PO teoria modulo 1· Java

Introduzione

Incapsulamento

L'incapsulamento è il processo di strutturare il codice in capsule (packages) e classi, al fine di creare un modello coerente e riutilizzabile. Obiettivi principali dell'incapsulamento sono:

- Nascondere l'implementazione, esponendo solo i componenti rilevanti all'utente.
- Ridurre le dipendenze tra componenti.
- Permettere l'utilizzo di codice senza comprenderne i dettagli implementativi.
- Favorire l'alta riutilizzabilità del codice, consentendo modifiche localizzate.

Packages

Ogni classe appartiene a un package specifico. Se una classe vuole utilizzare una classe da un altro package, deve importarla esplicitamente. Il compilatore di Java richiede che tutte le classi importate vengano compilate insieme.

Artifacts

Gli artifacts sono il risultato della compilazione del programma e vengono spesso rappresentati come file *jar*. Gli artifacts incapsulano come uno zip pacchetti, classi e risorse. Contengono:

1. Tutte le classi ottenute dalla compilazione.
2. I file di risorsa.
3. Un file manifesto che descrive l'artifact (versione, titolo, creatore, etc.).

Gli artifacts possono essere condivisi con la community e semplificano la distribuzione del software.

Aggregazione

L'aggregazione rappresenta la relazione "has-a" e si riferisce all'uso di istanze di oggetti in altre classi attraverso metodi.

Polimorfismo

Il polimorfismo consente di utilizzare lo stesso simbolo (metodo o attributo) con differenti comportamenti a seconda del contesto. Questo è realizzato attraverso l'ereditarietà (*is-a relationship*). Il concetto di ereditarietà è implementato tramite l'uso della keyword `extends`. Si crea una gerarchia di classi, dove le sottoclassi ereditano funzionalità dalla superclasse e possono aggiungerne di proprie. Si utilizza `super()` nei costruttori delle sottoclassi per inizializzare la superclasse prima delle operazioni aggiuntive. Java supporta il principio di subtyping, che permette di utilizzare un'istanza di una sottoclasse quando si richiede una istanza della classe base. Java è fortemente tipato, ma offre il dinamismo dei tipi grazie a `instanceof` e il casting.

Method Dispatching

Il method dispatching riguarda la selezione del metodo corretto da chiamare.

- **Static dispatching**: La selezione del metodo avviene a compile time basandosi sul tipo statico dell'oggetto. Non supporta il polimorfismo.
- **Dynamic dispatching**: La selezione del metodo avviene a runtime, cercando il metodo più specifico. Supporta il polimorfismo e il principio di sostituzione.

In Java, si utilizza dynamic dispatching sul receiver (`<receiver>.<method>`) per ottenere il polimorfismo. Per i metodi statici, si applica static dispatching. La scelta del metodo corretto si basa sulla signature del metodo e sul tipo statico del parametro quando si chiama quel metoso.

Classi, Campi e Metodi

Dati e Funzionalità

Una classe in OOP rappresenta un modello da cui possono essere creati oggetti. I dati vengono espressi tramite campi, che possono essere di tipi primitivi (come int, double, char, ecc.) o di tipi definiti da altre classi (tipi statici). Le funzionalità sono espresse tramite metodi.

Creazione di Oggetti

La creazione di oggetti avviene tramite l'uso della keyword `new`. Questa alloca la memoria necessaria per i campi della classe e restituisce un riferimento al nuovo oggetto. Il costruttore della classe è responsabile di inizializzare i campi e allocare la memoria necessaria. L'iter per la creazione di oggetti è il seguente:

1. Allocazione della memoria per l'oggetto.
2. Inizializzazione dei campi a valori di default (zero o null).
3. Esecuzione del costruttore.
4. Restituzione del riferimento al nuovo oggetto.

Keyword `this` e `super`

La keyword `this` viene utilizzata per accedere ai campi o metodi della classe corrente. Può essere utilizzata per passare il riferimento dell'istanza attuale o per accedere al costruttore (solo all'interno di altri costruttori e come primo statement). La keyword `super` è utilizzata per accedere ai campi o metodi della superclasse.

Istanziamento e Campi

Una classe può essere istanziata più volte, creando diversi oggetti. Ogni oggetto rappresenta un'istanza della classe. I campi dichiarati all'interno di una classe (non all'interno dei suoi metodi) sono memorizzati nello heap ogni volta che viene creata una nuova istanza.

Variabili Locali e Parametri

Le variabili locali istanziate in un metodo e i parametri passati al metodo sono memorizzati nello stack del metodo. Vengono rimossi dopo l'esecuzione del metodo. È una buona pratica evitare di sovrascrivere un parametro, in quanto può causare confusione.

