetti possono essere serializzati (es. connessioni, thread)
3. **Integrità dei dati**: Necessità di verificare che i dati non siano stati corrotti durante la trasmissione
4. **Compatibilità tra versioni**: Gestione delle differenze tra versioni della stessa classe
5. **Sicurezza**: La deserializzazione di dati non fidati può rappresentare un rischio di sicurezza

Strategie per Affrontare le Problematiche

1. **Efficienza**: Utilizzare formati di serializzazione più efficienti (es. JSON, Protocol Buffers)
2. **Oggetti non serializzabili**: Marcare i campi non serializzabili con `transient`
3. **Integrità dei dati**: Aggiungere checksum o firme digitali
4. **Compatibilità tra versioni**: Gestire esplicitamente `serialVersionUID` e implementare metodi `readObject` e `writeObject` personalizzati
5. **Sicurezza**: Validare i dati deserializzati e limitare le classi che possono essere deserializzate

```
// Esempio di campo transient
public class Connessione implements Serializable {
    private static final long serialVersionUID = 1L;

    private String url;
    private String username;
    private transient Socket socket; // Non sarà serializzato

    // Metodo chiamato dopo la deserializzazione
    private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
        // Legge i campi non transient
        in.defaultReadObject();

        // Ricrea i campi transient
        if (url != null) {
            try {
                URL serverURL = new URL(url);
                socket = new Socket(serverURL.getHost(), serverURL.getPort());
            } catch (Exception e) {
                // Gestione dell'errore
            }
        }
    }
}
```

Protocolli di Comunicazione

Un protocollo di comunicazione definisce le regole per lo scambio di messaggi tra sistemi.

Protocolli a Livello di Applicazione

1. **HTTP/HTTPS**: Per il web e le API REST
2. **FTP**: Per il trasferimento di file
3. **SMTP/POP3/IMAP**: Per la posta elettronica
4. **WebSocket**: Per la comunicazione bidirezionale in tempo reale
5. **MQTT**: Per l'Internet of Things (IoT)

Implementazione di un Protocollo Semplice

```
// Definizione del protocollo
public class Protocollo {
    // Tipi di messaggio
    public static final byte RICHIESTA = 1;
    public static final byte RISPOSTA = 2;
    public static final byte ERRORE = 3;

    // Formato del messaggio:
    // - 1 byte: tipo di messaggio
    // - 4 byte: lunghezza del payload
    // - N byte: payload (in UTF-8)

    public static void inviaMessaggio(OutputStream out, byte tipo, String payload) throws IOException {
        byte[] payloadBytes = payload.getBytes("UTF-8");

        // Scrive il tipo di messaggio
        out.write(tipo);

        // Scrive la lunghezza del payload
        out.write((payloadBytes.length >> 24) & 0xFF);
        out.write((payloadBytes.length >> 16) & 0xFF);
        out.write((payloadBytes.length >> 8) & 0xFF);
        out.write(payloadBytes.length & 0xFF);

        // Scrive il payload
        out.write(payloadBytes);
```

```
        out.flush();
    }

    public static Messaggio riceviMessaggio(InputStream in) throws IOException {
        // Legge il tipo di messaggio
        int tipo = in.read();
        if (tipo == -1) {
            return null; // Fine dello stream
        }

        // Legge la lunghezza del payload
        int length = 0;
        for (int i = 0; i < 4; i++) {
            int b = in.read();
            if (b == -1) {
                throw new IOException("Stream terminato prematuramente");
            }
            length = (length << 8) | b;
        }

        // Legge il payload
        byte[] payloadBytes = new byte[length];
        int bytesRead = 0;
        while (bytesRead < length) {
            int count = in.read(payloadBytes, bytesRead, length - bytesRead);
            if (count == -1) {
                throw new IOException("Stream terminato prematuramente");
            }
            bytesRead += count;
        }

        String payload = new String(payloadBytes, "UTF-8");
        return new Messaggio((byte) tipo, payload);
    }

    public static class Messaggio {
        private byte tipo;
        private String payload;

        public Messaggio(byte tipo, String payload) {
            this.tipo = tipo;
            this.payload = payload;
        }

        public byte getTipo() {
            return tipo;
        }

        public String getPayload() {
            return payload;
        }
    }
}
```

Utilizzo del Protocollo

```
// Server
ServerSocket serverSocket = new ServerSocket(8080);
Socket clientSocket = serverSocket.accept();

InputStream in = clientSocket.getInputStream();
OutputStream out = clientSocket.getOutputStream();

Protocollo.Messaggio messaggio = Protocollo.riceviMessaggio(in);
if (messaggio.getTipo() == Protocollo.RICHIESTA) {
    String risposta = elaboraRichiesta(messaggio.getPayload());
    Protocollo.inviaMessaggio(out, Protocollo.RISPOSTA, risposta);
} else {
    Protocollo.inviaMessaggio(out, Protocollo.ERRORE, "Tipo di messaggio non valido");
}

// Client
Socket socket = new Socket("localhost", 8080);

OutputStream out = socket.getOutputStream();
InputStream in = socket.getInputStream();

Protocollo.inviaMessaggio(out, Protocollo.RICHIESTA, "Dati richiesti");
Protocollo.Messaggio risposta = Protocollo.riceviMessaggio(in);

if (risposta.getTipo() == Protocollo.RISPOSTA) {
    System.out.println("Risposta: " + risposta.getPayload());
} else {
    System.err.println("Errore: " + risposta.getPayload());
}
```

Gestione delle Risorse

La corretta gestione delle risorse di rete è fondamentale per evitare memory leak e garantire il corretto funzionamento dell'applicazione.

Try-with-resources

Java 7 ha introdotto il costrutto try-with-resources, che semplifica la gestione delle risorse:

```
try (
    ServerSocket serverSocket = new ServerSocket(8080);
    Socket clientSocket = serverSocket.accept();
    InputStream in = clientSocket.getInputStream();
    OutputStream out = clientSocket.getOutputStream()
) {
    // Utilizzo delle risorse
} catch (IOException e) {
    e.printStackTrace();
}
// Le risorse vengono chiuse automaticamente
```

Come evidenziato negli appelli d'esame:

> "In questo codice di esempio:
>
> ```
> try(DatagramSocket socket= new DatagramSocket(8080)) {
>     byte[] buf = new byte[16];
>     DatagramPacket packet = new DatagramPacket(buf, 16);
>     // A
>     String input = new String(packet.getData(), 0, packet.getLength());
>     System.out.println(input);
>     // B
> }
> ```
>
> in che punto va inserita la chiusura della risorsa DatagramSocket:"

> Risposta: "Non è necessaria"

Con try-with-resources, la chiusura delle risorse avviene automaticamente al termine del blocco try.

Gestione delle Eccezioni

La gestione delle eccezioni è particolarmente importante nella programmazione di rete, dove possono verificarsi errori di connessione, timeout, ecc.

```
try {
    // Operazioni di rete
} catch (ConnectException e) {
    System.err.println("Impossibile connettersi al server: " + e.getMessage());
} catch (SocketTimeoutException e) {
    System.err.println("Timeout durante la connessione: " + e.getMessage());
} catch (IOException e) {
    System.err.println("Errore di I/O: " + e.getMessage());
} finally {
    // Chiusura delle risorse
    if (socket != null && !socket.isClosed()) {
        try {
            socket.close();
        } catch (IOException e) {
            // Ignora l'eccezione durante la chiusura
        }
    }
}
```

Timeout

È importante impostare timeout per le operazioni di rete, per evitare che l'applicazione rimanga bloccata indefinitamente:

```
// Timeout per la connessione
socket.connect(new InetSocketAddress("example.com", 80), 5000); // 5 secondi

// Timeout per la lettura
socket.setSoTimeout(3000); // 3 secondi
```

Programmazione di Rete Asincrona

La programmazione di rete asincrona permette di gestire molte connessioni contemporaneamente senza bloccare i thread.

CompletableFuture

Java 8 ha introdotto `CompletableFuture`, che semplifica la programmazione asincrona:

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    try {
        // Connessione al server
        Socket socket = new Socket("example.com", 80);

        // Invio della richiesta
        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
        out.println("GET / HTTP/1.1");
        out.println("Host: example.com");
        out.println();

        // Lettura della risposta
        BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        StringBuilder response = new StringBuilder();
        String line;
        while ((line = in.readLine()) != null) {
            response.append(line).append("\n");
        }

        // Chiusura della connessione
        socket.close();

        return response.toString();
    } catch (IOException e) {
        throw new CompletionException(e);
    }
});

future.thenAccept(response -> {
    System.out.println("Risposta ricevuta:");
    System.out.println(response);
}).exceptionally(e -> {
    System.err.println("Errore: " + e.getMessage());
    return null;
});
```

Reactive Programming

Le Reactive Extensions (Rx) offrono un approccio più dichiarativo alla programmazione asincrona:

```
Observable<String> observable = Observable.create(emitter -> {
    try {
        // Connessione al server
        Socket socket = new Socket("example.com", 80);

        // Invio della richiesta
        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
        out.println("GET / HTTP/1.1");
        out.println("Host: example.com");
        out.println();

        // Lettura della risposta
        BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        String line;
        while ((line = in.readLine()) != null) {
            emitter.onNext(line);
        }

