**INTERFACCE**

La interfacce vengono definite public, o se non viene specificata la visibilità saranno package access. In ogni caso i metodi devono essere sempre public, altrimenti classi di altri package che vogliono implementare l’interfaccia non potrebbero sfruttare il meccanismo di ereditarietà. Una interfaccia può estendere un’altra interfaccia. Si possono dichiarare delle variabili all’interno, che saranno implicitamente final e static, perché verranno inizializzato ad un valore costante e non potranno essere modificate dalle classi che implementano l’interfaccia.

**PASSAGGIO DI PARAMETRI**

* PASSAGGIO PER RIFERIMENTO 🡪 Quando passiamo un oggetto come argomento per un metodo, non passiamo mai l’oggetto in sé, ma passiamo un riferimento dell’oggetto. Quindi ogni modifica che apporteremo all’oggetto all’interno del metodo provocherà side-effect all’esterno del metodo, cioè sullo stato del programma.
* PASSAGGIO PER VALORE 🡪 Tutti i tipi primitivi vengono passati per valore ai metodi. Passaggio per valore significa esattamente effettuare una copia dell’oggetto e darla in pasto al metodo, perciò le operazioni eventualmente di modifica di tali variabili passati per valore non si ripercuoteranno all’esterno del metodo, avranno un campo d’azione prettamente locale.

**ARRAY**

La dimensione dell’array è nota a compile-time se dichiarata in maniera completa; altrimenti è nota a run-time, se prima viene dichiarato e poi viene inizializzato

**RIUTILIZZO DELLE CLASSI**

È possibile riutilizzare delle classi già scritte e testate in java piuttosto che scrivere classi completamente ex novo. Questo può avvenire tramite composizione o ereditarietà.

* Composizione 🡪 La nuova classe è composta da altre classi già esistenti; i riferimenti alle classi componenti sono inizializzati o nel costruttore della classe composta, o prima del costruttore, o prima che l’oggetto a cui si fa riferimento debba essere usato (lazy inizializzation).
* Ereditarietà 🡪 Il comportamento e/o lo stato di una nuova classe viene ereditato da una classe madre.

**INIZIALIZZAZIONE DELLA CLASSE BASE**

Se una classe che specializza una classe base non ha costruttore, automaticamente java chiama il costruttore della classe base (se il programmatore non l’ha specificato, crea automaticamente un costruttore con zero argomenti) SE LA CLASSE FIGLIA NON HA UN COSTRUTTORE. Se invece la classe figlia ha un costruttore, il programmatore deve effettuare manualmente la chiamata al costruttore della classe base tramite la parola chiave SUPER.

**PAROLA CHIAVE FINAL**

Applicata ai dati

* Tipi primitivi 🡪 Rende le variabili dichiarate final costanti. Il valore viene calcolato a compile time in modo da non sovraccaricare il run time;
* Oggetti 🡪 Rende IL RIFERIMENTO all’oggetto costante. L’oggetto può quindi variare, ma non può variare il binding fra la variabile e tale oggetto;
* Blank Final 🡪 Variabili con le caratteristiche di una costante ma non ancora inizializzati. Ciò permette una maggiore flessibilità perché possiamo avere un campo diverso di oggetto in oggetto che mantiene comunque le caratteristiche di un elemento final;
* Final arg 🡪 Java permette di rendere final anche gli argomenti dei metodi, in modo da non modificarne il riferimento durante l’esecuzione del metodo.

Applicato ai metodi

Se un metodo viene dichiarato final, questo non potrà essere sovrascritto nelle classi che estendono la classe in cui si trova, e ciò comporta un miglioramento in termini di efficienza. I metodi private sono tutti implicitamente final perché infatti non è possibile effettuare una operazione di override del metodo se si estende la classe in cui il metodo si trova.

Applicato alle classi

Una classe final non può essere estesa. Si dice “classe foglia”. I motivi che potrebbero spingerci a dichiarare una classe final sono quelli di affidabilità o efficienza.

