programmazione ad oggetti

JAVA

ii semestre ii anno 2023-2024

corso del prof. bicocchi

# E1.1 – java from functions to classes

### Programmazione procedurale

La **programmazione procedurale** punta ad avere una sequenza di procedure o funzioni che vengono chiamate in modo sequenziale per risolvere un determinato problema. Questo approccio non è adatto per grandi progetti.

Tra le varie problematiche della programmazione procedurale, abbiamo:

* ***Difficoltà nel riutilizzo del codice***, questo comporta ad avere:
  + **Codice riscritto**, nell’approccio procedurale è comune dover riscrivere codice perché le funzioni non sono facilmente riutilizzabili in contesti leggermente diversi.
  + **Difficile da manutenere**, la ripetizione del codice non solo aumenta la quantità di codice da mantenere, ma rende anche difficile trovare e correggere i bug. Se un bug è presente in un blocco di codice duplicato, è probabile che il bug esista in ogni duplicato di quel blocco.
  + **Meno testato**, l’approccio procedurale non incoraggia “la separazione delle responsabilità” il che porta a funzioni che fanno troppe cose e ciò rende difficile testare le unità separate.
* ***Non protezione dei dati***, all’aumentare della dimensione del codice è più semplice accedere ed usare una variabile fuori dallo scope per cui è stata creata (***violazione di sicurezza ed integrità***).

Immagine che contiene testo, Carattere, linea, bianco

Descrizione generata automaticamente

* ***I dati sono separati dalle operazioni***, questo rende difficile la decomposizione del codice. È più complesso suddividere e distribuire il lavoro efficacemente e può portare a conflitti e difficoltà di coordinamento, ad esempio:
  + Si può verificare una **mancanza di sincronizzazione** tra operazioni e dati, portando all’incongruenza dei dati.
  + **Complessità di debug**.
  + I **dati indipendenti** possono essere manipolati da più operazioni e ciò può portare all’inconsistenza dei dati.
  + **Operazioni non ottimizzate**, “mi dimentico della presenza di quel dato e ne alloco un altro”.

! Differenze: **Inconsistenza, incongruenza e incoerenza**

**Inconsistenza** 🡪 mancanza di affidabilità dei dati.

**Incongruenza** 🡪 mancanza di corrispondenza tra due o più elementi.

**Incoerenza** 🡪 mancanza di logica o di ordine tra i dati.

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

Immagine che contiene testo, schermata, software, Carattere

Descrizione generata automaticamente

### Programmazione ad oggetti

La **programmazione orientata agli oggetti**, punta ad un software sicuro, ottimizzabile e flessibile, adatto per grandi progetti. Si astrae ulteriormente rispetto alla programmazione procedurale tramite la definizione di un nuovo concetto noto come **classe** che unisce le variabili e i metodi in aggregazioni.

La programmazione ad oggetti è sviluppata su 3 fondamenta:

* **Encapsulation** (*incapsulamento*),
* **Inheritance** (*ereditarietà),*
* **Polymorphism** (*polimorfismo)*:
* **Polimorfismo** **statico**,(durante la compilazione)
* **Polimorfismo** **dinamico**, (durante l’esecuzione del programma).

# E2.1 – java basics

### Nozioni di base Java

Un programma Java è composto da **pacchetti** (*package*) contenenti uno o più file(*classi*).

Un **file** contiene una classe pubblica e volendo, zero o più classi private. Il nome del file deve essere necessariamente uguale al nome della classe pubblica.

In Java non esistono funzioni tradizionali ma metodi all’interno di classi. Un **metodo** è un blocco di codice che definisce il comportamento di un oggetto, quindi, è una funzione che opera sull’oggetto manipolando i dati al suo interno o influenzando lo stato della classe. I metodi possono avere parametri e possono ritornare *un valore*. In Java, quando si passa un argomento ad un metodo, che sia un tipo primitivo o un oggetto, esso viene passato come copia. Ci sono però significative differenze:

* **Tipi primitivi**, viene passata la copia effettiva del valore, questo significa che eventuali modifiche non influenzeranno il valore originale all’esterno del metodo.
* **Oggetti**, si passa una copia del riferimento all’oggetto, questo significa che punteranno allo stesso riferimento, quindi, eventuali modifiche saranno effettive anche al di fuori del metodo.

|  |
| --- |
| ! **Variabili di tipo riferimento** |
| Un **riferimento** è una variabile che permette di accedere ad un oggetto, senza di esso, non è possibile accedervi. I riferimenti sono variabili **primitive** e sono archiviate nello **stack**.  Gli oggetti puntati dai riferimenti sono allocati dinamicamente e risiedono nella memoria **heap**. |

Un programma Java inizia sempre con la chiamata al metodo ***main***.

I **tipi primitivi** in Java sono:

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

In Java ci sono diversi tipi di **letterali**:

Immagine che contiene testo, schermata, Carattere, design

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

In Java le **costanti** sono variabili il cui valore non può essere modificato una volta assegnato. Sono fondamentali anche per rendere il codice più sicuro e leggibile.

Immagine che contiene testo, Carattere, linea, bianco

Descrizione generata automaticamente

La classe **java.lang.Math** contiene metodi per eseguire operazioni numeriche di base, come le funzioni elementari esponenziale, logaritmo, radice quadrata e trigonometriche.

L'interfaccia **RandomGenerator** è progettata per fornire un protocollo comune per oggetti che generano sequenze casuali o (più tipicamente) pseudo-casuali di numeri (o valori booleani)

Immagine che contiene testo, Carattere, schermata

Descrizione generata automaticamente

Java ha a disposizione una serie di operatori. Gli **operatori** sono simboli o parole chiave che rappresenta un’azione da eseguire sui dati. Gli operatori sono organizzati in gerarchie di priorità:

Immagine che contiene testo, schermata, numero, Carattere

Descrizione generata automaticamente

Nella JDK 10 del 2008 è stata introdotta in Java la parola chiave **var**. Questa parola chiave permette al programmatore di non specificare il tipo, sarà il compilatore a dedurlo.

Immagine che contiene testo, Carattere, bianco, schermata

Descrizione generata automaticamente

**Casting.** Il casting può essere di:

* **implicit casting**, quando il tipo di destinazione è “più largo” del tipo di origine, viene eseguito automaticamente dal compilatore e senza perdita di informazione.
* **explicit casting,** quando il tipo di destinazione è “più piccolo” del tipo di origine e potrebbe causare la perdita di informazioni (es. double d = 2.00003; long l = (long) d; ).

**Variabili di riferimento**. Un riferimento è una variabile che fornisce un modo per accedere a un oggetto. In genere, non è possibile accedere a un oggetto senza un riferimento ad esso. I riferimenti sono variabili primitive e sono memorizzati nello stack.

Gli oggetti, invece, sono allocati dinamicamente e risiedono nella memoria heap. Per questo motivo, il ciclo di vita degli oggetti non dipende da alcun metodo specifico. (I riferimenti e gli oggetti sono l'equivalente Java dei puntatori struct e delle struct allocate dinamicamente in C.)

**Passaggio di parametri ai metodi**. È possibile passare parametri primitivi (compresi i riferimenti a oggetti) a metodi, i parametri sono sempre passati per valore (copiati nello stack del metodo ricevente).

### Convenzioni

In Java ci sono diverse **convenzioni di codifica**:

* I package sono scritti in minuscolo.
* Le interfacce devono essere rappresentate con un aggettivo.
* I metodi di accesso ai dati devono iniziare con la parola “get”.
* I metodi di modifica dei dati devono iniziare con la parola “set”.
* I metodi devono essere rappresentati da una frase imperativa.
* Un metodo booleano deve iniziare con la parola “is” o “has”.

Immagine che contiene testo, Carattere, schermata

Descrizione generata automaticamente

### Istruzioni per il controllo di flusso

|  |  |  |
| --- | --- | --- |
| Decision statements | | |
| if | switch | switch (enhanced) |
|  |  |  |
| Iterative statements | | |
| do-while | while | for |
|  |  |  |

L' istruzione **break** può essere usata per uscire da un ciclo.

L' istruzione **continue** interrompe un'iterazione (nel ciclo), ma continua con l'iterazione successiva invece di uscire.

# E2.2 – java basics: Arrays

In Java un **array** è una struttura dati che può memorizzare una sequenza di elementi di dimensione fissa dello stesso tipo di dati, è anch’esso un oggetto e l’indice parte da 0. La lunghezza di un array è fissa e non può essere modificata successivamente.

Gli array possono memorizzare elementi di qualsiasi tipo, inclusi i tipi primitivi e possono essere anche multidimensionali. Un array **multidimensionale** è un array che contiene altri array.