        // Chiusura della connessione
        socket.close();

        emitter.onComplete();
    } catch (IOException e) {
        emitter.onError(e);
    }
});

observable
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe(
        line -> System.out.println("Linea ricevuta: " + line),
        error -> System.err.println("Errore: " + error.getMessage()),
        () -> System.out.println("Risposta completata")
    );
```

8. Sistemi Distribuiti

Concetti Fondamentali

Un sistema distribuito è un insieme di componenti software indipendenti che operano su computer diversi ma appaiono all'utente come un unico sistema coerente. Questi sistemi sono diventati sempre più importanti con la diffusione del cloud computing, dei microservizi e delle applicazioni web scalabili.

Caratteristiche dei Sistemi Distribuiti

1. **Distribuzione geografica**: I componenti sono distribuiti su più macchine, potenzialmente in diverse località geografiche
2. **Comunicazione tramite messaggi**: I componenti comunicano scambiando messaggi attraverso la rete
3. **Autonomia dei componenti**: Ogni componente opera indipendentemente dagli altri
4. **Eterogeneità**: I componenti possono essere implementati con tecnologie diverse
5. **Trasparenza**: La complessità della distribuzione è nascosta all'utente finale

Vantaggi dei Sistemi Distribuiti

1. **Scalabilità**: Possibilità di aggiungere risorse per gestire carichi crescenti
2. **Disponibilità**: Resistenza ai guasti grazie alla ridondanza
3. **Prestazioni**: Elaborazione parallela e distribuzione del carico
4. **Flessibilità**: Possibilità di aggiornare o sostituire componenti indipendentemente

Sfide dei Sistemi Distribuiti

1. **Latenza**: Ritardi nella comunicazione tra componenti
2. **Coerenza dei dati**: Difficoltà nel mantenere una visione coerente dello stato
3. **Gestione dei guasti**: Necessità di gestire guasti parziali
4. **Sicurezza**: Protezione delle comunicazioni e dei dati distribuiti

Consenso Distribuito

Il consenso distribuito è il processo attraverso il quale i nodi di un sistema distribuito raggiungono un accordo su un valore o uno stato.

Problemi di Consenso

Come evidenziato negli appelli d'esame:

> "Se in un sistema distribuito i nodi non trovano un consenso sullo stato del sistema, può accadere che:"
>
> Risposta: "Le risposte del sistema siano incoerenti e dipendano da quale nodo viene contattato."

Quando i nodi non raggiungono un consenso, possono verificarsi:

1. **Incoerenza**: Nodi diversi forniscono risposte diverse
2. **Split-brain**: Il sistema si divide in sottosistemi che operano indipendentemente
3. **Perdita di dati**: Alcune operazioni possono essere perse o duplicate

Algoritmi di Consenso

Paxos

Paxos è uno degli algoritmi di consenso più noti e studiati:

1. **Fase di preparazione**: Un proponente invia una proposta con un numero di sequenza
2. **Fase di promessa**: Gli accettori promettono di non accettare proposte con numeri inferiori
3. **Fase di accettazione**: Il proponente invia il valore da accettare
4. **Fase di apprendimento**: Gli accettori informano i learner del valore accettato

```
// Pseudocodice semplificato di Paxos
class Paxos {
    private int highestProposalSeen = 0;
    private int acceptedProposal = 0;
    private Object acceptedValue = null;

    // Fase di preparazione (ricevuta dal proponente)
    public Response prepare(int proposalNumber) {
        if (proposalNumber > highestProposalSeen) {
            highestProposalSeen = proposalNumber;
            return new Response(true, acceptedProposal, acceptedValue);
        } else {
            return new Response(false, null, null);
        }
    }

    // Fase di accettazione (ricevuta dal proponente)
    public boolean accept(int proposalNumber, Object value) {
        if (proposalNumber >= highestProposalSeen) {
            highestProposalSeen = proposalNumber;
            acceptedProposal = proposalNumber;
            acceptedValue = value;
            return true;
        } else {
            return false;
        }
    }
}
```

Raft

Raft è un algoritmo di consenso progettato per essere più comprensibile di Paxos:

1. **Elezione del leader**: I nodi eleggono un leader che coordina le operazioni
2. **Replicazione del log**: Il leader replica le operazioni su tutti i nodi
3. **Safety**: Garantisce che solo i log completi possano essere applicati

```
// Pseudocodice semplificato di Raft
class RaftNode {
    enum State { FOLLOWER, CANDIDATE, LEADER }

    private State state = State.FOLLOWER;
    private int currentTerm = 0;
    private Integer votedFor = null;
    private List<LogEntry> log = new ArrayList<>();
    private int commitIndex = 0;

    // Timer scaduto: diventa candidato e richiede voti
    public void startElection() {
        state = State.CANDIDATE;
        currentTerm++;
        votedFor = nodeId;
        // Invia richieste di voto a tutti i nodi
    }

    // Riceve richiesta di voto
    public VoteResponse requestVote(int term, int candidateId, int lastLogIndex, int lastLogTerm) {
        if (term < currentTerm) {
            return new VoteResponse(currentTerm, false);
        }

        if ((votedFor == null || votedFor == candidateId) &&
            isLogUpToDate(lastLogIndex, lastLogTerm)) {
            votedFor = candidateId;
            return new VoteResponse(currentTerm, true);
        }

        return new VoteResponse(currentTerm, false);
    }

    // Leader: invia append entries a tutti i follower
    public void appendEntries() {
        // Per ogni follower, invia le voci di log mancanti
    }

    // Riceve append entries
    public AppendResponse appendEntries(int term, int leaderId, int prevLogIndex,
                                        int prevLogTerm, List<LogEntry> entries, int leaderCommit) {
        if (term < currentTerm) {
            return new AppendResponse(currentTerm, false);
        }

        // Verifica che il log sia coerente
        if (log.size() <= prevLogIndex || log.get(prevLogIndex).term != prevLogTerm) {
            return new AppendResponse(currentTerm, false);
        }

        // Aggiunge le nuove voci al log
        // ...

        // Aggiorna l'indice di commit
        if (leaderCommit > commitIndex) {
            commitIndex = Math.min(leaderCommit, log.size() - 1);
        }

        return new AppendResponse(currentTerm, true);
    }
}
```

Teorema CAP

Il teorema CAP (Consistency, Availability, Partition tolerance) afferma che un sistema distribuito può garantire al massimo due delle seguenti proprietà:

1. **Consistency (Coerenza)**: Tutti i nodi vedono gli stessi dati nello stesso momento
2. **Availability (Disponibilità)**: Ogni richiesta riceve una risposta, senza garanzia che sia la più recente
3. **Partition tolerance (Tolleranza alle partizioni)**: Il sistema continua a funzionare nonostante la perdita di messaggi o il guasto di parte della rete

In pratica, poiché le partizioni di rete sono inevitabili, i sistemi distribuiti devono scegliere tra coerenza e disponibilità:

- **Sistemi CP**: Privilegiano la coerenza a scapito della disponibilità (es. database relazionali distribuiti)
- **Sistemi AP**: Privilegiano la disponibilità a scapito della coerenza (es. database NoSQL)

Distribuzione dello Stato

La distribuzione dello stato è una tecnica fondamentale nei sistemi distribuiti, che consiste nel replicare lo stato del sistema su più nodi.

Vantaggi della Distribuzione dello Stato

Come evidenziato negli appelli d'esame:

> "Quali vantaggi si cercano nel distribuire lo stato di un sistema su più nodi:"
>
> Risposta: "Possibilità di gestire uno stato più grande della capacità di una singola macchina. Accesso più rapido da località differenti."

I principali vantaggi sono:

1. **Scalabilità**: Possibilità di gestire stati più grandi di quanto una singola macchina possa contenere
2. **Località dei dati**: Accesso più rapido posizionando i dati vicino agli utenti
3. **Disponibilità**: Resistenza ai guasti grazie alla ridondanza
4. **Bilanciamento del carico**: Distribuzione delle richieste su più nodi

Strategie di Replicazione

Replicazione Sincrona

Nella replicazione sincrona, un'operazione è considerata completata solo quando è stata applicata a tutte le repliche:

```
// Pseudocodice per la replicazione sincrona
public boolean write(String key, String value) {
    boolean success = true;

    // Scrive su tutte le repliche
    for (Node replica : replicas) {
        try {
            success &= replica.write(key, value);
        } catch (Exception e) {
            success = false;
        }
    }

    return success;
}
```

Vantaggi:

- Forte coerenza dei dati
- Garanzia che tutte le repliche siano aggiornate

Svantaggi:

- Latenza elevata
- Disponibilità ridotta (basta un nodo non disponibile per bloccare l'operazione)

Replicazione Asincrona

Nella replicazione asincrona, un'operazione è considerata completata quando è stata applicata alla replica primaria:

```
// Pseudocodice per la replicazione asincrona
public boolean write(String key, String value) {
    // Scrive sulla replica primaria
    boolean success = primaryReplica.write(key, value);

    if (success) {
        // Propaga l'aggiornamento alle altre repliche in modo asincrono
        for (Node replica : secondaryReplicas) {
            executor.submit(() -> {
                try {
                    replica.write(key, value);
                } catch (Exception e) {
                    // Gestione dell'errore
                }
            });
        }
    }

    return success;
}
```

Vantaggi:

- Latenza ridotta
- Maggiore disponibilità

Svantaggi:

- Coerenza eventuale (le repliche potrebbero non essere aggiornate immediatamente)
- Possibilità di perdita di dati in caso di guasto della replica primaria

Coerenza dei Dati

La coerenza dei dati si riferisce al grado di sincronizzazione tra le repliche di un sistema distribuito.

Modelli di Coerenza

1. **Coerenza forte**: Tutte le operazioni sono viste da tutti i nodi nello stesso ordine
2. **Coerenza sequenziale**: Le operazioni di ogni processo appaiono nell'ordine specificato dal programma
3. **Coerenza causale**: Le operazioni causalmente correlate sono viste nello stesso ordine da tutti i nodi
4. **Coerenza eventuale**: Il sistema garantisce che, in assenza di nuovi aggiornamenti, tutte le repliche convergeranno allo stesso stato

```
// Esempio di coerenza eventuale con risoluzione dei conflitti
class EventuallyConsistentStore {
    private Map<String, ValueWithVersion> data = new HashMap<>();

    public void write(String key, String value, long timestamp) {
        ValueWithVersion current = data.get(key);

        // Applica l'aggiornamento solo se è più recente
        if (current == null || timestamp > current.timestamp) {
            data.put(key, new ValueWithVersion(value, timestamp));
        }
    }

    public String read(String key) {
        ValueWithVersion value = data.get(key);
        return value != null ? value.value : null;
    }

    // Sincronizza con un altro nodo
    public void synchronize(EventuallyConsistentStore other) {
        for (Map.Entry<String, ValueWithVersion> entry : data.entrySet()) {
            String key = entry.getKey();
            ValueWithVersion value = entry.getValue();
            other.write(key, value.value, value.timestamp);
        }

        for (Map.Entry<String, ValueWithVersion> entry : other.data.entrySet()) {
            String key = entry.getKey();
            ValueWithVersion value = entry.getValue();
            this.write(key, value.value, value.timestamp);
        }
    }

    private static class ValueWithVersion {
        String value;
        long timestamp;

