Introduzione ai Paradigmi di Programmazione

I paradigmi di programmazione rappresentano differenti approcci e modelli concettuali utilizzati per organizzare e strutturare il codice. Durante il corso sono stati analizzati i seguenti paradigmi:

Risorse/Scopo	Diverso	Comune
Comuni	Concorrenza	Parallelismo
Isolate	Rete	Distribuzione

Oltre a questi, sono stati presentati anche i paradigmi:

- Reattivo
- Attori

Java: Fondamenti del Linguaggio

Classi e Package

In Java, l'unità principale di organizzazione del codice è la **Classe**. Ogni oggetto fa riferimento alla definizione di una Classe, che determina la struttura del suo stato e il codice che opera su tale stato.

Una classe in Java è dichiarata con la parola chiave `class` seguita dal nome e dalla definizione:

```
class App {
}
```

Per convenzione, le classi Java sono denominate in Pascal Case (iniziale maiuscola).

Una classe appartiene a un **Package**, che permette di organizzare le classi in gruppi gerarchici:

```
package it.unipd.pdp2023;
class App {
}
```

Visibilità

Una classe non può usare un'altra classe qualsiasi: deve averne visibilità e dichiarare l'intenzione di usarla. Ogni classe, nella sua definizione, indica la sua visibilità:

- **Visibilità di default**: in mancanza di indicazioni, una classe è visibile da tutte le classi dello stesso package, ma non dalle classi al di fuori di esso.
- **Visibilità pubblica**: una classe dichiarata `public` è visibile da qualsiasi altra classe caricata dalla JVM.

Modificatore	Classe	Package	Sottoclasse	Universo
public	✅	✅	✅	✅
protected	✅	✅	✅	❌
nessuno	✅	✅	❌	❌
private	✅	❌	❌	❌

Struttura delle Classi

Una classe può contenere:

- Variabili
- Metodi
- Altre classi
- Blocchi di codice anonimi

Variabili

Una classe può contenere diverse variabili che definiscono la struttura dello stato di ciascun oggetto della classe:

```
public class App {
    int a;
    String b;
}
```

Le variabili si dividono principalmente in due categorie:

- **statiche**: ne esiste "una sola" copia, legata alla classe.
- **di istanza**: ogni oggetto ha la propria e fa parte del suo stato.

```
public class App {
    static char c;  // Variabile statica
    int a;          // Variabile di istanza
```

```
    String b;       // Variabile di istanza
}
```

Metodi

Un metodo è definito da:

- Alcuni modificatori (opzionali)
- Parametri di tipo (opzionali)
- Un tipo di ritorno (richiesto, con una eccezione)
- Un nome (minuscolo, con una eccezione)
- Un elenco di parametri (richiesto)
- Un elenco di eccezioni (opzionale)
- Un blocco di codice da eseguire (opzionale)

```
public class App {
    public App() { };  // Costruttore

    int apply(char d) {
        return 0;
    }

    static boolean prepare(String target, int count) throws RuntimeException
{
        return false;
    }
}
```

Una classe può avere uno o più metodi denominati come la classe stessa che sono detti **costruttori**. Un costruttore viene chiamato quando si richiede la creazione di un oggetto della classe.

Ereditarietà

Java ha un meccanismo di ereditarietà singola: una classe può avere una sola superclasse, di cui eredita codice e (parte) dello stato. Una sottoclasse ha accesso ai membri pubblici, package e protected della superclasse, ma non ai membri private.

```
class App {
    private int a;
    protected int b;
}

class Foo extends App {
```

```
    private String c;
}
```

Una classe dichiara di essere sottoclasse di un'altra con la parola chiave `extends` dopo il nome della classe.

Interfacce

Un'Interfaccia dichiara le caratteristiche di un Tipo senza fornire una sua implementazione. Le classi possono dichiarare di implementare un'interfaccia fornendo l'implementazione richiesta.

```
interface Baz {
    int TEST = 1;  // Costante
    void bar();    // Metodo astratto
    String desc(boolean b);  // Metodo astratto
}

class Foo extends App implements Baz {
    private String c;

    public void bar() {}

    public String desc(boolean b) {
        return "";
    }
}
```

Le interfacce in Java permettono di avere una sorta di ereditarietà multipla mitigando il "Diamond Problem" (problema del diamante).