Creare in Java un array equivale ad allocare spazi contigui nell’ heap della dimensione del tipo di dato scelto, ma ***attenzione***:

* Se il tipo di dato è primitivo equivale ad avere in memoria valori contigui in memoria.
* Se il tipo di dato è un oggetto equivale ad avere in memoria i riferimenti (i **puntatori**) ai valori contigui in memoria (***NON*** i valori in se).

|  |
| --- |
| ! **Ricorda** |
| Creare un vettore significa allocare una zona di memoria fissa che viene inizializzata a 0 se non vengono specificati i valori. |
| Anche con gli array multidimensionali la memoria è contigua ed organizzata in questo modo:    ! **ATTENZIONE**: L’immagine descrive un array multidimensionale di tipi primitivi. In caso di oggetti i *dati contigui sono riferimenti*. |

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente

Si può trovare la dimensione dell'array tramite l'attributo associato *numbers.length*.

Se l'indice punta all'esterno dell'Array, ovvero l'elemento non esiste, otteniamo un *ArrayIndexOutOfBoundsException*. Questo errore indica che l'Array non contiene l'indice specificato.

**Passaggio di array ai metodi**. Si può usare gli array come parametro di un metodo proprio come qualsiasi altra variabile. Quando si usa l’array come parametro di un metodo, il metodo riceve una copia del riferimento all'array.

**Un array è un oggetto, quindi quando si modifica l'array all'interno del metodo, le modifiche persistono dopo l'esecuzione del metodo.**

### Array Utilities

*java.util.Arrays* contiene vari metodi per manipolare array come l'ordinamento, la ricerca, il riempimento, la stampa o la visualizzazione di collections:

* copyOf() / copyOfRange
* fill()
* equals() / deepEquals()
* hashCode() / deepHashCode()
* sort() – ordinare un vettore
* binarySearch() – ricercar di un elemento in un array ordinato
* toString() / deepToString() – stampare un array
* asList() – trasformarlo in una lista
* setAll() – trasformarlo in una collection

Altro metodo utile:

*System.arraycopy()* copia un array dall'array sorgente specificato, iniziando dalla posizione specificata, alla posizione specificata dell'array di destinazione. Il numero di componenti copiati è uguale all'argomento lunghezza.

# E2.3 JAVA BASICS: STRINGS

La classe **String** rappresenta stringhe di caratteri (sono oggetti e non sono puntatori a char). Sono **immutabili** e quindi i valori non possono essere modificati.

|  |
| --- |
| ! **Differenze nella creazione di stringhe** |
| Le stringhe inizializzate in questa modalità non vengono ricopiate in memoria, sono archiviate nella zona chiamata “**string pool**”. Quindi, quando viene inizializzata una stringa, il compilatore Java controllerà prima in questa zona di memoria. In caso esista la stringa verrà ritornato il riferimento alla stringa esistente. Il pool delle stringhe è gestito dalla **Java Virtual Machine** (JVM). |
| Le stringhe inizializzate con l’operatore new verranno archiviate in memoria heep come qualsiasi altro oggetto. Quindi, ogni stringa occupa la sia memoria. |
| ! **Confronto di stringhe** | |
| String è un oggetto quindi esso funziona allo stesso modo. Per confrontare delle stringhe non basta usare l’operatore “==”, ci sarà solo il confronto dei riferimenti. | |

Quando si parla di stringhe, fondamentale è l’operatore “**+**” per la concatenazione di stringhe:

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente

|  |
| --- |
| ! **Performance - STRINGBUILDER** |
| L’utilizzo dell’operatore “+” per la concatenazione di stringhe può essere utilizzato per concatenare poche stringhe. Ad ogni concatenazione le stringhe verranno scartate e ci sarà l’allocazione e la copia di una nuova stringa in memoria. All’aumentare delle concatenazioni questa operazione diventa **ordini di grandezza più lento** dell’uso di altri modi di concatenazione come la funzione “append()” di **StringBuilder**. |

### Standard streams

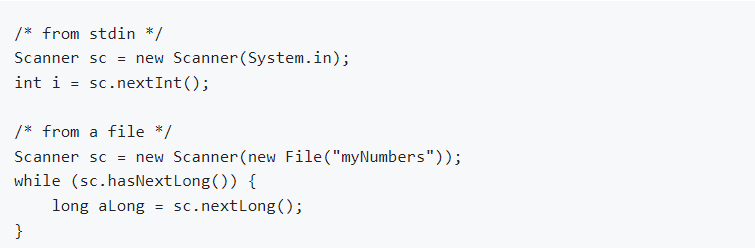
Java dispone di tre flussi che vengono utilizzati per l’input e l’output delle applicazioni Java. Vengono inizializzati dal runtime di Java all’avvio della JVM. I tre flussi sono:

* **System.in**, chiamato standard input, utilizzato per leggere i dati in ingresso da un flusso di input.
* **System.out**, chiamato standard output, utilizzato per scrivere dati di output su un flusso di output.
* **System.err**, chiamato standard error, utilizzato per inviare messaggi di errore o di avviso alla console o a un altro flusso di output destinato a gestire errori.

**Metodi PrintStream**. Stampa una stringa:



**Scanner** è una classe di utilità per leggere da vari input e ha metodi comodi per lavorare con l'input.Suddivide il suo input in token usando un modello di delimitazione, che per impostazione predefinita corrisponde agli spazi vuoti. I token risultanti possono quindi essere convertiti in valori di tipi diversi usando i vari metodi successivi.



# E3.1 – JAVA CLASSES

Una classe in Java è dichiarata con la parola chiave **class**, mentre, per crearne l’istanza, si utilizza la parola chiave **new**:

Immagine che contiene testo, Carattere, schermata

Descrizione generata automaticamente

Il corpo può includere:

* **Metodi**, funzioni definite all'interno della classe che consentono di definirne il comportamento e di manipolarne i dati.

Un metodo può essere di due tipi:

* **Statico**, per invocare il metodo non è necessario creare un oggetto. Invochiamo il metodo con il nome della classe.

Un metodo statico ha però delle differenze sostanziali:

* Può accedere solo a campi statici.
* Può invocare solo metodi statici.
* Non può fare riferimento alla parola this.

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

* **Di istanza**, per invocare il metodo è necessario creare un oggetto.

I metodi di istanza possono accedere a metodi statici e hanno il vantaggio di poter accedere ai campi dell’oggetto. Per riferirsi ai campi di una classe si usa la parola chiave **this** (rappresenta una particolare istanza della classe) ed è fondamentale per differenziarlo dai campi un altro oggetto.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

* **Campi** (**attributi**, **variabili di istanza**), sono variabili contenute in un oggetto e che ne definiscono lo stato. Possono essere di qualsiasi tipo, compresi tipi primitivi e classi stesse.

Anche i campi possono essere dichiarati statici, questo crea il vantaggio di poter condividere delle variabili tra le classi e risparmiare memoria, però, non è una buona idea dichiarare variabili statiche non costanti.

Immagine che contiene testo, schermata

Descrizione generata automaticamente

* **Costruttori**, creano e inizializzano nuovi oggetti della classe, cioè, impostano i valori iniziali dei campi.

Sono metodi speciali che vengono chiamati automaticamente quando viene creata un’istanza della classe con la parola chiave new.

Si differenziano dagli altri metodi perché:

* Hanno lo stesso nome della classe che lo contiene.
* Non hanno tipo di ritorno, nemmeno void.

Si possono definire infiniti costruttori, a patto che, i parametri siano diversi, questo concetto è nome come **sovraccarico del costruttore**.

Immagine che contiene testo, schermata, Carattere, software

Descrizione generata automaticamente

### Modificatori di accesso

Un modificatore di accesso è una parola chiave che specifica chi può usare il tuo codice o parti di esso, posizionabile davanti a campi, metodi o classi. Serve per la chiarezza del codice, permettendo di "nascondere" parti complesse e fornendo solo le interfacce necessarie (come i pulsanti di una lavatrice). Inoltre, garantisce la sicurezza del codice, impedendo ad altri sviluppatori di modificare variabili critiche, assicurando così che il comportamento del codice rimanga prevedibile.

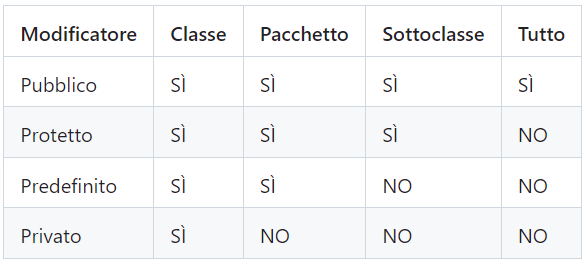
### VISIBILITà

**Visibiltà della classe**. Una classe può avere uno dei due seguenti modificatori:

* **package-private** (predefinito, nessun modificatore esplicito): visibile solo per le classi dello stesso pacchetto;
* **pubblico**: visibile a tutte le classi ovunque.

