Riassunto

Java e JVM

Java può essere considerato un linguaggio sia compilato che interpretato. Dopo aver scritto il codice sorgente utilizzeremo infatti il compilatore Java per compilarlo ottenendo però, non direttamente un file eseguibile in linguaggio macchina, ma un file che contiene la traduzione del nostro sorgente in un linguaggio molto vicino al linguaggio macchina detto **bytecode**. Una volta ottenuto questo file, il programmatore potrà passarlo poi in input alla JVM (*Java Virtual Machine*) che interpreterà il bytecode ed il nostro programma sarà finalmente eseguito.
E' chiaro quindi che una volta precompilato il file bytecode (con estenzione **.class**) questo potrà essere eseguito su una qualsiasi macchina con una JVM installata.
La JVM rende quindi possibile l'indipendenza del software in produzione dalla piattaforma che lo eseguirà, ricoprendo di fatto il ruolo di *interprete*.

JDK, JRE e jlink

Il **Java Development Kit** è l'ambiente di sviluppo ufficiale di Java e comprende il compilatore, debugger e altro.
Il **Java Runtime Environment** comprende soltanto l'ambiente di runtime.
Con l'introduzione dello strumento **jlink** è possibile creare delle versioni del JDK ottimizzate da distribuire insieme al programma stesso, rendendo non più necessario l'installazione di nulla, visto che l'applicazione stessa conterrà un ambiente di runtime personalizzato.

Firma di un metodo

Si definisce **firma** di un metodo la coppia (nomeMetodo, parametri). Tramite la firma si distingue univocamente ogni metodo da ogni altro e da ogni so overload.

Commenti di documentazione

Conosciuti anche come commenti *Javadoc*

```
/**
 *  Questo commento permette di produrre
 *  la documentazione del codice
 *  in formato HTML, nello standard Javadoc
*/
```

Si usa per spiegare il funzionamento dei componenti Java (principalmente le classi, i metodi, i costruttori, ma non solo), descrivendo i parametri in input, output e il comportamento con una sintassi standard e precisa.

Dati Primitivi: int, byte, float, ...

Vengono definiti **literal** i valori che vengono specificati nel codice sorgente invece che a run-time.
Una variabile primitiva corrisponde all'area di memoria dove è presente il valore primitivo.

Dati complessi: reference (oggetti)

La variabile che rappresenta un oggetto è una particolare variabile detta reference.
Una variabile complessa corrisponde ad una area di memoria dove è presente un indirizzo numerico, il quale individua un'altra area di memoria dove è memorizzato l'oggetto.
Un array in quanto collezione di dati sono oggetti, quindi reference.

Classi Wrapper

Le classi wrapper ci permettono di usare i tipi primitivi come oggetti e sono utilizzate ad esempio con i Collection objects in cui i tipi primitivi non possono essere utilizzati.
(Integer, Double, Character, ...)
Tutte le classi wrapper sono qualificate come *final* perciò non possono essere estese, inoltre sono immutabili cioè non è possibile cambiarne il valore dopo la loro costruzione. Effettuando una operazione che modifica il valore in realtà si andrà a creare un nuovo oggetto (nuova locazione di memoria) con il valore aggiornato.

La classe String

Java permette di usare la classe String come fosse un tipo primitivo.

Deduzione del tipo per variabili locali: *var*

```
var bool = false; // dedotto il tipo booleano
var string = "asd";
var byteInteger = 8; // dedotto il tipo byte
var oggetto = new Libro();
```

La deduzione del tipo funziona solo per variabili locali. Quindi non è applicabile per variabili d'istanza, per tipi di ritorno di un metodo, per tipo di parametro di un metodo, etc.
Inoltre non è possibile utilizzare la parola var per variabili locali che non siano inizializzate inline alla dichiarazione.

Il nuovo Switch: Switch expression

Lo switch expression è stata introdotta per risolvere i problemi del costrutto classico. Per esempio, dimenticare un *break* significherebbe causare un *fall-through*, ovvero l'esecuzione di tutti i *case* sottostanti fino al primo *break*.
Questo costrutto invece ritorna un valore, ecco un esempio:

```
import java.time.Month

public class SeasonSwitchExpressionEnumTest{
    public static void main(String args[]){
        Month month = Month.APRIL;

        String season = switch(month) {
            case DECEMBER, JANUARY, FEBRUARY -> "winter";
            case MARCH, APRIL, MAY -> "spring";
            case JUNE, JULY, AUGUST -> "summer";
            case SEPTEMBER, OCTOBER, NOVEMBRE -> {
                String value = "autumn";
                yield value;
            }
        };

        System.out.println("The season is " + season);
    }
}
```

La notazione '->' può anche puntare ad una espressione che solleva un'eccezione o un blocco di codice contenente diverse istruzioni. In questo caso usiamo lo *yield* per ritornare il valore. Lo *yield* può essere usata soltanto all'interno del blocco di codice.

Exhaustivness dello switch espression

```
enum Colore {
    VERDE,
    GIALLO,
    ROSSO
};

public void cambiaColore(Colore colore){
    stato = switch(colore) {
        case VERDE -> "La luce è verde";
        case GIALLO -> "La luce è gialla";
    }
}
```

Il compilatore può controllare l'enumerazione (Colore) per valutare quali sono tutti i suoi elementi e dare un errore nel caso in cui mancasse un *case* nello switch expression.
E' buona norma inserire un case *default*.

Classi Record

Un record è un nuovo tipo di classe che usufruisce di una sintassi minimale, progettata per essere utilizzata per le classi che rappresentano dati immutabili.

```
[modificatori] record identificatore(header) { }
// per esempio:
public record PuntoFisso(int x, int y) { }
```

Il compilatore compilerà il record come una classe con due attributi (x e y) dichiarate *private* e *final*, con i relativi metodi *getter* e un costruttore per impostare i due attributi. Inoltre saranno presenti anche i metodi: *equals*, *hashcode*, *toString*.

public

Con il modificatore pubblico si garantisce l'accesso da una qualsiasi classe situata in qualsiasi package.

protected

Con il modificatore protected si garantisce l'accessibilità solo all'interno dello stesso package, inoltre le sottoclassi erediteranno il membro così definito anche se non appartenenti allo stesso package. Non può essere usato nella dichiarazione di una classe.
I membri protetti saranno ereditati come fossero pubblici nelle sottoclassi.

nessun modificatore

Se non anteponiamo modificatori d'accesso ad un membro di una classe, esso sarà ereditato solo alle sottoclassi appartenenti al package dove è definito, inoltre la classe sarà visibile soltanto dalle classi appartenenti allo stesso package (**visibilità di package** o di **default**).

private

Questo modificatore restringe la visibilità di un membro di una classe alla classe stessa. Pertanto tali membri non saranno ereditati alle sottoclassi.

static

Metodi statici

Un metodo statico è un metodo che è condiviso da tutti gli oggetti istanziati dalla classe in cui è dichiarato. Appartiene perciò alla classe e non ad un'stanza particolare, e può essere richiamato quindi non da un oggetto ma dalla classe stessa.
I metodi statici possono usare soltanto membri statici.