Metodi di Default

A partire da Java 8, le interfacce possono avere dei metodi di default, cioè metodi implementati che si comportano in modo simile a quelli delle superclassi:

```
interface Top {
    default String name() {
        return "unnamed";
    }
}
```

Questo permette di estendere un'interfaccia con nuovi metodi senza che le implementazioni esistenti debbano essere modificate.

Tipi Generici

I tipi generici (o Generics) permettono di scrivere codice indipendente dal tipo specifico usato durante l'esecuzione:

```
interface MappableList<T> {
    void add(T element);
    T head();
    List<T> tail();
    <M> List<M> map(Function<T, M> xform);
}
```

All'interno della definizione, il parametro T può essere usato come un tipo. Al momento dell'uso, è necessario specificare un tipo concreto.

Records

Un record è uno speciale tipo di oggetto immutabile, per il quale il compilatore completa un insieme di metodi di default:

```
record Name(String firstName, String lastName) {}
```

Un record ottiene automaticamente:

- membri privati con metodi di accesso pubblici
- un costruttore con tutti gli elementi del record
- equals, hashCode, toString generati automaticamente dallo stato del record

Annotazioni

Un'annotazione è una speciale interfaccia che può essere usata per aggiungere metadati a strutture sintattiche nel codice:

```
@Override
public String toString() {
    return "Custom toString";
}
```

Le annotazioni forniscono informazioni al compilatore o al runtime senza alterare il comportamento del codice annotato.

Istruzioni ed Espressioni

Espressioni

Un'espressione è una sintassi che produce un valore. Possono contenere:

- **Valori letterali**: `12`, `3.14f`, `true`, `'a'`, `"abcdef"`
- **Operatori**: `+`, `-`, `*`, `/`, `%`, etc.
- **Chiamate di metodi**: `obj.method()`
- **Creazione di oggetti**: `new Object()`

Assegnamento

L'assegnamento è un operatore, quindi un'assegnazione è un'espressione e quindi un'istruzione:

```
int x;
x = 5;  // Assegnamento
```

Istruzioni Condizionali

If-Else

```
if (condizione) {
    // codice se condizione vera
} else {
    // codice se condizione falsa
}
```

Switch-Case

```
switch (variabile) {
    case valore1:
        // codice per valore1
        break;
    case valore2:
    case valore3:
        // codice per valore2 o valore3
        break;
    default:
        // codice di default
        break;
}
```

Pattern Matching

A partire da Java 16, è possibile utilizzare il pattern matching nei costrutti if e switch:

```
if (obj instanceof String s) {
    // s è già castato a String qui
```

```
    System.out.println(s.length());
}
```

Con i record:

```
switch(shape) {
    case Circle(var radius) -> Math.PI * radius * radius;
    case Rectangle(var width, var height) -> width * height;
    case Square(var side) -> side * side;
}
```

Istruzioni di Iterazione

While

```
while (condizione) {
    // corpo del ciclo
}
```

Do-While

```
do {
    // corpo del ciclo
} while (condizione);
```

For

```
for (int i = 0; i < 10; i++) {
    // corpo del ciclo
}

// For-each (enhanced for)
for (String item : collection) {
    // corpo del ciclo
}
```

Eccezioni

```
try {
    // codice che può generare eccezioni
} catch (ExceptionType1 e1) {
    // gestione eccezione di tipo 1
} catch (ExceptionType2 e2) {
    // gestione eccezione di tipo 2
```

```
} finally {
    // codice eseguito sempre
}
```

Try-with-resources

```
try (Resource resource = new Resource()) {
    // usa la risorsa
} // la risorsa viene chiusa automaticamente
```

Libreria Standard Java

Collections API

La libreria delle collezioni fornisce un insieme di interfacce e classi per gestire gruppi di oggetti.