        ValueWithVersion(String value, long timestamp) {
            this.value = value;
            this.timestamp = timestamp;
        }
    }
}
```

Fallacies of Distributed Computing

Le "Fallacies of Distributed Computing" sono un insieme di assunzioni errate che i programmatori spesso fanno quando sviluppano sistemi distribuiti.

Le Otto Fallacie

1. **La rete è affidabile**: In realtà, le reti possono fallire in molti modi
2. **La latenza è zero**: In realtà, la comunicazione di rete introduce sempre ritardi
3. **La banda è infinita**: In realtà, la banda è limitata e può variare
4. **La rete è sicura**: In realtà, la sicurezza richiede sforzi espliciti
5. **La topologia non cambia**: In realtà, la topologia di rete può cambiare nel tempo
6. **C'è un solo amministratore**: In realtà, i sistemi distribuiti spesso attraversano confini amministrativi
7. **Il costo di trasporto è zero**: In realtà, la serializzazione e il trasferimento dei dati hanno un costo
8. **La rete è omogenea**: In realtà, le reti sono composte da tecnologie e dispositivi diversi

Come evidenziato negli appelli d'esame:

> "La fallacia 'Network is homogeneus' è stata aggiunta alle prime sette da Gosling, proprio in seguito alle prime esperienze con Java. Oggi, la sua rilevanza:"
>
> Risposta: "È ancora maggiore, perché le tipologie e le caratteristiche delle reti sono sempre più varie."

> "La fallacia 'Latency is zero' è ancora rilevante perché:"
>
> Risposta: "Dipende da una grandezza fisica"

Implicazioni per lo Sviluppo

Per affrontare queste fallacie, è necessario:

1. **Progettare per i guasti**: Assumere che la rete possa fallire e gestire gli errori
2. **Minimizzare la comunicazione**: Ridurre il numero di round-trip di rete
3. **Utilizzare timeout**: Non aspettare indefinitamente le risposte
4. **Implementare retry con backoff**: Riprovare le operazioni fallite con intervalli crescenti
5. **Monitorare le prestazioni**: Tenere traccia di latenza, throughput e tasso di errore
6. **Utilizzare la compressione**: Ridurre la quantità di dati trasmessi
7. **Implementare la sicurezza**: Cifrare i dati e autenticare le comunicazioni
8. **Supportare l'eterogeneità**: Utilizzare protocolli e formati standard

```
// Esempio di gestione dei timeout e retry
public Response sendRequestWithRetry(Request request) {
    int maxRetries = 3;
    int retryCount = 0;
    long retryDelayMs = 100;

    while (retryCount < maxRetries) {
        try {
            // Imposta un timeout per la richiesta
            return sendRequestWithTimeout(request, 1000); // 1 secondo di timeout
        } catch (TimeoutException e) {
            // Timeout: riprova dopo un ritardo
            retryCount++;
            if (retryCount < maxRetries) {
                try {
                    Thread.sleep(retryDelayMs);
                    retryDelayMs *= 2; // Backoff esponenziale
                } catch (InterruptedException ie) {
                    Thread.currentThread().interrupt();
                    throw new RuntimeException("Interrupted during retry", ie);
                }
            } else {
                throw new RuntimeException("Request failed after " + maxRetries + " retries", e);
            }
        } catch (Exception e) {
            // Altri errori: non riprovare
            throw new RuntimeException("Request failed", e);
        }
    }

    throw new RuntimeException("Should not reach here");
}
```

Modello degli Attori

Il modello degli attori è un paradigma di programmazione concorrente che si basa sul concetto di "attori" come unità fondamentali di computazione.

Caratteristiche degli Attori

Un attore è un'entità che:

1. Ha uno stato interno privato
2. Può ricevere e inviare messaggi
3. Può creare nuovi attori
4. Può decidere come rispondere al prossimo messaggio

Come evidenziato negli appelli d'esame:

> "In reazione alla ricezione di un messaggio, un attore può:"
>
> Risposta: "Creare nuovi attori. Modificare il suo stato interno. Inviare messaggi ad attori di cui ha un riferimento. Modificare il suo comportamento per la ricezione dei prossimi messaggi."

Implementazione con Akka

Akka è una libreria Java/Scala che implementa il modello degli attori:

```
import akka.actor.AbstractActor;
import akka.actor.ActorRef;
```

```
import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.Props;

// Definizione dei messaggi
class Messages {
    static class Greeting {
        private final String who;

        public Greeting(String who) {
            this.who = who;
        }

        public String getWho() {
            return who;
        }
    }

    static class Greet {
    }
}

// Definizione dell'attore
public class GreeterActor extends AbstractActor {
    private String greeting = "Hello";

    @Override
    public Receive createReceive() {
        return receiveBuilder()
            .match(Messages.Greet.class, this::onGreet)
            .match(String.class, this::onChangeGreeting)
            .build();
    }

    private void onGreet(Messages.Greet greet) {
        // Invia un messaggio a un altro attore
        getSender().tell(new Messages.Greeting(greeting), getSelf());
    }

    private void onChangeGreeting(String message) {
        // Modifica lo stato interno
        greeting = message;
    }

    public static Props props() {
        return Props.create(GreeterActor.class);
    }

    public static void main(String[] args) {
        // Crea il sistema di attori
        ActorSystem system = ActorSystem.create("greeter-system");

        // Crea un attore
        ActorRef greeter = system.actorOf(GreeterActor.props(), "greeter");

        // Invia messaggi all'attore
        greeter.tell(new Messages.Greet(), ActorRef.noSender());
        greeter.tell("Ciao", ActorRef.noSender());
        greeter.tell(new Messages.Greet(), ActorRef.noSender());

        // Termina il sistema dopo un po'
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

```
        system.terminate();
    }
}
```

Vantaggi del Modello degli Attori

1. **Incapsulamento**: Lo stato di un attore è accessibile solo all'attore stesso
2. **Comunicazione asincrona**: Gli attori comunicano tramite messaggi asincroni
3. **Località**: Il modello degli attori astrae dalla posizione fisica degli attori
4. **Scalabilità**: Il modello si adatta naturalmente ai sistemi distribuiti
5. **Tolleranza ai guasti**: Gli attori possono essere supervisionati e riavviati in caso di errori

Framework per Applicazioni Distribuite

I framework per applicazioni distribuite forniscono astrazioni e strumenti per semplificare lo sviluppo di sistemi distribuiti.

Scelta di un Framework

Come evidenziato negli appelli d'esame:

> "Che tipo di rischi devono essere considerati nella scelta e nell'adozione di un framework per la costruzione di applicazioni distribuite?"
>
> Risposta: "Errori operativi dovuti a bachi nel codice del framework. Direzione di sviluppo non allineata con le esigenze dell'evoluzione dell'applicazione. Casi d'uso particolari non coperti dalle funzionalità del framework."

I principali rischi da considerare sono:

1. **Bug nel framework**: Errori nel codice del framework possono compromettere l'intera applicazione
2. **Evoluzione del framework**: Il framework potrebbe evolversi in direzioni non compatibili con le esigenze dell'applicazione
3. **Copertura funzionale**: Il framework potrebbe non supportare tutti i casi d'uso necessari
4. **Lock-in**: Dipendenza eccessiva da un framework specifico può rendere difficile cambiare in futuro
5. **Curva di apprendimento**: Tempo e risorse necessari per padroneggiare il framework
6. **Prestazioni**: Il framework potrebbe introdurre overhead non accettabili
7. **Supporto e comunità**: Disponibilità di supporto, documentazione e comunità attiva

Framework Popolari

Spring Cloud

Spring Cloud fornisce strumenti per lo sviluppo di applicazioni distribuite basate su Spring:

```
// Esempio di servizio con Spring Cloud
@SpringBootApplication
@EnableDiscoveryClient
public class ServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(ServiceApplication.class, args);
    }
}

@RestController
public class ServiceController {
    @Autowired
    private DiscoveryClient discoveryClient;

    @Autowired
    private RestTemplate restTemplate;

    @GetMapping("/service")
    public String service() {
        // Chiama un altro servizio
        List<ServiceInstance> instances = discoveryClient.getInstances("other-service");
        if (instances.isEmpty()) {
            return "Service not available";
        }

        ServiceInstance instance = instances.get(0);
        String url = instance.getUri().toString() + "/api";

        return restTemplate.getForObject(url, String.class);
    }
}
```

Akka

Akka fornisce strumenti per la costruzione di sistemi concorrenti e distribuiti basati sul modello degli attori:

```
// Esempio di cluster con Akka
public class ClusterWorker extends AbstractActor {
    private final Cluster cluster = Cluster.get(getContext().getSystem());

    @Override
    public void preStart() {
        cluster.subscribe(getSelf(), MemberEvent.class, UnreachableMember.class);
    }

    @Override
    public void postStop() {
        cluster.unsubscribe(getSelf());
    }

    @Override
    public Receive createReceive() {
        return receiveBuilder()
            .match(MemberUp.class, this::onMemberUp)
            .match(UnreachableMember.class, this::onUnreachableMember)
            .match(MemberRemoved.class, this::onMemberRemoved)
            .match(Work.class, this::onWork)
            .build();
    }

    private void onMemberUp(MemberUp event) {
        System.out.println("Member up: " + event.member());
    }

    private void onUnreachableMember(UnreachableMember event) {
        System.out.println("Member unreachable: " + event.member());
    }

    private void onMemberRemoved(MemberRemoved event) {
        System.out.println("Member removed: " + event.member());
    }

    private void onWork(Work work) {
        // Esegue il lavoro
        getSender().tell(new Result(work.getId(), "Done"), getSelf());
    }
}
```

Apache Kafka

Apache Kafka è una piattaforma di streaming distribuita che può essere utilizzata per costruire pipeline di dati in tempo reale:

```
// Esempio di producer Kafka
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

Producer<String, String> producer = new KafkaProducer<>(props);

for (int i = 0; i < 100; i++) {
    producer.send(new ProducerRecord<>("my-topic", "key-" + i, "value-" + i));
}

producer.close();

// Esempio di consumer Kafka
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "my-group");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

Consumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("my-topic"));

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("offset = %d, key = %s, value = %s%n",
                          record.offset(), record.key(), record.value());
    }
}
```

Reactive Manifesto

Il Reactive Manifesto è un documento che definisce i principi dei sistemi reattivi, particolarmente adatti per i sistemi distribuiti.