Attributi statici

Una variabile statica, essendo condivisa da tutte le istanze della classe assumerà lo stesso identico valore per ogni oggetto di una classe. Sarà condivisa tra le istanze della classe. Una variabile di questo tipo può essere utile ad esempio per contare il numero di oggetti istanziati di una classe (per esempio incrementandola nel costruttore e decrementandola nel distruttore).
Una variabile statica viene inizializzata al valore nullo del suo tipo quando viene caricata in memoria, e non al momento dell'istanza dell'oggetto.
Una variabile dichiarata *static* e *public* si considera variabile **globale**, queste possono essere rese costanti con la parola chiave *final*.

Inizializzatore statico

Il modificatore static può anche essere utilizzato per marcare un semplice blocco di codice definito all'interno di una classe. Un blocco di codice statico avrà la caratteristica di essere chiamato al momento del caricamento in memoria della classe stessa, quindi verrà eseguito prima di un costruttore.

```
// Sarà stampato 10 e non 20

public class Esempio {
    private static int a = 10;

    public Esempio(){
        a += 10 // Costrttore
    }

    static {
        System.out.println("valore statico " + a);
    }
}
```

Questo blocco di codice potrà utilizzare variabili definite fuori da esso se e solo se dichiarate static.

Inizializzatore d'istanza

Si implementa includendo codice in un blocco di parentesi graffe all'interno di una classe. La sua caratteristica è quella di essere eseguito quando viene istanziato l'oggetto, prima del costruttore.
Non essendo statico non ha il limite di usare solo membri statici.

E' possibile inserire nella stessa classe più inizializzatori statici e d'istanza. Questi saranno eseguiti in maniera sequenziale a seconda di come sono stati ordinatiall'interno del file, ma quelli statici avrano sempre la precedenza perchè saranno eseguiti nel momento in cui viene caricata la classe in memoria.

Singleton (design pattern)

E' un design pattern che permette di istanziare una unica istanza della classe che lo implementa e ogni volta che si richiama l'istanza verrà richiamata quindi la stessa istanza.

Caratteristiche:

- Costruttore privato, in questo modo il costruttore non può essere richiamato esternamente per creare altri oggetti (inibendo il concetto di singleton).
- Una variabile privata e statica dello stesso tipo della classe (chiamata *instance*) che viene inizializzata da *getInstance()*. Essendo statica appartiene alla classe.
- Un metodo statico pubblico (chiamato *getInstance()*) che ritorna quell'unica istanza della classe stessa, e se ancora non esiste la crea.

Ereditarietà

La classe *java.lang.Object* è la superclasse di tutte le classi. E' in cima alla gerarchia delle classi e quindi tutte le classi ereditano i membri di Object.
L'ereditarietà non è applicabile ai costruttori in quanto è un metodo speciale che quindi non viene ereditato, ma qualsiasi costruttore (anche di default) come prima istruzione invoca sempre un costruttore della superclasse.

modificatore *final*

- una variabile final diviene costante.
- un metodo dichiarato final non può essere riscritto in una sottoclasse (ovvero non è possibile applicare l'*override*)
- una classe dichiarata *final* non può essere estesa.

modificatore *sealed*

Questo modificatore può essere applicato a classi e interfacce.
Una *classe sealed* specifica da quali sottoclassi può essere estesa direttamente, per esempio:

```
public sealed class DiscoOttico permits CD, DVD {
    ...
}
```

Esistono però dei vinsoli da rispettare:

- Le classi che estendono una classe sealed devono essere esplicitamente; *final*, *sealed*, *non-sealed*.
- Le sottoclassi di una classe *sealed* devono risiedere nello stesso package della classe superclasse.
- *sealed* deve essere seguito da *permits*.

modificatore *abstract*

Può essere applicato solo a classi e metodi, mentre *non* può essere applicato a record e variabili.

Metodi astratti

Un metodo astratto è un metodo che non implementa un proprio blocco di codice e che di fatto non potrà essere utilizzato, ma sarà soggetto a riscrittura (override) in una sottoclasse.

Classi astratte

Allo stesso modo una classe astratta non può essere istanziata, ma è una classe generica creata con l'intento di essere estesa da delle sottoclassi con un legame comune ma più specifiche.
Una classe astratta può presentare sia metodi astratti che metodi implementati propri.
Una classe con tutti i metodi astratti viene epsressa nel concetto di interfaccia.

Interfacce

L'interfaccia è un'evoluzione del concetto di classe astratta ed è uno dei tipi di Java. Una classe può implementare più interfacce, simulando il concetto di ereditarietà multipla.
Una interfaccia può estendere una o più interfacce ma mai delle classi.
Una interfaccia può definire al suo interno metodi statici che quindi verrano chiamati direttamente sulla interfaccia, ma tutti i suoi metodi statici non verrano ereditati dalle classi che la implementano.

Metodi di default (Interfacce) (clausola *default*)

```
public interface Solista {
    default void eseguiAssolo() {
        System.out.println("DO RE MI FA SOL"); // questo metodo non è astratto
    }

    void accordaStrumento(); // questo metodo è astratto
}

public interface SolistaBlues extends Solista {
    ...
    // può riscrivere (override) eseguiAssolo()
    // deve riscrivere accordaStrumento()
}
```

Tramite questa clausola è quindi possibile definire dei metodi concreti in una interfaccia che possono comunque essere riscritti (override) dalle classi che la implementano.

Interfacce funzionali

Una interfaccia è detta funzionale quando dichiara un unico metodo definito astratto (ma potrebbe implementare anche altri metodi statici, di default, privati e costanti).
Questo metodo è chiamato **SAM** (Single Abstract Method).

Diamond Problem

Si verifica quando una classe implementa due o più interfacce che hanno un metodo default con la stessa firma.
Si può risolvere con un override del metodo default nella classe, usando anche il metodo super() per richiamare l'implementazione di default nelle interfacce se si desidera.

varargs

```
public class Aritmetica {
    public static double somma(double... nums) {
        double res = 0.0D;
        for(double tmp : nums) {
            res += tmp;
        }
    }

    return res;
}
```

Un metodo che specifica un parametro **varargs**, può accettare un numero non precisato di argomenti (anche zero), evitandoci di overloadarlo. I varargs è considerato un array, si può usare *length*, ecc.
Si può passare un solo varargs per metodo.