Il modo più comune di utilizzare i modificatori è:

* rendere pubbliche le classi contenenti metodi per gli utenti (i "pulsanti") ;
* rendere private tutte le altre classi con metodi logici di basso livello utilizzati da quelle pubbliche (coprire il motore con il corpo).

**Visibilità dei membri.** Un membro di una classe (un campo o un metodo, ad esempio un costruttore di classe) ha più opzioni tra cui scegliere:

* *privato* disponibile solo all'interno di una classe;
* *package-private* (noto anche come default) disponibile per tutte le classi nello stesso pacchetto;
* *protetto* disponibile per le classi nello stesso pacchetto e per le sottoclassi (sarà trattato in seguito);
* *pubblico*, accessibile per tutte le classi e ovunque.

### Incapsulamento dei dati (encapsulation)

Nella maggior parte dei casi, una classe non espone i suoi campi ad altre classi. Invece, rende i suoi campi accessibili tramite i cosiddetti metodi di accesso.

Modificatori d’accesso

Secondo il principio di incapsulamento dei dati, i campi di una classe non sono accessibili direttamente da altre classi. I campi sono accessibili solo tramite i metodi di quella particolare classe.

Per accedere ai campi nascosti, i programmatori scrivono tipi speciali di metodi:

* **Getter**, iniziano con get per convenzione, tranne per i boolean che iniziano per ***is***, servono per ottenere il valore di una variabile.
* **Setter**, iniziano con set per convenzione, servono per modificare il valore di una variabile, controllato dal progettista della classe.

Entrambi i tipi di metodi dovrebbero essere public. L'utilizzo di questi metodi ci offre alcuni vantaggi:

* i campi di una classe possono essere resi di sola lettura, di sola scrittura o entrambi;
* una classe può avere il controllo totale sui valori memorizzati nei campi;
* gli utenti di una classe non sanno come la classe memorizza i suoi dati e non dipendono dai campi.

**Perché l'Incapsulamento è importante?**

L'incapsulamento è una tecnica che nasconde i dettagli di implementazione di una classe, avvolgendola in uno "scudo protettivo", per proteggere i dati da modifiche indesiderate o da accessi non autorizzati. La classe mette a disposizione all’esterno solo un elenco di metodi pubblici.

* *Raggruppamento di dati e metodi*: All’interno della classe stessa si raggruppano i dati e i metodi che operano sugli stessi attributi.
* *Controllo d’accesso:* Attraverso l’uso di modificatori di accesso.

Se è presente un bug all’interno del codice, possiamo escludere a priori che l’errore dipenda da un utilizzo scorretto da parte dell’esterno, proprio perché gli unici modi per utilizzare metodi o richiedere delle variabili sono interni. Questo rende notevolmente più semplice trovare e sistemare il bug.

### Immutabilità degli oggetti

**Immutabilità** significa che una volta inizializzato un oggetto, esso non può essere cambiato. Per modificare un oggetto è necessario ricrearne uno nuovo con le caratteristiche specificate. I tipi immutabili permettono di scrivere programmi con meno errori. (le stringhe sono un esempio di oggetto immutabile)

|  |
| --- |
| Convezione nella creazione di classi |
| * privato, campo finale per ogni pezzo di dati; * getter per ogni campo; * costruttore pubblico con un argomento corrispondente per ogni campo; * metodo equals che restituisce true per oggetti della stessa classe quando tutti i campi corrispondono; * metodo hashCode che restituisce lo stesso valore quando tutti i campi corrispondono; * metodo toString che include il nome della classe e il nome di ciascun campo e il suo valore corrispondente. |

**I record sono classi di dati immutabili che richiedono solo il tipo e il nome dei campi**. I metodi equals, hashCode e toString, così come i campi private, final e il costruttore public, sono generati dal compilatore Java.

### Condivisione dei riferimenti

Più di una variabile può fare riferimento allo stesso oggetto. È importante capire che due variabili fanno riferimento agli stessi dati in memoria piuttosto che a due copie indipendenti. Poiché la classe degli oggetti è mutabile, possiamo modificare l'oggetto usando entrambi i riferimenti.

Per il riferimento all'oggetto sono definiti solo gli operatori relazionali == e !=:

* La condizione di uguaglianza indica se due riferimenti puntano allo stesso oggetto in memoria
* La condizione di uguaglianza viene valutata sui valori dei riferimenti (ovvero gli indirizzi in memoria)!

### Garbage Collector

Il Garbage Collector tiene traccia di ogni oggetto disponibile nell’heap e rimuove quelli inutilizzati. Il suo compito principale è:

* Rilevare gli oggetti non più raggiungibili.
* Liberare la memoria degli oggetti non referenziati.

|  |  |
| --- | --- |
| Vantaggi | Svantaggi |
| Nessuna gestione manuale della memoria | Richiede maggiore uso della CPU |
| Gestione automatica della perdita di memoria | I programmatori non hanno il controllo su quando avverrà la liberazione della memoria |
| I puntatori pendenti e i puntatori selvaggi nella programmazione del computer sono puntatori che non puntano a un oggetto valido del tipo appropriato. Questi sono casi speciali di violazioni della sicurezza della memoria. Più in generale, i riferimenti penzolanti e i riferimenti selvaggi sono riferimenti che non si risolvono in una destinazione valida. |  |

### Immagine che contiene testo, schermata, Carattere, numero Descrizione generata automaticamenteClassi Wrapper

Ogni tipo primitivo ha una classe a lui dedicata. Queste classi sono note come wrapper e sono **immutabili** (proprio come le stringhe). Le classi wrapper possono essere utilizzate in diverse situazioni:

* quando una variabile può essere null(assenza di un valore);
* quando si desidera utilizzare metodi speciali di queste classi (ad esempio per la conversione da e verso stringhe).
* quando è necessario memorizzare valori in raccolte generiche (sarà preso in considerazione nei prossimi argomenti);

Gli oggetti wrapper sono tipi di riferimento (puntatori).

Il **boxing** è la conversione di tipi primitivi in ​​oggetti delle corrispondenti classi wrapper. L'**unboxing** è il processo inverso. C'è un possibile problema durante l'unboxing. Se l'oggetto wrapper è null e viene utilizzato in una inizializzazione o in altre operazioni, l'unboxing genera un NullPointerException.

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente

# E3.2 – java classes: inheritance

L'ereditarietà è un meccanismo che permette di creare una nuova classe derivata da un'altra, acquisendo campi e metodi della classe base senza riscriverli. Rappresenta la relazione IS-A ed è fondamentale nella programmazione orientata agli oggetti, consentendo la creazione di gerarchie di classi e il riutilizzo del codice. La classe derivata è chiamata sottoclasse, mentre la classe da cui deriva è chiamata superclasse.

Immagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamente

Ci sono punti importanti sull'ereditarietà in Java:

* Java non supporta l'ereditarietà multi-classe, quindi una classe può ereditare solo da una singola superclasse.
* Una gerarchia di classi può avere più livelli (una classe C può estendere una classe B che estende una classe A).
* Una superclasse può avere più di una sottoclasse.
* Una sottoclasse eredita tutti i campi e i metodi pubblici e protetti dalla superclasse, ma non quelli privati. Tuttavia, i metodi pubblici o protetti della superclasse possono fornire accesso ai campi privati.
* I costruttori non sono ereditati, ma possono essere invocati dalla sottoclasse usando la parola chiave super.
* Se una classe è dichiarata con la parola chiave final, non può avere sottoclassi.

Esistono due modi per fare riferimento a un oggetto sottoclasse:

* Riferimento alla **sottoclasse,** creando istanze delle sottoclassi utilizzando il loro costruttore.
* Riferimento alla **superclasse,** dove una variabile della superclasse può fare riferimento a oggetti delle sue sottoclassi.

Accesso ai campi e ai metodi tramite un riferimento alla superclasse:

L'ereditarietà è un meccanismo che permette di creare una nuova classe derivata da un'altra, acquisendo campi e metodi della classe base senza riscriverli. Rappresenta la relazione IS-A ed è fondamentale nella programmazione orientata agli oggetti, consentendo la creazione di gerarchie di classi e il riutilizzo del codice. La classe derivata è chiamata sottoclasse, mentre la classe da cui deriva è chiamata superclasse.

