PROGRAMMAZIONE

Prof. Paolo Maresca – A.A. 2023/24

INDICE DEGLI ARGOMENTI

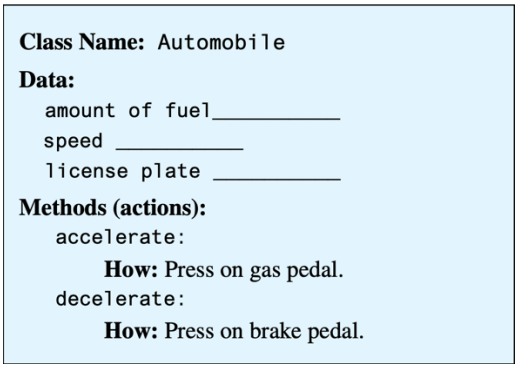
PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

ALGORITMI E STRUTTURE DATI

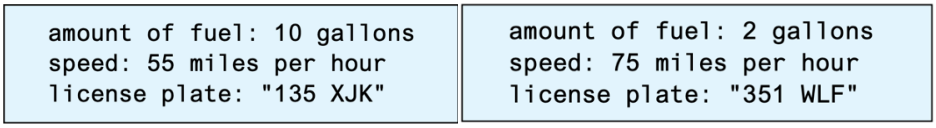
PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

LA PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

Le **definizioni di classe e di oggetto** sono in una **strettissima relazione**; **un oggetto** (**specializzazione**) **può essere la rappresentazione di un oggetto reale** (come un'automobile, una bottiglia …) **o l'astrazione di un concetto** (come un punto matematico, un colore …), mentre **la classe** (**generalizzazione**) è lo stampino, **il blueprint sul quale l'oggetto è definito**. Qui di seguito è presentata **la struttura con cui è definita una classe**:



Mentre **gli oggetti costruiti a partire da essa**:

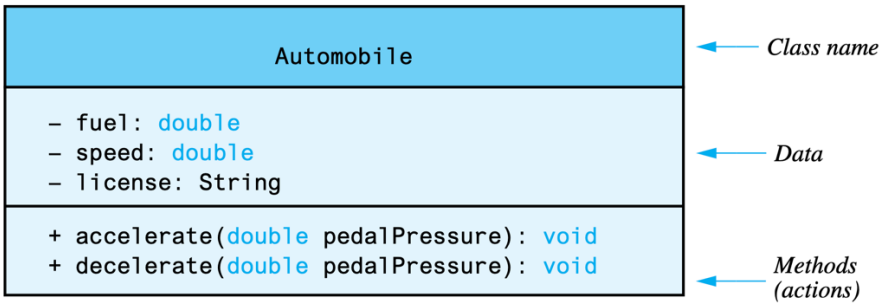


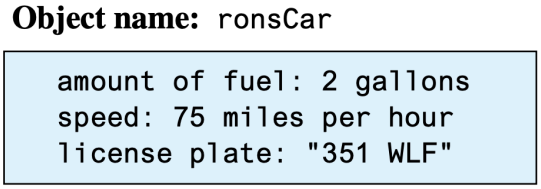
Ovviamente, **l'astrazione effettuata per definire la classe `Automobile` non è completa** (non sono presenti tutti gli attributi e tutti i comportamenti di un auto) ma è **circoscritta all'applicazione per la quale è usata**, al fine di **risparmiare tempo e risorse**. **L'astrazione è il primo dei quattro pilastri su cui si basa la Programmazione Orientata agli Oggetti**.

Ognuno di questi oggetti è **definito a partire dalla classe `Automobile`** e può essere considerato come un **oggetto `Automobile`**, composto da **diversi parametri** (come il valore di serbatoio) **ma dalle stesse strutture** (come la variabile serbatoio); sotto queste ipotesi, si può dire che **i due oggetti qui considerati sono due istanze della classe `Automobile`**, oggetti che **rispettano la definizione di classe `Automobile` ma sui quali sono stati applicati valori reali**. Per questo motivo non si può dire che alla classe sono associati i valori reali di serbatoio o di velocità, ma bensì le variabili alle quali sono associati i valori reali nell'istanza di una classe. Si può facilmente intuire che **sono gli oggetti, e non le classi, a partecipare attivamente all'elaborazione in un programma**.

Per **definire una classe** è necessaria **un'astrazione sul tipo di oggetto che si vuole creare**, individuando **due tipologie di informazioni intrinseche nell'oggetto stesso**: **attributi** e **azioni**; un **attributo** di una classe è definito come **una variabile che descrive una caratteristica dell'oggetto** (come la velocità massima o il volume del serbatoio di una macchina) mentre i **metodi** ne **descrivono i comportamenti** (come l'accelerazione o la decelerazione). **Due oggetti istanza della stessa classe condivideranno sempre gli stessi attributi e gli stessi metodi**, sebbene possano **agire su diversi set di dati**.

Per descrivere o rappresentare una classe ed i suoi oggetti al di fuori del codice si può usare il **linguaggio UML** (Universal Modeling Language) con il quale **ad ogni classe è associato un box diviso in `NomeClasse`, `Attributi` e `Metodi`**:





Nel codice, **ogni oggetto**, così come la relativa **classe**, **è determinato univocamente attraverso il suo identificativo** (`Automobile` e `ronsCar` in questo caso); **per quanto riguarda la classe**, questo **rappresenterà il "tipo di dato" che verrà definito**, mentre **per l'oggetto la variabile di tipo `Automobile`**.

Nel linguaggio di programmazione Java la programmazione ad oggetti viene implementata attraverso un meccanismo di eterarchizzazione e compilazione separata. **Ogni classe** viene definita e scritta **in un file `.java` separato dal programma principale**, con la possibilità di poter essere **compilato ancor prima che sia utilizzato**; successivamente, **quando si vuole implementare un oggetto di tipo classe nel programma principale**, anch'esso scritto in un file `.java` separato, **si potrà definire una nuova variabile di tipo classe ed utilizzare tutti i suoi attributi e metodi**. Quando si andrà a compilare il programma principale **non si dovrà ricompilare il file classe se questo è stato già compilato**. La comodità di questo approccio orientato agli oggetti risiede nel fatto che **non è più il programma principale ad occuparsi di eseguire tutte le operazioni e le chiamate ma si occupa solo di gestire il flusso di dati** che, dal punto di vista operazionale, è **decentralizzato nelle varie classi in esso menzionate**.

La **sintassi Java** con cui è possibile definire un oggetto a partire da una classe è la seguente:

```
ClassName objectName = new ClassName();
```

Viene **dichiarata una variabile di tipo `ClassName`**, alla quale è **associata l'inizializzazione `new ClassName()` che assegna in memoria l'intera struttura definita nella classe** (attributi e metodi); la variabile **`objectName` non funge da variabile contenente l'intera struttura** ma da **puntatore che punta alla locazione di memoria in cui è stata inserita la struttura della classe** (da qui l'utilizzo di `new`). Il motivo dietro all'uso dei puntatori risiede nella **rapidità con cui essi lavorano su strutture anche complesse** rispetto a variabili contenenti le strutture stesse.

Dal momento in cui la variabile che rappresenta l'oggetto usata a livello di codice è un puntatore, **l'accesso ai vari attributi/metodi dell'oggetto stesso viene effettuato tramite la sintassi a punti**:

```
objectName.attribute;
```

```
objectName.method();
```

Usando il puntatore `objectName` al quale sono stati aggiunti sufficienti indirizzi per raggiungere le locazioni desiderate **sulla base della struttura definita nel file `className.java`**.

Il metodo creato all'istanza dell'oggetto è detto **metodo di istanza** ed è una **funzione chiamata da o su un oggetto capace di alterare lo stato dell'oggetto stesso** (esistono metodi che alterano e metodi che osservano lo stato, sono due tipi di metodi diversi). **L'oggetto che è in grado di chiamare il metodo di istanza** (cioè l'oggetto nel quale è stato definito il metodo in accordo con la classe da cui deriva) è detto **oggetto chiamante** ed **instaura una relazione biunivoca con il metodo stesso**, per la quale **è possibile sia dire che l'oggetto effettua la chiamata sia che il metodo riceve l'invocazione dall'oggetto**.

Una stessa variabile attributo di una classe, richiamata in contesti diversi prevede sintassi diverse; ad esempio, **in un programma che usa una classe attraverso un oggetto** prevede la sintassi puntata **`objectName.attribute`**, mentre **all'interno della classe stessa può essere richiamata con il semplice identificativo `attribute`**. In determinate situazioni questa pratica può portare a **situazioni spiacevoli**; ad esempio, **un metodo della classe in questione può richiamare un attributo e assegnargli un valore dato**:

```
public class ClassName{
     public int value;
     public void setValue(int value){
          value = value;
     }
}
```

In fase di compilazione del file `className.java`, **il compilatore si rende conto che l'assegnazione `value = value` avviene su due elementi diversi**, una variabile attributo e un parametro formale, e **non solleva interruzioni o errori**; tuttavia, **in un eventuale programma principale il compilatore non riesce ad effettuare questa distinzione**, dal momento in cui **ciò che vede è la struttura nello stack frame e non la definizione di classe** presente nel relativo file `.java`, quindi **interpreta quell'istruzione come una variabile alla quale è assegnato il suo valore, non restituendo alcun cambiamento**.

Per ovviare a questo errore **si può utilizzare anche all'interno della classe la nomenclatura puntata simile a quella usata con gli oggetti**, `objectName.value`, ma **sfruttando l'identificativo `this`**, che **non fa riferimento ad un oggetto istanziato ma a quello che verrà eventualmente istanziato in futuro** (quando la classe si fa oggetto il `this` è sostituito virtualmente dal nome della classe). La sintassi diventa così:

```
public class ClassName{
     public int value;
     public void setValue(int value){
          this.value = value;
     }
}
```

E **nel `main` verrà a tutti gli effetti eseguita l'assegnazione correttamente**. In realtà **il `this` nella classe è un puntatore** (perciò permette di fare autoassegnamento) **e può essere utilizzato in qualsiasi situazione che coinvolge l'utilizzo degli attributi**, in modo da evitare alcun tipo di fraintendimento per il compilatore; tuttavia, **l'uso eccessivo del `this` è scoraggiato e limitato al caso in cui ci possa essere un tipo di aliasing tra le variabili usate**.

La necessità del `this` sorge dal fatto che, avendo la Programmazione Orientata ad Oggetti una **struttura non gerarchica**, **nelle classi non avviene la sovrascrittura** che un linguaggio di programmazione procedurale provvederebbe ad effettuare e **l'aliasing non è automaticamente evitato**.

Ovviamente, il non `this` vale unicamente per la chiamata di attributi ma può essere usata anche per chiamare metodi della stessa classe nel corpo di un altro metodo.

INFORMATION HIDING E INCAPSULAZIONE

Un programmatore che usa un metodo di una classe non ha la necessità di conoscere come il metodo viene eseguito ma solo cosa va messo in input e cosa va messo in output; conoscere il come può distrarre il programmatore da quello che deve fare perché potrebbe introdurre diversi altri parametri di cui occuparsi. Si può, quindi, dedurre che **nel design di un metodo** (ma anche più generalmente, di una classe) **è necessario separare il cosa dal come e permettere ad un eventuale utilizzatore di usare solo gli strumenti e i metodi di cui si deve interessare**; un processo di questo tipo è definito **information hiding** ed è una parte della cosiddetta **incapsulazione** (**il secondo dei quattro pilastri su cui si basa la Programmazione Orientata ad Oggetti**). Un esempio di information hiding è la **guida**: **quando un autista guida un'automobile non è necessario che sappia come funziona la frizione ed il cambio ma solo che esistono e che vanno usati in determinate circostanze per far andare avanti l'auto**.

Un modo efficiente per descrivere cosa fa un metodo, senza costringere l'utilizzatore a tuffarsi nel come, è costituito dall'uso di **precondition** e **postcondition**: **le prime sono delle affermazioni che specificano le condizioni da rispettare prima che il metodo sia invocato**, impedendone l'utilizzo se non fossero rispettate, mentre **le seconde sono affermazioni che descrivono tutti gli effetti prodotti dal metodo invocato**.

Un metodo/attributo/classe può essere caratterizzato, nella sua dichiarazione, **da un modificatore di visibilità, `public` o `private`**, che **modifica il grado di accessibilità** del metodo/attributo/classe **al di fuori della classe in cui è definito**. Il modificatore **`public` indica che qualsiasi altra classe può direttamente accedere o modificare il metodo/attributo/classe semplicemente utilizzando il suo identificativo**; sebbene sia una pratica **comune e comoda**, l'utilizzo di modificatori pubblici **non è una pratica incoraggiata e convenzionale**. Il modificatore **`private` indica che nessuna classe al di fuori di quella in cui si sta agendo può direttamente accedere o modificare il metodo/attributo/classe**; in generale, **conviene impostare gli attributi di una classe come privati e accedergli attraverso vie secondarie**, mentre **per i metodi e le classi ci si affida alle particolari esigenze**.