**POLIMORFISMO**

Java supporta:

* Polimorfismo per overloading 🡪 Definire tanti metodi con lo stesso nome che in base o al contesto o agli argomenti si riconosce a quale metodo si fa riferimento
* Polimorfismo per inclusione 🡪 Polimorfismo che permette di trattare un riferimento ad una classe derivata come se fosse un riferimento alla sua superclasse. Questa conversione implicita va sotto il nome di upcasting. Genera legami dinamici (late binding). A dir la verità i legami in java sono quasi tutti dinamici: fanno eccezione solamente ciò che è static, final o private. I legami statici danno una efficienza migliore ma una estendibilità minore. Questo spiega come mai non possono esistere metodi statici nelle interfacce, perché non può creare un legame in fase di compilazione se non sa a quale realizzazione creare il legame statico. Mentre nelle classi astratte possono esistere metodi statici.

**PAROLA CHIAVE ABSTRACT**

Permette di creare metodi astratti (metodi senza implementazione) e classi astratte (classi con almeno un metodo astratto). Nei costruttori delle classe astratte è fortemente sconsigliato inserire chiamate a metodi che non siano final.

**TRATTAMENTO DELLE ECCEZIONI**

FASE 1: SOLLEVAMENTO DELL’ECCEZIONE

* Creare l’oggetto di tipo eccezione, come un qualsiasi altro oggetto java;
* Viene interrotto il normale flusso dell’esecuzione e si espelle il riferimento all’oggetto creato;
* Si analizza il contesto e si cerca il punto più opportuno da cui far ripartire il programma (cioè si passa alla fase 2).

FASE 2: GESTIONE DELL’ECCEZIONE

Il punto da cui far ripartire programma si chiama “gestore dell’eccezione” (exception handler). Ha il compito di decidere da dove far ripartire l’esecuzione del programma.

Il blocco di codice che può generare una eccezione si chiama zona protetta (guarded zone, sostanzialmente il blocco TRY).

VANTAGGI

La gestione delle eccezione è utilissima perché consente al programmatore di concentrarsi sul problema da risolvere, rimandando eventuale gestione di eccezione ad altri blocchi di programma, senza mescolare il codice come si fa nei linguaggi di programmazione che non supportano la gestione delle eccezioni.

In Java si può o sollevare esplicitamente una eccezione sfruttando un IF o altre strutture di controllo con l’istruzione THROW, oppure utilizzare un blocco TRY - CATCH. Nel blocco TRY va il codice che può sollevare eccezione, e seguono tanti blocchi catch quante sono le eccezioni che si decide di gestire. L’eccezione viene sollevata, gestita e poi si riprende il flusso dell’esecuzione dalla riga di codice esattamente successiva al blocco TRY - CATCH. Il gestore dell’eccezione è unico. Se l’eccezione non viene gestita nel blocco in cui viene sollevata, essa viene propagata in tutto il programma venendo restituita di volta in volta finché non viene gestita o finché non si esce dal MAIN.

CONSIGLI PER LA QUALITA

Non conviene utilizzare un triplo blocco TRY – CATCH - FINALLY perché il codice che ne viene fuori risulta di difficile comprensione. Conviene utilizzare un blocco TRY – CATCH e all’interno del catch un secondo blocco TRY - FINALLY .

GERARCHIA DELLE ECCEZIONIE

Tutte le eccezioni sono classi derivate dalla classe madre THROWABLE. Le due classi derivate da THROWABLE da cui a loro volta si derivano tutte le altre, sono ERROR (si occupa degli errori a COMPILE TIME) e EXCEPTION (RUN TIME). Le eccezioni che derivano da EXCEPTION si suddividono a loro volta in non controllate e controllate (dove il compilatore ci obbliga a mettere o il TRY – CATCH o il THROW). Quando definiamo noi le eccezioni, possiamo decidere se renderle controllate o non controllate decidendo la classe da estendere. Un criterio molto generale per decidere se controllare o non controllare l’eccezione si basa su una osservazione: se dobbiamo gestire interazioni con elementi esterni al nostro programma (ad esempio la lettura di una pagina web) la controlliamo, se invece dobbiamo interagire con elementi interni allora non la controlliamo.