Annotazione *@override*

Sbagliare un override, ad esempio cambiando la firma o il tipo di ritorno, comporta la creazione di un nuovo metodo creato quindi per errore (di cui possiamo non accorgerci) perciò è opportuno usare questa annotazione per segnalare al compilatore la nostra intenzione.

operatore *instanceof*

E' un operatore binario che restituisce true se l'oggetto e il tipo con cui viene confrontato sono gli stessi o se è una sottoclasse, altrimenti false.

Variabile di binding

```
if (dip instanceof Programmatore pro) {
    pro.metodo();
}
```

La variabile pro viene definita come *variabile di binding*, lo scope di tale variabile dipende dal risultato del predicato.
La variabile di binding è *final*.

Asserzione

E' una condizione che deve essere verificata affinchè lo sviluppatore consideri corretta quella porzione di codice.
Esempio: `assert b > 0;` il programma va avanti se la condizione viene soddisfatta, altrimenti si arresta.

Eccezioni checked

Situazione imprevista ma gestibile dal programmatore che il compilatore ci obbliga a gestire, derivanti dalla classe `Exception` che a sua volta deriva da `Throwable`.

Eccezioni unchecked

Situazione imprevista che salta fuori a run-time `RuntimeException`.

Gestione delle eccezioni, clausola *finally*

Il codice definito in un blocco *finally* viene eseguito in qualsiasi caso, sia se viene lanciata l'eccezione, sia se non viene lanciata.

```
try {
    int c = 10 / 0;
} catch (ArithmeticException | NullPointerException exc) {
    System.out.println(exc.getMessage());
} catch (Exception exc) {
    exc.printStackTrace();
} finally {
    System.out.println("Operazione terminata");
}
```

Il multi-catch causa errore se usato tra due classi madre-figlia

try-with-resources

Consente la chiusura automatica degli oggetti che è necessario chiudere una volta che sono stati utilizzati:

```
try (Connection conn = DriverManager.getConnection(...);
        Statement stmt = conn.create(...);
        ResultSet rs = stmt.execute(...);
    ) {
        // fai qualcosa...
    } catch (SQLException exc) {
        System.out.println(exc.getMessage());
    }
```

Sia che venga sollevata o meno una eccezione, gli oggetti definiti nel *try* vengono deallocati.

throw e *throws*

La parola chiave `throw` è usata per generare una eccezione su una espressione personalizzata:

```
if (age < 18) {
    throw new ArithmeticException("Access denied");
}
```

La parola chiave `throws` viene invece usata nel prototipo del metodo e serve per specificare che il metodo potrebbe generare eccezioni del tipo specificato.

```
static void checkAge(int age) throws ArithmeticException {
    ...
    throw new ArithmeticException
}
```

Ricapitolo eccezioni

La parola chiave `throw` serve per generare una eccezione checked su una condizione ben precisa e specifica alle esigenze del programmatore. Questa eccezione ripercorrerà al ritroso le classi che l'hanno generata fin quando una di queste classi non la cattura con un blocco `try-catch`.
Se nessuna classe cattura e gestisce l'eccezione questa allora verrà intercettata dal *main* che, se a sua volta non gestisce l'eccezione, terminerà l'esecuzione (l'eccezione può essere quindi considerata unchecked) con un errore a run-time.
La parola chiave `throws` nel prototipo di un metodo serve per suggerire al programmatore che quel metodo può generare appunto una eccezione. Ha quindi il solo scopo informativo al programmatore.

ordinal (enumerazioni)

Metodo di Enum che ritorna la posizione dell'elemento all'interno della Enumerazione.

values (enumerazioni)

Metodo di Enum che ritorna un vettore contenente tutti gli elementi della enumerazione nell'ordine di inserimento.

import static (enumerazioni)

```
import static com.asd.GiornoDellaSettimana.*

// lo static ci permette di scrivere direttamente

System.out.println(SABATO); // anzichè GiornoDellaSettimana.SABATO
```

Enumerazion e Record

Le Enumerazioni estendono la classe Enum, pertanto non possono essere estese, ma possono implementare interfacce.

Allo stesso modo i Record estendno la classe Record.

- Le enmerazioni sono progettate per rappresentare un numero definito di istanze costanti dello stesso tipo.
- I record invece rappresentano contenitori di dati immutabili.

Il costruttore di default (con i parametri dello header) dei Record viene chiamato **costruttore canonico**:

```
// versione compatta, senza bisogno di specifcare parametri, gli attributi verrano settati di default in automatico
public record foto(String formato, boolean aColori) {
    public Foto {
        if (formato.length < 5) throw new IllegalArgumentException("Formato troppo breve");
    }
}
```

Generics <...>

Classe Generic

```
public class Contenitore<T> {
    private T object1;
    private T object2;

    public void setObject(T object) {
        this.object = object;
    }

    public T getObject() {
        return object;
    }
}

Contenitore<String> c1 = new Contenitore<String>();
c1.setObject("Stringa");
String tmp = c1.getObject();
```

`Contenitore<List> obj = new Contenitore<ArrayList>()` è illegale in quanto la gerarchia ereditaria non può essere applicata al parametro.
`List<Integer> obj = new ArrayList<>()` è invece legale perchè List è implementata da ArrayList e questa gerarchia e definita nel tipo e non nel parametro *Integer*.

Una classe generica è una classe che può operare su un tipo di dato parametrizzato, cioè specificato al momento della creazione dell'oggetto.

Collections

Collections è una interfaccia madre ridimensionabile che raggruppa più elementi. Viene estesa da due intefacce: *list* e *set*.

- *List* estende l'interfaccia con collezioni indicizzate ridimensionabili ed è implementata dalle classi *ArrayList* e *LinkedList*.
- *Set* estende l'interfaccia con collezioni ridimensionabili che non ammettono duplicati ed è implementata dalle classi *HashSet* e *TreeSet*.

Iterator

Iterator è una una interfaccia del framework Collections che permette di iterare sugli elementi di una collezione in modo sequenziale. L'iterator ha il vantaggio di poter modificare o eliminare a runtime gli elementi nel frattempo che li scorre.
Metodi più usati: *hasNext()*, *next()*.

Map

Map è una collezione che associa chiavi ad ogni elemento che contiene, che non sono ordinati e non sono neanche duplicati, è più veloce rispetto ad una lista, la più usata è HashMap.

HashSet

HashSet, è una collezione che estende Set e utilizza una HashMap per mappare i suoi elementi e recuperarli velocemente quando occorre.