Ci si chiede **perché non sia conveniente implementare attributi o (determinati) metodi pubblicamente**; si faccia l'esempio banale di una classe `Rettangolo`, dotata di attributi `altezza` e `larghezza` e di metodi `stampaArea` e `determinaArea`:

```
public class Rettangolo{
     public int altezza;
     public int larghezza;
     public int determinaArea(){
          return altezza*larghezza;
     }
     public int stampaArea(){
          System.out.println(determinaArea());
     }
}
```

Nel programma principale **si istanzia l'oggetto `quadrato` e si impostano i parametri `altezza = larghezza = 3`, con un area di `9`**. Se i modificatori fossero pubblici **si potrebbe, da qualsiasi parte del programma in cui è istanziato l'oggetto, modificare gli attributi o i metodi in modo che `larghezza` e `altezza` non siano più uguali o che l'area non si calcoli più come `altezza*larghezza`**, alterando e **corrompendo l'essenza dell'oggetto** o i suoi dati. Conviene impostare **`altezza`, `larghezza` e `determinaArea()` come `private`, in modo che solo all'interno della classe stessa siano accessibili e modificabili**. La nuova classe si configura come:

```
public class Rettangolo{
     private int altezza;
     private int larghezza;
     private int determinaArea(){
          return altezza*larghezza;
     }
     public int stampaArea(){
          System.out.println(determinaArea());
     }
}
```

Con questa modifica **ci si chiede si ci siano dei modi con cui, all'esterno della classe, si possa accedere agli attributi o ai metodi in questione**. La risposta risiede nei **metodi modificatori e accessori (pubblici)**, o **setter e getter**, metodi che **sfruttano la proprietà per cui gli elementi di una classe sono sempre accessibili da elementi della stessa classe**; un **setter di un attributo** è un **metodo pubblico che prende in ingresso un valore dello stesso tipo dell'attributo privato che si vuole modificare e lo associa a questo**, mentre il **getter di un attributo** è un **metodo pubblico che ritorna direttamente l'attributo al quale si vuole accedere**. Ovviamente **setter e getter sono usati al di fuori della classe in cui sono definiti per poter instaurare un ponte tra gli elementi privati e l'ambiente esterno** alla classe in cui sono definiti; ma allora **il problema non è risolto**, dal momento in cui si può accedere e modificare tranquillamente gli attributi privati? **No**, perché nei setter e nei getter **possono essere inserite delle strutture di selezione che indirizzano eventuali input o operazioni non ammissibili lontano dall'attributo in questione**. Nella loro forma primitiva, i setter e i getter sono definiti come segue:

```
public void setAttribute (type value){
     attribute = value;
}
```

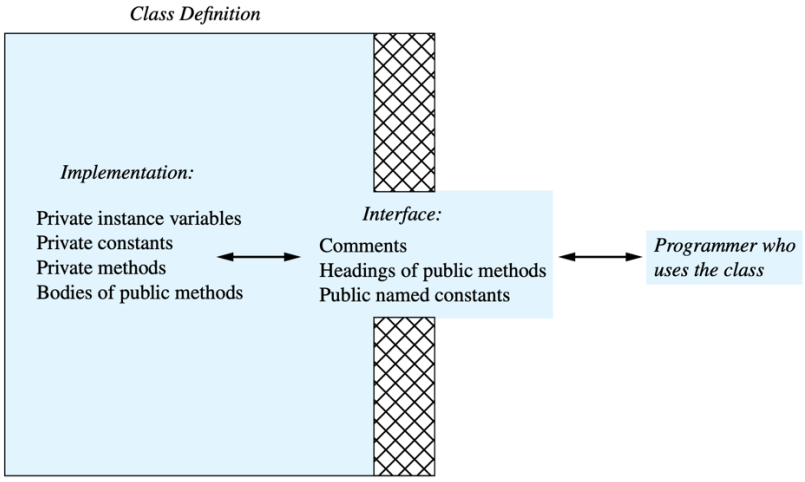
```
public type getAttribute (){
     return attribute;
}
```

La procedura di information hiding non è altro che una parte di uno dei quattro pilastri della Programmazione Orientata agli Oggetto: l'incapsulazione; **l'incapsulazione è il processo per il quale le informazioni non necessarie all'uso di un oggetto sono nascoste**. Durante la guida di un automobile non è necessario sapere il numero di cilindri o i litri di carburante iniettato ogni secondo durante l'accelerazione per poter far funzionare l'oggetto; questi dettagli sono essenziali per comprendere il funzionamento dell'auto ma non per guidarla. Allo stesso modo, **per usare un software non è necessario comprendere a fondo il meccanismo che lo muove alla base ma solo conoscere il modo di usare quello strumento**. **L'incapsulazione è il secondo dei quattro pilastri su cui si basa la Programmazione Orientata agli Oggetti**.

Affinché l'incapsulazione sia effettuata con successo, **la definizione di una classe deve essere tale da permettere il suo utilizzo senza necessità di andare a vedere come essa è composta**; ciò divide la definizione in due blocchi:

- **Interfaccia della classe**, composta da titoli per metodi e attributi (costanti) pubblici insieme a commenti che ne specificano le precondizioni e le postcondizioni, quindi il funzionamento generale;
- **Implementazione della classe**, composta da tutti gli elementi privati della classe e da tutte le definizioni di metodi (pubblici e privati).

La differenziazione di interfaccia e implementazione non avviene nel codice Java, dove i due sono **omogeneamente mischiati**, ma **nella concettualizzazione**: **quando si va a prototipare la classe**, per ottenere una buona incapsulazione, **si va a distinguere concettualmente l'interfaccia dall'implementazione attraverso uno schema simile a quello seguente**, cioè con un muro che separa l'utilizzatore dall'implementazione attraverso l'interfaccia.



Sono indicati di seguito dei consigli per una buona incapsulazione:

- Posizionare commenti prima della definizione della classe che descrivono il modo in cui il programmatore deve pensare i metodi e gli attributi della stessa;
- Dichiarare privatamente tutti gli attributi della classe;
- Implementare getters e setters (e simili funzioni per le operazioni elementari) su ogni dato privato della classe;

- Posizionare commenti prima di ogni metodo che ne specifichino le precondizioni e le postcondizioni;
- Posizionare commenti nella classe per eventuali dettagli di implementazione;

È importante annotare che **l'interfaccia della classe deve essere invariante rispetto ad eventuali cambiamenti nell'implementazione**; cioè, **se si cambia qualcosa nell'implementazione, qualsiasi altro programma che usa la classe in questione deve continuare a funzionare correttamente**.

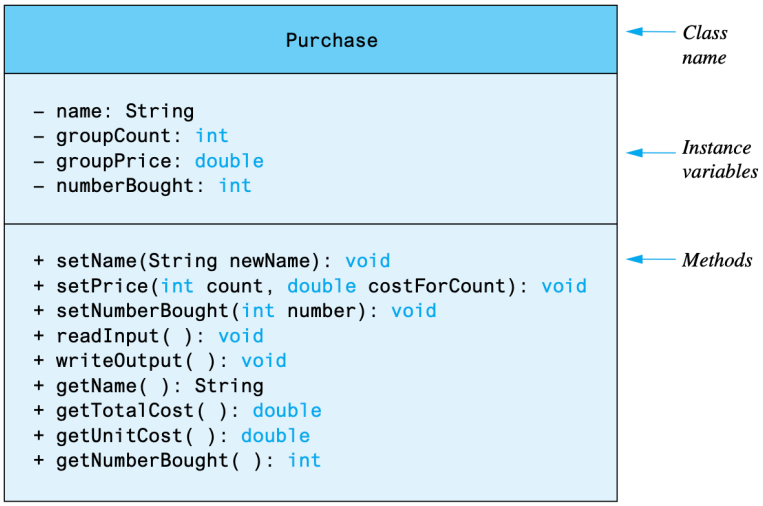
DOCUMENTAZIONE, PROTOTIPIZZAZIONE E TESTING IN JAVA

Quando **si inizializza un nuovo progetto Java** (nei principali sistemi) **viene inserito un file `javadoc` che andrà automaticamente a generare la documentazione per le interfacce delle classi del progetto**. Questa documentazione è **essenziale agli utilizzatori per comprendere più facilmente il funzionamento o le varie funzionalità della classe**, evitando di dover entrare ad osservare il codice (che spesso può confondere). Per poter ottenere una documentazione nel modo più semplice possibile, **`javadoc` converte i commenti inseriti tra `/***/` in un file HTML che verrà successivamente visualizzato in un Web browser o HTML viewer**.

Per prototipizzare una classe si usa convenzionalmente il sistema UML, Unified Modeling Language, che permette di ideal**izzare e concepire la struttura di una classe attraverso la suddivisione di una tabella in tre righe**:

- **Nome**, che rappresenta non solo il nome della classe ma anche il nome del puntatore ad oggetto;
- **Attributi**, per i quali è specificato nome e tipo;
- **Metodi**, per i quali è specificato nome, parametri formali e tipo.

Ogni attributo e ogni metodo è introdotto nella tabella da un segno – o +, a simboleggiare la **visibilità** (rispettivamente **privato** e **pubblico**).



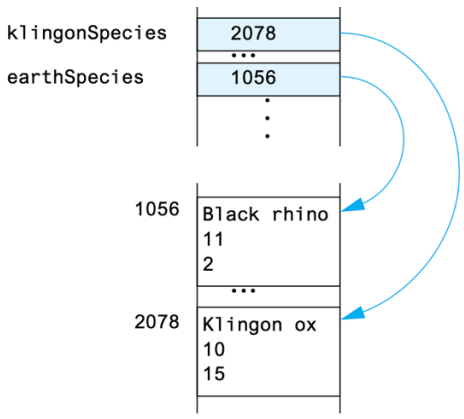
La tabella UML comunica molto di più riguardo l'interfaccia che l'implementazione, dal momento in cui è utilizzata dal programmatore per chiarire preliminarmente quali sono i dati su cui la classe dovrà lavorare e quali sono, in linea di principio, i modi con cui questi interagiscono.

Fino ad ora **non è stato indicato un vero e proprio metodo per il testing**, è stato eseguito un programma, vi è stato **inserito un input e si sono osservati gli output**; **questo metodo amatoriale è utile solo per programmi di dimensione ridotta**, sebbene presenti problemi anche sotto queste ipotesi. **Una strategia migliore per il testing dei programmi prende il nome di Unit Testing e prevede che il programmatore testi la correttezza di unità individuali del codice**; la collezione di unità di test prende il nome di **test suite**. **Ogni test è automaticamente generato in modo da non richiedere alcun input umano**, permettendo un **testing rapido e frequente** così da verificare la correttezza del programma **anche dopo qualche cambiamento**; il processo di testing ripetuto è chiamato **regression testing**.

Il fatto che il testing dia esito positivo non significa automaticamente che il programma è privo di errori, dal momento in cui **le combinazioni di input testate non saranno mai tutte quelle possibili ed una non testata potrebbe condurre ad un errore**. A tal proposito, **è importante costruire il software in modo che non crashi quando è sollevato un errore**, altrimenti operazioni essenziali potrebbero non essere più eseguite e ci potrebbero essere **conseguenze indesiderate o dannose** (come, ad esempio, lo spegnimento di un computer di bordo di un astronave nel bel mezzo dello spazio); quest'ultimo tipo di test viene detto **negative test**.

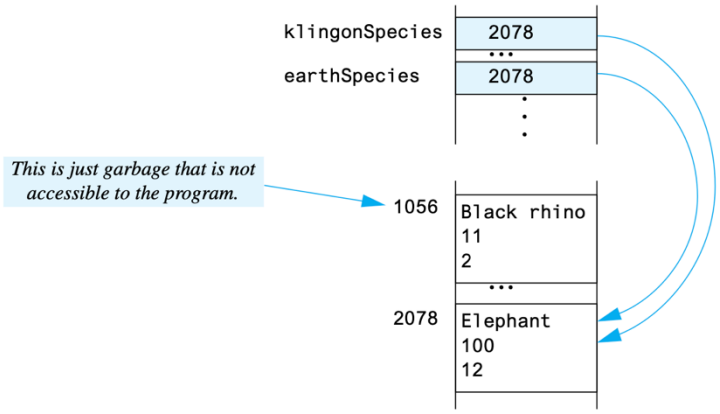
OGGETTI E RIFERIMENTI

Come è stato precedentemente introdotto, **una variabile di tipo classe denota gli oggetti e i dati in maniera differente rispetto a come fa una variabile di tipo primitivo**; entrambe allocano spazio in memoria ma, sebbene **per quella di tipo primitivo vi è inserito il valore assunto**, **per quella di tipo classe il discorso è diverso**. **L'oggetto non è allocato nello spazio di memoria che la variabile di tipo oggetto occupa ma in un area di memoria dinamica a cui punta la variabile in questione**; l'indirizzo di memoria in cui è contenuto l'oggetto (quindi il contenuto della variabile di tipo oggetto) è detto **riferimento dell'oggetto** (motivo per cui il tipo classe è detto anche tipo riferimento).