Principi dei Sistemi Reattivi

Come evidenziato negli appelli d'esame:

> "Quando di un sistema reattivo si indicano le sue qualità come Responsive, Resilient, Elastic, Message-Oriented, con la qualità 'Message-Oriented' si intende:"
>
> Risposta: "L'unica primitiva di comunicazione fra i componenti è il messaggio asincrono."

I quattro principi dei sistemi reattivi sono:

1. **Responsive (Reattivo)**: Il sistema risponde in modo tempestivo
2. **Resilient (Resiliente)**: Il sistema rimane reattivo anche in caso di guasti
3. **Elastic (Elastico)**: Il sistema rimane reattivo sotto carichi variabili
4. **Message-Driven (Guidato dai messaggi)**: Il sistema si basa sullo scambio di messaggi asincroni

Implementazione dei Principi

Responsive

```
// Esempio di timeout per garantire la reattività
CompletableFuture<Response> future = service.call()
    .orTimeout(1, TimeUnit.SECONDS)
    .exceptionally(ex -> {
        if (ex instanceof TimeoutException) {
            return Response.fallback("Timeout");
        }
        return Response.error(ex);
    });
```

Resilient

```
// Esempio di circuit breaker per la resilienza
CircuitBreaker circuitBreaker = CircuitBreaker.builder()
    .failureRateThreshold(50)
    .waitDurationInOpenState(Duration.ofMillis(1000))
    .permittedNumberOfCallsInHalfOpenState(2)
    .slidingWindowSize(10)
    .build();

Supplier<Response> decoratedSupplier = CircuitBreaker.decorateSupplier(
    circuitBreaker, () -> service.call());

try {
    Response response = decoratedSupplier.get();
    // Elabora la risposta
} catch (Exception e) {
    // Gestisce l'errore
}
```

Elastic

```
// Esempio di backpressure per l'elasticità
Flowable.range(1, 1000000)
    .onBackpressureBuffer(1000, () -> {}, BackpressureOverflowStrategy.DROP_LATEST)
    .observeOn(Schedulers.computation())
    .subscribe(
        i -> {
            // Elaborazione lenta
            Thread.sleep(1);
            System.out.println("Processed: " + i);
        },
        Throwable::printStackTrace,
        () -> System.out.println("Completed")
    );
```

Message-Driven

```
// Esempio di comunicazione basata su messaggi
@Component
public class OrderProcessor {
    @JmsListener(destination = "orders")
    public void processOrder(Order order) {
        // Elabora l'ordine
        OrderResult result = processOrderInternal(order);

        // Invia il risultato a un'altra coda
        jmsTemplate.convertAndSend("results", result);
    }
}
```

Vantaggi dei Sistemi Reattivi

1. **Migliore esperienza utente**: Tempi di risposta più rapidi e prevedibili
2. **Maggiore disponibilità**: Resistenza ai guasti e degradazione graduale
3. **Migliore utilizzo delle risorse**: Scalabilità efficiente in base al carico
4. **Maggiore manutenibilità**: Componenti disaccoppiati e più facili da evolvere

Microservizi

I microservizi sono un'architettura che struttura un'applicazione come un insieme di servizi indipendenti, ciascuno focalizzato su una specifica funzionalità di business.

Caratteristiche dei Microservizi

1. **Indipendenza**: Ogni servizio può essere sviluppato, distribuito e scalato indipendentemente

2. **Specializzazione**: Ogni servizio si concentra su una singola funzionalità
3. **Comunicazione via API**: I servizi comunicano tramite API ben definite
4. **Autonomia dei team**: Team diversi possono lavorare su servizi diversi
5. **Tecnologie eterogenee**: Ogni servizio può utilizzare la tecnologia più adatta

Esempio di Architettura a Microservizi

```
// Servizio Utenti
@SpringBootApplication
public class UserServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(UserServiceApplication.class, args);
    }
}

@RestController
@RequestMapping("/users")
public class UserController {
    @Autowired
    private UserRepository userRepository;

    @GetMapping("/{id}")
    public User getUser(@PathVariable Long id) {
        return userRepository.findById(id)
            .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND));
    }

    @PostMapping
    public User createUser(@RequestBody User user) {
        return userRepository.save(user);
    }
}

// Servizio Ordini
@SpringBootApplication
public class OrderServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(OrderServiceApplication.class, args);
    }

    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}

@RestController
@RequestMapping("/orders")
public class OrderController {
    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private RestTemplate restTemplate;

    @PostMapping
    public Order createOrder(@RequestBody Order order) {
        // Verifica l'utente chiamando il servizio Utenti
        User user = restTemplate.getForObject(
            "http://user-service/users/" + order.getUserId(),
            User.class
        );

        if (user == null) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "User not found");
        }

        return orderRepository.save(order);
    }
}
```

Vantaggi e Sfide dei Microservizi

Vantaggi:

1. **Scalabilità granulare**: Possibilità di scalare solo i servizi necessari
2. **Resilienza**: Un guasto in un servizio non compromette l'intero sistema
3. **Agilità**: Sviluppo e distribuzione più rapidi
4. **Tecnologie diverse**: Possibilità di utilizzare la tecnologia più adatta per ogni servizio

Sfide:

1. **Complessità distribuita**: Gestione di un sistema distribuito
2. **Consistenza dei dati**: Mantenimento della coerenza tra servizi
3. **Testing end-to-end**: Test dell'intero sistema più complessi
4. **Monitoraggio**: Necessità di strumenti avanzati per il monitoraggio
5. **Deployment**: Gestione di molti servizi indipendenti

9. Domande d'Esame Organizzate per Argomento

Questa sezione raccoglie le domande d'esame estratte dai materiali forniti (appelli, simulazioni, file di domande) e le organizza per argomento, seguendo la struttura del corso. Questo permette uno studio mirato e la verifica della comprensione dei concetti chiave per ciascun modulo.

9.1 Paradigmi di Programmazione

- **Quale delle seguenti affermazioni riguardante le Lambda Expression è vera?** (Appello 1)
  - *Risposta (implicita):* Non rendono Java un linguaggio funzionale perché non sono una entità del linguaggio, ma solo una convenienza sintattica risolta dal compilatore.
- **Quale di queste caratteristiche è propria della sintassi switch-case come espressione?** (Appello 1, Tutteledomande.txt)
  - *Risposta:* L'elenco delle opzioni deve essere esaustivo.

9.2 Java OOP

- **Date le seguenti classi: [class Foo, class Bar extends Foo con inizializzatori e costruttori]. Ordinare le strutture indicate secondo la sequenza con cui verranno eseguite.** (Appello 1, Tutteledomande.txt)
  - *Risposta:* Inizializzatore statico -> Costruttore Foo -> Inizializzatore di istanza -> Costruttore Bar
- **Come molti altri linguaggi Object-Oriented, Java ha a disposizione un meccanismo di ereditarietà [...]. L'ereditarietà in Java ha le seguenti caratteristiche:** (Tutteledomande.txt)
  - *Risposte corrette:* A causa dell'introduzione dei default methods, è possibile causare un Diamond Problem. Una classe può ereditare da una sola altra classe. Una interfaccia può ereditare da un'altra interfaccia. Una classe può implementare più interfacce.
- **Quando si dice che il compilatore Java ha delle capacità di Type Inference si intende che:** (Appello 1, Tutteledomande.txt)
  - *Risposta:* È in grado di dedurre il tipo di alcune espressioni senza che sia necessario indicarlo esplicitamente.
- **I membri di una interfaccia Java:** (Tutteledomande.txt)
  - *Risposta:* Sono tutti pubblici, senza necessità di indicarlo.
- **Perché nel linguaggio Java si è deciso di introdurre i metodi di default nelle Interfacce?** (Appello 1, Tutteledomande.txt)
  - *Risposta:* Per poter estendere delle interfacce consolidate senza richiedere l'aggiornamento del codice esistente.
- **Nel linguaggio Java, una variabile final:** (Appello 1, Tutteledomande.txt)
  - *Risposta:* Deve essere inizializzata contestualmente alla definizione, ed il suo valore non può essere cambiato.
- **Una classe Java dichiarata abstract è visibile:** (Appello 1)
  - *Risposta:* abstract non è un modificatore di visibilità.

9.3 Concorrenza

- **Quando si dice che la parola chiave synchronized introduce una relazione di happens-before nel codice, si intende che:** (Appello 1, Tutteledomande.txt)
  - *Risposta:* Il compilatore viene istruito a garantire che il codice sorvegliato dalla parola chiave synchronized venga effettivamente eseguito prima del codice che lo segue, e da un solo Thread alla volta.
- **Selezionare quali delle seguenti sono condizioni di Coffman necessarie per l'instaurarsi di un deadlock.** (Tutteledomande.txt)
  - *Risposte corrette:* Possesso e attesa, Mutua Esclusione, Risorse non riassegnabili, Attesa circolare.
- **Il compilatore Java può riordinare le istruzioni di un blocco di codice [...]. Per imporre un ordinamento preciso è possibile:** (Tutteledomande.txt)
  - *Risposta:* Usare la parola chiave synchronized per introdurre un ordinamento specifico, con una relazione di happens-before.
- **Un Thread esce dallo stato blocked quando:** (Tutteledomande.txt)
  - *Risposta:* Ottiene la risorsa di sistema che aveva richiesto.
- **Un Thread esce dallo stato timed waiting quando:** (Appello 1)
  - *Risposta:* Ottiene il lock che stava aspettando, oppure viene interrotto o trascorre il timeout impostato.
- **Il modello dei Thread permette ad un Processo di organizzare più linee di esecuzione [...]. Tuttavia, si ritrova a dover gestire:** (Tutteledomande.txt)
  - *Risposta:* L'accesso e la condivisione delle risorse.
- **Le classi del package java.concurrent.atomic:** (Appello 1, Tutteledomande.txt)
  - *Risposta:* Sono particolarmente efficienti in caso di modifica concorrente del dato che rappresentano perché usano (se disponibili) delle funzionalità fornite direttamente dall'hardware.
- **Quale delle seguenti affermazioni riguardo ai rapporti fra Processi, Thread e Fiber è vera:** (Appello 1)
  - *Risposta:* Le risorse dei Processi sono controllate dal Sistema Operativo, mentre all'interno dei Processi i Thread devono direttamente controllare il loro accesso. Le Fiber rendono esplicita la concorrenza con lo scopo di essere ancora più leggere dei Thread.
- **Quale delle seguenti frasi è falsa per una struttura dati non Thread-Safe in caso di accesso concorrente:** (Appello 1)
  - *Risposta:* È più performante nel caso generale.
- **Associare ad ogni condizione di Coffmann una strategia utile per rimuoverla.** (Appello 1)
  - *Risposta:* Attesa circolare → Ordinamento dell'acquisizione, Mutua Esclusione → Algoritmi lock-free, Risorse non riassegnabili → Pre-emption, Possesso e attesa → Assegnazione delle risorse transazionale.