Erasure

Erasure, è il fenomeno di cancellazione dei tipi generici durante la compilazione, questo per poter adattare il codice anche al Java prima della versione 5. L'ereditarietà dei generics non si basa infatti sul tipo del parametro in quanto l'erasure va a cancellare il tipo parametro al momento della compilazione (questo per mantenere la retro-compatibilità del novo codice con il codice pre-java5).

```
ArrayList<Number> list;
// equivale a
ArrayList list
// a compile time
```

WildCard

E' una sintassi speciale per i tipi generici. Mettendo il *<?>* come parametro trasforma i tipi generici in sola lettura senza la possibilità di modificarli. Il parametro sarà valorizzato solo a Runtime. **Viene usata quando si vuole che un metodo accetti come parametro un tipo generico**. Supponiamo il seguente codice:

```
public static <E> List<? extends E> mergeWildcard(List<? extends E> listOne, List<? extends E> listTwo) {
    return Stream.concat(listOne.stream(), listTwo.stream())
            .collect(Collectors.toList());
}


    List<Double> numbers1 = new ArrayList<>();
    numbers1.add(5);
    numbers1.add(10L);

    List<Byte> numbers2 = new ArrayList<>();
    numbers2.add(15f);
    numbers2.add(20.0);

    List<Byte> numbersMerged = mergeWildcard(numbers1, numbers2);

    /*
    incompatible types: inference variable E has incompatible bounds
    equality constraints: java.lang.Byte
    lower bounds: java.lang.Byte,java.lang.Double
    */
```

Poiché stiamo trasmettendo due elenchi di numeri, ci aspetteremmo di ricevere indietro lo stesso tipo di elenco. Questo potrebbe non succedere perchè ritorniamo un tipo jolly, che potrebbe essere qualsiasi tipo. Inoltre listOne e listTwo possono essere di due tipi diversi.

We can use wildcards with bounds in three ways:

- Unbounded Wildcards: List<?> – represents a list of any type
- Upper Bounded Wildcards: List<? extends Number> – represents a list of Number or its subtypes (for instance, Double or Integer).
- Lower Bounded Wildcards: List<? super Integer> – represents a list of Integer or its supertypes, Number, and Object

Questo serve a limitare il range di tipi possibili per la wildcard e a garantire la compatibilità tra i tipi.
`prendiBottiglia(Bottiglia< ? > b1, Bottiglia< ? > b2)`**
In questo caso b1 e b2 possono essere di tipi diversi.

Metodi Generici (parametrici)

Sono dei metodi che ci permettono di usare i parametri generics.
`public static <T> String copiaOggetti(Contenitore<T> obj1, Contenitore<T> obj2)` senza il < T> prima del tipo di ritorno *String* non si potrebbe usare un parametro generics Contenitore< T>.
In questo caso obj1 e obj2 sono dello stesso tipo generico.

We can bound type parameters in two ways:

- Unbounded Type Parameter: List< T> represents a list of type T
- Bounded Type Parameter: List< T extends Number & Comparable> represents a list of Number or its subtypes such as Integer and Double that implement the Comparable interface

```
public static <E> List<E> mergeTypeParameter(List<? extends E> listOne, List<? extends E> listTwo) {
    return Stream.concat(listOne.stream(), listTwo.stream())
            .collect(Collectors.toList());
}

// ritorna una superclasse dei due parametri in input (i due parametri possono essere di due sottotipi diversi)
```

Comparable

L'interfaccia Comparable mette a disposizione un metodo chiamato compareTo che permette di applicare un criterio di ordinamento per il confronto di 2 oggetti.
Tale metodo è implementata (facendo override) da molte classi come String e le classi Wrapper.
Essendo un'interfaccia, qualsiasi classe che Implementa Comparable deve necessariamente fornire un'implementazione del metodo compareTo.

Il "contratto" di implementazione deve far sì che il metodo deve ritornare:

- Un valore negativo se l'oggetto corrente è minore dell'oggetto fornito in input
- Valore 0 se l'oggetto è uguale a quello passato in input
- Un valore positivo se l'oggetto corrente è maggiore dell'oggetto passato in input

La classe *Class*

La classe *class* appartiene al package *java.lang* e astrae il concetto di classe con metodi come getConstructor, getMethods... che ritornano oggetti di tipo Constructor, Method, che si trovano in *java.lang.reflect*
Ognuna definisce metodi per ricavare informazioni precise, ad esempio la classe Field ha il metodo getModifiers che restituisce i modificatori di na variabile di istanza. E' un tipo generico dicharato con `public Class<T>`.

Istanziare un oggetto *Class*

1. A partire da una stringa usando il metodo `forName(String name)`

```
try {
    Class classe = Class.forNames("java.lang.String");
} catch (ClassNotFoundException exc) {
    ...
}
```

2. A partire da un oggetto invocando il metodo `getClass()` definito in *Object*

```
String a = "stringa";
Class prova = a.getClass();
```

3. Utilizzando un *class literal* `class prova = java.lang.String.class`

Reflection

Un caso pratico di Reflection è nel creare una classe che stampa tutti i metodi(e i propri parametri) di una classe passata come parametro dalla riga di comandi, potremo stamparci anche tutti i metodi privati andando contro l'idea stessa di incapsulamento.

Annotazioni

è un tipo java che viene usato per astrarre metadati(informazioni che descrivono informazioni) che vengono letti tramite reflection da altro codice.

@FunctionalInterface

E' una annotazione marcatrice che ha il compito di annotare una interfaccia funzionale, cioè con un solo metodo astratto.

@Documented

Marcatrice, annota che una documentazione JavaDoc dovrà essere prodotta

@Deprecated

segnala che quell'elemento potrà essere rimosso nelle versioni successive di Java quindi ne sconsiglia l'utilizzo e il compilatore produrrà un warning.

Differenza tra *@Autowired* e *@Inject*

L'autowiring è il processo per mezzo del quale Spring definisce le dipendenze tra bean, sollevando lo sviluppatore dal doverle definire esplicitamente. L'annotazione @autowired definisce i punti di iniezione.

L'annotazione @inject ha la stessa funzione. L'unica differenza sta nel fatto che @autowired è specifica di Spring mentre @inject è standard di Java.