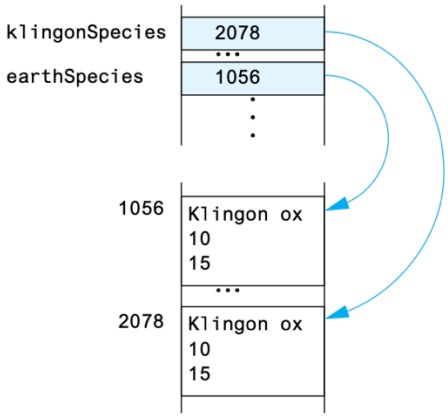
Il motivo per cui è stata necessaria questa distinzione riguarda principalmente **le dimensioni dei due tipi di dati**: **le variabili di tipo primitivo allocano sempre lo stesso spazio perché i dati di tipo primitivo saranno sempre descritti dallo stesso numero di bit**, mentre **le classi non definiscono una dimensione a priori**, due classi possono avere tranquillamente dimensioni molto diverse tra di loro. In merito a ciò, **si evidenzia la necessità di creare variabili sempre dello stesso tipo**, affinché sia finito e chiaro a priori lo spazio di memoria da allocare e quello usato; **per le variabili di tipo primitivo non ci sono problemi**, **per quelle di tipo classe risulta comodo l'uso dei puntatori**, che avranno **sempre la stessa dimensione** (la dimensione di un indirizzo) **e non la dimensione non predicibile di una classe**.

Le conseguenze di questo diverso comportamento sono due: **in primis non è possibile stampare o richiamare automaticamente un dato di un oggetto a partire dal suo riferimento**, sarà sempre necessario **spostarsi lungo la memoria attraverso il puntatore di riferimento** (e quindi usare la dotted notation), e poi **non è possibile eseguire agilmente un'operazione di assegnazione** (=) **tra due oggetti**, perché **saranno sempre uguagliati i puntatori** (cosa sintatticamente corretta ma pericolosa perché **si possono perdere i riferimenti**, e quindi i collegamenti alle strutture in memoria, conducendo ad un **memory leak**).



Con le variabili di tipo primitivo questi due problemi non si presentano e il motivo risiede sempre nel come sono gestite quelle variabili in memoria.

In maniera del tutto analoga all'operatore =, a**nche l'operazione booleana == non è utile quando si lavora con gli oggetti** (per gli stessi motivi); infatti, **uguagliando due variabili di tipo classe si stanno uguagliando i riferimenti**, che non saranno mai uguali, e **l'operazione sarà sempre `false`**. Per eseguire un'operazione di questo tipo **non ci si può affidare a strumenti offerti dal linguaggio**, dato che **due classi possono definire relazioni di equivalenza diverse**; è necessario, all'interno della classe, implementare un me**todo booleano `equals` che verifichi la relazione di equivalenza tra due oggetti per il singolo dominio applicativo della classe** (due classi diverse possono avere metodi `equals` diversi), ovvero **verifica se due oggetti in aree di memoria diverse sono uguali**. Ad esempio, sia considerata una classe che descrive una specie del regno animale; due specie saranno uguali se condivideranno il nome, la popolazione e il tasso di crescita:



Si noti che le variabili di tipo classe (cioè i riferimenti) sono diverse nonostante i due oggetti associati siano uguali. Per una classe automobile, due oggetti saranno uguali se condivideranno modello e targa, individuando due metodi `equals` diversi per classi diverse.

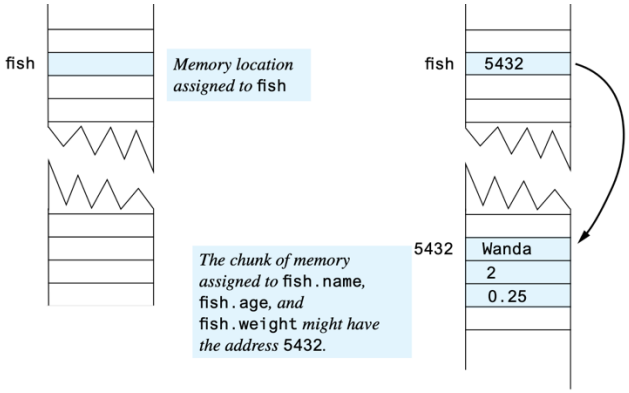
Un'ultima differenza tra le variabili di tipo primitivo e di tipo classe è nel **passaggio dei parametri**; entrambe possono essere usate come parametri per un metodo ma il modo in cui esse vengono trattate è diverso. **Per le variabili di tipo primitivo**, **il parametro formale rappresenta una "variabile locale"** che viene **inizializzata al valore del parametro effettivo al momento della chiamata**; ne consegue che **le modifiche effettuate sul parametro formale durante l'esecuzione del metodo non si propagano sul parametro effettivo** perché **parametro formale ed effettivo rappresentano due aree di memoria diverse e non collegate**. Quando **il parametro formale è una variabile di tipo classe** la situazione cambia perché **si sta passando al metodo non un dato ma l'indirizzo nel quale il dato è conservato**; segue che **la modifica del parametro formale durante l'esecuzione equivale all'accesso e alla modifica del valore** registrato nell'indirizzo di memoria indicato dal parametro (formale ed effettivo) **e la propagazione, in questo caso, avviene**. Ovviamente, **anche per i parametri di tipo classe viene fatta l'inizializzazione del parametro formale al valore contenuto nel parametro effettivo ma sarà l'assegnazione di un indirizzo**, per la quale **si avranno due variabili** (parametro formale e parametro effettivo) **in aree di memoria diverse con cui accedere alla stessa area di memoria**. In altre parole, **il passaggio di parametri di tipo classe è del tutto equivalente al passaggio di parametri di tipo primitivo per riferimento**.

IL COSTRUTTORE DI UNA CLASSE

Durante la definizione di un oggetto si invoca automaticamente un particolare metodo, il **costruttore**. Il costruttore è **il metodo che si occupa di allocare all'indirizzo di memoria specificato nel riferimento la struttura necessaria per l'esistenza dell'oggetto e**, volendo, **anche di inizializzare i valori dei vari attributi**. Fino ad ora la definizione di un nuovo oggetto è stata:

```
ClassName objectName = new ClassName();
```

Il `new` si occupa di allocare dinamicamente un'area di memoria e di inserirne l'indirizzo in `objectName`; **l'area di memoria in questione è allocata secondo la struttura della classe `ClassName` tramite il metodo `ClassName()`**, che ne è il **costruttore**. È così delineato il compito generale di un costruttore: **allocare spazio in memoria e inizializzare i valori delle variabili**; è molto **simile al funzionamento di un setter, se non per il fatto che quest'ultimo non ha la stessa abilità e autorità nel gestire la memoria di un costruttore**, che andrebbe **preferito**. L'unico limite di un costruttore rispetto ai metodi setter è **l'inizializzazione delle variabili di tipo classe**: **un setter lo può fare tranquillamente, il costruttore inizializza solo variabili di tipo primitivo**; questo svantaggio è **limitato alle singole applicazioni** ed è **controbilanciato dalla possibilità di invocare un setter nel costruttore stesso**. Infatti, **un costruttore può chiamare al suo interno altri metodi della stessa classe, a patto che siano privati** (per motivi che saranno successivamente esplicati, i metodi pubblici chiamati in un costruttore sono possibile causa di errori).



Negli schemi e nei programmi finora sviluppati non c'è stata alcuna menzione al costruttore, sebbene sia stato sempre usato per definire gli oggetti di una classe; il motivo risiede nel fatto che, se non è implementato ad hoc, **il costruttore è nascosto all'interno della struttura di Java e, pertanto, non necessita né di essere inserito nel diagramma UML**. Il costruttore in questione è detto **costruttore di default e si occupa unicamente di allocare la memoria e inizializzare a valori di default gli attributi**.

Per una stessa classe **il costruttore non è univoco**, possono esistere **infiniti tipi di costruttore ma tutti dovranno avere lo stesso nome della classe**; **ciò che distingue un costruttore da un altro è il funzionamento**, uno può inizializzare alcune variabili e uno altre, e **il modo in cui Java si rende conto del fatto che si sta usando un costruttore piuttosto che un altro è tramite la firma**: **costruttori diversi prendono in ingresso parametri diversi**. Quindi, **quando si implementano due o più costruttori**, si ha **la possibilità di usare uno di questi o quello di default semplicemente specificando più o meno parametri nella definizione dell'oggetto**.

Si osservi la definizione di un costruttore:

```
class ClassName{
     private int val1;
     private int val2;
     public ClassName(){
          System.out.prinln("Hai creato un oggetto");
          // val1 = 0
          // val2 = 0
     }
     public ClassName(int val1, int val2){
          this.val1 = val1;
          this.val2 = val2;
     }
}
```

Il primo costruttore implementato sarà il costruttore di default e lo si può riconoscere dal fatto che è privo di parametri; in realtà è **omissibile**, **dal momento in cui Java ne implementa uno a priori**, ma **nel caso in cui fosse inserito si andrebbe a sovrascrivere a quello già presente**. Il secondo costruttore non si occupa solo di allocare in memoria la struttura ma inizializza anche gli attributi sulla base dei valori che gli sono passati come parametri effettivi all'atto della definizione di un oggetto.

Finora **l'unica proprietà che rende il costruttore atipico è la sovrascrivibilità** (che sarà poi ampliata anche ad altri metodi) **ma non è l'unica**; infatti, **un altro motivo per cui il costruttore è un metodo particolare è l'impossibilità di un oggetto di chiamare il proprio costruttore dopo la sua definizione**:

```
objectName.ClassName();  // Non è possibile
```

Un'altra peculiarità dei costruttori è il fatto che **nella loro firma non presentino tipi di ritorno**: **i costruttori non ritornano (e non possono ritornare) in alcun modo dati**, neanche void (è comunque un tipo di ritorno).

Si consideri una **classe dotata di diversi costruttori**, è possibile **chiamare un costruttore all'interno di un altro semplicemente ricorrendo alla keyword `this` seguita dalla firma del costruttore desiderato**, ma a patto che questa sia **la prima istruzione eseguibile nel corpo del costruttore** in cui avviene la chiamata:

```
this(param1, param2, param3);
```

Si vuole concludere il discorso sui costruttori menzionando un **grave errore logico**, quello di **pending pointer**:

```
objectName = null;
```

L'operazione è lecita, poiché **`objectName` è un puntatore** e può essere inizializzato al puntatore nullo, ma è un **pericoloso strumento perché punta a qualcosa che non è in memoria e rischia di perdere delle informazioni che l'eventuale oggetto possiede**.

METODI STATICI E VARIABILI STATICHE

Una **variabile statica** è una variabile che **appartiene alla classe piuttosto che ad un singolo oggetto** e **nasce dalla necessità di non duplicare determinate informazioni quando si istanzia un nuovo oggetto**; ad esempio, **ogni campo da calcio condivide la stessa lunghezza** e il relativo attributo potrà essere **generalizzato alla classe `CampoCalcio` piuttosto che associato al singolo oggetto istanza di `CampoCalcio`**, dal quale è accessibile in quanto particolarizzazione della propria classe. **Gli oggetti di una classe non avranno a disposizione nella loro struttura in memoria una propria copia della variabile statica** ma vi potranno **accedere semplicemente accedendo in memoria alla classe da cui provengono**.

Le variabili statiche **possono essere anch'esse private o pubbliche**, sebbene valga lo stesso principio del buon senso per le variabili di istanza; inoltre, **vale la convenzione per cui variabili statiche sono cambiate da metodi statici**.

Se le variabili statiche **servono per trascendere un informazione dal singolo oggetto alla classe, i metodi statici nascono dalla necessità di codificare in una classe dei comportamenti che non hanno alcuna relazione con gli oggetti della stessa**, come **la conversione di una valuta** in una classe `ContoBancario`: la conversione **non è tipica di ogni oggetto**, bensì della classe, e non necessita di avere un proprio spazio nella struttura in memoria dell'oggetto.

Sia variabili statiche che metodi statici sono accessibili non mediante il riferimento all'oggetto ma con il riferimento alla classe:

```
// NO
objectName.staticVariabile
objectName.staticMethod()
// Si
ClassName.staticVariable
ClassName.staticMethod()
```

Nella forma più generale, **una classe può essere caratterizzata da**:

- Variabili di istanza;
- Variabili statiche;
- Costanti statiche;
- Metodi di istanza;
- Metodi statici.

Nelle astrazioni che conducono alla creazione di una classe **non è una buona pratica inserire delle costanti di istanza**, visto che sono informazioni non mutevoli comuni a tutti gli oggetti.

Poiché **un metodo statico può essere invocato senza l'istanza di un oggetto**, è necessario che **nel corpo di tale metodo non ci sia riferimento a variabili o metodi di istanza**, che **all'atto della chiamata non esisteranno in memoria per quella classe**. Analogamente, **un metodo non statico non può essere chiamato da un metodo statico prima che non sia istanziato un oggetto** dal quale prelevare il metodo in questione. Al contrario, **un metodo di istanza di una classe può richiamare una variabile statica o un metodo statico della stessa classe**, senza neanche la necessità di specificare questa.