Tipi di Dati

I tipi di dati possono essere ValueType (tipi di valore) o ReferenceType (tipi di riferimento). I ValueType rappresentano dati di base come numeri e sono allocati in memoria con un numero fisso di byte. I ReferenceType sono riferimenti a oggetti, consentendo di cambiare lo stato dell'oggetto anche dopo il passaggio del riferimento come parametro (attenzione agli errori).

Controllo della Memoria e Garbage Collector

In Java, l'ambiente runtime traccia cosa è raggiungibile dal programma e cosa no. Il garbage collector è responsabile di deallocare la memoria automaticamente, liberando gli oggetti che non sono più raggiungibili. Questo processo avviene in modo automatico, ma può comportare variazioni inattese nel tempo di esecuzione del programma.

Evitare Modifiche Indesiderate

Per evitare modifiche indesiderate agli oggetti passati a un metodo, è possibile:

1. Passare una copia dell'oggetto invece dell'oggetto stesso.
2. Utilizzare campi `final` nella classe per rendere immutabili i campi.
3. Sfruttare il meccanismo di tracciamento automatico dell'ambiente Java per deallocare la memoria in modo efficiente.

Modifiers

Static

- **Metodi e Campi static**: Utilizzando la keyword `static`, è possibile condividere memoria tra tutte le istanze di una classe. I metodi statici possono accedere solo a campi statici e invocare metodi statici, poiché non fanno riferimento a un'istanza specifica.
- **Costruttore statico**: Un costruttore statico non può ricevere parametri ed è utilizzato per inizializzare campi statici o invocare metodi statici.
- **Accesso a metodi e campi statici**:
  1. Utilizzando la variabile che punta all'oggetto istanziato, ad esempio `ferrari.resetCounter()`.
  2. All'interno della classe, utilizzando direttamente il nome del metodo o della classe.
  3. Utilizzando il nome della classe seguito dal punto e il nome del componente statico, ad esempio `Car.resetCounter()`.
- **Classi statiche**: Le classi statiche non esistono.

Final

- **Campi final**: I campi final possono essere assegnati solo una volta, sia nel costruttore che nella definizione. Dopo l'assegnazione, diventano accessibili solo in lettura e non possono essere modificati.
- **Metodi final**: I metodi final impediscono l'override, quindi non possono essere sovrascritti nelle sottoclassi.
- **Classi final**: Le classi final impediscono l'ereditarietà, quindi non possono essere estese da altre classi. Vengono utilizzate quando si desidera fornire un'implementazione finale e non personalizzabile di un oggetto.

Abstract

- **Classi astratte**: Le classi astratte possono contenere metodi con dichiarazioni astratte o concrete. Non possono essere istanziate direttamente ma possono avere un costruttore per inizializzare lo stato. Le classi che estendono una classe astratta con metodi astratti devono implementare o sovrascrivere quei metodi.
- **Metodi astratti**: I metodi astratti sono dichiarazioni senza implementazione e appartengono a classi astratte. Devono essere implementati nelle sottoclassi.
- **Campi astratti**: I campi astratti non esistono.

Access Modifier

Gli access modifier determinano la visibilità di componenti (campi, metodi, classi) all'interno della classe e nel contesto esterno.

- **Public**: I componenti pubblici rappresentano l'interfaccia per l'utente finale della classe.
- **Private**: I componenti privati sono accessibili solo all'interno della classe stessa.
- **Protected**: I componenti protetti sono accessibili all'interno della classe e dalle sottoclassi.
- **Default (Package-private)**: I componenti senza un modificatore sono accessibili solo all'interno del package.

	Same class	Same package	Subclasses	Everywhere
public	👍	👍	👍	👍
protected	👍	👍	👍	👎
<default>	👍	👍	👎	👎
private	👍	👎	👎	👎

(P.S.: Se si utilizzano pubblic allora valori arbitrari possono essere assegnati e non possiamo cambiare in futuro la rappresentazione di quel componente (si pensi all'ereditarietà). Sarebbe meglio renderlo privato e in caso fornire dei metodi get / set in base alle esigenze)

Scelta dei Modificatori

La scelta dei modificatori dovrebbe seguire un iter:

1. Scegliere componenti pubblici per l'utente finale.
2. Rendere tutto il resto privato.
3. Analizzare e, se necessario, allentare la visibilità.

Le classi non possono essere `protected` o `private`, ma i metodi e i campi possono avere tutti i modificatori.

Documentazione

La documentazione è un elemento essenziale nell'artefatto software poiché descrive il comportamento delle funzioni e ne facilita la comprensione senza dover analizzare l'implementazione del codice. In Java, la documentazione può essere realizzata seguendo uno specifico formato.