Sebbene simili, in quanto @Autowired e @Inject fondamentalmente servono allo stesso scopo, ci sono diverse differenze tra di loro, che vogliamo riepilogare brevemente:

1. La prima e più importante differenza tra l' annotazione @Autowired e @Inject è che l'annotazione @Inject è disponibile solo dalla versione 3.0 di Spring in poi. Quindi se si desidera utilizzare l'iniezione attraverso annotazioni nella versione 2.5 del framework, allora deve essere obbligatoriamente utilizzata l'annotazione @Autowired.
2. La seconda differenza tra queste due annotazioni è che, diversamente da @Autowired, l'annotazione @Inject richiede l'attributo booleano required. Questo indica al framework se sollevare o meno una eccezione nel caso in cui la dipendenza non può essere risolta, rendendo di fatto la dipendenza

opzionale.
3. La terza e più importante differenza tra le due annotazioni è che @Autowired è specifica di Spring mentre @Inject è lo standard per l'iniezione di dipendenza definita in JSR-330. In generale, è quindi raccomandabile utilizzare l'annotazione standard per la Dependency Injection, a maggior ragione se si pensa che è supportata da Spring e che può essere utilizzata in combinazione con altre annotazioni del framework come @Value e @Lazy.
4. L'annotazione @Autowired è stata aggiunta nella versione 2.5 di Spring al fine di guidare l'iniezione delle dipendenze per mezzo di annotazioni. Lavora in combinazione con l'annotazione @Component e la direttiva `<context: component-scan/>`. Dalla versione 3.0, Spring supporta la specifica JSR-330 e quindi le relative annotazioni, quali @Inject, @Named e @Singleton. Sono inoltre state aggiunte ulteriori annotazioni, quali @Primary, @Lazy e @DependsOn (riferirsi al libro Spring in Action 4th Edition per saperne di più sulle annotazioni specifiche di Spring).
5. L' annotazione @Inject portabile. Poiché @Autowired è specifico per il framework Spring, nel momento in cui si decide di utilizzare un framework alternativo per la DI sarà necessario re-implementare tutta la logica di iniezione delle dipendenza, anche se l'applicazione rimane invariata.

In conclusione poiché il framework Spring è il più popolare container per DI e IOC per le applicazione Java, l'annotazione @Autowired è più comune di @Inject che è meno conosciuta. Ma dal punto di vista della portabilità, è meglio utilizzare @Inject.

Creare una annotazione

```
public @interface DaCompletare {
    String descrizione();
    String assegnataA() default "da assegnare";
}
```

La parola chiave `@interface` serve per definire un'annotazione analgamente a come la parola chiave `class` serve a definire una classe.

Thread

In Java i meccanismi della gestione dei thread risiedono nella classe Thread e nell'interfaccia Runneable.
La classe Thread estrae un concetto che non solo è dinamico, ma addirittura rappresenta l'esecuzione stessa dell'applicazione. Il currentThread non è l'oggetto corrente(il this) ma l'oggetto che esegue l'oggetto corrente. Un oggetto di Thread si trova in un'altra dimensione rispetto agli altri oggetti.

E' possibile assegnare una priorità che va dalla minima 1 alla massima 10 e quella di default è 5.
Per creare altri Thread, dobbiamo implementare l'interfaccia funzionale Runneable che ha il metodo run().

Nel metodo inseriamo tutte le azioni che vogliamo fare e successivamente lanciamo il thread con il metodo start() che non eseguirà subito run(), ma indicherà che quel thread è eseguibile e successivamente verrà lanciato run, in base alle priorità e alla gestione dei thread.
Ogni JVM definisce un thread scheduler che si occuperà del flusso del multithreading, ma esso dipende dal sistema operativo quindi non possiamo definire a priori le priorità dei thread dato che ogni sistema operativo li gestisce in maniera diversa.

L'interfaccia Runnable estesa dalle Thread ci impone di ridefinire il metodo *run()* che definirà il compotamento che vogliamo assegnare l thread creato.

Il metodo run() è lanciato in base allo scheduler ma soltanto dal momento che viene startato con start() messo in stato *ready*

Il modificatore *volatile* è applicabile solo alle variabili di istanza ed è poco utilizzato. Java nel suo utilizzo della memoria esegue diverse ottimizzazioni fra cui quella di creare diverse copie di variabile di istanza a cui accedono vari thread. Il modificatore volatile non permette questa ottimizzazione, garantendo a tutti i thread la stessa area di memoria in quanto non è possibile creare delle copie durante l'esecuzione. Questo garantirà l'accesso in lettura e scrittura in maniera **atomica**, per cui se un thread sta facendo delle operazioni read-write, allora un altro thread dovrà aspettare che il primo finisca prima di poter accedere a quell'area di memoria. Per gestire degli accessi sequenziali di thread alla stessa area di memoria, bisogna utilizzare il modificatore *synchronized* che gestisce l'ingresso e l'uscita grazie ai lock-unlock.
Syncronized e sleep non sono sufficienti a gestire la comunicazione fra thread.
Introduciamo altri metodi fondamentali per questo scopo:

- **wait**, comunica al thread corrente di abbandonare il monitor e mettersi in pausa fino a quando un altro thread non entri nello stesso monitor e chiami il metodo notify.
- **notify**, richiama dallo stato di pausa il primo thread che ha chiamato wait nello stesso oggetto.
- **notifyAll**, richiama dallo stato di pausa tutti i thread che hanno chiamato wait in quello stesso oggetto.

Monitor di un oggetto

In java ogni oggetto ha associato un proprio monitor che è un oggetto utilizzato come blocco di mutua esclusione per i thread, il che significa che solo un thread può entrare in n monitor in un determinato istante. Java non implementa fisicamente il concetto di monitor ma se un thread *t1* entra in un metodo sincronizzato *ms1* di n determinato oggetto *o1* , nessun altro thread potrà entrare in alcun modo metodo sincronizzato dell'oggetto o1 sino a quando t1 non avrà terminato l'esecuzione del metodo ms1.
t1 ha il lock dell'oggetto o1.

Oggetti immutabili

Il motivo per cui si preferiscono degli oggetti immutabili durante la programmazione è che essi sono thread safe, cioè non corrono il rischio di essere modificati(malamente) da dei thread concorrenti.
Affinchè una classe sia immutabile deve accadere che:

- Tutti gli attributi devono essere final e private;
- Non fornire i metodi setter che possono modificare l'oggetto;

- Se dobbiamo passare valori delle variabili d'istanza di tipo reference, bisogna creare delle copie e passare le copie, per evitare che all'esterno si modifichi la variabile d'istanza originale usando il metodo clone() che è un metodo della classe madre Object;
- Impedire l'override dei metodi alle sottoclassi mettendo final come modificatore della classe.

Espressione lambda

Tipi Innestati

Un tipo innestato è un tipo dichiarato all'interno di un altro tipo. La Inner Class è un esempio ed è una qualunque classe interna ad una calsse principale. Essa può essere *protected* o *static*, cosa che normalmente non è possibile per le classi e viene messa all'interno poiché andrà a comunicare con la classe principale tilizzando direttamente le variabili di istanza e non i suoi get/set.

```
public class Outer {
    private String messaggio = "prova";
    private void stampaMessaggio() {
        System.out.println(messaggio + "-Esterna");
    }

    public class Inner {
        public void metodo() {
            System.out.println(messaggio + "-Interna");
        }

        public void print() {
            stampaMessaggio();
        }
    }
}
```

Dal punto di vista dellincapsulamento è una pratica sconsigliata in quanto il tipo contenitore e il tipo contenuto sono strettamente dipendenti l'uno dall'altro, riducendo la loro riusabilità.