LA CLASSE `Math` E I WRAPPER

Implementata e importata automaticamente nel linguaggio Java insieme a molti altri strumenti, **la classe predefinita `Math` consente l'esecuzione di numerose operazioni matematiche standard**; di seguito sono elencati alcuni dei più importanti metodi della classe, tutti statici dal momento in cui non è necessario un reale uso di un oggetto `Math`:

Name	Description	Argument Type	Return Type	Example	Value Returned
`pow`	Power	`double`	`double`	`Math.pow(2.0,3.0)`	8.0

`abs`	Absolute value	`int`, `long`, `float`, or `double`	Same as the type of the argument	`Math.abs(-7)`
				`Math.abs(7)`
				`Math.abs(-3.5)`
					7
					7
					3.5

`max`	Maximum	`int`, `long`, `float`, or `double`	Same as the type of the arguments	`Math.max(5, 6)` `Math.max(5.5, 5.3)`	6 5.5
`min`	Minimum	`int`, `long`, `float`, or `double`	Same as the type of the arguments	`Math.min(5, 6)` `Math.min(5.5, 5.3)`	5 5.3
`random`	Random number	none	`double`	`Math.random()`	Random number in the range ≥ 0 and < 1
`round`	Rounding	`float` or `double`	`int` or `long`, respectively	`Math.round(6.2)` `Math.round(6.8)`	6 7
`ceil`	Ceiling	`double`	`double`	`Math.ceil(3.2)` `Math.ceil(3.9)`	4.0 4.0
`floor`	Floor	`double`	`double`	`Math.floor(3.2)` `Math.floor(3.9)`	3.0 3.0
`sqrt`	Square root	`double`	`double`	`Math.sqrt(4.0)`	2.0

Si noti la presenza di **tre metodi apparentemente simili ma che svolgono la stessa operazione con diversi approcci: `round()`, `ceil()` e `floor()`**. Nonostante gli ultimi due ritornino dei valori di tipo `double`, tutti restituiscono un valore che in linea di principio è intero; infatti, **i tre metodi in questione effettuano un'approssimazione**, solo che **`round()` approssima al numero intero più vicino**, **`ceil()` al successivo e `floor()` al precedente**. La peculiarità di `round` è nei tipi di ingresso e ritorno: in ingresso può entrare un `double` (come gli altri due metodi) ma vi sarà restituito un `long`, un intero con più bit per la rappresentazione, ma se vi entra un `float` sarà restituito un semplice `int`.

Un altro metodo interessante e spesso usato è il metodo **`random()`**, che **restituisce un double compreso tra 0 (compreso) e 1 (non compreso)**; quando il numero che si vuole randomizzare deve essere maggiore di 1, **si può moltiplicare l'estrazione in questione per il massimo valore estraibile e si aggiunge 1**:

```
//Estrae un numero intero tra 1 e 6
int randomNumber = (int)(6.0 * Math.random()) + 1
```

In aggiunta ai metodi sopra elencati, **ci sono due costanti matematiche statiche: π ed e**, definite approssimativamente **3.14159 e 2.71828** ed accessibili tramite gli identificativi **`Math.PI`** e **`Math.E`**. Quando, nell'astrazione da effettuare, sono previste delle operazioni che impiegano costanti matematiche, è buona pratica non definirle ex novo ma prelevarle dalla classe `Math`.

Nei precedenti capitoli è stato fatto un ampio discorso sulla differenza tra tipi primitivi e tipi classe, in particolar modo sul modo in cui questi due vengono considerati in memoria. **Ci sono particolari situazioni in cui conviene lavorare su un'informazione attraverso i puntatori o le classi piuttosto che attraverso le variabili di tipo primitivo**; ad esempio, **un metodo può richiedere in ingresso una classe ma l'informazione che serve manipolare con esso è di tipo primitivo**. In queste situazioni, **Java mette a disposizione determinate classi standard, dette classi wrapper, con le quali è possibile convertire dei dati di tipo primitivo in oggetti di tipo classe mantenendone la capacità informativa** (quindi da variabile di tipo `int` a variabile di tipo classe che contiene un intero); **le classi wrapper contengono anche dei metodi con i quali è possibile effettuare tutta una serie di operazioni che altrimenti richiederebbero una progettazione ad hoc**.

Le classi wrapper **funzionano attraverso un meccanismo di boxing – unboxing**, **un particolare caso di incapsulazione**: per **boxing** si intende l'operazione di **wrapping**, cioè di **associazione di una struttura ad un informazione primitiva**, mentre per **unboxing** si intende **l'operazione di prelievo dell'informazione primitiva a partire da una struttura complessa**. Facendo l'esempio con un numero intero:

- **Boxing**

```
Integer n = new Integer(42);
```

- **Unboxing**

```
int numero = n.IntValue();
```

Nella prima istruzione è stata creata una classe `Integer` contenente il valore 42, con la seconda è stata presa l'informazione boxed nella classe `Integer` ed è stata associata ad una variabile di tipo primitivo `int`.

Il boxing e l'unboxing possono essere anche automatizzati e unificati in istruzioni del tipo:

```
Integer n = 42;
int numero = n;
int numero = new Integer(42);
```

Le classi wrapper contengono anche una serie di metodi e variabili statiche con le cui possibili applicazioni sono infinite; ad esempio, le variabili `Integer.MAX_VALUE`, che restituisce il valore `int` massimo che può essere registrato, o il metodo `Double.parseDouble("199.98")` che converte una stringa in double.

Esistono classi wrapper per ogni tipo primitivo: **`char` → `Character`, `int` → `Integer`, `long` → `Long`, `float` → `Float`, `double` → `Double`**. **Le "eccezioni" alla regola sono due**: **`Bool` e `String`**; in particolare, **`Bool` non è usata perché sono più pratiche le variabili primitive `true` e `false`** (al posto di `Bool.TRUE` e `Bool.FALSE`), mentre **`String` sostituisce il tipo primitivo `char[]` in quasi tutte le applicazioni per la sua praticità d'uso**.

OVERLOADING

Inconsapevolmente sono state osservate diverse situazioni in cui **in una stessa classe erano presenti metodi con lo stesso nome ma con una firma diversa** (ad esempio, i costruttori), il che ha permesso **l'adattamento di quel metodo ai vari tipi che gli venivano posti in ingresso all'uso**. Una pratica di questo tipo viene detta **overloading** e si può **osservare e implementare solo tra metodi con lo stesso nome e della stessa classe**, non tra metodi di diverse classi ma con nomi uguali; quindi, **si sta effettuando overloading quando all'interno della stessa classe sono implementati due o più metodi con lo stesso nome**. Java ha bisogno di qualche informazione in più per poter distinguere i diversi metodi in questione e richiamarli solo quando necessario; quindi, **serve avere nella definizione del metodo qualcosa che distingue due metodi in overloading**.

Si può facilmente intuire come l'overloading sia un sovraccarico semantico che va ad intaccare la fase di compilazione del programma; inoltre, **poiché a livello di compilatore due funzioni sono distinte dalle rispettive firme**, si può concludere che questa è il **discriminante tra due funzioni con lo stesso nome**. La **firma di una funzione** è composta da:

- **Nome**;
- **Parametri formali:**
  - **Tipo del parametro**;
  - **Nome del parametro**;
  - **Posizione del parametro**.

Poiché **alla base dell'overloading c'è la corrispondenza dello stesso nome e poiché un'operazione che può essere effettuata su diversi tipi di dati deve ritornare sempre lo stesso tipo di dato**, si può dedurre che **due metodi in overloading avranno di diverso i parametri**; in particolare, **saranno diversi i nomi, i tipi e l'ordine**. Anche **nel caso in cui si volessero diversi tipi di ritorno per una stessa operazione**, come per un metodo getter, **l'overloading non è possibile perché il compilatore non sarebbe in grado di distinguere i due metodi a partire dalla firma** (visto che il tipo di ritorno non è parte della firma).

Si supponga di avere diversi metodi in overloading; durante la compilazione, **se uno di essi è usato, Java osserva la chiamata e la compara con la firma di ogni metodo**: se **coincide** (ovvero se coincidono i tipi e l'ordine dei parametri), **viene eseguito**, se **non coincide osserva la firma del metodo successivo e così via**; nel caso in cui **non si trovasse alcuna corrispondenza**, **Java effettuerebbe delle semplici conversioni di tipo dei parametri per adattare la chiamata ad uno dei metodi a disposizione**.

Ma **perché è utile l'overloading?** Perché **permette di creare una facile associazione tra il nome del metodo e il suo utilizzo anche quando questo può essere chiamato da diversi parametri** (come il metodo **`println()`**, **usato per stampare a schermo una stringa, ma anche un intero o un float**): invece che dare un nome diverso ad ogni funzione che adopera diversi tipi di ingresso ma che esegue la stessa task (per poi doverne ricordare ogni nome), **si preferisce implementare un solo identificativo ma che può essere esteso a diverse combinazioni di input**. A questo punto, si illustra **la lista di possibili metodi che possono essere overloaded**:

- Metodi void;
- Metodi con tipo di ritorno;
- Metodi di istanza;
- Metodi statici;
- Combinazione di metodi sopra elencati;
- Costruttori.

Sulla base di quanto detto finora, **gli errori che si possono commettere quando si fa overloading sono solo errori semantici**, errori commessi da chi scrive il codice e che il compilatore non solleva; **sono i più difficili errori da rilevare e quelli più facili da ottenere**. Un errore semantico durante l'overloading può essere rappresentato **dall'ordine o dal tipo dei parametri** durante la chiamata del metodo overloaded: **se si sbaglia l'ordine dei parametri ma c'è un metodo che ha la stessa firma, viene comunque chiamato un metodo (che non sarà quello desiderato)**, altrimenti il programma non trova alcun metodo corrispondente e solleva errore; **se, invece, si sbaglia il tipo dei parametri, Java verifica prima che ci sia un metodo con quella firma, altrimenti cerca di effettuare un type casting per adattare la chiamata** o altrimenti, se neanche questa soluzione risolve il problema, solleva errore.

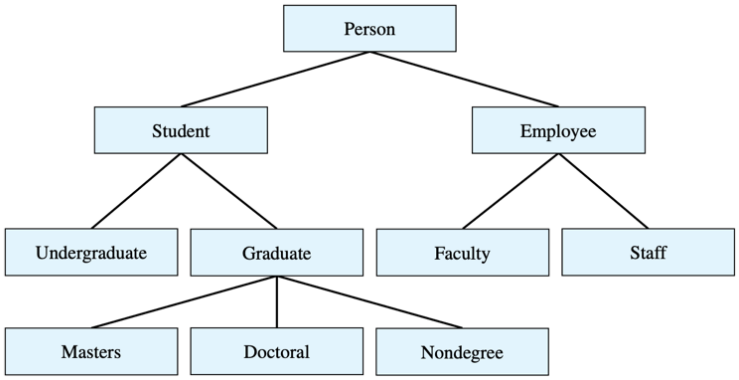
Si potrebbe pensare che **l'errore più scomodo sia quello che occorre quando il compilatore non trova metodi con la firma specificata**, quando in realtà **sono molto più tediosi i casi in cui l'errore semantico porta Java a chiamare un metodo diverso da quello desiderato**. Si supponga di avere due metodi overloaded che prendono in ingresso un solo parametro, uno intero e uno double; se la chiamata viene fatta sul secondo metodo ma, per errore, si inserisce un numero intero al posto di un double, il compilatore non farà type casting e non chiamerà il metodo corretto ma andrà a chiamare il primo metodo, che magari svolge un'operazione del tutto differente rispetto a quella del metodo che si voleva chiamare.

È ovvio che **questi errori possono essere evitati con un po' di attenzione**, però **non si può mai predire cosa vuole fare un programmatore che scrive usando classi scritte da altri**; ne consegue che **l'overloading è un ottimo strumento quando viene meticolosamente dosato e pensato**, ovvero **quando non ne si abusa e quando si pensa ad ogni possibile evenienza**. In genere, il consiglio convenzionalmente dato è di **non fare overloading dei metodi setter e di limitare questa pratica ai costruttori**.

EREDITARIETÀ

L'ereditarietà è il terzo dei quattro pilastri su cui si basa la Programmazione Orientata agli Oggetti; permette di **definire una classe a partire da una sua versione generalizzata**, senza la necessità di ridefinire alcuni attributi e alcuni metodi che sono in comune ma con la possibilità di **specializzare alcuni comportamenti che la classe originaria non può assumere**. **La classe ereditiera non potrà accedere ai comportamenti specializzati della classe ereditaria ma quest'ultima potrà accedere ai comportamenti della prima**.