- **Commenti interni**: Utilizzati per spiegare il comportamento di una specifica linea o porzione di codice. Questi commenti non sono inclusi nella documentazione destinata all'utente finale e si dividono in:
  - `//`: Commenti su singola linea.
  - `/* ... */`: Commenti su più righe.

- **Commenti JavaDoc Standard (`/** ... */`)**: Questi commenti multilinea sono destinati agli utenti finali che utilizzeranno il software. Dovrebbero precedere i componenti (campi, classi e metodi) che intendono documentare. I commenti JavaDoc consentono di comprendere le funzionalità di un componente in modo astratto, evitando dettagli implementativi. Questi commenti, seguendo lo standard JavaDoc, faranno parte della documentazione ufficiale.
- **Tag JavaDoc**:
  - Per le classi:
    - `@author`: Specifica l'autore della classe.
    - `@since`: Indica dalla quale versione del software la classe è presente.
    - `@version`: Specifica la versione della classe (da evitare per evitare conflitti con il manifesto dell'artefatto).
  - Per i metodi:
    - `@param`: Documenta un parametro del metodo. È possibile utilizzarne più di uno.
    - `@requires` $a \geq 0$: Utilizzato nella programmazione a contratto, rappresenta una precondizione per il metodo.
    - `@return`: Documenta ciò che il metodo restituirà.
    - `@ensures` $return \geq 0$: Utilizzato nella programmazione a contratto, rappresenta una postcondizione per il metodo.
    - `@throws`: Documenta le eccezioni che il metodo potrebbe lanciare.
  - Per i campi/oggetti:
    - `@invariant` $speed \geq 0$: Utilizzato nella programmazione a contratto, specifica una condizione che deve essere sempre verificata per l'oggetto durante l'esecuzione del programma.

Design by Contract

Il concetto di Design by Contract afferma che una capsula, l'ereditarietà, un'interfaccia e la documentazione possono essere considerati come un contratto. In altre parole, i developer hanno il diritto di ricevere valori conformi a criteri specifici e, allo stesso tempo, hanno l'obbligo di restituire valori secondo la firma del metodo e la documentazione fornita. Questo pattern, sebbene potente, non è spesso utilizzato a causa del tempo e dello sforzo necessari per implementarlo correttamente.

Override e Overload

Firma e Definizione dei Metodi: La firma di un metodo comprende tutte le informazioni necessarie per chiamare il metodo, inclusi il nome, il numero e i tipi dei parametri e il tipo statico del receiver. La definizione di un metodo include anche il tipo di ritorno e la visibilità.
Overload: Il concetto di overload si basa sulla definizione di più metodi con lo stesso nome all'interno di una classe. Questi metodi devono avere firme diverse, ovvero devono differire per numero o tipo di parametri. La scelta di quale metodo eseguire avviene solo a runtime, in base ai parametri forniti durante la chiamata. L'overload è un modo per fornire diverse versioni di un metodo, ciascuna adattata a un insieme specifico di parametri.
Override: L'override si verifica quando una classe ridefinisce la funzionalità di un metodo ereditato da una superclasse. La firma del metodo ridefinito deve coincidere con la firma del metodo nella superclasse. Tuttavia, la visibilità del metodo può essere resa più rilassata (ad esempio, da `protected` a `public`), ma non il contrario. L'override consente a una sottoclasse di personalizzare o estendere il comportamento del metodo ereditato senza dover modificare la superclasse stessa.

Interfacce

Le interfacce in Java sono un modo per definire tipi che possono essere utilizzati nel programma. Una caratteristica chiave delle interfacce è che definiscono una lista di firme di metodi senza fornire un'implementazione. È possibile aggiungere una `default implementation` per un metodo in un'interfaccia, ma ogni classe che implementa l'interfaccia dovrà comunque fornire la propria implementazione per quei metodi. Alcuni punti importanti sulle interfacce includono:

- Le interfacce consentono l'ereditarietà multipla in Java, il che significa che una classe può implementare più interfacce, ma estendere al massimo una sola classe.
- Un'interfaccia può estendere più interfacce, creando una gerarchia complessa di interfacce. Tuttavia, se due interfacce estese hanno metodi con la stessa firma ma diverse implementazioni predefinite, si verificherà un errore di compilazione.
- Tutti i membri di un'interfaccia sono implicitamente `public`, quindi non è possibile definire un membro come `private`.
- Una classe che implementa un'interfaccia ma non fornisce implementazioni per tutti i suoi metodi deve essere dichiarata come `abstract`.
- Non si può assegnare un valore ad un campo anche se si può definere (bad practice si consiglia uso di get e set).