Ci sono punti importanti sull'ereditarietà in Java:

* Java non supporta l'ereditarietà multi-classe, quindi una classe può ereditare solo da una singola superclasse.
* Una gerarchia di classi può avere più livelli (una classe C può estendere una classe B che estende una classe A).
* Una superclasse può avere più di una sottoclasse.
* Una sottoclasse eredita tutti i campi e i metodi pubblici e protetti dalla superclasse, ma non quelli privati. Tuttavia, i metodi pubblici o protetti della superclasse possono fornire accesso ai campi privati.
* I costruttori non sono ereditati, ma possono essere invocati dalla sottoclasse usando la parola chiave super.
* Se una classe è dichiarata con la parola chiave final, non può avere sottoclassi.

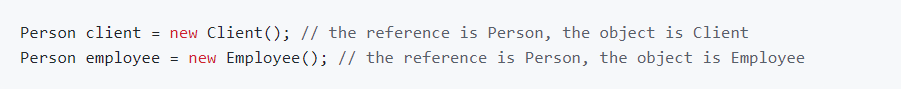
Esistono due modi per fare riferimento a un oggetto sottoclasse:

* Riferimento alla **sottoclasse,** creando istanze delle sottoclassi utilizzando il loro costruttore.

Immagine che contiene testo, Carattere, linea, schermata

Descrizione generata automaticamente

* Riferimento alla **superclasse,** dove una variabile della superclasse può fare riferimento a oggetti delle sue sottoclassi.



Accesso ai campi e ai metodi tramite un riferimento alla superclasse: Un **riferimento di superclasse** può essere usato per qualsiasi oggetto di sottoclasse derivato da essa, ma non può accedere ai membri specifici della sottoclasse.

**Casting tra superclasse e sottoclasse**: È possibile eseguire il cast di un oggetto di una sottoclasse nella sua superclasse e viceversa, ma il cast alla sottoclasse deve essere fatto con attenzione per evitare ClassCastException.

Immagine che contiene testo, Carattere, schermata, algebra

Descrizione generata automaticamente

**Quando utilizzare il riferimento alla superclasse**: Utile per l'elaborazione di una raccolta di oggetti di tipi diversi della stessa gerarchia e per metodi che accettano oggetti della classe base ma funzionano con le sue sottoclassi.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

**Accesso ai campi e ai metodi della superclasse**: La parola chiave super può essere usata per accedere ai metodi di istanza o ai campi della superclasse, utile per distinguere tra membri con lo stesso nome. Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

**Invocazione del costruttore della superclasse:** L'invocazione super(...) deve essere la prima istruzione in un costruttore di sottoclasse.

Il costruttore predefinito di una sottoclasse chiama automaticamente il costruttore senza argomenti della superclasse.

Inoltre, nelle sottoclassi è possibile sovrascrivere i metodi equals() e hashCode() per confrontare campi aggiuntivi.

**Tipi di ereditarietà:**

* Ereditarietà singola: una sottoclasse eredita da una singola classe.
* Ereditarietà multipla: una sottoclasse eredita da più classi (non supportata direttamente in Java).
* Ereditarietà multilivello: una sottoclasse può ereditare da un'altra sottoclasse.
* Ereditarietà gerarchica: più classi derivano da una singola classe.

**!** In Java, l'ereditarietà multipla diretta non esiste, ma è possibile ottenere un comportamento simile implementando più interfacce.

### La classe Object

La Java Standard Library ha una classe denominata Object che è il genitore predefinito di tutte le classi standard e delle tue classi personalizzate. Ogni classe estende questa implicitamente, quindi è la radice dell'ereditarietà nei programmi Java. La classe appartiene al *java.lang* pacchetto che viene importato di default.

La Object classe può fare riferimento a un'istanza di qualsiasi classe perché qualsiasi istanza è un tipo di Object( upcasting ).

La Object classe fornisce alcuni metodi comuni a tutte le sottoclassi. Ha nove metodi di istanza (esclusi i metodi sovraccaricati) che possono essere divisi in quattro gruppi:

* **identità dell'oggetto**:
  + *hashCode* (valore del codice hash intero dell'oggetto, questo può cambiare in più esecuzioni della stessa applicazione),
  + *equals* (confronto tra due riferimenti)

Se due oggetti sono uguali in base al equals()metodo, allora il loro codice hash deve essere lo stesso. Se due oggetti sono disuguali secondo il equals()metodo, non è necessario che il loro codice hash sia diverso. Il valore del loro codice hash può essere uguale o meno;

* **rappresentazione leggibile dall'uomo**: *toString*;
* **sincronizzazione thread**:
  + wait,
  + notify,
  + notifyAll;
* **gestione degli oggetti**:
  + finalize,
  + clone,
  + getClass.

### polimorfismo

Immagine che contiene testo, diagramma, linea, schermata

Descrizione generata automaticamenteIl polimorfismo (polimorfismo) è la capacità di fornire diverse implementazioni di un metodo ereditato a seconda del tipo di parametro passato al metodo, quindi sono concetti generici che si comportano diversamente in base alla situazione. Il polimorfismo si divide in:

* Polimorfismo **statico**: si verifica durante la **compilazione**, il compilatore determina quale metodo o sovraccarico di metodo chiamare in base al tipo statico delle variabili coinvolte nell’invocazione del metodo. Java lo supporta come **overloading** di metodi, consentendo ai programmatori di definire più metodi con lo stesso nome ma parametri diversi.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

* Polimorfismo **dinamico**: la decisione su quale metodo eseguire dipende dal tipo dinamico dell’oggetto passato durante l’esecuzione del programma. Java lo supporta tramite **l’override** dei metodi. Il polimorfismo in fase di **esecuzione** funziona quando un metodo sovrascritto viene chiamato tramite la variabile di riferimento di una superclasse. Java determina in fase di esecuzione quale versione del metodo (superclasse/sottoclassi) deve essere eseguita in base al tipo di oggetto a cui si fa riferimento, non al tipo di riferimento. Utilizza un meccanismo noto come *dynamic* *method* *dispatching*.

Immagine che contiene testo, schermata, software, Carattere

Descrizione generata automaticamente

* Polimorfismo **parametrico**: quando il codice è scritto senza menzionare alcun tipo specifico e quindi può essere utilizzato in modo trasparente con qualsiasi numero di nuovi tipi. Java lo supporta come **generici** o **programmazione** **generica**.

### METODI DI ISTANZA DI OVERRIDE

L'override dei metodi di Java è una caratteristica fondamentale che consente alle sottoclassi di ridefinire i comportamenti ereditati dalle loro superclassi. Java offre l'opportunità di dichiarare un metodo in una sottoclasse con lo stesso nome di un metodo nella superclasse. Il *vantaggio dell'override* è che una sottoclasse può fornire la propria implementazione specifica di un metodo della superclasse. L'override dei metodi nelle sottoclassi consente a una classe di ereditare da una superclasse il cui comportamento è sufficientemente simile e quindi di modificare tale comportamento in base alle esigenze della sottoclasse.

I metodi di istanza possono essere sovrascritti se sono ereditati dalla sottoclasse. Il metodo sovrascritto deve avere lo stesso nome, parametri (numero e tipo di parametri) e tipo di ritorno (o una sottoclasse del tipo) del metodo sovrascritto.

**Regole per l'override dei metodi**:

* Il metodo deve avere lo stesso nome della superclasse.
* Gli argomenti dovrebbero essere esattamente gli stessi del metodo della superclasse.
* Il tipo di ritorno dovrebbe essere lo stesso tipo o un sottotipo del tipo di ritorno dichiarato nel metodo della superclasse.
* Il livello di accesso deve essere uguale o più aperto del livello di accesso del metodo sovrascritto.
* Un metodo privato non può essere sovrascritto perché non è ereditato dalle sottoclassi.
* Se la superclasse e la sua sottoclasse sono nello stesso pacchetto, allora i metodi privati del pacchetto possono essere sovrascritti.
* I metodi statici non possono essere sovrascritti.

Per verificare queste regole, c'è un'annotazione speciale @Override. Se il compilatore decide che il metodo non può essere sovrascritto, genererà un errore. Tuttavia, l'annotazione non è obbligatoria, è solo per comodità. Se si desidera **impedire** **l'override** di un metodo, dichiararlo con la parola chiave *final.*

L’override e l’overload dei metodi possono essere combinati insieme anche se sono due meccanismi differenti.

**Nascondere i metodi statici:** i metodi statici non possono essere sovrascritti. Se una sottoclasse ha un metodo statico con la stessa firma (nome e parametri) di un metodo statico nella superclasse, allora il metodo nella sottoclasse nasconde quello nella superclasse. Questo è completamente diverso dall'override del metodo.