L'ereditarietà **è uno strumento profittevole quando applicato a delle astrazioni che si trovano in un rapporto gerarchico** (o relazione **is – a**, "è – un" letteralmente); ad esempio, in un sistema di gestione informazioni universitario si possono trovare diverse figure (studenti, impiegati, laureandi, dottorandi, specializzandi, professori, personale amministrativo …) ognuna delle quali presenta delle caratteristiche in comune (come il nome, il cognome, la possibilità di apporre la propria firma …). **Grazie all'ereditarietà è possibile racchiudere in una classe generale** (la classe Persona) **tutte le caratteristiche e i comportamenti in comune a tutte le altre classi** (il nome, il cognome e la possibilità di apporre la propria firma) **per poi specializzarle in comportamenti proprietari** (l'insegnante la possibilità di sedersi alla cattedra e il corso che insegna, lo studente i corsi che segue e la possibilità di dare esami, il dottorando la propria laurea e la possibilità di fare ricerca …). **La specializzazione e la generalizzazione non si fermano ad un solo livello**, una classe specializzata può essere generalizzazione di un'altra specializzazione e così via. Per comprendere meglio questa struttura gerarchica si propone la seguente immagine:



La **classe specializzata** può anche essere detta **subclass** (o **classe derivata** o **classe figlio** ed è quella che **riceve**, o eredita) mentre la **classe generalizzata superclass** (o **classe base** o **classe padre** ed è quella che **diffonde**); la Persona è la classe padre e lo Studente la classe figlio. **Una classe derivata eredita dalla propria classe padre tutti i metodi di istanza e i metodi pubblici, rendendo non necessaria la loro definizione**; verranno definiti solo quegli strumenti che sono presenti nella classe figlio ma non in quella padre, ovvero le specializzazioni.

Riprendendo l'esempio, la classe Dottorando è figlia della classe Laureato ma non è figlia né della classe Studente né della sua superclass, Persona; nonostante ciò, le variabili di istanza e i metodi pubblici di Persona e Studente sono comunque accessibili alla classe Dottorando. Il motivo di questa stranezza risiede nella capacità dell'ereditarietà di propagarsi a relazioni concatenate, rendendo la classe Dottorando comunque in una relazione is – a con la classe Persona e potendone definire l'ereditarietà (sebbene in maniera indiretta); in questa ottica, la classe Persona è la classe antenato (non più padre) e la classe Dottorando è la classe discendente (non più classe figlia).

Dal punto di vista di codice, **in Java una classe figlia eredita da una classe padre quando è specificata la keyword `extends`**:

```
public class Child extends Parent{
      …
}
```

Può presentarsi la necessità di **definire in una classe padre un metodo che descrive un certo comportamento** (attraverso una certa firma) **e in una sua classe figlia un metodo che descrive un comportamento analogo** (con la stessa firma, quindi stesso nome, stessi parametri dello stesso tipo e con lo stesso ordine); **poiché la classe figlio eredita tutti i metodi della classe padre, essa sarà caratterizzata da entrambe le copie del metodo**, senza la possibilità di effettuare overloading (le firme sono uguali). **Java è in grado di distinguere le due funzioni e di utilizzare quella che viene ridefinita nella classe figlio**; l'operazione appena descritta è detta **overriding**.

Quando una funzione è ridefinita con il processo di overriding è impossibile, nella classe figlia, **cambiare la firma o il tipo di ritorno del metodo**, altrimenti sarebbero accessibili dall'oggetto **due funzioni diverse** (cadendo nel caso dell'**overloading**); esiste, però, una sola eccezione ed è disponibile dalla versione 5 di Java: **quando il metodo da ridefinire restituisce un tipo oggetto, è possibile cambiare l'oggetto di ritorno nella classe figlia**. Ad esempio, nella classe Persona c'è un metodo che restituisce un oggetto Persona, mentre nella classe figlia Studente può esserci un metodo che esegue le stesse operazioni ma che restituisce un oggetto Studente; al posto di definire due metodi diversi, con firme diverse e in classi diverse (così che poi Studente erediti entrambi), si può ridefinire il metodo della classe Persona cambiandone il tipo di ritorno in Studente. **Il tipo in questione è detto tipo di ritorno covariante**.

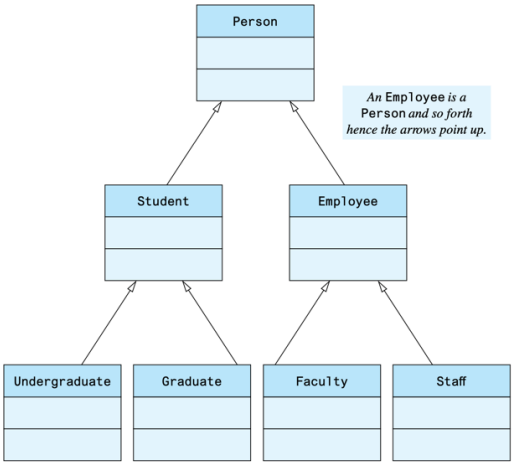
Un'operazione di questo tipo è **fattibile perché nella ridefinizione non si va a sostituire il tipo classe con un tipo più restrittivo** (quale possa essere un intero o un carattere); inoltre, **visto che il tipo classe è stato dimostrato essere un particolare caso di tipo puntatore, è come se si stesse sostituendo un tipo puntatore con un altro tipo puntatore**, rendendo la situazione praticamente invariata.

Per quanto riguarda **la modifica dei modificatori d'accesso di un metodo overridden**, la regola generale prevede che **si possa sempre cambiare il modificatore in modo che l'accesso sia più permissivo** (quindi si può modificare da `private` a `public` ma non da `public` a `private`); **d'altronde se sussiste un meccanismo di ereditarietà non ha senso rendere meno permissivo l'accesso ad un metodo**. Analogamente, **è possibile invocare un metodo pubblico laddove in precedenza era invocato un metodo privato ma non è possibile invocare un metodo privato laddove in precedenza era invocato un metodo pubblico**. In realtà, **l'ereditarietà permette di definire un terzo modificatore d'accesso, `protected`**, con il quale **il metodo o l'attributo è accessibile solo da classi figlie**.

Può essere necessario, in una classe padre, **impedire la ridefinizione di un metodo in una classe figlia**; questa necessità è codificata dalla **keyword `final`**, che usata **in accoppiamento alle variabili le rende costanti** ma che **usata per un metodo lo rende non – ridefinibile**; allo stesso modo, **la keyword `final` associata ad una classe impedisce che qualsiasi altra classe erediti da essa**.

Con queste informazioni a disposizione, **è bene distinguere precisamente overloading e overriding**; **l'overloading occorre nella stessa classe e con funzioni che coincidono solo nel tipo di ritorno e nel nome** (quindi non in tutta la firma), mentre **l'overriding occorre in classi diverse ma imparentate e con funzioni che hanno lo stesso tipo di ritorno** (ad eccezione del tipo classe) **e la stessa firma**. Dal punto di vista **eziologico**, **l'overloading rappresenta un sovraccarico semantico** mentre **l'overriding una ridefinizione del metodo**.

Per rappresentare una relazione di **ereditarietà nei diagrammi UML** si usa **una freccia che punta dalla specializzazione** (quindi dalla classe figlio) **alla generalizzazione** (quindi alla classe padre), **come per indicare da chi provengono i metodi e gli attributi ereditati**. Per risalire alla definizione di un metodo o di una variabile ereditati è spesso utile ricorrere al diagramma UML, in quanto risulta particolarmente arduo risalire alla stessa informazione osservando solo il codice in cui questi sono usati.



In una relazione di ereditarietà, **non vengono ereditati i costruttori della classe padre** ma, in genere, **quando si va a costruire una classe che eredita da un'altra classe, si dovranno sempre definire le strutture che appartengono alla classe padre**; per evitare di dover riscrivere le istruzioni di costruzione, **nella classe figlio si può richiamare automaticamente il costruttore della classe padre** attraverso l'istruzione **`super()`**, che **accede al costruttore padre che ha la stessa firma specificata** (quindi si possono richiamare anche costruttori con parametri). **L'istruzione `super` deve essere sempre specificata come prima istruzione**, dal momento in cui **è necessario costruire prima le strutture su cui la classe erediterà si basa**; inoltre, in maniera analoga a come funziona il costruttore di default di una comune classe, **se non si specificasse alcun richiamo al costruttore della classe padre ne verrebbe chiamato quello di default**.

L'istruzione `super` non è tanto differente dall'istruzione `this`, entrambe agiscono come **placeholder**, **`super` per la classe padre e `this` per l'oggetto istanza di quella classe**. Anche gli usi delle due istruzioni sono simili; infatti, **`super` può essere usato per accedere anche ai metodi** (soprattutto quelli overridden e quelli che accedono agli attributi privati, non ereditati) **e alle variabili di istanza** (pubbliche) **della classe padre**, mentre **`this` può essere chiamato per chiamare un costruttore** (della classe stessa):

```
this(name) // Chiama il costruttore della classe attuale
super.getName() // Invoca la funzione della classe padre getName()
```

Quando si usano `this` e `super` per chiamare un costruttore (della classe attuale o di un'eventuale classe padre), **devono entrambi essere inseriti come prima istruzione**; ciò significa che **in uno stesso metodo/costruttore non possono essere inseriti insieme**, o si inserisce uno o un altro.

Si supponga di **scrivere in una classe un metodo che ne ridefinisce uno della classe padre da cui eredita**; in entrambe le classi, **il metodo prende come parametro un oggetto della stessa classe** (come un metodo `equals`) **e nel metodo della classe figlia è richiamato il metodo della classe padre**. Nella chiamata che viene effettuata nel corpo della funzione figlia, **non è necessario**

effettuare un type casting che trasforma il parametro `ClasseFiglia` in `ClassePadre`, dal momento in cui sussiste una relazione di eredità: **la classe figlia contiene tutte le strutture della classe padre ed è in relazione is – a**, **`ClasseFiglia` è una `ClassePadre`**.

```
// Metodo overridden
public boolean equals(ClasseFiglio altroOggetto){
     // Type casting, usando la funzione equals di ClassePadre
     return equals((ClassePadre)altroOggetto) && altre condizioni;
}
```

Si potrebbe benissimo anche ricorrere alla chiamata esplicita della funzione padre con l'istruzione `super`:

```
// Metodo overridden
public boolean equals(ClasseFiglio altroOggetto){
     // Chiamata al padre, usando la funzione equals di ClassePadre
     return super.equals(altroOggetto) && altre condizioni;
}
```

Questo principio **si applica a qualsiasi funzione in cui è necessario un parametro di tipo classe**, potendo immettervi qualsiasi oggetto di tipo classe ad essa figlio, **e anche su diversi livelli di eredità**. Per generalizzare la regola, **un oggetto che è istanza di una classe discendente può comportarsi in tutto e per tutto come un oggetto istanza della classe antenata**. **Ciò che**, invece, **non è possibile fare**, è inizializza**re un oggetto come istanza di una classe e chiamarne il costruttore di un suo antenato**:

```
ClasseFiglio nomeOggetto = new ClassePadre(); // NO
```

Poiché **l'eredità è una relazione non biunivoca** (in Java non esistono, così come non esiste l'eredità multipla), **tutte le operazioni appena elencate funzionano solo se si fa agire una classe come una sua antenata**, non il contrario perché non sussiste la relaizone is – a.

LA CLASSE `Object`

In Java il meccanismo di ereditarietà è automatico quando non è esplicito; la classe da cui questa **"eredità di default"** è estratta è la **classe `Object`**, detta anche **classe Eva**. La classe `Object` è **la classe antenato di qualsiasi classe si possa creare in Java** e dalla quale o**gnuna di queste preleva determinate strutture**; sulla base di quanto è stato precedentemente detto, **anche se si ereditasse una classe a partire da un'altra, questa sarebbe comunque discendente della classe `Object` perché la sua classe padre ne è discendente**.

Il fatto che esiste una classe **antenato comune** permette di co**dificare dei metodi o delle funzioni che prendono in ingresso un oggetto nel suo senso più generale**, senza specificarne una particolare struttura; allo stesso modo, **la classe `Object` prevede l'eredità di alcuni metodi che tutte le classi possono usare senza doverne riscrivere il corpo**, o meglio, **dovendolo solo adattare al particolare utilizzo**. Un esempio di questi metodi particolari sono quelli che sono stati definiti come **`equals()` e `toString()`**, sebbene nella maggior parte dei casi applicativi considerati in questa sede andranno adattati ad hoc; infatti, **i metodi `equals()` e `toString()` definiti nella classe `Object` sono un**

po' complicati e generalisti, adatti all'oggetto nel senso più generale del termine, mentre fino ad ora sono stati usati in accezioni più particolari.

Il metodo `toString()` della classe `Object` non prende nessun argomento in ingresso e restituisce una stringa contenente una formattazione in stringa di tutti i dati presenti nella classe; ad esempio, sia considerata una classe che ha come variabili di istanza un nome e un identificativo (tipo una matricola), il relativo metodo `toString()` ereditato dalla classe `Object` sarbbe del tipo:

```
public String toString(){
       return "Nome: " + getNome() +
                  "\nMatricola: " + getMatricola();
}
```

Restituendo una formattazione del tipo:

```
Nome: nome
Matricola: matricola
```

Nella maggior parte dei casi, però, **non è utile questa formattazione e conviene adottare un implementazione ad hoc** ridefinendo la funzione attraverso l'overriding.