Generics

I generics in Java consentono ai programmi di lavorare con tipi diversi passati come argomenti, fornendo una maggiore flessibilità e riusabilità del codice. Alcuni concetti importanti relativi ai generics includono:

- I generics possono essere applicati alle classi e ai metodi. Quando si istanzia una classe che utilizza i generics, è necessario specificare il tipo tra le parentesi angolari, ad esempio `List<Oggetto> lista = new List<Oggetto>();`.
- I generics possono essere usati anche per parametrizzare singoli metodi anziché intere classi. Questi sono spesso utilizzati con metodi statici che non sono legati a un'istanza specifica della classe, ad esempio `public static <T> Integer metodo (T el)...` .
- **Generics inference**: Il compilatore Java è in grado di dedurre il tipo dei generics durante la creazione di un'istanza, quindi in molte situazioni non è necessario specificare il tipo esplicitamente, ad esempio `List<Oggetto> lista = new List<>();`.
- **Generic invariance**: I generici sono invarianti, il che significa che non è possibile assegnare una lista di un tipo generico a una lista di un tipo diverso, nemmeno se il secondo tipo è una sottoclasse del primo. Gli array, al contrario, sono covarianti.
- **Binding Generics**: È possibile limitare il tipo di generici utilizzati con vincoli come `<T extends OggettoX>` o `<? super OggettoX>`.
- **Wildcards**: Il simbolo `?` rappresenta un tipo sconosciuto. È utilizzato per garantire che il codice sia compilabile anche quando non si ha certezza sul tipo.

Object

La classe `Object` è la superclasse di tutte le altre classi in Java. Ogni oggetto creato in Java estende implicitamente la classe `Object` e ne eredita i suoi metodi principali. Alcuni punti importanti relativi a `Object` includono:

- `equals`: Questo metodo pubblico riceve un oggetto come parametro e restituisce `true` o `false` se l'oggetto è uguale all'oggetto corrente. È importante notare che `equals` è diverso dall'operatore `==`, che confronta solo le reference. Se si ridefinisce `equals`, è necessario ridefinire anche il metodo `hashCode` per rispettare le regole di coerenza. La ridefinizione di `equals` dovrebbe rispettare le proprietà di riflessività, simmetria e transitività.
- `clone`: Questo metodo protetto ritorna un nuovo oggetto identico all'oggetto corrente. È possibile scegliere di rilassarne la visibilità a `public`. Esistono due approcci principali per il cloning: *shallow cloning*, che crea nuove reference per gli oggetti interni, e *deep cloning*, che crea copie profonde di tutti gli oggetti interni.
- `hashCode`: Questo metodo pubblico non riceve parametri e restituisce un intero. Viene utilizzato per fornire un valore di hash per l'oggetto, spesso utilizzato nelle strutture dati basate su hashtable. È importante che se due oggetti sono uguali, il metodo `hashCode` dovrebbe restituire lo stesso valore, sebbene non valga l'opposto.
- `toString`: Questo metodo pubblico ritorna una rappresentazione in forma di stringa dell'oggetto. È spesso utilizzato per scopi di debug e di solito ha il formato `nome_classe@hash_oggetto`.

Collection

Tutte le collezioni in Java implementano l'interfaccia `Iterable`, il che significa che possono essere utilizzate in un ciclo `for-each`. Alcuni punti rilevanti sulle collezioni includono:

- Il metodo `contains()` si basa sull'implementazione del metodo `equals`. Questo metodo determina se un elemento è presente nella collezione.
- Le strutture dati basate su hashtable, come `HashMap`, si basano sull'implementazione del metodo `hashCode`. È importante che due oggetti uguali abbiano lo stesso valore di hash.
- Il metodo `sort()` delle collezioni si basa sull'implementazione dell'interfaccia `Comparable`. Questo metodo ordina gli elementi in base all'ordine naturale definito dalla classe.

String

La classe `String` in Java è utilizzata per rappresentare, codificare e decodificare sequenze di caratteri. Le stringhe in Java sono immutabili, il che significa che una volta create, il loro stato interno non può essere modificato. Tuttavia, esiste anche la classe `StringBuffer`, che supporta la modifica del suo contenuto. È importante notare che l'operatore di concatenazione (`+=`) per le stringhe crea una nuova stringa ad ogni passaggio del processo di concatenazione, il che può comportare un consumo significativo di memoria.
Nel processo di concatenazione con `+=`, i passaggi sono i seguenti:

1. Viene creato un `StringBuffer`.

2. Vengono aggiunti al `StringBuffer` i caratteri della prima stringa.
3. Vengono aggiunti al `StringBuffer` i caratteri della seconda stringa.
4. Viene restituita la stringa rappresentata dal `StringBuffer`.

In generale, è importante fare attenzione alla gestione delle stringhe, poiché un uso inefficiente può portare a un consumo eccessivo di memoria.