Principali interfacce:

- **Collection**: radice della gerarchia
- **List**: elenco ordinato di elementi
- **Set**: insieme senza duplicati
- **Map**: associazione chiave-valore
- **Queue/Deque**: code e code double-ended

Implementazioni comuni:

- **ArrayList**: implementazione di List basata su array
- **LinkedList**: implementazione di List basata su nodi collegati
- **HashSet**: implementazione di Set basata su hash
- **HashMap**: implementazione di Map basata su hash
- **TreeSet/TreeMap**: implementazioni ordinate

Stream API

Gli Stream permettono di esprimere operazioni di elaborazione sequenziale o parallela su una sorgente di dati:

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

Gli Stream hanno tre parti:

1. **Sorgente**: da dove provengono i dati (collezione, array, ecc.)
2. **Operazioni intermedie**: trasformazioni (filter, map, etc.)
3. **Operazione terminale**: produce un risultato (collect, reduce, etc.)

Optional

La classe `Optional` rappresenta un valore che potrebbe essere presente o assente:

```
Optional<String> optional = Optional.of("value");
optional.ifPresent(System.out::println);
String result = optional.orElse("default");
```

Time API

Java 8 ha introdotto una nuova API per la gestione del tempo nel package `java.time`:

- `Instant`: un punto preciso nella timeline
- `LocalDate`, `LocalTime`, `LocalDateTime`: data/ora senza fuso orario
- `ZonedDateTime`: data/ora con fuso orario
- `Duration`, `Period`: intervalli di tempo

Programmazione Concorrente

Definizione e Problematiche

La programmazione concorrente si occupa della gestione di più processi sulla stessa macchina che operano contemporaneamente condividendo le risorse disponibili.

Principali problematiche:

- **Non determinismo**: l'ordine di esecuzione non è garantito
- **Starvation**: un thread non riceve risorse sufficienti
- **Race conditions**: il risultato dipende dall'ordine di esecuzione
- **Deadlock**: due o più thread si bloccano a vicenda aspettando risorse

Condizioni di Coffman

Le condizioni necessarie per un deadlock sono (Coffman):

1. **Mutual exclusion**: le risorse non possono essere condivise
2. **Hold and wait** (Resource holding): un processo può mantenere risorse mentre ne attende altre
3. **No preemption**: le risorse non possono essere forzatamente rilasciate
4. **Circular wait**: esiste una catena circolare di processi in attesa

Thread in Java

I Thread in Java sono rappresentati dalla classe `Thread`:

```
Thread thread = new Thread(() -> {
    // Codice da eseguire nel thread
    System.out.println("Thread in esecuzione");
});
thread.start();
```

Stati di un Thread:

- **NEW**: il thread è stato creato ma non avviato
- **RUNNABLE**: il thread è in esecuzione o pronto per l'esecuzione
- **BLOCKED**: il thread è in attesa di acquisire un monitor lock
- **WAITING**: il thread è in attesa indefinita
- **TIMED_WAITING**: il thread è in attesa per un tempo specificato
- **TERMINATED**: il thread ha completato l'esecuzione

Sincronizzazione

synchronized

La parola chiave `synchronized` permette di creare blocchi di codice mutuamente esclusivi:

```
synchronized (object) {
    // Sezione critica
}

// Oppure
public synchronized void method() {
    // Metodo sincronizzato
}
```

wait/notify

I metodi `wait()`, `notify()` e `notifyAll()` permettono ai thread di coordinarsi:

```
synchronized (object) {
    while (!condition) {
        object.wait();
    }
    // Codice da eseguire quando la condizione è vera

    // Modifica lo stato e notifica altri thread
```

```
    object.notify();
}
```

Lock

L'interfaccia `Lock` fornisce meccanismi più flessibili di sincronizzazione:

```
Lock lock = new ReentrantLock();
try {
    lock.lock();
    // Sezione critica
} finally {
    lock.unlock();
}
```

Semaphore

La classe `Semaphore` controlla l'accesso a risorse condivise:

```
Semaphore semaphore = new Semaphore(permits);
try {
    semaphore.acquire();
    // Accesso alla risorsa
} finally {
    semaphore.release();
}
```

Classi Thread-Safe

Variabili Atomiche

Il package `java.util.concurrent.atomic` fornisce classi per operazioni atomiche:

```
AtomicInteger counter = new AtomicInteger(0);
counter.incrementAndGet(); // Operazione atomica
```