Di seguito verranno mostrate delle **procedure atte a migliorare l'implementazione di un metodo `equals()` nelle classi scritte ad hoc per una determinata evenienza**; ognuno di questi consigli può essere **trasposto per la ridefinizione del metodo `toString()` e di ogni altro metodo notevole che viene ereditato dalla classe `Object`** ma che richiede alcuni accorgimenti per essere adattato alla corrente applicazione. Chiaramente, **il tipo di ritorno del metodo `equals()` deve essere** lo stesso per la classe proprietaria e per la classe `Object`, ovvero **`boolean`**: restituirà **`true` se i due oggetti sono uguali** e **`false` se non lo sono**; **la firma però differisce**, il nome è lo stesso ma **i parametri di ingresso non lo saranno**: p**er la classe `Object` andrà messo in ingresso un parametro di tipo classe `Object`**, **per la classe proprietaria un parametro di tipo classe proprietaria**. Tuttavia, come è stato precedentemente mostrato, **se si dispone di un oggetto di tipo classe proprietaria si disporrà automaticamente dello stesso oggetto ma di tipo `Object`**, visto che sussiste una relazione is – a tra classe proprietaria e `Object`, ma **non varrà il contrario**, un oggetto di tipo `Object` non sarà necessariamente un oggetto di tipo classe proprietaria.

Al momento attuale, le due firme del metodo `equals()` sono:

```
public boolean equals(Object oggetto)
public boolean equals(ClassName oggetto)
```

Per il momento, però, non si può parlare di overriding ma di overloading, non si sta ridefinendo ma sovraccaricando perché **la classe `ClassName` avrà entrambi i metodi**. Nella maggior parte dei casi non ci saranno problemi, però **può capitare che venga chiamato il metodo sbagliato**; ad esempio, si dispongano di due oggetti, `oggettoObject` e `oggettoClasse`, e si vuole verificare se i due siano uguali. Si potrà fare la chiamata:

```
oggettoClasse.equals(oggettoObject);
```

Che invocherà dalla classe `ClassName` il metodo della classe `Object` con il parametro `Object`. Per poter comunque operare su questo metodo, **andrà fatto un type casting dell'oggetto `oggettoObject` verso il tipo `ClassName`**:

```
public boolean equals(Object oggettoObject){
       ClassName castedObject = (ClassName)oggettoObject;
       // Verifica l'uguaglianza
}
```

Tuttavia, **nel caso in cui `oggettoObject` non sia un oggetto `ClassName`** (la relazione is – a è al contrario), **verrà sollevato errore**; per non far fermare il programma, e visto che se `oggettoObject` non è di tipo `ClassName` sicuramente non sarà uguale a `oggettoClasse`**, si può usare un'istruzione booleana che ritorna `false` se l'oggetto inserito non è un'istanza della classe specificata**:

```
oggettoObject istanceof ClassName;
```

Se, invece, `oggettoObject` è anche di tipo `ClassName`, l'istruzione ritornerà `true` e potrà essere eseguito sia il type casting che la verifica dell'uguaglianza. In ultima istanza, **si verifica anche**, **per una questione di sicurezza, che l'oggetto in ingresso come parametro non sia nullo**.

Il metodo `equals()` finale assume la forma:

```
public boolean equals(Object oggetto){
      boolean isEqual = false;
       if ((oggettoObject != null) &&
           (oggettoObject istanceof ClassName)){
              ClassName castedObject = (ClassName)oggettoObject;
              isEqual = condizioni uguaglianza;
        }
          return isEqual;
}
```

POLIMORFISMO E BINDING DINAMICO

Mentre l'ereditarietà permette di definire una classe base il cui codice è usabile non solo dagli oggetti istanza di tale classe ma anche quelli istanza di classi da essa derivate, **il polimorfismo è la proprietà che consente di applicare modifiche alla definizione del metodo per le classi derivate e di applicare tali modifiche ai metodi scritti nella classe base**.

Il concetto di polimorfismo passa per quello di **dynamic binding**. **Per binding si intende il processo per cui l'invocazione di un metodo viene associato alla sua specifica definizione** e può essere implementato in due diverse modalità:

- **Binding statico** (o **early binding**), l'associazione avviene **al momento della compilazione**;
- **Binding dinamico** (o **late binding**), l'associazione avviene **al momento dell'esecuzione**.

Java implementa il binding dinamico per quasi tutti i metodi, salvo alcune eccezioni.

Si supponga di disporre di **diverse classi che rappresentano delle figure geometriche, ognuna delle quali eredita da `FiguraGeometrica`**, e che si voglia cre**are un metodo `centra()` per centrare e visualizzare l'oggetto in uno schermo**:

```
public class FiguraGeometrica {
    public void centra() {
        // Istruzioni per centrare la figura
        visualizza();
    }
    public void visualizza(){
        // Istruzioni per visualizzare
    }
}
```

In realtà **ogni figura geometrica avrà il proprio modo di essere visualizzata**, sarà quindi necessario ridefinirlo nella classe derivata:

```
public class Rettangolo extends FiguraGeometrica {
    public double larghezza;
    public double altezza;
    public double centroX;
    public double centroY;
    public void visualizza(){
        // Istruzioni per visualizzare il rettangolo
    }
}
```

Si istanzi un nuovo rettangolo e **si invochi il metodo `centra()`**, disponibile perché ereditato da `FiguraGeometrica`:

```
Rettangolo r = new Rettangolo();
r.centra();
```

Nel caso in cui Java usasse il binding statico, quando l'oggetto `r` invoca il metodo `centra()` il compilatore si sposta nella classe `FiguraGeometrica`, dove **vede sia il metodo in questione che il metodo `visualizza()`**, il quale **non sarà associato alla classe che l'ha chiamato** (`Rettangolo`) **ma a quella che il compilatore vede** (`FiguraGeometrica`); questo comportamento **non è quello desiderato**, visto che si vuole centrare un rettangolo, ma poiché Java usa il binding dinamico problemi di questo tipo non si incontrano. Con un meccanismo di **binding dinamico**, **la chiamata a `centra()` e `visualizza()` non viene fatta in fase di compilazione** (dove il compilatore vedrebbe il metodo `visualizza()` nella stessa classe di `centra()`) **ma in fase di esecuzione**, dove **non si accoppierà il metodo alla definizione più vicina ma a quella contenuta nella classe dell'oggetto chiamante**.

Polimorfismo e binding dinamico sono **due concetti ben diversi** ma che **si sfruttano a vicenda** per ampliare le possibilità di Programmazione Orientata agli Oggetti; **il primo esprime la possibilità**

di associare ad uno stesso metodo diversi significati (il metodo `centra()` lavora in modo diverso in base a chi lo chiama), **il secondo opera l'accoppiamento tra chiamata e definizione con lo scopo di indirizzare il polimorfismo**. **Il polimorfismo è il quarto dei quattro pilastri su cui si basa la Programmazione Orientata agli Oggetti**.

Si approfondisca il tema del polimorfismo e del binding dinamico con un ulteriore esempio. **La classe `Object`**, da cui derivano tutte le classi, **fornisce una prima definizione del metodo `toString()`** che, però, è **inutile nella maggior parte dei casi**, in quanto non fornisce una opportuna formattazione della rappresentazione testuale della classe; **accadrà spesso**, quindi, **che si debba ridefinire tale metodo in modo che restituisca un'appropriata rappresentazione degli oggetti della classe**.

Si supponga di voler stampare a schermo un oggetto di tipo `Vendita`:

```
Vendita unaVendita = new Vendita();
Sytem.out.println(unaVendita.toString());
```

Dove il metodo `unaVendita.toString()` è una ridefinizione di quello della classe `Object`; l'output sarà:

```
Componente = pneumatico, Prezzo e costo totale = E9.95
```

Ma **può accadere che al posto di passare al metodo `unaVendita.toString()`, si passi solo l'oggetto `unaVendita`**; **l'output rimarrà invariato** perché `println()` è un metodo affetto da **overloading nell'oggetto `System.out`** e Java implementa **binding dinamico**, quindi sarà in grado di **richiamare lo stesso il metodo `toString()` della classe Vendita**:

```
public void println(Object oggetto){
    System.out.println(oggetto.toString);
}
```

Con la chiamata:

```
System.out.println(unaVendita);
```

Si sta passando al metodo un parametro di tipo `Object` (visto che `Vendita` is – a `Object`) dotato di metodo `toString()`; tuttavia, **senza il binding dinamico l'output cambierebbe**, perché **il metodo `toString()` associato alla chiamata sarebbe quello dello stesso tipo del parametro che il compilatore vede in fase di compilazione**, cioè `Object`, mentre con il binding dinamico si aspetterebbe fino al runtime, dove il parametro visto non è di tipo `Object` ma di tipo `Vendita`, che al suo interno ridefinisce `toString()`.

Come precedentemente mostrato, **per disabilitare l'overriding** (principale responsabile di polimorfismo e ambiente di utilizzo principale del binding dinamico**) è sufficiente impostare un metodo come costante tramite la keyword `final`**. Il modificatore `final` **può essere visto come uno strumento per disabilitare il binding dinamico e forzare Java ad usare, per quel metodo, il binding statico** (sebbene **in realtà fa qualcosa in più**, ovvero impedisce di ridefinire il metodo nelle classi derivate). Un motivo per cui si potrebbe preferire un metodo `final` è quello delle prestazioni; infatti, **il binding statico è leggermente più performante del binding dinamico ma l'incremento nelle prestazioni non giustifica nella maggior parte dei casi la preferenza per il modificatore**.

I metodi per i quali in Java il binding dinamico viene sostituito da quello statico sono quelli privati e `final`, per i quali non avrebbe utilità il binding dinamico, **e quelli statici**, in cui l'assenza di binding dinamico può essere significativa quando il metodo viene invocato utilizzando un oggetto chiamante. In genere, **quando si invoca un metodo statico nella definizione di un metodo non statico senza usare né il nome di una classe né di un oggetto chiamante**, **si suppone implicitamente usato il `this` e il tipo di binding usato è**, quindi, quello **statico**.

DOWNCAST E UPCAST

Si consideri il seguente set di istruzioni, considerando le classi `Vendita` (padre) e `VenditaScontata` (figlia):

```
Vendita variabileVendita;
VenditaScontata variabileScontata = new VenditaScontata();
variabileVendita = variabileScontata;
System.out.println(variabileVendita.toString());
```

Non solo queste operazioni sono possibili nonostante i due oggetti sono diversi, ma nella chiamata al metodo `toString()` sarà anche chiamato quello della classe `VenditaScontata`. L'assegnazione tra i due oggetti è possible nel momento in cui tra le due classi c'è una relazione is – a con la quale **è possibile affermare che un oggetto `VenditaScontata` è un oggetto `Vendita`**; inoltre, **visto che è utilizzato il binding dinamico, Java associa il metodo non in funzione del tipo della variabile puntatore ad oggetto ma in funzione dell'oggetto stesso**.

Si può pensare che operazioni di questo tipo siano inutili e superflue, quando in realtà sono più comuni di quanto si pensi, solo che vengono effettuate "dietro le quinte" come nel passaggio dei parametri oppure nei costruttori di copia. **Il costruttore di copia è un particolare tipo di costruttore che ammette come parametro un oggetto istanza della stessa classe che si sta costruendo con lo scopo di copiarne i contenuti**; ricorrendo all'esempio delle classi `Vendita` e `VenditaScontata`, **è possibile costruire un oggetto `Vendita` a partire da un oggetto `VenditaScontata` chiamando il costruttore di copia (in ingresso va un oggetto `Vendita`)**:

```
super(venditaScontata);
```

Un altro motivo per cui le assegnazioni di variabili di tipo classe diverse sono lecite risiede nel fatto che queste **non rappresentano gli oggetti in sé e per sé ma solo i riferimenti**, con i quali è possibile invocare i metodi della relativa classe; quindi, **quando si assegna un oggetto `VenditaScontata` ad una variabile di tipo `Vendita` si stanno solo riducendo i metodi invocabili con quella variabile ai metodi della classe `Vendita`**, **l'oggetto** (e quindi i suoi metodi, con tutte le conseguenze sul binding dinamico) **non viene alterato**. Per comprendere meglio questo meccanismo, si pensi che **sono gli oggetti che conoscono la definizione dei propri metodi, i riferimenti conoscono solo quali metodi invocare in base al proprio tipo**.