# E3.3 java classes: interface, abstract classes

### Interfaccia

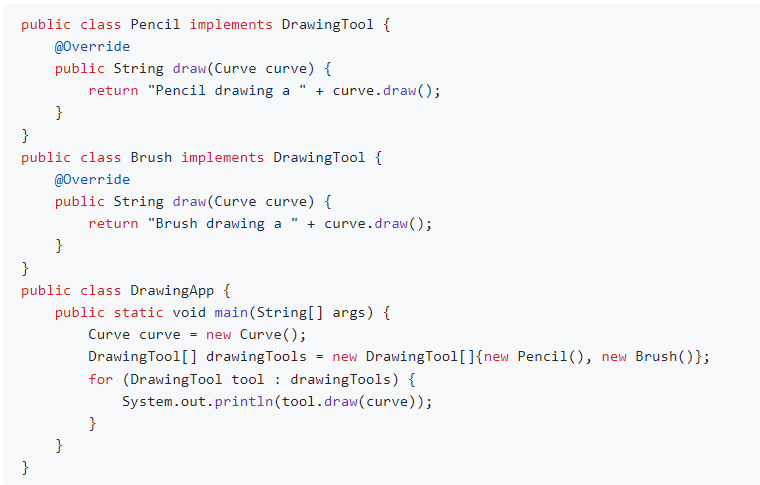
Un'interfaccia può essere considerata un tipo speciale di classe che non può essere istanziata. Per dichiarare un'interfaccia, dovresti usare la parola chiave *interface* invece di *class*. Un'interfaccia può contenere:

* **costanti** **pubbliche**.
* metodi **astratti** *senza* *implementazione* (la parola chiave abstract non è richiesta qui).
* metodi **statici** con implementazione (la parola chiave static è obbligatoria).
* metodi **predefiniti** con implementazione (la parola chiave default è obbligatoria), i metodi di interfaccia sono *astratti* per impostazione predefinita, ciò significa che **non** possono avere un corpo ma dichiarano solo una firma. Con l’eccezione dei metodi di default che possono avere un corpo.
* metodi **privati** ​​con implementazione.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Se i modificatori non vengono specificati una volta dichiarato il metodo, i suoi parametri saranno **pubblici** e **astratti** per impostazione predefinita. I metodi statici, predefiniti e privati ​​devono avere un'implementazione nell'interfaccia.



L’utilizzo delle interfacce è un altro modo di supportare **il polimorfismo**.

Una delle caratteristiche più importanti dell'interfaccia è **l'ereditarietà multipla**.

Immagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamente

Perché sono necessari i metodi predefiniti o di default?

L'idea principale di un'interfaccia è dichiarare la funzionalità. I ​​metodi predefiniti estendono quell'idea. Non solo dichiarano la funzionalità, ma la implementano anche. La ragione principale è supportare la **retrocompatibilità**.

Lo scopo principale dei metodi statici dell'interfaccia è definire funzionalità di utilità comuni a tutte le classi che implementano l'interfaccia. Aiutano a evitare la duplicazione del codice e a creare classi di utilità aggiuntive.

In alcune situazioni, un'interfaccia *non ha alcun membro*. Tali interfacce sono chiamate interfacce **marcatore** o **taggate**.

Qual è lo scopo dei metodi statici?

### Classi astratte

Una classe astratta è una classe dichiarata con la parola chiave **abstract**. Rappresenta un concetto astratto che viene utilizzato come classe base per le sottoclassi. Le classi astratte hanno alcune caratteristiche speciali:

* è impossibile creare un'istanza di una classe astratta;
* una classe astratta può contenere metodi astratti che devono essere implementati in sottoclassi non astratte;
* può contenere campi e metodi non astratti (inclusi quelli statici);
* una classe astratta può estendere un'altra classe, inclusa una astratta;
* può contenere un costruttore.

Una classe astratta presenta due differenze principali rispetto alle classi normali (concrete): **nessuna** **istanza** e metodi **astratti**.

I metodi astratti vengono dichiarati aggiungendo la parola chiave *abstract.* Hanno una dichiarazione (modificatori, un tipo di ritorno e una firma) ma non hanno un'implementazione. Ogni sottoclasse concreta (non astratta) deve implementare questi metodi.

**!** i metodi statici **non** possono essere astratti.

Immagine che contiene testo, schermata, Carattere, diagramma

Descrizione generata automaticamenteImmagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Le classi e le interfacce astratte sono entrambe strumenti per ottenere l'astrazione che consentono di dichiarare metodi astratti. Non si possono creare istanze di classi e interfacce astratte direttamente, possiamo farlo solo tramite classi che le ereditano.

**Utilizzo di classi astratte e interfacce insieme.** Spesso interfacce e classi astratte vengono utilizzate insieme per rendere più flessibile una gerarchia di classi. In questo caso, una classe astratta contiene membri comuni e implementa una o più interfacce, mentre le classi concrete estendono la classe astratta e possibilmente implementano altre interfacce.

### Classi anonime

Java fornisce un meccanismo per creare una classe in una singola istruzione senza dover dichiarare una nuova classe denominata. Tali classi sono chiamate anonime perché **non** hanno identificatori di nome. La classe anonima viene dichiarata e istanziata contemporaneamente a un'espressione. Sostituisce entrambi i metodi dell'interfaccia.

Immagine che contiene testo, diagramma, linea, Carattere

Descrizione generata automaticamenteImmagine che contiene testo, schermata

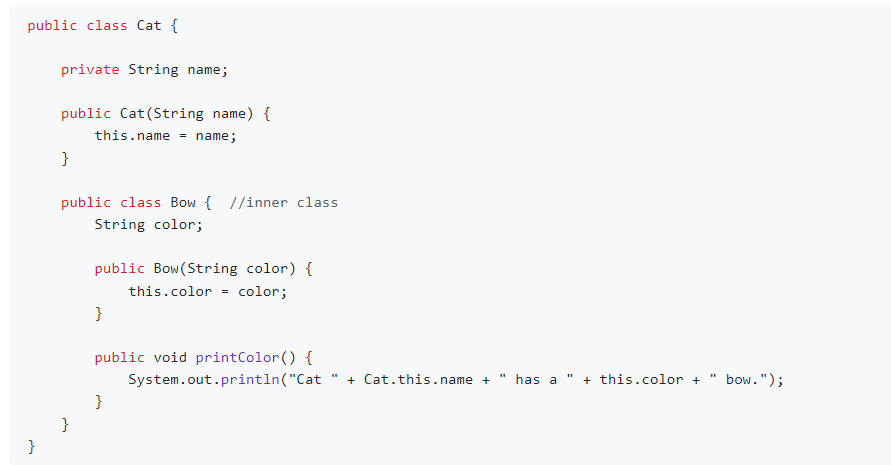
Descrizione generata automaticamente

### Classi nidificate (nasted)

È possibile definire una **classe nasted** quando è dichiarata all'interno di un'altra classe.

**Classi interne** Un'inner class è una classe dichiarata all'interno di un'altra classe (non statica). Ha accesso diretto ai membri della classe esterna, inclusi quelli privati. Può essere istanziata solo in relazione a un'istanza della classe esterna.

Dall'interno della classe interna, si possono vedere tutti i metodi e i campi della classe esterna, anche se sono *private.* Una classe interna è associata a un'istanza della sua classe di contenimento. Quindi, per creare un'istanza di una classe interna e accedervi, devi prima creare un'istanza della classe esterna. Se si crea una classe interna *private,* allora è possibile accedervi solo dall'interno della classe esterna. Lo stesso vale per i campi e i metodi.



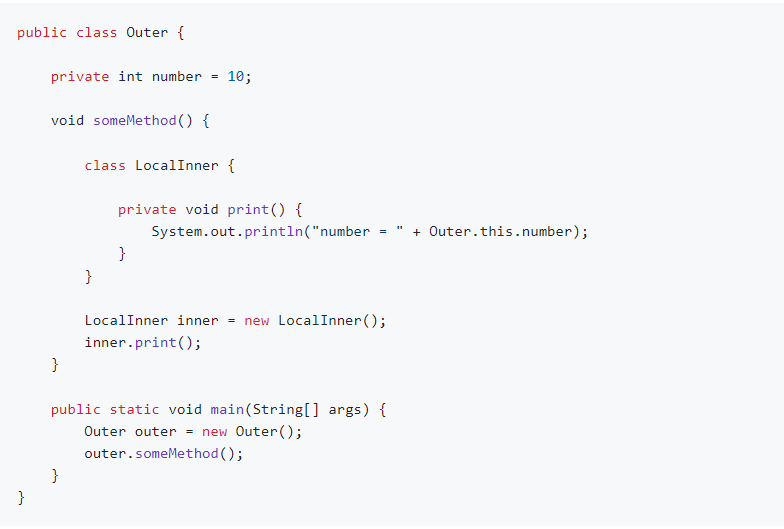
**Classi nidificate statiche** (static nested class)

Una static nested class è una classe dichiarata all'interno di un'altra classe con l'uso della parola chiave static. Non ha accesso diretto ai membri istanza della classe esterna (può accedere solo ai membri statici). Può essere istanziata senza un'istanza della classe esterna.