Primitive

I tipi primitivi in Java sono quei tipi che non sono sottotipi di `Object`. Questi includono `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` e `double`. I tipi primitivi rappresentano valori numerici e non sono oggetti di riferimento. Ogni tipo primitivo ha un numero fisso di bit che può rappresentare.
È importante notare che non è possibile assegnare direttamente un valore di un tipo primitivo a un altro tipo primitivo con una dimensione diversa. È necessario utilizzare un'operazione di *casting* e l'operatore `()` per effettuare la conversione.
Per ogni tipo primitivo esiste un *wrapper class*, che è un oggetto che estende `Object` e rappresenta il tipo primitivo corrispondente. Questi wrapper sono necessari quando si ha bisogno di trattare valori primitivi come oggetti. Tuttavia, il compilatore supporta l'auto-boxing e l'unboxing, ovvero la conversione automatica tra tipi primitivi e wrapper, se necessario. È importante notare che queste operazioni hanno un costo in termini di prestazioni e memoria, quindi è consigliabile utilizzarle con parsimonia.

Eccezioni

Le eccezioni in Java consentono di gestire situazioni anomale o errori durante l'esecuzione di un programma. Le eccezioni sono oggetti che raccolgono e descrivono informazioni utili sul contesto dell'errore. In Java, tutte le eccezioni sono sottoclassi dell'interfaccia `Throwable`.
Le eccezioni sono utili per distinguere tra vari tipi di errori e problemi durante l'esecuzione del programma:

- **Checked Exceptions**: Queste eccezioni devono essere esplicitamente gestite nel codice tramite try-catch o dichiarate nel metodo con la keyword `throws`. Sono sottoclassi della classe `Exception`. Esempi includono errori di input/output o di connessione di rete. Possono essere documentate per aiutare gli sviluppatori a comprendere le situazioni in cui possono verificarsi.
- **Unchecked Exceptions**: Queste eccezioni sono sottoclassi di `RuntimeException` e `Error`. Le `RuntimeException` sono sollevate durante esecuzioni anormali e non rappresentano errori logici, ma piuttosto inconsistenze nell'esecuzione. Gli `Error`, al contrario, indicano problemi gravi a livello di runtime e portano spesso alla terminazione del programma. Le unchecked exceptions non richiedono una gestione esplicita.

Java permette di creare eccezioni personalizzate estendendo classi esistenti o implementando interfacce. La creazione di un'eccezione personalizzata richiede l'override dei costruttori e la gestione del messaggio di errore attraverso la superclasse. È inoltre possibile dichiarare che un metodo lancia una specifica eccezione utilizzando la keyword `throws`, e all'interno del metodo lanciare l'eccezione con `throw new NomeClasseEccezione()`.
Per gestire le eccezioni durante l'esecuzione, è necessario racchiudere il codice potenzialmente problematico all'interno di un blocco `try-catch`. Il blocco `finally` viene eseguito sempre, indipendentemente dall'occorrenza di un'eccezione. Questa struttura permette di catturare e gestire le eccezioni senza interrompere completamente il programma.
La sequenza di esecuzione di un blocco `try-catch-finally` è la seguente:

1. Il blocco `try` viene eseguito.
2. Se non si verifica alcuna eccezione, viene eseguito il blocco `finally`.
3. Se si verifica un'eccezione non catturata, viene eseguito il blocco `finally`, seguito dal sollevamento dell'eccezione.
4. Se si verifica un'eccezione catturata, viene eseguito il blocco `catch`, seguito dal blocco `finally`.

Le eccezioni possono essere annidate, catturando un'eccezione all'interno di un blocco `catch` e lanciando una nuova eccezione.
Un altro modo per gestire situazioni in cui si desidera terminare immediatamente il programma in base a condizioni specifiche è l'uso delle *assertions*. Gli statement di assertion sono utilizzati per testare condizioni e lanciare un errore di asserzione se la condizione non è soddisfatta. Gli statement di asserzione vengono abilitati con l'opzione `-ea` (enable assertions) e sono particolarmente utili per il testing e il debugging.

Annotations

Le annotazioni (annotations) in Java sono simili alle classi, ma sono utilizzate per fornire metadati al codice anziché definire comportamenti. Le annotazioni possono avere attributi che rappresentano informazioni aggiuntive sul componente annotato. Le annotazioni possono essere utilizzate dalla JVM durante la compilazione e l'esecuzione del programma. Devono essere applicate direttamente sopra il componente che si desidera annotare.
Le annotazioni predefinite principali in Java includono:

- `@Override`: Indica che un metodo sta cercando di eseguire l'override di un metodo nella superclasse. Se il metodo non è suscettibile di override, viene generato un errore.
- `@Deprecated`: Indica che un metodo o una classe sono obsoleti e dovrebbero essere evitati. Il codice continuerà a compilare, ma verrà emesso un avviso dal compilatore.
- `@SuppressWarnings("stringa")`: Sopprime specifici warning segnalati dal compilatore. È possibile specificare la stringa di avviso da sopprimere, ad esempio `"all"`, `"unused"`, ecc.
- `@Test`: Utilizzato dal framework di testing JUnit per identificare i metodi di test. I metodi annotati con `@Test` vengono eseguiti e testati da JUnit.

