Programmazione a oggetti Java

**Reflection**

Recuperare classe:

*Class<?> myclass =Class.forName(className);*

Creare un oggetto

*Object obj=myclass.newInstance();*

Abilitare i metodi alla modifica se non è pubblico

*metodo.setAccessible(true);*

*getModifiers()* ->restituisce il modificate del metodo 1=publico 2=privato ….

*getFields()* ->ritorna solo i campi pubblici guardando gerarchia

*getDecalredFields()* ->ritorna tutti i campi senza guardare gerarchia

*getField(nome), getDeclaredField(nome*) si può prendere un campo specifico, le operazioni sono analoghe per i metodi

Tipi importanti:

* Fields
* Method
* Parameter
* Constructor

Per invocare un metodo usa *metodo.invoke(oggetto, parametri)*

Per invocare costruttori *Costruttore.newInstance(…parametri…)*

*Classe.newInstance()* solo se esiste il costruttore di default

*Myclass.getSuperClass()* ritorna le classe da cui si eredita, con *getDeclaringClass()* si ottengono le classi nested, ritorna null per le classi anonime, mentre con *getEnclosingClass()* si ottengono le classi interne anonime locali

È possibile modificare e creare dinamicamente array uasndo il package *java.lang.reflect.Array*

**Annotation**

@Annotation

Esempio

*@Target(ElementType.METHOD)*

*@Retention(RetentionPolicy.RUNTIME)*

*@Repeatable(ToDoContainer.class)*

*public @interface Todo {*

*String dueDate();*

*String task() default "";*

*}*

*@Target(ElementType.METHOD)*

*@Retention(RetentionPolicy.RUNTIME)*

*@interface ToDoContainer { Todo[] value(); }*

Nella classe si usano i metdo di reflection per le annotation

Esempio per i metodi annotati

*method.isAnnotationPresent(ToDoContainer.class)*

Il container si usa quando ci sono più annotation su un metodo

Il target del container deve essere lo stesso della classe da contenere, mentre la retention deve essere almeno come quella della classe contenuta

Dichiarare una annotation:

*@Todo(dueDate = "01/02/2021", task = "add jsdoc comments")*

**Generics**

Raw type: tipo generico senza specificazione, quindi Object, da evitare

Instaceof non funziona, e non funziona nemmeno il cast classico usare classe.class.isInstace(obj)==OggettoClasse per verificare la compatibilità e classeDestinazione.cast(oggetto) per effettuare il cast.

Wildcard

*G<? extends tipo>* accetta i tipi che estendono la classe tipo(utile per gli input)

*G<? super tipo>* accetta i tipi che vanno da Object fino a tipo non oltre (utile per gli output)

Generics e Collections: extends blocca le operazioni di aggiunta, ma consente le letture. Super blocca le letture, ma consente le aggiunte

**Classi nested**

Una classe innestata può essere statica oppure interna, quest’ultima deve essere instanziato un oggetto per essere usata **e non può contenere elementi statici.**

Ci possono essere anche interfacce nested

**Lambda**

Per poter implementare una lambda bisogna avere un interfaccia con annotation *@FunctionalInterface* e deve contenere un solo metodo.

Immagine che contiene tavolo

Descrizione generata automaticamenteClosures sono funzioni che combinano due cose e ritornano una funzione implementata e l’ambiente dove la funzione deve essere eseguita

Side effects: modifica di variabili locali o parametri che possono portare ad errori, nelle lambda bisognerebbe evitare.

**Streams**

Utili per evitare cicli espliciti o operazioni su collection particolari.

Creare una stream:

*Stream<G> stream1=Stream.of(elements)*

Stream infinite

*Stream<G> stream2 =Stream.generate(Supplier()) il supplier deve essere un metodo che crea un elemento*

*Stream2.Iterate(indice primo elemento, funzione di avanzamento)*

Unendo più stream si creano delle pipeline.

Leazy: esegue alla fine Eager esegue appena arriva l’input.

Collect(Collector.toList()) restituisce una lista

*Collect(Collectors.joining(“, ”))* deve essere fatto su una lista di stringhe, nel caso fare un .*map(Object::toString)* a e trasformarlo in stringa prima del collect. Tramite *Collectors.summarizingDouble(caratteristica)* è possibile avere delle statistiche di una determinata caratteristica nella stream, *collectors.groupby(lambda)* raggruppa in base alla lambda passata per parametro.

*Filter* consente di filtrare gli elementi, *dinstinct* elimina i duplicati, *findFirst* ritorna un Optional contente il primo elemento della stream

*FlatMap* unisce due stream in un'unica, *Reduce* è simile al *collect* ma si passa una lambda BinaryOperation, *skip* salta n elementi *limit* tonca la lista a una x lunghezza, *count* restituisce il numero di elementi, *max,* *min* trovano massimo e minimo, *peek* processa i dati come *ForEach* ma è egear, *AnyMatch*, *AllMatch*, *NoneMatch* restituiscono se ci sono elementi che soddisfano una determinata condizione, *sorted* ordina