Ricordati dei modificatori di accesso: se crei una classe nidificata statica private, allora è possibile accedervi solo all'interno della classe esterna. Lo stesso vale per i campi e i metodi.

**Classe interna locale** (Local inner class)

Una local inner class è una classe definita all'interno di un metodo, di un costruttore o di un blocco di inizializzazione. È visibile solo all'interno del suo ambito e può accedere ai membri della classe esterna e alle variabili locali finali o effettivamente finali del metodo in cui è dichiarata.



**Cosa possiamo vedere all'interno della classe interna locale?**

Membri della classe esterna, incluso il campo *number*. E variabili locali del blocco di chiusura, come void *someMethod*. Le variabili locali devono essere dichiarate come *final* o essere effettivamente *final*, quest'ultima significa che il loro valore non viene mai modificato dopo l'inizializzazione e non c'è bisogno della parola chiave *final*.

**!** una classe interna locale può essere istanziata solo all'interno del blocco in cui è definita la classe interna. Quindi altre parti del codice non sanno che esiste.

# E4.1 Java data structures

La scelta della struttura dati è fondamentale quando si progetta un software, tra i motivi più importanti:

* **Efficienza algoritmica**, la capacità di risolvere un problema in modo rapido ed efficiente. Questa efficienza dipende da diversi fattori:
* **Complessità** **temporale**, il tempo necessario per eseguire l’algoritmo.
* **Complessità** **spaziale**, quanta memoria è necessaria per eseguire l’algoritmo.

Ogni struttura dati è ottimizzata per eseguire determinate operazioni, ad esempio una linked-list è efficiente nell’aggiunta in testa e così via.

* **Utilizzo della memoria**:
* **Dimensione della struttura dati**, le strutture dati possono richiedere una quantità di memoria variabile, altre richiedono una quantità di memoria fissa e contigua.
* **Overhead di memoria**, ogni struttura dati ha un certo overhead di memoria associato, che include informazioni di gestione della memoria, altre strutture o informazioni di supporto.
* **Efficienza di accesso alla memoria**, strutture dati contigue offrono maggiore efficienza nell’accesso all’informazione rispetto a strutture che utilizzano l’ausilio di puntatori.
* **Facilità di implementazione e manutenzione**, alcune strutture dati sono più complesse di altre da utilizzare ed implementare.
* **Adattabilità ai requisiti del problema**, ogni problema ha requisiti unici, quindi, bisogna scegliere la struttura dati in base a:
* Le operazioni più frequentemente eseguite.
* Vincoli che possono impedire la scelta dell’utilizzo di una struttura dati.
* **Scalabilità**, scegliere una struttura dati che permetta di mantenere le performance all’aumentare del volume dei dati e che sia adattabile ai cambiamenti.

Java Collection Framework (JCF) è un insieme di classi e interfacce che implementano strutture dati comunemente riutilizzabili.

Il JCF (pacchetto java.util) fornisce

* Immagine che contiene testo, schermata, diagramma, numero

  Descrizione generata automaticamenteinterfacce che definiscono le funzionalità
* classi astratte per l'aggregazione di codice condiviso
* classi concrete che implementano funzionalità
* algoritmi (java.util.Collections)

**Array Dinamico**

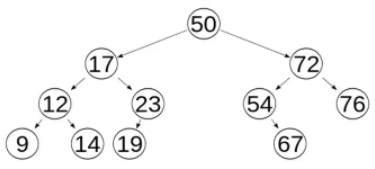
Lo svantaggio degli array regolari è che non possiamo modificarne le dimensioni nel mezzo dell'esecuzione del codice. Risolviamo questo problema usando l'idea dell'array dinamico ~O(n) in cui possiamo aumentare la dimensione dell'array dinamicamente quando ne abbiamo bisogno.

**Linked list**

Immagine che contiene testo, diagramma, linea, schermata

Descrizione generata automaticamenteUna Liked list **~O(n)** è una raccolta lineare di elementi di dati il ​​cui ordine non è dato dalla loro collocazione fisica nella memoria. Invece, ogni elemento punta al successivo. È una struttura di dati costituita da una raccolta di nodi che insieme rappresentano una sequenza. Nella sua forma più elementare, ogni nodo contiene dati e un riferimento (in altre parole, un collegamento) al nodo successivo nella sequenza. Questa struttura consente un inserimento o una rimozione efficiente di elementi da qualsiasi posizione nella sequenza durante l'iterazione. Varianti più complesse aggiungono collegamenti aggiuntivi, consentendo un inserimento o una rimozione più efficiente di nodi in posizioni arbitrarie. Uno svantaggio delle liste concatenate è che il tempo di accesso ai dati è una funzione lineare del numero di nodi per ogni lista concatenata (vale a dire, il tempo di accesso aumenta linearmente man mano che i nodi vengono aggiunti a una lista concatenata), perché i nodi sono collegati in serie; quindi, è necessario accedere a un nodo prima per accedere al nodo successivo (quindi difficile da gestire). Un accesso più rapido, come l'accesso casuale, non è fattibile. Gli array hanno una migliore località della cache rispetto alle liste concatenate.

**Binary tree**

Un albero binario di ricerca autobilanciante ~**O(log(n))** è un albero binario di ricerca basato su nodi che mantiene automaticamente la sua altezza (numero massimo di livelli sotto la radice) piccola di fronte a inserimenti ed eliminazioni di elementi arbitrari. Queste operazioni, quando progettate per un albero binario di ricerca autobilanciante, contengono misure precauzionali contro l'aumento illimitato dell'altezza dell'albero, in modo che queste strutture dati astratte ricevano l'attributo "autobilanciante".

**Tabella hash**

Immagine che contiene testo, diagramma, schermata, linea

Descrizione generata automaticamenteUna tabella hash **~O(1)** , o hash map o hash map, è una struttura dati che implementa un array associativo o un dizionario. È un tipo di dati astratto che mappa le chiavi ai valori. Una tabella hash utilizza una funzione hash per calcolare un indice, chiamato anche codice hash, in un array di bucket o slot, da cui è possibile trovare il valore desiderato. Durante la ricerca, la chiave viene sottoposta a hash e l'hash risultante indica dove è archiviato il valore corrispondente.

Immagine che contiene testo, schermata, Rettangolo, diagramma

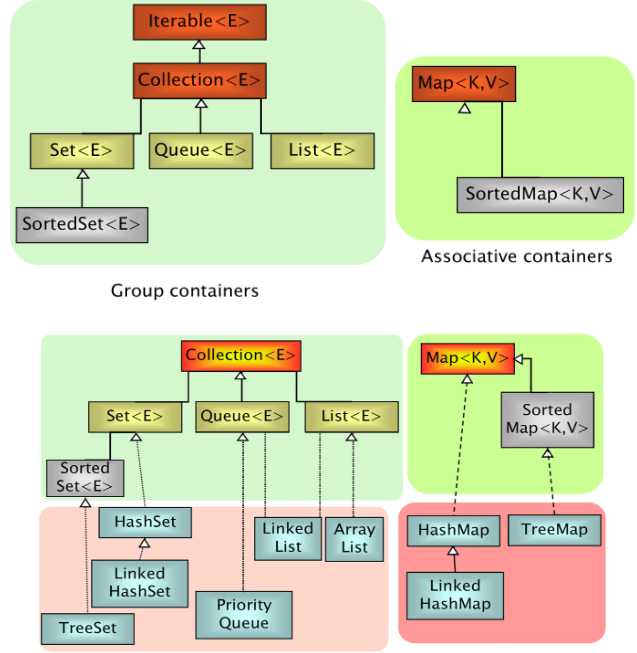
Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

### Interfacce Iterabili e Iteratore

L'interfaccia **Iterable** (java.lang.Iterable) è l'interfaccia radice del framework di raccolta Java. Iterable, letteralmente, significa che "può essere iterato". Tecnicamente, significa che può essere restituito un Iterator . Gli oggetti Iterable (oggetti che implementano l'interfaccia iterable) possono essere utilizzati all'interno di cicli for-each.

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

L' interfaccia **Iterator** estrae il comportamento di attraversamento di una raccolta in un oggetto separato denominato iteratore.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

### Collection interface

L'interfaccia root nella gerarchia della collection. Una collectiom rappresenta un gruppo di oggetti, conosciuti come i suoi elementi. Alcune collection consentono elementi duplicati, altre no. Alcuni sono ordinati e altri non ordinati. Il JDK non fornisce alcuna implementazione diretta di questa interfaccia: fornisce implementazioni di sottointerfacce più specifiche come Set e List. Questa interfaccia viene generalmente utilizzata per passare le raccolte e manipolarle laddove si desidera la massima generalità.