Il procedimento di assegnazione di un oggetto di una classe derivata ad una variabile del tipo della classe base (o una qualsiasi superclasse) **è detto upcasting** ("conversione verso l'alto", si sta salendo il diagramma UML), mentre **la conversione da una classe base a una classe derivata** (o da una superclasse ad una discendente) **è detta downcast** ("conversione verso il basso", si sta scendendo il diagramma UML). **L'upcasting è semplice ed immediato**, visto che **sussiste la relazione is – a**,

mentre **proprio questa relazione rende il downcast decisamente più problematico**, **soprattutto dal punto di vista logico**.

Java intercetta gli errori di downcast il prima possibile ma ci sono alcuni errori che non sono rilevabili prima del runtime:

```
// errore di compilazione
oggettoDiscendente = oggettoSuperclasse;
// errore a runtime
oggettoDiscendente = (classeDiscendente)oggettoSuperclasse;
```

Benché sia pericoloso, **a volte il downcast è necessario**; ad esempio, **quando si invoca il metodo `equals()`** si deve effettuare un downcast:

```
Vendita altraVendita = (Vendita)altroOggetto;
return (nome.equals(altraVendita.nome) &&
       (pezzo == altraVendita.prezzo);
```

Senza il downcasting, gli attributi `nome` e `prezzo` non sarebbero esistenti nell'istanza `altroOggetto` e il loro utilizzo sarebbe illegale. **Verificare che il downcast sia legale non è una responsabilità del compilatore**, **che non ha strumenti per il controllo**, **ma del programmatore**. Il **downcast verso un tipo specifico** funziona **se l'oggetto che deve essere convertito è di quel tipo**; questo controllo **può essere fatto dall'istruzione `istanceof`**, in modo tale da non sollevare errori (ma indirizzare il programma verso altre istruzioni) se il downcast non è legale.

CLASSI ASTRATTE E INTERFACCE

Una classe astratta è una classe dotata di alcuni metodi non completamente definiti, che **non può essere costruita in un oggetto** e che **può fungere solo da classe base** per un meccanismo di eredità.

Una classe astratta **ha senso di esistere quando rappresenta un'astrazione troppo astratta**, **della quale non è possibile immaginarsi un oggetto concreto** (ad esempio, `FiguraGeometrica` può essere un'astrazione ma non si può immaginare una figura geometrica particolare prima di definire le classi `Rettangolo`, `Triangolo` …); in questa classe **ci saranno determinati metodi che rappresentano comportamenti comuni alle classi derivate ma che**, comportandosi diversamente in ogni classe figlia, **non necessitano una definizione precisa e ad hoc**, anche perché non vendendo istanziato alcun oggetto a partire da essa **non saranno mai utilizzati**. I metodi in questione, che vengono **definiti nella classe antenata solo per poter sfruttare il polimorfismo**, vengono detti **metodi astratti** e **necessitano solo della firma**, **la definizione verrà implementata nelle classi figlie con un meccanismo di override**:

```
public abstract metodoDaRidefinire();
```

Rendendo un metodo astratto, si posticipa la sua definizione, delegata a tutte le classi figlie in modo che non possa essere in alcun modo usato da un oggetto un metodo che non ha definizione. Ovviamente, **la ridefinizione è possibile solo se la classe derivata è in grado di definire il metodo**, **altrimenti l'astrazione si propaga anche ad essa**; nel momento in cui i**n una classe esiste anche**

solo un metodo astratto, l'intera classe è definita come astratta (rendendo necessaria anche la keyword `abstract`) **e non si potranno istanziare oggetti a partire da essa**:

```
public abstract ClasseDerivataAstratta{
    …
}
```

Non potendo istanziare oggetti, l'unico scopo di una classe astratta è l'ereditarietà; può **funzionare solo da classe padre** per eventuali classi derivate e **seminare determinati comportamenti da ereditare**; per questo motivo, **una classe astratta è detta anche classe base astratta**, mentre **la controparte è detta classe concreta**. Quindi, sebbene non ce ne sia alcuna esigenza, **la seguente istruzione non è valida**:

```
ClasseAstratta oggetto = new ClasseAstratta();
```

Ovviamente, **in una classe astratta non sono preclusi metodi definiti** (non astratti); in generale, **in una classe andranno definiti solo metodi che, a quel livello di astrazione del problema reale, sono definibili** (ad esempio, il quadrilatero non si può disegnare ma può dire di quanti vertici è composto). I metodi definiti in una classe astratta saranno ereditati (a meno di eventuali ridefinizioni) dalle classi derivate con la definizione incorporata e, perciò, saranno usabili.

Sebbene non abbia senso logico istanziare un oggetto di una classe astratta, ha perfettamente senso disporre di una variabile del tipo di una classe astratta, costruita a partire da una delle sue classi derivate:

```
FormaGeometrica r = new Rettangolo();
r.disegna();
```

Però, **la decisione su quale metodo usare**, tra due metodi ridefiniti, **non dipende dal tipo della variabile di riferimento ma dalla classe a partire dalla quale è stato istanziato l'oggetto referenziato** con tale variabile; quindi, **l'invocazione precedente sarà accoppiata con il metodo della classe rettangolo**. Il modo con cui Java determina la classe da cui invocare il metodo è **l'osservazione di quale costruttore costruisce la classe dopo l'istruzione `new`**: in questo caso è `Rettangolo`, l'oggetto sarà un `Rettangolo` e la variabile che lo referenzia una `FormaGeometrica`; **nel caso in cui Java non trovi nella classe `Rettangolo` il metodo invocato, risale la catena di ereditarietà finché non trova una classe dotata del metodo invocato da cui `Rettangolo` eredita**.

Tuttavia, **nel momento in cui la classe `Rettangolo` implementi un metodo ad hoc al di fuori dell'ereditarietà**, questo **non potrà essere invocato con la variabile di tipo `FormaGeometrica`** perché **in essa non è presente quel metodo**. Si può dire che **il riferimento di un tipo classe può essere associato ad un oggetto istanza di una sua classe derivata ma non può accedere agli elementi che non sono stati ereditati da quest'ultima**.

Ogni classe è dotata di due componenti fondamentali: **interfaccia** e **implementazione**; le due, finora, sono sempre state specificate nello stesso momento ma **Java mette a disposizione una serie di strumenti con i quali è possibile separarle**. Un motivo per cui questa separazione è utile risiede nella **possibilità di dare al programmatore uno strumento pronto all'uso senza la necessità di andare a vedere come tale strumento è fatto**; infatti, **le interfacce sono composte unicamente dalle firme dei metodi pubblici e dalle dichiarazioni delle costanti pubbliche**, non vi è alcuna

menzione all'implementazione, il che **rende il lavoro più facile al programmatore perché non ha altro di cui occuparsi**.

Le interfacce permettono anche di **definire una serie di comportamenti che diverse classi possono avere in comune senza specificarne i dettagli di implementazione**, che **possono essere diversi** nelle varie classi; **il modo in cui il metodo dell'interfaccia sarà chiamato è lo stesso per ogni classe**, sebbene il risultato dell'elaborazione possa essere diverso.

In Java, **un'interfaccia è definita in maniera del tutto analoga alle classi**, **si sostituisce solo alla keyword `class` la keyword `interface`**; dopodiché, n**el corpo dell'interfaccia si scrive la firma dei metodi** (o la definizione di costanti) **pubblici da implementare in una classe separata**, distanziati dal punto e virgola e con la possibilità di inserirvi commenti:

```
public interface NomeInterfaccia{
    public final float COSTANTE1;
    public void primoMetodo();
    public boolean secondoMetodo(String param1, int param2);
    …
}
```

Questa interfaccia verrà salvata in un file `.java` apposito, con lo stesso nome dell'interfaccia stessa. Si conclude aggiungendo che **le interfacce possono contenere quanti metodi e costanti si vuole, a patto che non ne sia mai definita una** (non devono avere corpo) **e che non ci sia alcun costruttore o variabile di istanza**.

Una classe che definisce i metodi solo dichiarati in un'interfaccia implementa tale interfaccia; se tale classe **non implementa tutti i metodi**, va definita **astratta**. Sebbene una classe debba implementare tutti i metodi di un'interfaccia, **un'interfaccia non deve necessariamente dichiarare tutti i metodi della classe**, questa **può avere dei metodi diegetici**. L'implementazione passa per due step:

- **Specificare le interfacce che si stanno implementando** tramite la keyword `implements` nella definizione della classe;

```
public class NomeClasse implements Interfaccia1, Interfaccia2 {…}
```

- **Definire i metodi solo dichiarati nell'interfaccia**;

```
public void primoMetodo(){…}
```

Si deduce da quanto appena detto che **una stessa classe può implementare più interfacce** ma vale anche il contrario, cioè **è possibile che più classi implementino la stessa interfaccia**.

Un'interfaccia è un tipo riferimento così come una classe; si può, quindi, **definire un metodo che passi come argomento un'interfaccia**. Sussiste la **stessa relazione tra classi antenate e classi derivate quando si parla di interfaccia e classi implementanti** quando si vanno a creare delle variabili di tipo classe antenata/interfaccia inizializzate con l'istanza ad una classe derivata/classe implementante:

```
ClasseAntenata nomeRiferimento = new ClasseDerivata();
Interfaccia nomeRiferimento = new ClasseImplementante();
```

Allo stesso modo in cui il riferimento non può chiamare i metodi in `ClasseDerivata` che non sono stati ereditati da `ClasseAntenata`, non può chiamare i metodi in `ClasseImplementante` che non sono stati dichiarati in `Interfaccia`. Si può intuire che, in conformità con quanto detto finora, **anche il meccanismo di binding dinamico e di polimorfismo si applicano alle interfacce**.

Avendo già a disposizione un'interfaccia, **è possibile crearne una che la estende** (da qui la keyword `extends`) **utilizzando una sorta di ereditarietà**. L'interfaccia "derivata" disporrà sia dei metodi dichiarati in essa che dei metodi "ereditati"; di conseguenza, **una classe che implementa questa nuova interfaccia dovrà definire anche quelli dell'interfaccia "padre"**. A differenza dell'ereditarietà per classi, **le interfacce possono effettuare eredità multiple**:

```
public interface Padre1 {…}
public interface Padre2 {…}
public interface Nuova extends Padre1, Padre2 {…}
```

ALGORITMI E STRUTTURE DATI

LE FASI DI SVILUPPO DI UN SOFTWARE

Paradossalmente, **per sviluppare un software non è conveniente procedere direttamente allo sviluppo di un programma che risolva il problema alla sua base**; si individua, quindi, una **pipeline** che **parte dall'analisi del problema e termina con la produzione del software che lo risolve**, detta **design strategy**:

1. **Specifica e analisi del problema**, si isola ciò che si ha a disposizione e ciò che va prodotto senza ancora specificare il modo in cui si connettono queste informazioni;
2. **Design dell'algoritmo**, si astrae la soluzione escludendo i dettagli di realizzazione e specificando le opportune strutture e gli opportuni dati;
3. **Implementazione in codice**, si traduce l'algoritmo sviluppato in un linguaggio che può essere compreso dai calcolatori;
4. **Test e Debug**, si valutano possibili errori e le lori soluzioni.

Questi passi non sono costituiscono procedimenti a compartimenti stagni, interagiscono e si sovrappongono; **l'importante è procedere con rigore e non saltare direttamente alla risoluzione del problema**. Una tecnica efficace, che permette di semplificare il carico di lavoro che la pipeline induce, è il **metodo divide et impera** (o sovrapposzione degli effetti), con il quale **il problema macroscopico è diviso in problemi microscopici la cui somma delle soluzioni restituisce la soluzione al problema principale**.

Esistono diversi paradigmi di programmazione, uno di questi è la **Programmazione Orientata agli Oggetti** (o OOP). La OOP ha riscosso negli anni molto successo grazie alla sua **vicinanza con la realtà**; infatti, **si basa sulla possibilità di astrarre qualsiasi oggetto della vita reale e di distinguerne i comportamenti (il cosa) e gli attributi (il come)**. Ha i vantaggi di **poter costruire il cosa separato dal come**, nascondendo all'utente le informazioni di cui non deve necessitare; segue che **i software progettati con paradigmi OOP non sono più gerarchici**, non esiste più il blocco locale ed il blocco globale, bensì **eterarchici**, **ogni blocco di codice ha la stessa importanza** (sebbene ci siano gradi di visibilità diversi), **la costruzione è orizzontale e non più verticale**.

Quando si effettua il **design dell'algoritmo**, si deve prestare particolare **attenzione ai dati che possono e quelli che non possono entrare e uscire dal programma**; **si identifica un dominio**, **l'insieme di possibili dati di input**, **e un codominio**, **l'insieme di possibili dati di output**, **al di fuori dei quali l'algoritmo non deve accettare o produrre informazioni**. Il vincolo è posto formalmente dalle **pre – condition** e dalle **post – condition**, condizio**ni per le quali i dati in questione vengono controllati al fine di non processare e comunicare risultati che sono a priori sbagliati**; ad esempio, in un algoritmo di conversione di temperature da gradi Kelvin a gradi Celsius, non vanno accettati dati numerici negativi, visto che temperature Kelvin negative non esistono. Le pre/post – condition possono essere dei puri costrutti formali (come un commento posto in testa alla funzione ed utile al programmatore) o delle vere e proprie istruzioni che fermano il programma (come `assert`) o non eseguono il blocco di codice che restituirebbe errori (come `try`).