9.4 Stream API

- **Nella implementazione degli Stream della libreria standard, che vantaggio si ottiene dal fatto che la API consente di costruire la catena di elaborazione separatamente dalla sua esecuzione?** (Appello 1, Tutteledomande.txt)
  - *Risposta:* L'implementazione può analizzare le operazioni della catena, e prendere decisioni su come applicarle in funzione delle loro caratteristiche.
- **Una operatore short-circuiting all'interno di una catena di elaborazione di uno Stream può:** (Appello 1, Tutteledomande.txt)
  - *Risposta:* Produrre il risultato prima che lo Stream sia stato interamente consumato.
- **Quale dei seguenti non è uno Stream Flag, cioè una caratteristica che uno Stream può dichiarare ed uno operatore (intermedio o terminale) utilizzare per organizzare l'esecuzione:** (Tutteledomande.txt)
  - *Risposta:* UNTYPED - non è noto a priori il tipo degli elementi.
- **L'interfaccia Collector permette di eseguire la riduzione ad un risultato di uno Stream parallelo in modo più efficiente perché:** (Appello 1)
  - *Risposta:* Perchè gestisce un accumulatore mutabile, che riduce la pressione sulla Garbage Collection.

9.5 Reactive Extensions

- **Nelle Reactive Extensions, quali di queste operazioni non è necessario (o possibile) specificare per elaborare gli oggetti emessi da un observable:** (Appello 1, Tutteledomande.txt)
  - *Risposte corrette:* Il numero di oggetti che l'observable è autorizzato ad inviare. Il comportamento alla richiesta di separazione di uno stream parallelo.
- **Lo scopo delle Reactive Extensions è:** (Appello 1, Tutteledomande.txt)
  - *Risposta:* Fornire una semantica per definire elaborazioni asincrone di sequenze di oggetti.
- **Usando i Reactive Stream, la gestione più granulare della composizione della pipeline di elaborazione dello stream permette di:** (Appello 1, Tutteledomande.txt)
  - *Risposta:* Isolare la parte di pipeline che si desidera sia resa parallela; gli Stream della libreria standard sono o completamente paralleli, o completamente seriali.
- **Nell'astrazione delle Reactive Extensions, un subject può:** (Tutteledomande.txt)
  - *Risposta:* Osservare diversi observable, e comportarsi da observable esso stesso, modificando la struttura dell'elaborazione dello stream.

9.6 Programmazione di Rete

- **Implementare un programma di rete usando l'astrazione dei "Channels" della libreria standard di Java permette di non occuparsi di molti dettagli [...], ma:** (Appello 1, Tutteledomande.txt)
  - *Risposta:* È necessario ristrutturare il nostro codice riorganizzando in metodi che vengono richiamati all'avvenire di specifici eventi di I/O.
- **È importante gestire velocemente l'accettazione di un pacchetto da una DatagramSocket perché:** (Tutteledomande.txt)
  - *Risposta:* I pacchetti ricevuti vengono conservati in buffer limitati; se si riempiono, i messaggi successivi sono scartati.
- **Quali dei seguenti possono essere indicati come vantaggi dell'uso dei Datagram:** (Appello 1, Tutteledomande.txt)
  - *Risposte corrette:* Un singolo Datagram può essere inviato a molti indirizzi con una sola istruzione. Il singolo Datagram è isolato, quindi non è necessario introdurre nel protocollo dei separatori fra messaggi diversi.
- **È importante gestire velocemente l'accettazione di una nuova connessione su di un ServerSocket perché:** (Appello 1)
  - *Risposta:* Finché non c'è un thread che attende in ServerSocket::accept(), le nuove richieste di connessione si accodano su di un buffer del sistema operativo che può avere una lunghezza molto limitata.
- **La comunicazione su Socket ha diversi vantaggi, ma il fatto che i dati vengano presentati come uno stream di byte ha i seguenti svantaggi:** (Appello 1)
  - *Risposte corrette:* Deve essere definito un protocollo con cui i due lati della comunicazione riconoscono l'inizio e la fine dei messaggi. Le due parti devono concordare in qualche modo l'encoding delle stringhe all'interno del protocollo di comunicazione.
- **Indicare quali fra le seguenti sono problematiche che rendono la serializzazione un processo complesso, che richiede molte attenzioni:** (Appello 1, Tutteledomande.txt)
  - *Risposte corrette:* Il processo di serializzazione deve essere efficiente nel tempo e nello spazio impiegati. Un oggetto può contenere altri oggetti, che potrebbbero non essere rappresentabili. È necessario disporre di un metodo per verificare integrità e affidabilità dei dati ricevuti. Mittente e ricevente possono avere versioni diverse dell'oggetto serializzato.
- **In questo codice di esempio: [try-with-resources con DatagramSocket]. in che punto va inserita la chiusura della risorsa DatagramSocket:** (Appello 1)
  - *Risposta:* Non è necessaria.

9.7 Sistemi Distribuiti

- **La fallacia "Network is homogeneus" è stata aggiunta alle prime sette da Gosling [...]. Oggi, la sua rilevanza:** (Tutteledomande.txt)
  - *Risposta:* È ancora maggiore, perché le tipologie e le caratteristiche delle reti sono sempre più varie.
- **Se in un sistema distribuito i nodi non trovano un consenso sullo stato del sistema, può accadere che:** (Appello 1, Tutteledomande.txt)
  - *Risposta:* Le risposte del sistema siano incoerenti e dipendano da quale nodo viene contattato.
- **Quando di un sistema reattivo si indicano le sue qualità come Responsive, Resilient, Elastic, Message-Oriented, con la qualità "Message-Oriented" si intende:** (Tutteledomande.txt)
  - *Risposta:* L'unica primitiva di comunicazione fra i componenti è il messaggio asincrono.
- **Che tipo di rischi devono essere considerati nella scelta e nell'adozione di un framework per la costruzione di applicazioni distribuite?** (Tutteledomande.txt)
  - *Risposte corrette:* Errori operativi dovuti a bachi nel codice del framework. Direzione di sviluppo non allineata con le esigenze dell'evoluzione dell'applicazione. Casi d'uso particolari non coperti dalle funzionalità del framework.
- **Quali vantaggi si cercano nel distribuire lo stato di un sistema su più nodi:** (Appello 1, Tutteledomande.txt)
  - *Risposte corrette:* Possibilità di gestire uno stato più grande della capacità di una singola macchina. Accesso più rapido da località differenti.
- **La fallacia "Latency is zero" è ancora rilevante perché:** (Appello 1)
  - *Risposta:* Dipende da una grandezza fisica.
- **Che tipo di vantaggi si possono avere dall'adottare un framework per la costruzioni di applicazioni distribuite?** (Appello 1)
  - *Risposte corrette:* Facilità di realizzazione perché i dettagli dei protocolli di comunicazione sono nascosti da API di livello più elevato. Facilità di realizzazione perché le parti più strutturali sono già implementate.
- **In reazione alla ricezione di un messaggio, un attore può:** (Appello 1)
  - *Risposte corrette:* Creare nuovi attori. Modificare il suo stato interno. Inviare messaggi ad attori di cui ha un riferimento. Modificare il suo comportamento per la ricezione dei prossimi messaggi.

(Nota: Questa sezione è stata compilata estraendo le domande dai file forniti. Potrebbero esserci sovrapposizioni o domande simili tra i diversi appelli. Le risposte indicate sono quelle suggerite o corrette nei materiali originali, ove disponibili.)

10. Simulazioni d'Esame Complete

Questa sezione contiene simulazioni d'esame complete, organizzate secondo la struttura degli appelli reali del corso di Paradigmi di Programmazione. Ogni simulazione include domande che coprono tutti gli argomenti principali del corso, con le relative risposte e spiegazioni.

Simulazione 1

Parte 1: Paradigmi di Programmazione e Java OOP

1. **Quale delle seguenti affermazioni riguardo alle interfacce funzionali in Java è vera?**

   - a) Possono contenere più di un metodo astratto
   - b) Non possono contenere metodi di default
   - c) Sono annotate con `@FunctionalInterface`
   - d) Possono essere implementate solo tramite lambda expression

   Risposta corretta: c) Sono annotate con `@FunctionalInterface`

   Spiegazione: Un'interfaccia funzionale in Java è un'interfaccia che contiene esattamente un metodo astratto. Può essere annotata con `@FunctionalInterface`, che fa sì che il compilatore verifichi che l'interfaccia rispetti i requisiti di un'interfaccia funzionale. Può contenere metodi di default e metodi statici, oltre al singolo metodo astratto.

2. **Quale delle seguenti non è una caratteristica del pattern Builder?**

   - a) Permette la costruzione di oggetti complessi passo dopo passo
   - b) Separa la costruzione di un oggetto dalla sua rappresentazione
   - c) Garantisce l'immutabilità dell'oggetto costruito
   - d) Richiede sempre l'uso di una classe interna statica

   Risposta corretta: c) Garantisce l'immutabilità dell'oggetto costruito

   Spiegazione: Il pattern Builder non garantisce di per sé l'immutabilità dell'oggetto costruito. L'immutabilità dipende da come è progettata la classe dell'oggetto. Il Builder permette di costruire oggetti complessi passo dopo passo, separa la costruzione dalla rappresentazione e spesso (ma non sempre) utilizza una classe interna statica.