Principali metodi di collecton

* int size()
* boolean isEmpty()
* boolean contains(Object element)
* boolean containsAll(Collection c)
* boolean add(Object element)
* boolean addAll(Collection c)
* boolean remove(Object element)
* boolean removeAll(Collection c)
* void clear()
* Immagine che contiene testo, schermata, Carattere, linea

  Descrizione generata automaticamenteObject[] toArray()
* Iterator iterator()

**LIST INTERFACE**

* Può contenere **elementi duplicati**
* L'ordine di inserimento viene mantenuto
* L'utente può selezionare punti di inserimento arbitrari
* Gli elementi sono accessibili **tramite posizione**

Elenca i metodi principali

* Object get(int index)
* Object set(int index, Object o)
* Immagine che contiene schermata, diagramma, design

  Descrizione generata automaticamenteObject remove(int index)
* void add(int index, Object o)
* boolean addAll(int index, Collection c)
* int indexOf(Object o)
* int lastIndexOf(Object o)
* List subList(int fromIndex, int toIndex)

**!** Il metodo *List.of* in Java è un metodo statico all'interno della classe List che permette di creare una lista immutabile (non modificabile una volta creata) in modo conveniente e conciso.

Esistono due implementazioni List di uso generale:

* **ArrayList** implementa List
  + get(indice) -> Tempo costante
  + add(index, obj) -> Tempo lineare
* **LinkedList** implementa List, Deque
  + get(indice) -> Tempo lineare
  + add(index, obj) -> Tempo lineare (ma più leggero)

La maggior parte delle volte probabilmente utilizzerai ArrayList, che offre accesso posizionale in tempo costante ed è semplicemente veloce. Non è necessario allocare un oggetto nodo per ciascun elemento nell'elenco e può trarre vantaggio da System.arraycopy quando deve spostare più elementi contemporaneamente.

Se aggiungi frequentemente elementi all'inizio dell'Elenco o esegui un'iterazione sull'Elenco per eliminare elementi dal suo interno, dovresti prendere in considerazione l'utilizzo di LinkedList. Queste operazioni richiedono tempo costante in una LinkedList e tempo lineare in un ArrayList. Ma paghi un caro prezzo in termini di prestazioni. L'accesso posizionale richiede tempo lineare in una LinkedList e tempo costante in un ArrayList. Inoltre, il fattore costante per LinkedList è molto peggiore. Se pensi di voler utilizzare una LinkedList, misura le prestazioni della tua applicazione sia con LinkedList che con ArrayList prima di fare la tua scelta. **ArrayList è solitamente più veloce.**

****

**SET INTERFACE**

Una collection che non contiene elementi duplicati. Più formalmente, gli insiemi non contengono coppie di elementi e1 ed e2 tali che *e1*.*equals*(*e2*) e al massimo un elemento nullo. Come suggerisce il nome, questa interfaccia modella l'astrazione dell'insieme matematico.

Metodi principali di set: L' interfaccia Set non aggiunge alcun metodo all'interfaccia Collection.

Implementazioni di set:

* **HashSet** implementa Set
  + Tabelle hash come struttura dati interna (veloce!)
  + Ordine di inserimento non conservato
* **LinkedHashSet** estende HashSet
  + Ordine di inserimento conservato
* **TreeSet** implementa SortedSet (un'estensione di Set)
  + Alberi RB come struttura dati interna (fornire l'ordinamento)
  + Ordinamento interno definibile dall'utente TreeSet (Comparatore c)
  + Lento rispetto alle implementazioni basate su hash

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

**Ordinamento interno di TreeSet**

A seconda del costruttore utilizzato, i TreeSet possono utilizzare ordinamenti diversi:

* InsiemeAlberi()
  + Ordinamento ascendente naturale
  + Gli elementi devono implementare l' interfaccia comparabile
* TreeSet(Comparatore c)
* L'ordinamento è definito dal comparatore c

**Confronto tra HashSet e TreeSet**

|  |  |
| --- | --- |
| HASHSET | TREESET |
| memorizza gli oggetti in ordine casuale | applica l'ordine naturale degli elementi. |
| può memorizzare oggetti nulli | non lo consente |
| garantisce prestazioni in tempo costante per la maggior parte delle operazioni come add(), remove() e contains() | prestazioni in tempo log(n) |
| Ha meno funzionalità | più ricco di funzionalità, implementando metodi aggiuntivi come: first(), last(), ceiling(), lower(), … |

**INTERFACE QUEUE/DEQUE**

**Queue** Una raccolta progettata per contenere elementi prima dell'elaborazione. Oltre alle operazioni di raccolta di base, le code forniscono operazioni aggiuntive di inserimento, estrazione e ispezione. Ciascuno di questi metodi esiste in due forme: uno lancia un'eccezione se l'operazione fallisce, l'altro restituisce un valore speciale (null o false, a seconda dell'operazione).

**Deque** Una raccolta che supporta l'inserimento e la rimozione di elementi su entrambe le estremità. Il nome deque è l'abbreviazione di "coda a doppia estremità" e viene solitamente pronunciato "mazzo". La maggior parte delle implementazioni di Deque non pongono limiti fissi al numero di elementi che possono contenere, ma questa interfaccia supporta la rimozione dalla coda con capacità limitata così come quelle senza limite di dimensione fissa.

Implementazioni di coda/deque

* **LinkedList** implementa List, Queue, Deque
  + Nessuna limitazione di capacità.
  + Gli elementi non sono ordinati.
* **ArrayDeque** implementa Queue, Deque
  + Nessuna limitazione di capacità.
  + Gli elementi non sono ordinati.
* **PriorityQueue** implementa Queue
  + Nessuna limitazione di capacità.
  + Gli elementi sono ordinati.
* **LinkedBlockingDeque** implementa Queue, Deque
  + Capacità limitata.
  + Gli elementi non sono ordinati.
* **ArrayBlockingQueue** implementa Queue
  + Capacità limitata.
  + Gli elementi non sono ordinati.
* **ConcurrentLinkedDeque** implementa Queue, Deque
  + Nessuna limitazione di capacità.
  + Gli elementi non sono ordinati.

**MAP INTERFACE**

Un oggetto che associa le chiavi ai valori. Una mappa non può contenere chiavi duplicate; ciascuna chiave può essere associata al massimo a un valore. L'interfaccia Mappa fornisce tre visualizzazioni di raccolta, che consentono di visualizzare il contenuto di una mappa come un insieme di chiavi, una raccolta di valori o un insieme di mappature di valori-chiave. L'ordine di una mappa è definito come l'ordine in cui gli iteratori nelle visualizzazioni di raccolta della mappa restituiscono i propri elementi. Alcune implementazioni di mappe, come la classe TreeMap, forniscono garanzie specifiche riguardo al loro ordine; altri, come la classe HashMap, no.

Metodi principali di map

* Object put(Object key, Object value)
* Object get(Object key)
* Object remove(Object key)
* boolean containsKey(Object key)
* boolean containsValue(Object value)
* public Set keySet()
* public Collection values()
* public Set entrySet()
* int size()
* boolean isEmpty()
* void clear()

Implementazioni di map

* **HashMap** implementa Map
* Tabelle hash come struttura dati interna (veloce!)
* Ordine di inserimento non conservato
* **LinkedHashMap** estende HashMap
  + Ordine di inserimento conservato
* **TreeMap** implementa SortedMap
  + Alberi RB come struttura dati interna
  + Ordinamento interno definibile dall'utente
  + Lento rispetto alle implementazioni basate su hash



### MANIPOLAZIONE DELLE COLLECTIONS

*UTILIZZO DEGLI ITERATORI*

**!** Non è sicuro modificare (aggiungendo o rimuovendo elementi) una raccolta mentre si esegue l'iterazione su di essa.

Interface **Iterator** fornisce un mezzo trasparente per scorrere tutti gli elementi di una Collection (solo forward ) e rimuovere elementi

* + boolean hasNext()
  + Object next()
  + void remove()

Interface **ListIterator** fornisce un mezzo trasparente per scorrere tutti gli elementi di una Collection (avanti e indietro ) e rimuovere e aggiungere elementi

* boolean hasNext()
* boolean hasPrevious()
* object next()
* object previous()
* void add()
* void set()
* void remove()
* int nextIndex()
* int previousIndex()

*UTILIZZANDO JAVA.UTIL.COLLECTIONS:*

ALTER-EGO DI JAVA.UTIL.ARRAYS PER RACCOLTE

Questa classe contiene vari metodi per manipolare gli array come l'ordinamento, la ricerca, il riempimento, la stampa o la visualizzazione come un array.