È possibile definire anche annotazioni personalizzate. Ad esempio:

```
public @interface Speed {
    String type() default "kmh";
    boolean forward();
}
```

Esempio di utilizzo:

```
public abstract class Vehicle {
    @Speed(forward = true) private double speed;
    public void accelerate(@Speed(forward = true) double a) {...}
    @Speed(forward = true, type = "kph") public double getSpeed() {...}
}
```

È possibile specificare su quali componenti le annotazioni possono essere applicate utilizzando l'array di `ElementType` nell'annotazione `@Target`. Ad esempio:

```
@Target({
    ElementType.METHOD,
    ElementType.FIELD,
    ElementType.PARAMETER
})
public @interface Speed {...}
```

Inoltre, è possibile limitare la visibilità delle annotazioni nelle diverse fasi di esecuzione del programma (ad esempio, `SOURCE`, `CLASS`, `RUNTIME`) utilizzando l'annotazione `@Retention`. Esempio:

```
@Retention(RetentionPolicy.SOURCE)
public @interface Speed {...}
```

Le annotazioni possono essere utilizzate anche per altri scopi. Ad esempio, la libreria JAXB utilizza le annotazioni per mappare classi in XML. Le annotazioni come `@XmlType`, `@XmlRootElement`, ecc., consentono di definire come le classi devono essere convertite in XML e viceversa. Questo semplifica il processo di conversione tra rappresentazione XML e oggetti Java.

Reflections

La reflection è la capacità di un programma di accedere alle informazioni relative a una classe e ai suoi componenti, come campi, metodi e costruttori. Questo consente ad altri servizi, come JavaBeans, di accedere ai membri di una classe specifica. Il principale punto di ingresso per la reflection è la classe `java.lang.Class`, che rappresenta qualsiasi tipo di oggetto. Ci sono due modi per ottenere un'istanza di `Class`:

1. `Object.getClass()` (valido solo per oggetti istanziabili)
2. `<nome_tipo>.class` (valido per qualsiasi tipo)

La classe `Class` fornisce diversi metodi per accedere alle informazioni relative a quella classe:

- Informazioni sul tipo: `isPrimitive`, `isInterface`, `isAnnotation`, `getModifiers`, ecc.
- Informazioni sulla gerarchia: `getSuperclass`, `getPackage`, ecc.
- Informazioni sulle funzionalità: `getDeclaredFields` (restituisce tutti i campi, inclusi quelli privati), `getDeclaredMethods`, ecc.

L'utilizzo della reflection rappresenta un approccio dinamico, in cui i vincoli vengono verificati a runtime. Ciò significa che non vengono generati errori di compilazione, ma possono verificarsi diverse eccezioni a runtime. Inoltre, la reflection può violare l'incapsulamento e l'information hiding. Ad esempio, le classi `Field`, `Method` o `Constructor` forniscono un metodo `setAccessible(true)` che consente di accedere e modificare arbitrariamente lo stato degli oggetti, ignorando la visibilità dichiarata. È possibile invocare metodi utilizzando la reflection. Per farlo, è necessario:

1. Specificare la firma del metodo e il numero e i tipi statici dei parametri.
2. Invoare il metodo tramite `invoke(Object obj, Object... args)`.

Anche i costruttori possono essere invocati tramite reflection, ma solo dopo che la classe è stata inizializzata. Possono essere identificati in base ai loro parametri.
Tramite reflection, è possibile anche accedere alle annotazioni se presenti. Tuttavia, se un'annotazione non è disponibile a runtime a causa di una retention diversa, non verrà recuperata tramite reflection.
Vantaggi dell'uso della reflection includono l'accesso a campi e metodi privati e la trasformazione di oggetti in XML. Tuttavia, ci sono anche svantaggi, come la violazione dell'incapsulamento e l'information hiding, oltre all'introduzione di errori e eccezioni che vengono rilevati solo a runtime.
Ecco alcuni esempi di utilizzo della reflection:

```
public class MyClass {
    private int myField;

    public MyClass(int value) {
        myField = value;
    }

    private void myPrivateMethod() {
        System.out.println("Private␣method␣called");
    }
}

public class ReflectionExample {
    public static void main(String[] args) throws Exception {
        Class<?> clazz = MyClass.class;
        Field field = clazz.getDeclaredField("myField");
        field.setAccessible(true);
        int value = (int) field.get(new MyClass(42));
        System.out.println("Field␣value:␣" + value);

        Method method = clazz.getDeclaredMethod("myPrivateMethod");
        method.setAccessible(true);
        method.invoke(new MyClass(42));
    }
}
```