**RTTI**

RTTI sta per Run Time Type Identificator. Java permette di venire a conoscenze di informazioni circa gli oggetti e le loro classi di appartenenza in tempo di esecuzione, e ciò è possibile tramite due differenti approcci:

* Approccio tradizionale 🡪 è possibile scoprire informazioni sui tipi sia durante la compilazione sia durante l’esecuzione;
* Approccio riflessivo (reflection) 🡪 è possibile scoprire informazioni sui tipi esclusivamente durante la fase di esecuzione.

RTTI TRADIZIONALE

Durante la fase di compilazione, per ogni classe presente in una applicazione viene creato un oggetto, contenente tutte le informazioni del file.class, figlio della classe Class. Questa classe talvolta è detta meta-classe. Questo oggetto ci permette di rappresentare le classi a run-time.

Non tutti i Class object vengono caricati in memoria nel momento della compilazione. Quando a run-time viene istanziato un oggetto, la JVM verifica se il Class object della classe corrispondente è stato caricato in memoria, e se così non è, lo carica ricercando il file.class corrispondente. Per ottenere il riferimento ad una classe possiamo utilizzare o la notazione puntata Class.forName(“nomeClasse”) oppure il class literal nomeClasse.class. Quest’ultimo metodo ci permette di avere un forte guadagno in semplicità ed efficienza (non si invoca un metodo e il Class object viene caricato a compile-time). Talvolta la scelta è obbligata, se infatti volessimo ottenere un riferimento alla classe di un oggetto in un blocco TRY - CATCH che viene valutato a runtime saremmo obbligati ad utilizzare .forName() e non name.class. Esempio dell’utilizzo del RTTI in java sono il cast o l’istruzione istanceof. isIstance() è un metodo che svolge lo stesso compito di istanceof ma viene utilizzato in contesti dinamici.

MECCANISMO DI RIFLESSIONE

Java supporta il meccanismo di riflessione. Tramite un pacchetto java.lang.reflect abbiamo a disposizione delle classi Field, Method, Constructor che ci vengono in aiuto per permette di estrarre tutte le informazione che vogliamo su un oggetto esclusivamente a run-time. La differenza sostanziale fra ii due tipi di RTTI sta nel fatto che il Class object è accessibile anche a compile-time dal primo, ed esclusivamente a run-time dal secondo. Il meccanismo di riflessione non è spesso utilizzato nelle applicazioni particolari.

**JAVA GENERICS**

Le java GENERICS sono un meccanismo che permette la realizzazione del concetto di astrazione generica. Tramite le java GENERICS è possibile:

* implementare codice che lavori su oggetti indipendentemente dalla loro classe di appartenenza;
* Permettono di specificare il tipo degli oggetti contenuti al momento della istanziazione/passaggio di parametri;
* Effettuano controlli di tipo (in base al tipo specificato) a compile-time, evitando di causare eccezione di ClassCastException a run-time;
* Generalizzare le interfacce.

È possibile generalizzare anche i metodi, indipendentemente se questi appartengano ad una classe generalizzata oppure no. Per di più, se il metodo è statico non potrà accedere al parametro di tipo della classe. Java permette una semplificazione che permette di inferire il tipo in maniera automatica, ma ciò funziona solo per l’assegnazione. Per superare questo limite per esempio nel passaggio di parametri, è indispensabile specificare esplicitamente i tipi. Le java GENERICS possono essere utilizzate anche in combinazione di numero variabile di argomenti di un metodo.