3. **Data la seguente dichiarazione: `var list = List.of(1, 2, 3);`, quale delle seguenti affermazioni è vera?**

   - a) `list` è di tipo `ArrayList<Integer>`
   - b) `list` è di tipo `List<Integer>` e può essere modificata
   - c) `list` è di tipo `List<Integer>` e non può essere modificata
   - d) Il codice non compila perché manca il tipo esplicito

   Risposta corretta: c) `list` è di tipo `List<Integer>` e non può essere modificata

   Spiegazione: Il metodo `List.of()` restituisce una lista immutabile di tipo `List<Integer>`. La parola chiave `var` permette al compilatore di inferire il tipo dalla parte destra dell'assegnamento, quindi `list` è di tipo `List<Integer>`.

Parte 2: Concorrenza

4. **Quale delle seguenti affermazioni riguardo alla classe `CompletableFuture` è falsa?**

   - a) Permette di comporre operazioni asincrone
   - b) Implementa sia `Future` che `CompletionStage`
   - c) Può essere completata esplicitamente chiamando il metodo `complete()`
   - d) Non può gestire eccezioni durante l'esecuzione asincrona

   Risposta corretta: d) Non può gestire eccezioni durante l'esecuzione asincrona

   Spiegazione: `CompletableFuture` può gestire eccezioni durante l'esecuzione asincrona tramite metodi come `exceptionally()`, `handle()` o `whenComplete()`. Implementa sia `Future` che `CompletionStage`, permette di comporre operazioni asincrone e può essere completata esplicitamente.

5. **Quale delle seguenti situazioni può portare a un deadlock?**

   - a) Due thread che accedono a una variabile volatile
   - b) Due thread che utilizzano strutture dati thread-safe
   - c) Due thread che acquisiscono lock in ordine diverso
   - d) Due thread che utilizzano operazioni atomiche

   Risposta corretta: c) Due thread che acquisiscono lock in ordine diverso

   Spiegazione: Un deadlock può verificarsi quando due o più thread acquisiscono lock in ordine diverso. Ad esempio, se il thread A acquisisce il lock 1 e poi tenta di acquisire il lock 2, mentre il thread B acquisisce il lock 2 e poi tenta di acquisire il lock 1, entrambi i thread possono rimanere bloccati indefinitamente.

Parte 3: Stream API e Reactive Extensions

6. **Quale delle seguenti operazioni su stream è terminale?**

   - a) `map()`
   - b) `filter()`
   - c) `sorted()`
   - d) `reduce()`

   Risposta corretta: d) `reduce()`

   Spiegazione: `reduce()` è un'operazione terminale che produce un risultato a partire dagli elementi dello stream. `map()`, `filter()` e `sorted()` sono operazioni intermedie che restituiscono un nuovo stream.

7. **Nell'ambito delle Reactive Extensions, quale delle seguenti affermazioni riguardo a un `Subject` è vera?**

- a) Può solo emettere elementi, non può sottoscriversi ad altri Observable
- b) È sia un Observable che un Observer
- c) Può emettere solo un elemento prima di completarsi
- d) Non può emettere errori

Risposta corretta: b) È sia un Observable che un Observer

Spiegazione: Un `Subject` nelle Reactive Extensions è sia un Observable (può essere osservato da Observer) sia un Observer (può sottoscriversi ad altri Observable). Questo lo rende utile come ponte tra il mondo imperativo e quello reattivo.

Parte 4: Programmazione di Rete e Sistemi Distribuiti

8. **Quale delle seguenti affermazioni riguardo ai `DatagramSocket` è vera?**

   - a) Garantiscono la consegna dei pacchetti
   - b) Mantengono una connessione persistente
   - c) Sono basati sul protocollo TCP
   - d) Possono inviare pacchetti a più destinatari con una sola operazione

   Risposta corretta: d) Possono inviare pacchetti a più destinatari con una sola operazione

   Spiegazione: I `DatagramSocket` sono basati sul protocollo UDP, che non garantisce la consegna dei pacchetti e non mantiene una connessione persistente. Tuttavia, permettono di inviare pacchetti a più destinatari (broadcast o multicast) con una sola operazione.

9. **Nel contesto dei sistemi distribuiti, cosa afferma il teorema CAP?**

   - a) Un sistema distribuito può garantire contemporaneamente Consistenza, Disponibilità e Tolleranza alle partizioni
   - b) Un sistema distribuito può garantire al massimo due tra Consistenza, Disponibilità e Tolleranza alle partizioni
   - c) Un sistema distribuito deve scegliere tra Consistenza e Disponibilità, poiché la Tolleranza alle partizioni è sempre necessaria
   - d) Un sistema distribuito può garantire solo una tra Consistenza, Disponibilità e Tolleranza alle partizioni

   Risposta corretta: b) Un sistema distribuito può garantire al massimo due tra Consistenza, Disponibilità e Tolleranza alle partizioni

   Spiegazione: Il teorema CAP (Consistency, Availability, Partition tolerance) afferma che un sistema distribuito può garantire al massimo due delle tre proprietà. In pratica, poiché le partizioni di rete sono inevitabili, i sistemi distribuiti devono scegliere tra consistenza e disponibilità.

10. **Nel modello degli attori, quale delle seguenti operazioni non può essere eseguita da un attore in risposta a un messaggio?**

    - a) Creare nuovi attori
    - b) Inviare messaggi ad altri attori
    - c) Modificare direttamente lo stato di un altro attore
    - d) Modificare il proprio comportamento

    Risposta corretta: c) Modificare direttamente lo stato di un altro attore

    Spiegazione: Nel modello degli attori, un attore può creare nuovi attori, inviare messaggi ad altri attori e modificare il proprio stato o comportamento, ma non può modificare direttamente lo stato di un altro attore. La comunicazione tra attori avviene esclusivamente tramite messaggi.

Simulazione 2

Parte 1: Paradigmi di Programmazione e Java OOP

1. **Quale delle seguenti affermazioni riguardo ai record in Java è falsa?**

   - a) Sono implicitamente final
   - b) Possono estendere altre classi
   - c) Forniscono automaticamente metodi equals(), hashCode() e toString()
   - d) I loro campi sono implicitamente final

   Risposta corretta: b) Possono estendere altre classi

   Spiegazione: I record in Java non possono estendere altre classi, ma possono implementare interfacce. Sono implicitamente final, i loro campi sono implicitamente final e forniscono automaticamente metodi equals(), hashCode() e toString() basati sui campi del record.

2. **Quale delle seguenti non è una caratteristica del pattern Singleton?**

   - a) Garantisce che una classe abbia una sola istanza
   - b) Fornisce un punto di accesso globale a tale istanza
   - c) Permette di creare più istanze in caso di necessità
   - d) Può essere implementato con un campo statico privato e un metodo statico pubblico

   Risposta corretta: c) Permette di creare più istanze in caso di necessità

   Spiegazione: Il pattern Singleton è progettato specificamente per garantire che una classe abbia una sola istanza e fornire un punto di accesso globale a tale istanza. Non permette di creare più istanze, poiché ciò contraddirebbe lo scopo del pattern.

3. **Quale delle seguenti affermazioni riguardo alle classi anonime in Java è vera?**

   - a) Possono implementare più interfacce
   - b) Possono estendere una classe e implementare un'interfaccia
   - c) Non possono accedere alle variabili locali del metodo in cui sono definite
   - d) Non possono definire costruttori

   Risposta corretta: d) Non possono definire costruttori

   Spiegazione: Le classi anonime in Java non possono definire costruttori espliciti, poiché non hanno un nome. Possono implementare una sola interfaccia o estendere una sola classe, ma non entrambe. Possono accedere alle variabili locali del metodo in cui sono definite, purché queste siano effettivamente finali o effettivamente finali.

Parte 2: Concorrenza

4. **Quale delle seguenti affermazioni riguardo alla keyword `volatile` in Java è vera?**

   - a) Garantisce l'atomicità delle operazioni
   - b) Garantisce la visibilità delle modifiche tra thread
   - c) È equivalente all'uso di `synchronized`
   - d) Impedisce il riordinamento delle istruzioni solo nel thread corrente

   Risposta corretta: b) Garantisce la visibilità delle modifiche tra thread

   Spiegazione: La keyword `volatile` in Java garantisce che le letture e le scritture di una variabile avvengano direttamente nella memoria principale, assicurando che le modifiche siano visibili a tutti i thread. Non garantisce l'atomicità delle operazioni composte (come `i++`) e non è equivalente a `synchronized`, che fornisce anche mutua esclusione.

5. **Quale delle seguenti classi non fa parte del framework Executor in Java?**

   - a) `ThreadPoolExecutor`
   - b) `ScheduledThreadPoolExecutor`
   - c) `ForkJoinPool`
   - d) `ThreadGroup`

   Risposta corretta: d) `ThreadGroup`

   Spiegazione: `ThreadGroup` non fa parte del framework Executor, ma è una classe che rappresenta un gruppo di thread. `ThreadPoolExecutor`, `ScheduledThreadPoolExecutor` e `ForkJoinPool` sono tutte implementazioni dell'interfaccia `ExecutorService` nel framework Executor.

Parte 3: Stream API e Reactive Extensions

6. **Quale delle seguenti operazioni su stream non è short-circuiting?**

   - a) `findFirst()`
   - b) `limit()`
   - c) `sorted()`
   - d) `anyMatch()`

   Risposta corretta: c) `sorted()`

   Spiegazione: `sorted()` non è un'operazione short-circuiting, poiché deve processare tutti gli elementi dello stream per poterli ordinare. `findFirst()`, `limit()` e `anyMatch()` sono operazioni short-circuiting, che possono terminare l'elaborazione prima di processare tutti gli elementi dello stream.