Classi Anonime

Le classi anonime sono particolari **classi interne** che spesso si trovano all'interno dei metodi e nascono e muoiono lì dentro.
La classe anonima si può istanziare una sola volta, quindi può essere considerata un'implementazione del pattern Singleton. Essa fa sempre un override di qualche metodo della superclasse o di una interfaccia, nel secondo caso va ad utilizzare un costruttore virtuale dell'interfaccia per istanziare l'oggetto di tipo classe anonima. Risultano utili quando dobbiamo dichiarare al volo qualcosa che useremo come parametro del metodo, anche se questa tecnica è fortemente sconsigliata perché poco Object Oriented.

Espressioni lambda

Un'espressioni Lambda è detta anche funzione anonima o *closure* (*chiusura*), questo perché si tratta di una vera e propria funzione senza nome (non di un metodo) definita al volo come fatto con le classi anonime.
La sintassi è molto complicata `( [lista_parametri] ) -> { blocco di codice }` l'unica cosa che può fare un'espressione lambda è sostituire l'implementazione di una interfaccia funzionale.
E' possibile istanziare oggetti a partire da una interfaccia funzionale usando una espressione lambda, che è una funzione anonima che può essere trattata come un oggetto.

Esempio classe anonima:

```
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Prima di Java 8: Classe anonima");
    }
})
```

Esempio espressione lambda equivalente:

```
new Thread( () -> System.out.println("Java 8: Funzione anonima")).start();
// Oppure
Runnable r = () -> System.out.println("asdasd");
new Thread(r).start();
```

Ma come fa il compilatore a capire il significato di quello che stiamo passando al costruttore di Thread?
Il costruttore di Thread accetta un'istanza di *Runnable* che è una *interfaccia funzionale*.
L'interfaccia Runnable dichiara il metodo *run* e con la sintassi dell'espressione lambda, il compilatore è capace di dedurre automaticamente che stiamo riscrivendo proprio il metodo run. Infatti run è l'unico metodo astratto definito in Runnable.

A livello di sintassi l'espressione lambda definisce sempre il codice di un metodo SAM do una interfaccia funzionale, ma a livello logico sostituisce l'implementazione di una classe anonima, istanziata a partire dall'espressione lambda.
Oltre ad omettere la dichiarazione dell'oggetto, omette parentesi e la parola chiave return. Per le espressioni lambda non è possibile usare *var* e la deduzione dei tipi poiché esse vengono già dedotte.

Esempio classe anonima vs. espressione lambda

```
JButton button1 = new JButton("Classe anonima");
button1.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Prima di java 8");
    }
})
```

```
JButton button2 = new JButton("Espr Lambda");
button2.addActionListener( e->System.out.println("Con java 8") );
```

Sono stati omessi sia il tipo (*ActionEvent*) del parametro *e*, sia le parentesi tonde che circondano la lista dei parametri (questo è possibile perchè c'è un solo parametro), e anche le graffe che delimitano il corpo della funzione (possibile sempre perchè c'è una sola istruzione). Le parti sono omissibili, ma è possibile comunque inserirle: `(ActionEvent e)->{System.out.println(...);};`

Regole e visibilità

Il reference *this*, all'interno di una classe anonima, si riferisce all'oggetto corrente istanziato dentro la classe interna (classe anonima).
`this.nomeMembro` si riferisce ad un membro della classe anonima.
`NomeClasseEsterna.this.nomeMembro` si riferisce ad un membro della classe esterna. Se non c'è ambiguità tra i due identificatori è possibile non usare il *this*.

Con le espresioni lambda invece il reference *this* si riferisce direttamente alla classe esterna perchè l'espressione lambda non è una classe:

```
// La variabile locale ha scope interno al metodo e vive con esso
public class LambdaThis {
    private String s1 = "variabile d'istanza classe esterna";
    public void MetodoContenenteLambda() {
        String s1 = "variabile locale";
        new Thread( ()->System.out.println(this.s1) ).start();
        new Thread( ()->System.out.println(s2) ).start();
    }
}
```

```
> variabile d'istanza classe esterna
> variabile locale
```

```
// La variabile locale dell'espressione lambda ha invece lo scope all'esterno, nella classe esterna.
public class LambdaThis {
    private String s1 = "variabile d'istanza classe esterna";
    new Thread( ()->{
        String s1 = "wer"
        System.out.println(this.s1)
    }).start();
}
```

```
> error: variable s1 already defined in method...
```

Le espressioni lambda possono usare anche variabili locali esterne, cioè dichiarate nel metodo, se e solo se queste sono final o implicitamente final, in altre parole il riferimento alla variabile non può cambiare e neppure il suo valore. Invece se la variabile locale è di tipo reference allora anche dichiarandola final potremmo sempre modificarne la struttura interna.
Ogni variabile d'istanza invece è sempre utilizzabile e modificabile.

```
public void startCount() {
    int count = 0;
    new Thread( () -> {
        while(count < 100) {
            cont++;
        }
    }).start();
}
```

> error: local variable referenced from lambda expression must be final...

```
public void startCount() {
    Integer count = 0;
    int array[] = new int[1];
    array[0] = count;
    new Thread( () -> {
        while(count < 100) {
            array[0]++;
        }
    }).start();
}
```

Reference a metodi

E' una sintassi ancora più compatta delle espressioni lambda, che fa uso di metodi già esistenti. `List<Film> films = v.getFilmFiltrati(FiltroFilm::isBeiFilm);`
FiltroFilm::isBeiFilm è il reference a metodi.
Questo metodo di passagio è usato in ogni occasione in cui bisogna passare del codice dinamicamente ad un metodo. Si pò esseffettuare in 4 modi:

- Reference a un metodo statico, `NomeTipo::nomeDelMetodoStatico`
- Reference ad un metodo d'istanza, `nomeOggetto::nomeMetodoIstanza`
- Reference ad un metodo d'istanza per tipo, `NomeTipo:nomeMetodo` es: `Collections.sort(filmNames, String::compareToIgnoreCase)`
- Reference ad un costruttore usando direttamente la parola chiave *new*, `Chitarra::new`, ci permette di richiamare il costruttore di NomeClasse implementando una sorta di **Pattern Factory**, che è uno dei più famosi design pattern e ha lo scopo di istanziare determinati oggetti.