Variabili volatile

La parola chiave `volatile` garantisce che le letture e scritture avvengano direttamente dalla memoria principale:

```
volatile int counter = 0;
```

Strutture Dati Concorrenti

- `ConcurrentHashMap`: versione thread-safe di HashMap
- `CopyOnWriteArrayList`: lista thread-safe ottimizzata per letture
- `BlockingQueue`: interfaccia per code che supportano operazioni bloccanti

ThreadLocal

Permette di avere variabili con istanze separate per ogni thread:

```
ThreadLocal<Integer> threadLocalValue = ThreadLocal.withInitial(() -> 0);
threadLocalValue.set(42); // Imposta valore per il thread corrente
Integer value = threadLocalValue.get(); // Ottiene il valore per il thread
corrente
```

Executor Framework

L'Executor Framework semplifica la gestione dei thread:

```
ExecutorService executor = Executors.newFixedThreadPool(10);
executor.submit(() -> {
    // Task da eseguire
});
executor.shutdown();
```

Future e Callable

`Future` rappresenta il risultato di un calcolo asincrono:

```
Callable<Integer> task = () -> {
    // Calcolo
    return result;
};
Future<Integer> future = executor.submit(task);
Integer result = future.get(); // Blocca finché il risultato non è
disponibile
```

Parallel Streams

Gli Stream possono essere eseguiti in parallelo:

```
list.parallelStream()
    .filter(predicate)
    .map(mapper)
    .collect(Collectors.toList());
```

Virtual Threads (Project Loom)

I Virtual Thread sono una feature recente di Java che permette di creare thread molto leggeri:

```
Thread vt = Thread.startVirtualThread(() -> {
    // Codice da eseguire
});
```

Programmazione Distribuita

Motivazioni e Caratteristiche

La programmazione distribuita si occupa della gestione di più processi su macchine diverse che operano in modo coordinato.

Motivazioni:

- **Affidabilità**: resistenza ai guasti
- **Suddivisione del carico**: elaborazione distribuita
- **Diffusione**: accesso da più punti

Caratteristiche:

- **Concorrenza dei componenti**: i nodi operano in parallelo
- **Asincronia totale**: non c'è un ordine temporale globale
- **Fallimenti imperscrutabili**: difficile distinguere guasti da ritardi

8 Fallacies of Distributed Computing

1. **The network is reliable**: le reti possono fallire in molti modi
2. **Latency is zero**: c'è sempre un limite fisico alla velocità
3. **Bandwidth is infinite**: la banda disponibile è sempre limitata
4. **The network is secure**: le reti sono vulnerabili a vari tipi di attacchi
5. **Topology doesn't change**: i nodi possono entrare e uscire dalla rete
6. **There is one administrator**: diverse organizzazioni gestiscono diverse parti
7. **Transport cost is zero**: il trasporto di dati ha sempre un costo
8. **The network is homogeneous**: le reti sono composte da tecnologie diverse

Socket

I Socket permettono la comunicazione bidirezionale punto-punto:

```
// Server
ServerSocket serverSocket = new ServerSocket(8080);
Socket clientSocket = serverSocket.accept();
PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
```

```
BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));

// Client
Socket socket = new Socket("localhost", 8080);
PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
```

Datagram

I Datagram permettono l'invio di pacchetti singoli senza connessione:

```
// Sender
DatagramSocket socket = new DatagramSocket();
byte[] buf = "Hello".getBytes();
InetAddress address = InetAddress.getByName("localhost");
DatagramPacket packet = new DatagramPacket(buf, buf.length, address, 8080);
socket.send(packet);

// Receiver
DatagramSocket socket = new DatagramSocket(8080);
byte[] buf = new byte[256];
DatagramPacket packet = new DatagramPacket(buf, buf.length);
socket.receive(packet);
String received = new String(packet.getData(), 0, packet.getLength());
```

Channels (NIO)

I Channels forniscono un'API per operazioni di I/O non bloccanti:

```
AsynchronousServerSocketChannel server =
AsynchronousServerSocketChannel.open()
    .bind(new InetSocketAddress("localhost", 8080));

server.accept(null, new CompletionHandler<AsynchronousSocketChannel, Void>()
{
    @Override
    public void completed(AsynchronousSocketChannel client, Void attachment)
{
        // Gestione della connessione
        server.accept(null, this); // Accetta nuove connessioni
    }

    @Override
    public void failed(Throwable exc, Void attachment) {
        // Gestione degli errori
```

```
    }
});
```