7. **Nell'ambito delle Reactive Extensions, quale delle seguenti non è una strategia di backpressure?**

   - a) Buffer
   - b) Drop
   - c) Latest
   - d) Parallel

   Risposta corretta: d) Parallel

   Spiegazione: "Parallel" non è una strategia di backpressure nelle Reactive Extensions. Le strategie di backpressure comuni includono Buffer (memorizzare gli elementi in eccesso), Drop (scartare gli elementi in eccesso) e Latest (mantenere solo l'elemento più recente).

Parte 4: Programmazione di Rete e Sistemi Distribuiti

8. **Quale delle seguenti affermazioni riguardo ai `SocketChannel` in Java NIO è falsa?**

   - a) Supportano operazioni di I/O non bloccanti
   - b) Possono essere registrati con un `Selector`
   - c) Sono sempre in modalità bloccante per impostazione predefinita
   - d) Possono essere utilizzati per leggere e scrivere dati

   Risposta corretta: c) Sono sempre in modalità bloccante per impostazione predefinita

   Spiegazione: I `SocketChannel` in Java NIO sono in modalità bloccante per impostazione predefinita, ma possono essere configurati in modalità non bloccante chiamando il metodo `configureBlocking(false)`. Supportano operazioni di I/O non bloccanti, possono essere registrati con un `Selector` e possono essere utilizzati per leggere e scrivere dati.

9. **Quale delle seguenti non è una delle otto fallacie della computazione distribuita?**

   - a) La rete è affidabile
   - b) La latenza è zero
   - c) La banda è infinita
   - d) I sistemi sono sempre compatibili

   Risposta corretta: d) I sistemi sono sempre compatibili

   Spiegazione: "I sistemi sono sempre compatibili" non è una delle otto fallacie della computazione distribuita. Le otto fallacie sono: la rete è affidabile, la latenza è zero, la banda è infinita, la rete è sicura, la topologia non cambia, c'è un solo amministratore, il costo di trasporto è zero, la rete è omogenea.

10. **Nel contesto dei sistemi reattivi, quale delle seguenti non è una delle quattro caratteristiche definite nel Reactive Manifesto?**

    - a) Responsive
    - b) Resilient
    - c) Elastic
    - d) Concurrent

Risposta corretta: d) Concurrent

Spiegazione: "Concurrent" non è una delle quattro caratteristiche definite nel Reactive Manifesto. Le quattro caratteristiche sono: Responsive (reattivo), Resilient (resiliente), Elastic (elastico) e Message-Driven (guidato dai messaggi).

Simulazione 3

Parte 1: Paradigmi di Programmazione e Java OOP

1. **Quale delle seguenti affermazioni riguardo ai metodi di default nelle interfacce Java è falsa?**
   - a) Possono essere sovrascritti dalle classi che implementano l'interfaccia
   - b) Possono accedere ai campi dell'interfaccia
   - c) Possono chiamare altri metodi dell'interfaccia
   - d) Possono essere dichiarati `private` a partire da Java 9

   Risposta corretta: b) Possono accedere ai campi dell'interfaccia

   Spiegazione: I metodi di default nelle interfacce Java non possono accedere ai campi dell'interfaccia, poiché le interfacce possono contenere solo costanti (campi `public static final`). Possono essere sovrascritti dalle classi che implementano l'interfaccia, possono chiamare altri metodi dell'interfaccia e possono essere dichiarati `private` a partire da Java 9.

2. **Quale delle seguenti non è una caratteristica del pattern Observer?**
   - a) Definisce una dipendenza uno-a-molti tra oggetti
   - b) Notifica automaticamente i cambiamenti a tutti gli oggetti dipendenti
   - c) Garantisce l'ordine di notifica degli osservatori
   - d) Permette l'accoppiamento debole tra soggetto e osservatori

   Risposta corretta: c) Garantisce l'ordine di notifica degli osservatori

   Spiegazione: Il pattern Observer non garantisce l'ordine di notifica degli osservatori. Definisce una dipendenza uno-a-molti tra oggetti, notifica automaticamente i cambiamenti a tutti gli oggetti dipendenti e permette l'accoppiamento debole tra soggetto e osservatori.

3. **Data la seguente dichiarazione: `List<? extends Number> list = new ArrayList<Integer>();`, quale delle seguenti operazioni è valida?**
   - a) `list.add(new Integer(1))`
   - b) `list.add(new Double(1.0))`
   - c) `Number n = list.get(0)`
   - d) `list.add(null)`

   Risposta corretta: c) `Number n = list.get(0)`

   Spiegazione: Con un wildcard `? extends Number`, è possibile leggere elementi dalla lista come `Number` (o qualsiasi supertipo di `Number`), ma non è possibile aggiungere elementi alla lista (tranne `null`), poiché il compilatore non può garantire che il tipo specifico della lista accetti l'elemento.

Parte 2: Concorrenza

4. **Quale delle seguenti affermazioni riguardo alla classe `ReentrantLock` è falsa?**
   - a) Supporta l'interruzione durante l'attesa di un lock
   - b) Permette di tentare l'acquisizione di un lock senza bloccarsi
   - c) Supporta lock equi (fair locking)
   - d) È più efficiente di `synchronized` in tutti i casi

   Risposta corretta: d) È più efficiente di `synchronized` in tutti i casi

   Spiegazione: `ReentrantLock` non è più efficiente di `synchronized` in tutti i casi. In scenari semplici, `synchronized` può essere più efficiente grazie alle ottimizzazioni della JVM. `ReentrantLock` offre funzionalità aggiuntive come l'interruzione durante l'attesa, il tentativo di acquisizione non bloccante e il lock equo.

5. **Quale delle seguenti situazioni può portare a una condizione di starvation?**
   - a) Due thread che acquisiscono lock in ordine diverso
   - b) Thread con priorità più alta che occupano continuamente una risorsa
   - c) Thread che cambiano continuamente stato senza fare progressi
   - d) Thread che attendono una condizione che non si verificherà mai

   Risposta corretta: b) Thread con priorità più alta che occupano continuamente una risorsa

   Spiegazione: La starvation si verifica quando un thread non riceve mai accesso alle risorse di cui ha bisogno, tipicamente perché thread con priorità più alta le occupano continuamente. L'acquisizione di lock in ordine diverso può portare a deadlock, thread che cambiano stato senza progredire descrivono un livelock, e thread che attendono una condizione che non si verificherà mai sono in deadlock.

Parte 3: Stream API e Reactive Extensions

6. **Quale delle seguenti operazioni su stream è stateful?**
   - a) `filter()`
   - b) `map()`
   - c) `sorted()`
   - d) `forEach()`

   Risposta corretta: c) `sorted()`

Spiegazione: `sorted()` è un'operazione stateful, poiché deve memorizzare tutti gli elementi dello stream per poterli ordinare. `filter()`, `map()` e `forEach()` sono operazioni stateless, che processano ogni elemento indipendentemente dagli altri.

7. **Nell'ambito delle Reactive Extensions, quale delle seguenti affermazioni riguardo a un `BehaviorSubject` è vera?**

   - a) Emette solo l'ultimo elemento prima del completamento
   - b) Emette tutti gli elementi emessi, indipendentemente da quando avviene la sottoscrizione
   - c) Emette l'elemento più recente e tutti gli elementi successivi
   - d) Non emette elementi ai nuovi sottoscrittori se è già stato completato

   Risposta corretta: c) Emette l'elemento più recente e tutti gli elementi successivi

   Spiegazione: Un `BehaviorSubject` nelle Reactive Extensions emette l'elemento più recente (o un valore iniziale) e tutti gli elementi successivi ai nuovi sottoscrittori. `AsyncSubject` emette solo l'ultimo elemento prima del completamento, `ReplaySubject` emette tutti gli elementi emessi, e un `BehaviorSubject` completato emette il suo ultimo valore seguito da un segnale di completamento.

Parte 4: Programmazione di Rete e Sistemi Distribuiti

8. **Quale delle seguenti affermazioni riguardo alla serializzazione in Java è vera?**

   - a) Tutti gli oggetti Java sono serializzabili per impostazione predefinita
   - b) I campi `transient` vengono serializzati ma con valori predefiniti
   - c) La serializzazione preserva l'intero grafo di oggetti raggiungibili
   - d) Le classi serializzabili devono implementare un costruttore senza argomenti

   Risposta corretta: c) La serializzazione preserva l'intero grafo di oggetti raggiungibili

   Spiegazione: La serializzazione in Java preserva l'intero grafo di oggetti raggiungibili dall'oggetto serializzato, purché tutti siano serializzabili. Non tutti gli oggetti sono serializzabili per impostazione predefinita (devono implementare `Serializable`), i campi `transient` non vengono serializzati, e le classi serializzabili non devono necessariamente implementare un costruttore senza argomenti.

9. **Nel contesto dei sistemi distribuiti, quale delle seguenti affermazioni riguardo all'algoritmo di consenso Raft è falsa?**

   - a) Divide il problema del consenso in tre sottoproblemi: elezione del leader, replicazione del log e safety
   - b) Garantisce la consistenza anche in presenza di partizioni di rete
   - c) È più semplice da comprendere rispetto a Paxos
   - d) Può tollerare fino a metà dei nodi guasti

   Risposta corretta: d) Può tollerare fino a metà dei nodi guasti

   Spiegazione: Raft può tollerare fino a meno della metà dei nodi guasti (f in un sistema di 2f+1 nodi), non fino a metà. Divide il problema del consenso in tre sottoproblemi, è progettato per essere più comprensibile di Paxos e garantisce la consistenza in presenza di partizioni di rete (a scapito della disponibilità, secondo il teorema CAP).

10. **Quale delle seguenti non è una caratteristica dei microservizi?**

    - a) Ogni servizio ha il proprio database
    - b) I servizi comunicano tramite API ben definite
    - c) I servizi condividono lo stesso modello di dati
    - d) Ogni servizio può essere distribuito indipendentemente

    Risposta corretta: c) I servizi condividono lo stesso modello di dati

    Spiegazione: I microservizi non condividono lo stesso modello di dati; al contrario, ogni servizio ha il proprio modello di dati e spesso il proprio database. I servizi comunicano tramite API ben definite e possono essere distribuiti indipendentemente.

11. Approfondimenti e Risorse Aggiuntive

Questa sezione fornisce approfondimenti su argomenti specifici del corso di Paradigmi di Programmazione e risorse aggiuntive per lo studio e la preparazione all'esame.