Interfacce funzionali principali

1. *Predicate< T>*, che ha il metodo *boolean test(T t)*, è un'interfaccia perfetta per fare i test.
2. *Consumer< T>*, che ha il metodo *void accept(T t)*, è utilizzata per aggiornare lo stato di un oggetto.
3. *Supplier< T>*, che ha il metodo *T get()*,si adatta bene per gestire una factory e restituisce un'istanza del tipo parametro dichiarato, quindi è pensato per creare e restituire un oggetto.
4. *Function< T, R>*, che ha il metodo *R apply(T t)*, permette di astrarre il concetto classico di funzione dove c'è un input t e un output r.
5. *UnaryOperator< T>* è un'estensione di Function e ha il metodo T apply(T t) è un'operazione che viene fatta su un singolo parametro che è di I-O. Si adatta bene quando abbiamo bisogno di trasformare un oggetto, ad esempio una stringa che entra in un modo e esce in un altro.

Stream API

`Java.util.stream.Stream` è un'interfaccia generica che semplifica la programmazione Java e offre supporto alla programmazione parallela. Uno stream rappresenta un flusso che ha come sorgente una collezione di dati su cui si possono eseguire operazioni complesse con uno sforzo di programmazione minimo.

Le operazioni che ritornano stream sono dette di *aggregazione*, mentre quelle che non ritornano stream sono metodi *terminali*. Una **Pipeline** è una concatenazioni di Stream, può essere intesa come un tubo composto da diversi tubi collegati in sequenza per far passare un flusso di qualcosa. Essa è costituida da 3 elementi:

1. Una source che di solito coincide con una collection o un array.
2. Un numero N di operazioni di aggregazione dette operazioni intermedie che hanno il compito di ritornare un altro stream, possiamo usare i metodi filter(..), map(..), che svolgono operazioni di aggregazione. Tutti i metodi di aggregazione sono chiamati solo all'ultimo momento quando sarà invocato il metodo terminale e si dicono di tipo lazy(pigri).
3. Almeno un metodo terminale, cioè che non restituisce altri oggetti di tipo stream.

In italiano potremmo dire che gli stream non immagazzinano dati, ma trasportano gli elementi dalla sorgente(collection) attraverso la tubatura(pipeline), trasformandoli(filtrandoli) in un risultato finale.

Classi Optional

Sono delle classi wrapper che consentono di evitare **NullPointerException** incapsulando valori di un certo tipo e richiamando i propri metodo senza bisogno di fare un controllo di nullità. Esistono OptionalInt, OptionalDouble, OptionalLong e Optional< T> che può essere usata con tutti gli oggetti.
NON si possono istanziare oggetti *Optional* con l'operatore *new*, ma occorre sempre passare tramite la chiamata ad un metodo che ne restituisce un'istanza (per esempio *.ofNullable(object)*) e usare il metodo *.orElse(...)* per per passare l'oggetto contenuto nell'Optional o qualcos'altro nel caso in cui l'Optional sia vuoto.

```
// ritorna "Nessun titolo" se titolo è null
public static String getTitoloMaiusc(String titolo) {
    Optional<String> opt = Optional.ofNullable(titolo);
    return opt.orElse("Nessun titolo");
}
```

Metodi get, filter, isPresent, map...

Il metodo map(..) consente di mappare oggetto1 ad oggetto2 restituendo un Optional contenente oggetto2.
Il metodo get() restituisce l'oggetto contenuto nell'optional. Se l'oggetto è null sarà lanciata l'eccezione `NoSuchElementException`.

Esistono anche i metodi getAsLong per OptionalLong e getAsInt per OptionalInt

Stream Sources

Esistono 3 metodi per ottenere degli stream:

1. Stream da valori
2. Stream da funzioni *iterate* e *generate*
3. Stream da array e collection

Stream da valori (metodi *of*)

Permette di creare uno Stream partendo da array o da un varargs di valori.
`Esempio Stream<Integer> nomiStream = Stream.of(arrayDiInteger);`
`Esempio2 Stream<String> stream = Stream.ofNullable("ciao", "asd", "qwe");`

Stream da funzioni (*iterate* e *generate*)

iterate è un metodo statico che prende in input 3 parametri: un valore di *inizializzazione*, un *Predicate* e una *UnaryOperator*.
`Stream<Integer> pari = Stream.iterate(2, n -> n <= 10, n -> n + 2)` lavora sui primi 5 numeri pari.
Esiste anche un overload a cui manca il secondo parametro (*Predicate*).
`Stream<Integer> mult5 = Stream.iterate(0, n -> n + 10).skip(5).limit(6)` produrrà:

> 50, 60, 70, 80, 90, 100

---

generate è metodo statico che fornisce invece uno stream infinito (quindi è necessario chiamare poi il limit)

`Stream<Double> randomDoubles = Stream.generate(Math::random).limit(5).foreach(System::println)`
Tale metodo permette di generare 5 numeri pseudo-casuali e stamparli. Applicati a tali metodi ne possiamo aggiungere altri che, in alcuni casi, sono necessari:

- Limit: ha lo stesso effetto del break ma prende in input un intero che indica quante iterazioni devono essere fatte prima di stoppare il ciclo dello stream.
- Skip: simile al continue e prende sempre un intero che indica quanti elementi generati devono essere ignorati.

Stream da array e collection

Stream da Collection può generare uno stream per qualunque collection, ad esempio:
`Set<String> nome = Set.of("Matteo", "Antonio");`
`Stream<String> nomiStream = nomi.stream();`

Metodi di aggregazione

Oltre i metodi skip() e limit() anche il metodo filter() come si può intuire dal nome, ha il compito di filtrare uno stream restituendo un altro stream, prende in input un *Predicate*.
Anche il metodo distinct() ha il compito di filtrare uno stream restituendo uno stream senza duplicati.

Metodi peek e parallel

Data la complessità nel fare il debug sugli stream, esiste il metodo peek che ci permette di dare un'occhiata all'interno dello stream senza alterarne il valore. Il metodo parallelStream divide lo stream tramite fork-join e consente di effettuare degli stream paralleli che se usati correttamente migliorano le prestazioni.

Metodi terminali

Sono quei metodi che trasformano uno stream in qualcosa di diverso e sono necessari quando abbiamo una pipeline(sequenza di Stream), fra i più importanti troviamo il metodo forEach(), metodi find() (findAny(), findFirst()), metodi match() (anyMatch(), allMatch(), noneMatch()).

Metodi di riduzione

Prendono in input uno stream e tornano un unico valore: min, sum, average, count, max.

Metodo *reduce*

Ha più overload, la più completa prende in input due parametri: un oggetto **identity** che rappresenta sia il valore iniziale sia il valore da restituire nel caso non ci siano elementi nello stream, un **oggetto BinaryOperator** chiamato *accumulator*.
BinaryOperator è una interfaccia funzionale che estende BiFunction
`.reduce(0, (x,y) -> x+y)` è il corrispettivo di `.reduce(0, Integer::Sum)`

Character Stream

Reader e *Writer* sono le superclassi astratte che hanno il compito di obbligare le sottoclassi a leggere e scrivere compatibilmente con il tipo *char* di Java. Vengono usati per le informazioni di tipo testuale.