HTTP Client

Java 11 ha introdotto un nuovo HTTP Client:

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://example.com"))
    .build();

HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
System.out.println(response.body());
```

Framework Web

I framework semplificano lo sviluppo di applicazioni distribuite gestendo molti dettagli di basso livello:

- **Vert.x**: framework asincrono event-driven
- **Micronaut**: framework leggero orientato ai microservizi
- **Spring Boot**: framework completo per applicazioni enterprise

Stato Distribuito

CAP Theorem

Il teorema CAP afferma che un sistema distribuito può garantire solo due delle seguenti tre proprietà:

- **Consistency**: ogni lettura riceve il valore più recente
- **Availability**: ogni richiesta riceve una risposta
- **Partition tolerance**: il sistema continua a funzionare nonostante le partizioni di rete

Algoritmi di Consenso

- **Paxos**: algoritmo di consenso che garantisce l'assenza di blocchi in caso di guasto singolo
- **Raft**: algoritmo di consenso progettato per essere più comprensibile di Paxos

CRDT (Conflict-Free Replicated Data Type)

Le CRDT sono strutture dati che possono essere replicate su più nodi e riconciliate automaticamente:

- **Grow-only Counter**: contatore che può solo incrementare
- **Last-Write-Wins Set**: set in cui l'ultima scrittura ha precedenza

Programmazione Reattiva

Reactive Extensions (Rx)

Le Reactive Extensions forniscono una semantica per elaborazioni asincrone di sequenze di oggetti:

```
Observable.just(1, 2, 3, 4, 5)
    .map(n -> n * n)
    .filter(n -> n % 2 == 0)
    .subscribe(
        System.out::println,  // onNext
        Throwable::printStackTrace,  // onError
        () -> System.out.println("Completed")  // onCompleted
    );
```

Componenti principali:

- **Observable**: emette una sequenza di valori
- **Scheduler**: controlla il contesto di esecuzione
- **Subscriber**: consuma i valori emessi
- **Subject**: sia Observable che Observer

Reactive Streams

Reactive Streams estende il modello Rx aggiungendo il concetto di back-pressure:

```
Flowable.range(1, 1000000)
    .onBackpressureBuffer()
    .observeOn(Schedulers.computation())
    .subscribe(new Subscriber<Integer>() {
        @Override
        public void onSubscribe(Subscription s) {
            s.request(Long.MAX_VALUE); // Richiesta di dati
        }

        @Override
        public void onNext(Integer t) {
            // Elaborazione dei dati
        }

        @Override
        public void onError(Throwable t) {
            // Gestione degli errori
```

```
        }

        @Override
        public void onComplete() {
            // Completamento
        }
    });
```

Componenti principali:

- **Publisher**: fornisce dati
- **Subscriber**: consuma dati
- **Subscription**: rappresenta la relazione tra Publisher e Subscriber
- **Processor**: sia Publisher che Subscriber

Reactive Manifesto

Il Reactive Manifesto definisce le caratteristiche dei sistemi reattivi:

- **Responsive**: pronti alla risposta
- **Resilient**: resistenti ai guasti
- **Elastic**: scalabili in base al carico
- **Message Driven**: basati su messaggi asincroni

Modello ad Attori

Caratteristiche

Un Attore è un'unità indipendente di elaborazione con stato privato che comunica solo tramite messaggi.

In reazione a un messaggio, un attore può:

- Mutare il proprio stato interno
- Creare nuovi attori
- Inviare messaggi ad attori noti
- Cambiare il suo comportamento

Implementazioni

Attori minimali

```
public interface Address<T> {
    Address<T> tell(T msg);
}
```

```
public interface Behavior<T> extends Function<T, Effect<T>> {}

public interface Effect<T> extends Function<Behavior<T>, Behavior<T>> {}
```

Akka

Akka è un framework completo per la programmazione basata su attori:

```
class HelloWorld extends AbstractBehavior<HelloWorld.Greet> {
    public static Behavior<Greet> create() {
        return Behaviors.setup(HelloWorld::new);
    }

    private HelloWorld(ActorContext<Greet> context) {
        super(context);
    }

    @Override
    public Receive<Greet> createReceive() {
        return newReceiveBuilder()
            .onMessage(Greet.class, this::onGreet)
            .build();
    }

    private Behavior<Greet> onGreet(Greet command) {
        getContext().getLog().info("Hello {}!", command.whom);
        command.replyTo.tell(new Greeted(command.whom,
getContext().getSelf()));
        return this;
    }

    public static final class Greet {
        public final String whom;
        public final ActorRef<Greeted> replyTo;

        public Greet(String whom, ActorRef<Greeted> replyTo) {
            this.whom = whom;
            this.replyTo = replyTo;
        }
    }

    public static final class Greeted {
        public final String whom;
        public final ActorRef<Greet> from;

        public Greeted(String whom, ActorRef<Greet> from) {
            this.whom = whom;
            this.from = from;
        }
```

```
    }
}
```

Esecuzione Alternativa

GraalVM

GraalVM è un sistema che include:

- Un nuovo compilatore JIT
- Un compilatore ahead-of-time per Java
- Una specifica di codice intermedio ("Truffle")
- Supporto per altri linguaggi

Consente la compilazione di bytecode Java in un eseguibile nativo, riducendo il tempo di avvio e la dimensione dell'eseguibile.

Coordinated Restore at Checkpoint (CRaC)

CRaC è un progetto che permette di:

- "Congelare" lo stato di una JVM
- Riavviarla in un altro momento, in un'altra macchina
- Eliminare il tempo di startup mantenendo le prestazioni a regime

Altri Runtime

- **TornadoVM**: permette di eseguire codice Java su hardware eterogeneo (GPU, FPGA)
- **KotlinNative**: produce eseguibili nativi direttamente dal codice Kotlin

Protocolli di Rete

OSI Layers

Il modello OSI (Open Systems Interconnection) identifica 7 livelli di comunicazione:

1. **Livello fisico**: trasmissione di bit (cavi, segnali)
2. **Livello data link**: gestione dei frame (Ethernet, WiFi)
3. **Livello rete**: routing dei pacchetti (IP)
4. **Livello trasporto**: connessioni end-to-end (TCP, UDP)
5. **Livello sessione**: gestione delle sessioni (RPC)
6. **Livello presentazione**: codifica dei dati (TLS, MIME)
7. **Livello applicazione**: interfaccia con l'utente (HTTP, FTP)

Protocolli Comuni

FTP (File Transfer Protocol)

Protocollo per il trasferimento di file che opera in due modalità:

- **Attiva**: il server si connette al client
- **Passiva**: il client si connette al server

SMTP (Simple Mail Transfer Protocol)

Protocollo per l'invio di email, con supporto per autenticazione e contenuti MIME.

HTTP (Hypertext Transfer Protocol)

Protocollo client-server per il World Wide Web:

- **Metodi**: GET, POST, PUT, DELETE, etc.
- **Stato**: 1xx (Info), 2xx (Successo), 3xx (Redirect), 4xx (Errore client), 5xx (Errore server)
- **Versioni**: HTTP/1.0, HTTP/1.1, HTTP/2, HTTP/3

BitTorrent

Protocollo peer-to-peer per la distribuzione di file:

- Divide i file in pezzi
- Utilizza hash per verificare l'integrità
- Permette lo scambio di pezzi tra client (seed e peer)

Conclusioni

I paradigmi di programmazione studiati offrono diverse strategie per affrontare problemi complessi:

- La **concorrenza** permette di sfruttare meglio le risorse di una singola macchina
- La **distribuzione** consente di estendere l'elaborazione oltre un singolo nodo
- La **reattività** minimizza la latenza di risposta
- Il modello ad **attori** fornisce un'astrazione di alto livello per sistemi concorrenti e distribuiti

Ciascun paradigma ha i suoi punti di forza e le sue limitazioni. La scelta del paradigma più adatto dipende dai requisiti specifici del problema da risolvere e dalle caratteristiche dell'ambiente di esecuzione.