11.1 Approfondimento: Esecuzione Nativa in Java

L'esecuzione nativa è un argomento importante nel corso, come evidenziato nei materiali d'esame forniti. Questo approfondimento spiega i concetti chiave relativi all'esecuzione nativa in Java.

Cos'è l'Esecuzione Nativa

L'esecuzione nativa in Java si riferisce all'esecuzione di codice compilato direttamente per l'architettura hardware specifica, invece di essere interpretato dalla Java Virtual Machine (JVM) o compilato just-in-time (JIT).

Java Native Interface (JNI)

JNI è un framework che consente al codice Java in esecuzione nella JVM di chiamare e essere chiamato da applicazioni native e librerie scritte in altri linguaggi come C, C++ e Assembly.

```
public class NativeExample {
    // Dichiarazione del metodo nativo
    private native void helloNative();

    // Caricamento della libreria nativa
    static {
        System.loadLibrary("native_example");
    }

    public static void main(String[] args) {
        NativeExample example = new NativeExample();
        example.helloNative();
    }
}
```

Il corrispondente codice C potrebbe essere:

```
#include <jni.h>
#include <stdio.h>
#include "NativeExample.h"

JNIEXPORT void JNICALL Java_NativeExample_helloNative(JNIEnv *env, jobject obj) {
    printf("Hello from native code!\n");
}
```

GraalVM e Native Image

GraalVM è una macchina virtuale Java che include un compilatore ahead-of-time (AOT) chiamato Native Image. Questo strumento consente di compilare applicazioni Java in eseguibili nativi autonomi.

Vantaggi dell'utilizzo di GraalVM Native Image:

- Avvio più rapido delle applicazioni
- Minore consumo di memoria
- Nessuna necessità di una JVM per l'esecuzione
- Distribuzione più semplice

Limitazioni:

- Riflessione limitata
- Serializzazione Java limitata
- Generazione di codice dinamico limitata
- Tempo di compilazione più lungo

Project Panama

Project Panama è un'iniziativa OpenJDK che mira a migliorare e modernizzare il modo in cui Java interagisce con il codice nativo. Include:

- Foreign Function Interface (FFI): Un'API per chiamare funzioni native senza JNI
- Foreign Memory Access API: Per accedere alla memoria al di fuori dell'heap Java
- Vector API: Per sfruttare le istruzioni SIMD dell'hardware

```
// Esempio di Foreign Memory Access API (Java 16+)
import jdk.incubator.foreign.*;

public class ForeignMemoryExample {
    public static void main(String[] args) {
        try (MemorySegment segment = MemorySegment.allocateNative(100)) {
            MemoryAddress address = segment.baseAddress();
            // Utilizzo della memoria nativa
        }
    }
}
```

11.2 Approfondimento: Pattern Funzionali in Java

Funzioni di Ordine Superiore

Le funzioni di ordine superiore sono funzioni che possono accettare altre funzioni come argomenti o restituire funzioni come risultati.

```
import java.util.function.Function;

public class HigherOrderFunctions {
    // Funzione che restituisce una funzione
    public static Function<Integer, Integer> multiplier(int factor) {
        return x -> x * factor;
    }

    // Funzione che accetta una funzione come argomento
    public static <T, R> R applyTwice(Function<T, R> function, T input) {
        return function.apply(function.apply(input));
    }

    public static void main(String[] args) {
        Function<Integer, Integer> triple = multiplier(3);
        System.out.println(triple.apply(5)); // 15

        Function<Integer, Integer> square = x -> x * x;
        System.out.println(applyTwice(square, 2)); // 16 (2² = 4, 4² = 16)
    }
}
```

Currying

Il currying è una tecnica che trasforma una funzione con più argomenti in una sequenza di funzioni, ciascuna con un singolo argomento.

```
import java.util.function.Function;

public class Currying {
    // Funzione tradizionale con due argomenti
    public static int add(int a, int b) {
        return a + b;
    }

    // Versione curried della funzione add
    public static Function<Integer, Function<Integer, Integer>> curriedAdd() {
        return a -> b -> a + b;
    }

    public static void main(String[] args) {
        // Utilizzo della funzione tradizionale
        System.out.println(add(2, 3)); // 5

        // Utilizzo della funzione curried
        Function<Integer, Function<Integer, Integer>> curriedAdd = curriedAdd();
        Function<Integer, Integer> add2 = curriedAdd.apply(2);
        System.out.println(add2.apply(3)); // 5

        // Applicazione diretta
        System.out.println(curriedAdd.apply(2).apply(3)); // 5
    }
}
```

Composizione di Funzioni

La composizione di funzioni permette di combinare più funzioni per crearne una nuova.

```
import java.util.function.Function;

public class FunctionComposition {
    public static <T, R, V> Function<T, V> compose(Function<R, V> f, Function<T, R> g) {
        return x -> f.apply(g.apply(x));
    }

    public static void main(String[] args) {
        Function<Integer, Integer> square = x -> x * x;
        Function<Integer, Integer> addOne = x -> x + 1;

        // Composizione manuale
        Function<Integer, Integer> squareThenAddOne = x -> addOne.apply(square.apply(x));
        System.out.println(squareThenAddOne.apply(3)); // 10 (3² = 9, 9 + 1 = 10)

        // Utilizzo del metodo compose
        Function<Integer, Integer> composed = compose(addOne, square);
        System.out.println(composed.apply(3)); // 10

        // Utilizzo dei metodi andThen e compose di Function
        Function<Integer, Integer> squareThenAddOne2 = square.andThen(addOne);
        Function<Integer, Integer> addOneThenSquare = square.compose(addOne);

        System.out.println(squareThenAddOne2.apply(3)); // 10 (3² = 9, 9 + 1 = 10)
        System.out.println(addOneThenSquare.apply(3)); // 16 (3 + 1 = 4, 4² = 16)
    }
}
```

11.3 Approfondimento: Reactive Programming vs Functional Programming

Reactive Programming

La programmazione reattiva è un paradigma di programmazione che si concentra sulla gestione di flussi di dati asincroni e sulla propagazione dei cambiamenti.

Caratteristiche principali:

- Basata su eventi
- Asincrona
- Non bloccante
- Orientata ai dati

```
// Esempio con RxJava
Observable<String> observable = Observable.create(emitter -> {
    emitter.onNext("Hello");
    emitter.onNext("Reactive");
    emitter.onNext("World");
    emitter.onComplete();
});

observable
    .map(String::toUpperCase)
    .filter(s -> s.length() > 5)
    .subscribe(
        s -> System.out.println("Received: " + s),
        Throwable::printStackTrace,
        () -> System.out.println("Completed")
    );
```

Functional Programming

La programmazione funzionale è un paradigma di programmazione che tratta la computazione come la valutazione di funzioni matematiche ed evita il cambiamento di stato e i dati mutabili.

Caratteristiche principali:

- Funzioni pure
- Immutabilità

- Funzioni di ordine superiore
- Valutazione lazy

```
// Esempio con Stream API
List<String> list = Arrays.asList("Hello", "Functional", "World");

list.stream()
    .map(String::toUpperCase)
    .filter(s -> s.length() > 5)
    .forEach(s -> System.out.println("Processed: " + s));
```

Confronto

Aspetto	Reactive Programming	Functional Programming
Focus	Flussi di dati asincroni	Trasformazioni di dati
Modello	Push (Observable emette)	Pull (Consumer richiede)
Temporalità	Gestisce eventi nel tempo	Trasforma dati esistenti
Errori	Gestione integrata	Try-catch espliciti
Uso tipico	UI, I/O asincrono	Elaborazione dati

11.4 Risorse Online Consigliate

Documentazione Ufficiale

- Java SE Documentation
- RxJava Documentation
- Project Reactor Reference Guide

Libri

- "Java Concurrency in Practice" di Brian Goetz
- "Functional Programming in Java" di Venkat Subramaniam
- "Reactive Programming with RxJava" di Tomasz Nurkiewicz e Ben Christensen

Corsi Online

- Coursera: Parallel, Concurrent, and Distributed Programming in Java
- Pluralsight: Java Fundamentals: Functional Programming
- Udemy: Reactive Programming in Modern Java using Project Reactor

Repository GitHub

- Awesome Java
- Java Design Patterns
- RxJava Samples

11.5 Consigli per la Preparazione all'Esame

Strategie di Studio

1. **Comprensione dei concetti fondamentali**: Assicurati di comprendere i concetti di base di ogni argomento prima di passare agli aspetti più avanzati.
2. **Pratica con il codice**: Implementa esempi pratici per ogni concetto teorico.
3. **Risolvi esercizi**: Lavora su esercizi e problemi che richiedono l'applicazione dei concetti studiati.
4. **Rivedi le domande d'esame**: Analizza le domande degli appelli precedenti e cerca di comprendere il ragionamento dietro le risposte corrette.
5. **Gruppi di studio**: Discuti i concetti con altri studenti per identificare e chiarire eventuali dubbi.

Preparazione per l'Esame Scritto

1. **Rivedi gli appunti**: Concentrati sui concetti chiave e sulle definizioni.
2. **Esercitati con le simulazioni**: Svolgi le simulazioni d'esame in condizioni simili a quelle dell'esame reale.
3. **Gestione del tempo**: Impara a gestire il tempo durante l'esame, dedicando più tempo alle domande con punteggio maggiore.

4. **Leggi attentamente le domande**: Assicurati di comprendere cosa viene richiesto prima di rispondere.
5. **Rispondi prima alle domande più facili**: Inizia con le domande di cui sei sicuro per guadagnare fiducia e punti.

Preparazione per l'Esame Pratico (se previsto)

1. **Esercitati con progetti reali**: Sviluppa piccoli progetti che utilizzano i concetti del corso.
2. **Rivedi gli errori comuni**: Familiarizza con gli errori comuni e come evitarli.
3. **Usa gli strumenti di debugging**: Impara a utilizzare efficacemente gli strumenti di debugging per risolvere problemi.
4. **Testa il tuo codice**: Scrivi test per verificare che il tuo codice funzioni come previsto.
5. **Ottimizza il tuo codice**: Rivedi il tuo codice per migliorarne la leggibilità, l'efficienza e la manutenibilità.