InputStream e *OutputStream* sono due classi parallele a Reader e Writer, ma sono però destinati alle informazioni di tipo non testuale (file binari, immagini, suoni).
ObjectInputStream e *ObjectOutputStream* sono due classi usate per la serializzazione degli oggetti.

Tutti gli stream sono automaticamente aperti quando sono creati ma devono essere chiusi manualmente invocando il metodo *close()* altrimenti il GarbageCollector non potrà deallocarne la memoria. E' una best-practice chiudere uno stream in un *finally* per essere sicuri che verrà chiuso sempre anche in caso di eccezioni checked, cosa che capita molto spesso con l'utilizzo degli stream. Forse si può usare anche il try-with-resources.

Scanner è una classe di *java.util* che consente di semplificare la lettura di sorgenti di input, siano esse stringhe, tipi primitivi o altro.

File è la classe che astrae il concetto fisico di file, i metodi più importanti sono:

- boolean exists()
- String getPath()
- String getName()
- String getParent(), restituisce il nome della directory che contiene il file.
- boolean isDirectory()
- boolean delete(), tenta di eliminare il file corrente.
- boolean createNewFile()

Serializzazione degli oggetti

Per *Serializzazione di oggetti* si intende il processo per rendere persistente un oggetto java. Rendere persistente un oggetto significa far sopravvivere l'oggetto oltre la terminazione del programma, salvando l'oggetto (ovvero il suo stato interno) all'interno di un file o di un database.
Per fare ciò si utilizzano solitamente le classi *FileOutputStream* e *ObjectOutputStream* con il metodo writeObject(Object), mentre *FileInputStream* e *ObjectInputStream* con il metodo *readObject()* per deserializzare un oggetto.
try-with-resources.

```java
public class Persona implements java.io.Serializable {
    private String nome;
    private String cognome;
    // questi due attributi non saranno serializzati:
    private transient Thread t = new Thread();
    private transient String codiceSegreto;

    // costruttore, getter e setter, toString...
}
```

```java
import java.io.*;

public class main {
    // non inserisco il throws perchè gestisco l'eccezione localmente...
    public static void Serializza() {
        Persona p = new Persona(...);

        // utilizzo del try-with-resources
        try (
            // prova ad aprire queste risorse...
            FileOutputStream f = new FileOutputStream(new File("persona.ser"));
            ObjectOutputStream s = new ObjectOutputStream(f);
        ) {
            // se ci riesci allora fai...
            s.writeObject(p);
        } catch (IOException e ) {
            // se si solleva una eccezione allora chiuderà in automatico le risorse
            e.printStackTrace();
        }
    }

    public static void De_serializza() throws IOException {
        Persona p = null;

        // utilizzo del try-with-resources
        try (
            // prova ad aprire queste risorse...
            FileInputStream f = new FileInputStream(new File("persona.ser"));
            ObjectInputStream s = new ObjectInputStream(f);
        ) {
            // se ci riesci allora fai...
            p = (Persona)s.readObject();
        } catch (IOException e) {
            // se si solleva una eccezione allora chiuderà in automatico le risorse
            e.printStackTrace();
            // gestisco parzialmente l'eccezione e la rimando alla classe chiamante
            throw e;
        }
    }
}
```

Interfaccia *Serializable*

Questa interfaccia non contiene metodi ma serve solo a distinguere ciò che è serializzabile da ciò che non lo è. Ci sono delle classi che non possono estendere Serializable, come ad esempio la classe Thread o Stream (un processo che esegue codice su dati non può essere serializzato). Provando a serializzare un oggetto non serializzabile, ovvero che non estende l'interfaccia *Serializable*, otterremo un NotSerializableException.

modificatore *transient*

Per ovviare a questo problema esiste il modificatore *transient*, che può essere anteposto solo alle variabili di istanza che non vogliamo serializzare (per esempio se la nostra classe presenta una variabile di istanza Thread) e avvertirà la JVM che la variabile transient non deve essere serializzata, evitandoci l'exception. Le variabili statiche sono implicitamente transient.

Interfaccia *Externalizable*

E' una interfaccia che estende l'interfaccia *Serializable*, e faciliterà il processo di serializzazione e deserializzazione di un oggetto definendo 2 metodi: *writeExternal(ObjectOutput)* per serializzare gli oggettie *readExternal(ObjectInput)*.

JDBC

Appunti

> switch può testare solo una variabile intera (o compatibile) confrontandone l'uguaglianza con costanti (in realtà dalla versione 5 si possono utilizzare come variabili di test anche le enumerazioni e il tipo Integer, e dalla versione 7 anche le stringhe). Il costrutto if permette di svolgere controlli incrociati sfruttando oggetti, espressioni booleane etc.

> il continue non si può utilizzare nello switch ma solo nei cicli.

> una interfaccia non può essere final, in quanto può sempre essere estesa o implementata.

> E' buona norma aggiungere la clausola default ad un costrutto switch per gestire tutti quei case che il programmatore potrebbe non aver previsto. Stesso discorso per la clausola else dell'if.

> se una classe include anche un solo metodo astratto, la classe deve essere dichiarata come *abstract*

> static non si può utilizzare con la dichiarazione di una classe

> ordine istanziazione di una classe:
> 1. Inizializzatore statico
> 2. Metodi statici
> 3. Inizializzatore di defalt
> 4. Costruttore
> 5. Metodi

> il modificatore statico può essere usato solo per le classi interne

> Un metodo protected viene ereditato in ogni sottoclasse qualsiasi sia il suo package e anche dalle classi dello stesso package

> L'incapsulamento è un principio fondamentale della programmazione orientata agli oggetti, che consiste nel nascondere i dettagli interni di una classe e fornire un'interfaccia pubblica per interagire con essa. L'incapsulamento serve a garantire la robustezza, l'indipendenza e la riusabilità del codice, oltre a facilitarne la manutenzione e l'evoluzione. Per applicare l'incapsulamento in Java, si usa il meccanismo dei modificatori di accesso, che permettono di definire il livello di visibilità dei membri di una classe (dati e metodi). In generale, si consiglia di dichiarare i dati privati, in modo da proteggerli da accessi indesiderati o errati, e di fornire dei metodi pubblici per leggerli o modificarli, eventualmente con dei controlli di validità. Questi metodi sono spesso chiamati "getter" e "setter", ma non sono parole chiave del linguaggio.

> le variabili incapsulate di una superclasse in una superclasse possono essere private ed essere utilizzate tramite i metodi accessor e mutator

> solo un altro costruttore della stessa classe può usare quella sintassi la sintassi this() per chiamare un costruttore