Questa classe implementa metodi come:

* sort() unisci l'implementazione dell'ordinamento, n log(n)
* BinarySearch() richiede la raccolta ordinata
* shuffle() mescola la raccolta
* reverse() richiede la raccolta ordinata
* rotate() ruota gli elementi di una determinata distanza
* min(), max() in una raccolta

*ORDINAMENTO DELLE COLLECTIONS*

Per ordinare raccolte di oggetti, è necessario implementare l'interfaccia **Comparable** per rendere gli oggetti confrontabili tra loro.

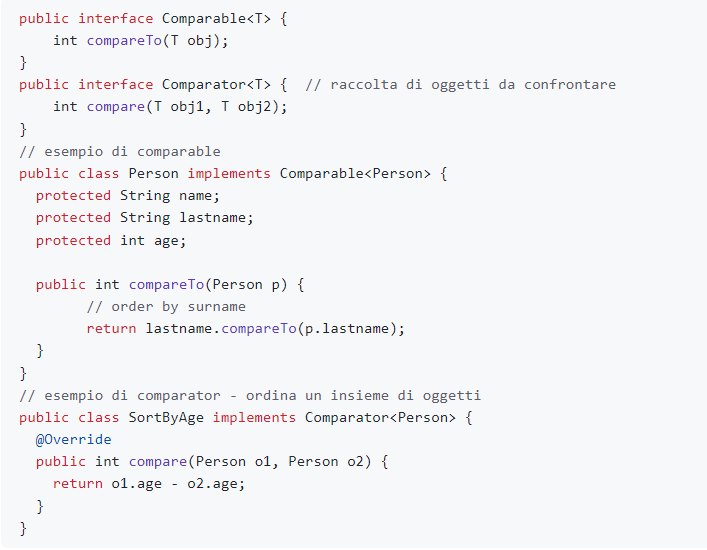
L'interfaccia comparabile è implementata di **default** nei tipi comuni nei pacchetti java.lang e java.util

Una raccolta di T può essere ordinata se T implementa Comparable. Il metodo *compareTo*() confronta l'oggetto con l'oggetto passato come parametro. Il valore restituito deve essere:

< 0 se questo oggetto precede obj

== 0 se questo oggetto ha la stessa posizione di obj

> 0 se questo oggetto segue obj



# E5.1 JAVA GENERICS

***-Generics***

***DOMANDA***: “*Come converto una classe o una funzione specializzata su un determinato tipo su un altro tipo? Come faccio a riutilizzare del codice per più tipi diversi?*”

Esistono situazioni in cui classi e metodi non dipendono strettamente dal tipo, quindi, l’algoritmo che li gestisce è esattamente identico.

Questo problema in Java può essere trattato in più modi:

* Metodi ***SBAGLIATI***:

|  |  |
| --- | --- |
| **METODO** | **Perché è sbagliato?** |
| Copiare il metodo o la classe per ogni tipo che ammette. | Violazione del principio **DRY** (Don’t repeat yourself). |
| Utilizzare la classe Object. | Può provocare **errori di runtime**. |

**Come funziona Object come tipo generico?**

Object è la classe padre, tutte le classi derivano da lui; quindi, è una valida scelta come tipo generico.

Tuttavia, quando assegneremo il risultato ad un oggetto più specifico, il compilatore si lamenterà, non sa che tipo di dato viene restituito quindi richiede un cast esplicito. Non si può garantire che l’oggetto restituito sia del tipo indicato e ciò potrebbe causare un **errore di runtime**.

ESEMPIO: Object = [ String, Double, String ] DOUBLE NON AMMESSO

* Metodo ***GIUSTO***:
* Utilizzo i **Generics**.

I **Generics** permettono di risolvere le problematiche runtime e di copia aumentando la chiarezza, la sicurezza ma anche la complessità.

**Come risolve il problema della copia?**

Con i Generics il programmatore stabilisce il tipo dei dati e il compilatore garantirà che questo vincolo sia rispettato. Questo comporta che l’errore a runtime viene convertito in un errore di compilazione.

***Sintassi ed esempi***:

***Generics alla compilazione***:

I tipi generici sono stati introdotti in Java nel suo secondo decennio, quindi, per garantire la compatibilità con le versioni precedenti, i tipi generici non vengono compilati ma cancellati (**erasure**). Cioè, vengono sostituiti con i normali tipi di Java più generici che rispettino il vincolo e a tempo di esecuzione viene sostituito il tipo generico con un cast.

In Java le strutture dati sono implementate nella JCF (**Java Collection Framework**), cioè, un insieme di classi e interfacce che implementano le strutture dati comunemente utilizzate. Tra le strutture dati, le più utilizzate sono:

* Hash table, è una struttura dati che implementa un array o un dizionario. È un tipo di dato astratto che associa un delle chiavi a dei valori. Utilizza una funzione hash per calcolare un indice chiamato anche codice hash. Che indica dove è memorizzato l’elemento.

Fondamentale per le collection è l’interfaccia radice del framework Iterable. Cioè, permette di restituire un oggetto Iterable che può essere utilizzato in un ciclo for-each.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

***-Programmazione funzionale***

La programmazione funzionale è un paradigma di programmazione

***-Exceptions***

Le **eccezioni** sono errori catturati durante l’esecuzione di un programma e che ne interrompono la normale esecuzione. Non sono funzionali per l’utente finale che utilizza il programma ma sono essenziali per il programmatore per avere maggiore comprensione degli errori.

Le eccezioni sono oggetti di una classe speciale organizzate, come tutte le classi, in gerarchie:

Immagine che contiene testo, Carattere, ricevuta, bianco

Descrizione generata automaticamente

Dove:

* **Throwable**, è la classe padre, fornisce una struttura comune per rappresentare le anomalie che si verificano durante l’esecuzione del programma.
* **Error**, rappresenta errori gravi ed irrecuperabili che possono riguardare la JVM o l’ambiente operativo sottostante.
* **Exception**, rappresenta situazioni anomale relative ad esempio a:
* *Errori di logica*.
* *Input non valido*.
* *Risorse non disponibili*.

|  |
| --- |
| ! **Possibile domanda dell’orale: Quali sono le eccezioni non controllate?** |
| Quelle più comuni |

Si suddivide a sua volta in:

* **Unchecked** **exceptions** (*eccezioni non controllate*), sono eccezioni che non richiedono una gestione esplicita del programmatore.

Immagine che contiene testo, Carattere, linea, schermata

Descrizione generata automaticamente

* **Checked** **exceptions** (*eccezioni controllate*), un meccanismo attraverso il quale il compilatore impone al programmatore di gestire le eccezioni che si possono verificare durante l’esecuzione del programma.

Se un metodo lancia un’eccezione controllata, deve essere contrassegnata nella dichiarazione dalla parola chiave **throws** altrimenti il programma non verrà compilato.

|  |
| --- |
| ! **Throws** |
| Questa parola chiave viene utilizzata nella dichiarazione di un metodo per segnalare che il blocco di codice potrebbe lanciare una particolare eccezione ma **NON** la gestirà.  Quindi throws permetterà di propagare l’errore al chiamante che a sua volta avrà due scelte:   * Contenere throws nella dichiarazione e far **propagare** ancora **l’eccezione** (risalire la gerarchia delle chiamate). * **Gestire l’eccezione**. |

Modelli di delaga, completa, parziale e nulla (chiede sempre a cosa serve la parziale)

Sia le checked che le unchecked possono essere intercettate e controllate manualmente.

Quindi, quando parliamo di eccezioni controllate, fondamentale è il concetto di **delega**, che può essere:

* **Nulla**, un metodo che potrebbe generare un’eccezione sceglie di non gestirlo e non propagarlo.

A volte l’eccezione può essere ignorata perché non rilevante e non genera conseguenze.

* **Parziale**, un metodo gestisce l’eccezione in parte.

Modificare o aggiungere informazioni sull’eccezione prima di propagarla trasformandola in una eccezione più significativa prima di propagarla.

* **Completa**, un metodo trasferisce completamente la responsabilità di gestione dell’eccezione al chiamante.

Permette ad un metodo di concentrarsi solo sulle proprie operazioni senza doversi preoccupare di gestire le eccezioni.

***-Threads***

Immagine che contiene testo, schermata, software, Pagina Web

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, software

Descrizione generata automaticamente

Buona norma non fare troppe sottoclassi//Vedere bene

Albero delle dipendenze funzionale e ad oggetti, forse

Inversione di dipendenza, slide 1