Library Management

Java offre i mezzi per aggiungere e utilizzare librerie a runtime nelle nostre applicazioni. È importante stabilire la giusta classpath per consentire alla JVM di individuare tutte le classi necessarie. La classpath principale è composta da più classpath, ognuna delle quali può essere un file JAR (artifact) o una directory contenente classi. Se una classe non viene trovata, viene lanciata un'eccezione apposita.
Una libreria può avere sia un'interfaccia interna che una esterna. Quella interna rappresenta campi privati o pacchetti con visibilità di default, mentre quella pubblica rappresenta le funzionalità accessibili agli utenti finali.
Poiché le librerie possono evolversi nel tempo per apportare miglioramenti, il processo di una nuova release consiste nella produzione di un nuovo artifact con una specifica versione della libreria che deve essere retrocompatibile. Una nuova versione può sostituire quella precedente se offre tutte le funzionalità precedenti e ne aggiunge di nuove. In questi contesti, l'annotazione `@Deprecated` è spesso utilizzata per segnalare che alcuni metodi non sono più raccomandati o sono stati sostituiti.
È anche importante utilizzare numeri di versione (esempio: 3.0.1) per indicare le variazioni apportate nella nuova release:

- **Punti o correzioni di bug**, in cui l'interfaccia esterna rimane invariata (1)
- **Cambiamenti minori**, in cui i cambiamenti sono compatibili con le versioni precedenti e le funzionalità esistenti sono mantenute (0)
- **Cambiamenti maggiori**, in cui l'interfaccia esterna subisce modifiche e la retrocompatibilità potrebbe non essere garantita (3)

Per automatizzare il processo di costruzione delle applicazioni Java, esistono diversi strumenti come Gradle. In generale, questo processo si compone di due fasi:

1. Gestione delle librerie (ad esempio, Apache Ivy)
2. Compilazione ed esecuzione del programma

Gradle, che gestisce entrambe le fasi, si compone di tre componenti principali: plugins (tipi di programmi che vogliamo compilare), repositories (repository di artefatti come Maven Central) e dependencies (versione, gruppo, configurazioni, ecc.). Una volta specificate queste informazioni, il file `build.gradle` è in grado di utilizzarle per la compilazione, l'esecuzione o i test.

Raccolta domande

Commenti e documentazione

- commenti interni spariscono dopo la compilazione
- commenti di documentazione fanno parte dell'interfaccia esterna della libreria

- commenti interni con // e /*
- commenti esterni con /**
- javadoc costruisce la documentazione partendo dai commenti di documentazione
- i commenti Javadoc sono strutturati e possono contenere diversi tag

Abstract

- su metodo e classe
- classe abstract non puo' essere istanziata
- metodo abstract non contiene il codice
- metodo abstract deve essere all'interno di una classe abstract
- non e' necessario che una classe abstract contenga metodi abstract

Annotazioni

- meccanismo per aggiungere informazioni alle componenti di una classe accessibili a runtime tramite reflection
- Una annotazione puo' essere definita con @interface seguito dal nome dell'annotazione
- come una classe con i propri campi/attributi, La definizione puo' contenere campi/attributi definiti con $< tipo >< nome\_campo >$ ()
- @Target restringe le componenti del programma ad oggetti a cui applicare una annotazione
- @Retention definisce a che livello l'annotazione e' visibile (codice sorgente, bytecode, esecuzione)

Reflection

- le reflection permette di vedere la struttura del programma (classi, metodi, campi, ...)
- class Class, Method, Field
- getField/Method/Constructor
- setField per assegnare un campo, invoke method
- vedere come il codice e' annotato

Invocare metodo tramite reflection

- tramite un oggetto Class possiamo accedere a tutti metodi della classe
- tramite un oggetto Class possiamo chiedere un metodo particolare specificandone la firma
- tramite l'invoke dobbiamo passare tutti i parametri del metodo invocato
- tramite il metodo invoke possiamo invocare un metodo
- se non abbiamo accesso al metodo o il metodo e' astratto/non esiste, otteniamo delle eccezioni ad hoc

Getter e Setter

- getter ritorna il valore di un campo
- setter modifica il valore di un campo ricevendo tale valore come parametro
- potrebbero non lavorare su un campo ma sullo stato dell'oggetto
- il setter puo' controllare che il valore rispetti determinate condizioni
- come svantaggio richiedono l'invocazione di metodi, mentre la lettura e scrittura diretta è computazionalmente meno costosa

Tipaggio forte e statico, sottotipaggio

- statico significa che il tipo di ciascuna espressione/variabile è noto a tempo di compilazione (tramite tipi dichiarati e regole di inference)
- strongly significa che un'espressione puo' essere assegnata a un'altra solo se e' "compatibile" (stesso tipo o sottotipo)
- la relazione di sottotipaggio significa che un sottotipo puo' "sostituire" un altro
- una classe che estende un'altra ne e' sottotipo
- una classe che implementa un'interfaccia ne e' sottotipo

Principio di sostituzione e correlazione con ereditarietà

- sun oggetto di una classe puo' sostiture un oggetto di un'altra classe (in linea generale)
- l'interfaccia dell'oggetto sostituito deve essere piu' piccola dell'interfaccia dell'oggetto che lo sostituisce
- una classe che estende un'altra classe ha un'interfaccia piu' ampia
- una una classe che estende un'altra classe e' suo sottotipo
- una classe che estende un'altra classe la puo' quindi sostituire

Boxing and Unboxing

- una classe wrapper/mirror per ogni tipo primitivo
- una classe Number supertipo di tutti i numeri
- con la conversione implicita si puo' assegnare il valore di un tipo primitivo a una variabile di un altro tipo
- non e' pero' possibile assegnare un tipo con "maggiore precisione" ad uno con minore (tipo un double a un float) a meno di non fare un casting/conversione esplicita
- con le classi wrapper/mirror non c'e' conversione implicita da una all'altra

Override di metodi e final

- override quando un metodo di una sottoclasse ridefinisce un metodo ereditato
- l'override si ottiene implementando un metodo con la stessa firma di un metodo erreditato
- in questo contesto, final puo' essere applicato a classi e metodi (sui campi non ha a che vedere con l'ereditarieta')
- final su una classe semplicemente impedisce che ci sia una sottoclasse
- final su un metodo impedisce l'override di un metodo

Keyword final

- a campi, metodi e classi
- una classe final non puo' essere estesa
- non si puo' fare l'override di un metodo final
- non si puo' assegnare piu' di una volta un campo final
- un campo final può essere inizializzato solo nel costruttore (o nella sua definizione)
- un campo final non puo' essere assegnato nei metodi

Implementare metodi dentro le interfacce

- si' si puo' implementare un metodo
- e' necessario definirlo di default
- si' si possono definire campi
- i campi sono static e final
- l'implementazione dei metodi non ha restrizioni

Differenza tra int e Integer

- uno e' un tipo valore e l'altra e' un tipo reference (numero/oggetto)
- Integer eredita i metodi di Object
- Integer e' un wrapper di int
- è possibile assegnare un int a integer e vice
- autoboxing e autounboxing

Modificatori di accesso

- Elenco completo di tutti e 4 gli access modifiers (public, private, protected, e default/package) e nessun altro modifier di altro tipo
- Descrizione corretta modificatore public (accessibile da ovunque)
- Descrizione corretta modificatore private (accessibile solo all'interno della classe)
- Descrizione corretta modificatore protected (accessibile dal package e dalle sottoclassi)
- Descrizione corretta modificatore default/package (accessibile dal package)

Definizione e firma di un metodo e correlazione con override

- La firma e' composta da nome metodo, numero di parametri e tipo (eventualmente classe che contiene il metodo)
- La dichiarazione comprende anche altro i (return type, visibilita', lista delle eccezioni lanciate, ...)
- L'override ridefinisce il comportamento di un metodo ereditato
- L'override significa definire un metodo con la stessa firma di un metodo ereditato
- Il resto della dichiarazione di un metodo di cui si fa override puo' variare ma deve essere meno "restrittivo"

Versionamento e compatibilità (backward compatibiliy)

- un sistema nuovo e' compatibile con uno vecchio/legacy
- sulle librerie significa che forniscono una interfaccia piu' ampia della precedente
- ciascuna versione e' identificato da un numero univoco
- un numero di versione e' composto da piu' numeri distinti
- la major cambia l'interfaccia e non e' backward compatible

Tipi valore e tipi referenza

- tipi valore sono int double long float ... (almeno 4)
- i tipi valore vengono passati per valore
- tipi referenza sono tutti quelli definiti da class o interface
- i tipi referenza sono tutti sottotipo di Object
- i tipi referenza vengono passati per referenza/indirizzo e creano quindi problematiche di aliasing

Eccezioni

- quando lanciata, un'eccezione termina il flusso sequenziale di esecuzione ed entra in una sorta di stato di errore
- classe che estende Exception/Throwable
- eccezioni lanciate con throw
- try... catch(E e) ... cattura eccezioni di tipo E o sottotipo lanciate dal blocco nel try
- finally viene sempre eseguito, se eccezione poi si riprende in stato "eccezionale"