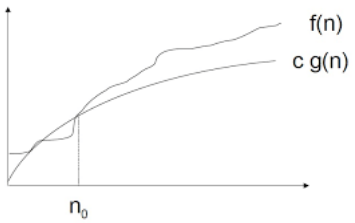
La pre – condition si compone come una asserzione che deve essere necessariamente vera ancor prima di eseguire l'algoritmo affinché sia prodotto un risultato, mentre **la post – condition come una asserzione che deve essere necessariamente vera in corrispondenza dell'output affinché il risultato sia accettabile**; dal momento in cui sono **asserzioni antecedenti alla scrittura del codice**, esse **devono essere invarianti rispetto a procedure equivalenti**. Poiché **le pre/post – condition sono condizioni che hanno lo scopo di evitare eventuali errori** che potrebbero far fermare

l'esecuzione di un'operazione, **è importante inserirle in punti strategici del codice**, dove c'è una probabilità maggiore di incontrare errori, come i cicli.

Nel processo di sviluppo di un software è necessario valutare adeguatamente anche quanto l'algoritmo è veloce in relazione alla procedura che gli abbiamo associato. Ci possono essere algoritmi che assolvono ad un compito in minor tempo ma con maggior consumo delle risorse, mentre altri che gestiscono bene il carico di dati sebbene siano lenti; in questo caso, **si modula la progettazione dell'algoritmo in base alla necessità**: se **serve un software veloce e reattivo** (come un ipotetico SO di un computer di bordo spaziale) **non ci si preoccupa di eventuali sprechi di risorse** che sarebbero **eliminati con software più lenti**. Da queste premesse, si può pensare che il tempo di esecuzione di un programma sia un buon parametro per valutare la rapidità del programma stesso; tuttavia, questa assunzione è errata perché computer diversi hanno diverse memorie, diversi processori e diversi sovraccarichi. Quindi, **il tempo di esecuzione di un programma è un parametro soggettivo in relazione alla rapidità del programma stesso**, che **necessita di parametri oggettivi per essere valutato**.

L'analisi della complessità temporale di un algoritmo è un'operazione da effettuare in fase di design dell'algoritmo ed è uno studio che comprende la **valutazione del carico di dati che il programma deve gestire**. **L'oggettivizzazione della misura del tempo passa per uno strumento matematico portabile**, che può essere usato su qualsiasi macchina, detto **ordine degli infiniti** (o **notazione Big – O** o **complessità asintotica**), per il quale **un algoritmo $\boldsymbol{T(n)}$ che riceve un carico di dati $\boldsymbol{n}$ viene classificato in base a come varia il numero di operazioni da eseguire quando il carico aumenta verso l'infinito**:

$$\forall\big(f(n)\big)_{n\in\mathbb{N}}, T(n) = O\big(f(n)\big) \Leftrightarrow \exists n_0, c > 0 : \forall n \geq n_0, T(n) \leq cf(n)$$



Quindi $\boldsymbol{T(n) = O\big(f(n)\big)}$ **se è al più una costante moltiplicata per la funzione $\boldsymbol{f(n)}$**. Chiaramente, **il meccanismo appena enunciato è un meccanismo matematico di convergenza**, che associa un algoritmo alla funzione a cui converge quando il carico di dati è infinito.

La valutazione della complessità asintotica di un algoritmo può essere fatta a partire dalla struttura che è stata designata per l'algoritmo stesso. Vanno **presi in considerazione il numero di volte con cui un'operazione è effettuata in relazione al carico da gestire**; quindi, **avranno un peso chiamate a funzioni, operazioni e condizionamenti** (cicli e strutture di selezione).

Sia fatto il seguente esempio: **si vogliono calcolare il numero di gradini della torre Eiffel**. Si possono implementare diversi algoritmi:

1. **Ogni gradino in basso si fa un segno**;
2. **Ogni gradino in basso si risale e si fa fare un segno ad una seconda persona**;
3. **Ogni dieci gradini si fa un segno**.

Indicando come **operazioni rilevanti al calcolo della complessità** la **salita** o la **discesa di un gradino e il segno**, **ognuno di questi tre algoritmi esegue**:

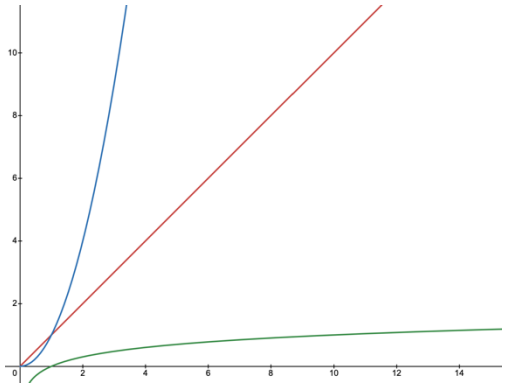
1. $2689\ (up) + 2689\ (down) + 2689(segni) = \mathbf{8067}$ **operazioni**;
2. $[1 + 2 + \cdots + 2679(down)] + [1 + 2 + \cdots + 2679(up)] + 2679(segni) = \mathbf{7236099}$ **operazioni**;
3. $\mathbf{10}$(segni) **operazioni**;

Se il numero di gradini fosse $\boldsymbol{n}$, si sarebbero effettuate:

1. $\boldsymbol{3n}$ **operazioni**;
2. $\boldsymbol{n^2 + 2n}$ **operazioni**;
3. $\boldsymbol{\log_{10} n}$ **operazioni**.

E le complessità temporali diventano:

1. $\boldsymbol{O(n)}$;
2. $\boldsymbol{O(n^2)}$;
3. $\boldsymbol{O(\log_{10} n)}$.



L'algoritmo più conveniente è logaritmico (come spesso capita).

È inutile andare a sviluppare un algoritmo estremamente efficiente sulla base di una quantità spropositata di dati quando nella realtà la quantità concreta è decisamente minore, a quel punto conviene **sviluppare un algoritmo che nei casi con meno dati performi meglio** (ad esempio il linear sort è più efficace di un binary sort per n piccolo ma non per n grande); si individuano, quindi, **tre situazioni che permettono di modulare lo sviluppo dell'algoritmo in base alla quantità di dati che verrebbero eventualmente processati**: **best case**, **average case**, **worst case**. Il **best case** fa riferimento a **quantità di dati minori** o al caso in cui **l'operazione sia completata in pochissime operazioni**, dualmente il **worst case** fa riferimento a **quantità di dati grande** o a casi in cui **l'operazione sia completata con il maggior numero di operazioni possibili**. Un algoritmo può essere **ottimale sul best case ma non sul worst case e viceversa**. Comunemente, si incontrano i seguenti tempi di operazione (in ordine di convenienza):

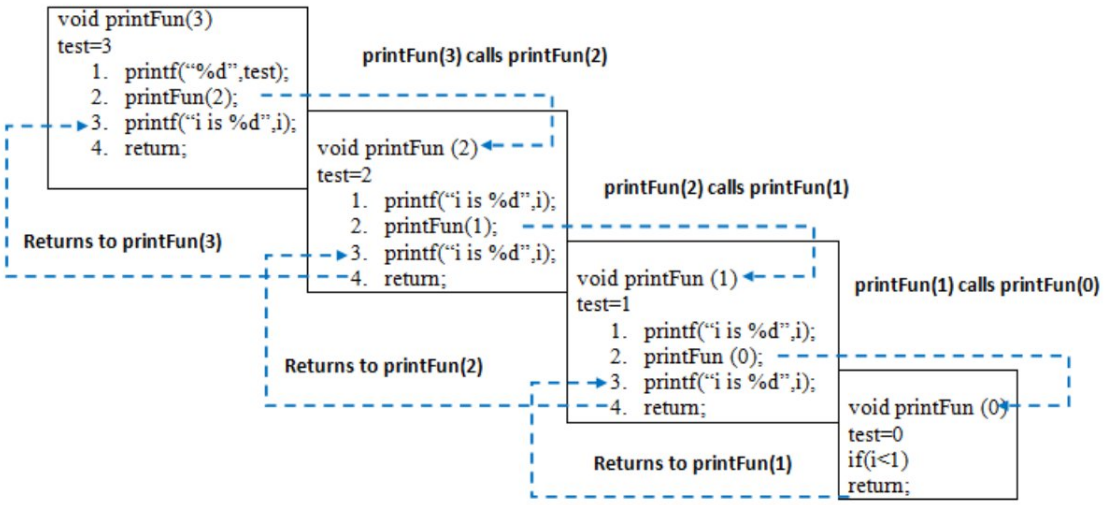
$O(1)$	Costante
$O(\log n)$	Logaritmico
$O(n)$	Lineare
$O(n \log n)$	N – log
$O(n^2)$	Quadratico

$O(n^3)$	Cubico
$O(2^n)$	Esponenziale

Una volta **progettato l'algoritmo** e **scritto il codice che lo descrive**, è importante **valutarne la sua efficacia** andando ad effettuare una serie di **prove e di test**; in quest'ottica si inserisce la fase di Test e Debug: **si esegue il programma con lo scopo di trovare degli errori e successivamente risolverli**. La **fase di Testing** si **occupa di portare in ingresso al programma dei dati al limite del dominio applicativo** (**boundary values**, ad esempio valori prossimi allo zero per l'algoritmo di conversione Kelvin – Celsius), **tenendo in conto che se il codice cambia devono rimanere corretti i testing effettuati prima del cambiamento**, mentre la **fase di Debug** si occupa di **trovare all'interno del codice i blocchi o le istruzioni che causano eventuali errori rilevati nella fase precedente**.

LA RICORSIONE

Sebbene sia spesso associata ad un codice, **la ricorsione è un concetto che affonda le proprie radici nella matematica**; **un algoritmo è detto ricorsivo se contiene una subtask** che è una **versione ridimensionata di sé stessa**, un'invocazione a sé stessa. Esempi notevoli di algoritmi matematici ricorsivi sono il **fattoriale**, la **potenza** e la **serie di Fibonacci**; **una funzione che esegue uno di questi algoritmi non è una ricorsione, bensì uno strumento con il quale può essere attuata una procedura ricorsiva**. Così enunciata, la ricorsione presenta già quelle che sono le sue condizioni di applicazione: **una procedura ricorsiva dovrà sempre agire su porzioni di dominio ridotte ad ogni chiamata ricorsiva** ($n! = n(n-1)! \wedge n-1 < n$), altrimenti **l'algoritmo non avrebbe un limite inferiore oltre il quale fermarsi**. Pertanto, **si individua una base** (o **stopping case**) **della ricorsione**, un livello **oltre il quale non si può andare** e che **svolge l'operazione ricorsiva con il dato più semplice**; nel progetto di un algoritmo ricorsivo **è fondamentale chiarire lo stopping case**, **senza di esso ci sarebbero infinite chiamate ricorsive** che **occuperebbero tempo e spazio** senza restituire alcun risultato. Generalmente, **se nell'algoritmo sono presenti delle pre – condition, esse sono proprio la base della ricorsione**.



Una ricorsione avviene in maniera differente da una chiamata ciclica: **quando si entra nella funzione e si giunge alla chiamata ricorsiva si congela la chiamata attuale**, si entra nella **seconda chiamata** e **si esegue il controllo per verificare che ci si trovi o meno nel caso base**, se il controllo

restituisce **esito negativo** si procede ad una **nuova chiamata ricorsiva** e **il percorso viene ripetuto fintantoché non si giunge al caso base** (winding), **a partire dal quale le chiamate congelate sono rievocate ed eseguite** fino alla loro terminazione (unwinding); **una volta terminate tutte le chiamate ricorsive**, sono state **eseguite anche tutte le operazioni ricorsive** e **si può avere a disposizione l'output**.

Segue che, se **il numero di iterazioni di un ciclo sono n**, **sono tali anche il numero di ricorsioni di un algoritmo equivalente**; tuttavia, **la complessità dell'algoritmo non è più $O(n)$ ma $O(1)$** perché **la singola chiamata è isolata dalle altre**, a differenza di come accade nei cicli. Di conseguenza, si può dedurre che **il programma principale non è a conoscenza di tutte le chiamate ricorsive**, le considera come delle semplici chiamate a funzione; **ciò che risente della ricorsione è il compilatore**, dal momento in cui **è colui che si occupa della creazione dei descrittori ad ogni chiamata di funzione**. Ogni **chiamata di ricorsione** è una **chiamata a funzione**, quindi **necessita di un proprio descrittore** (con firma, parametri, variabili locali, indirizzi di rientro e risultati) e per funzioni particolarmente complesse e/o con molti parametri/variabili locali **si può rischiare di incontrare l'errore di stack overflow**, che occorre quando **si cerca di scrivere nello stack** (dove vengono conservati i dati relativi al descrittore di una funzione) **oltre la sua capienza massima**; una possibile **soluzione** è **l'aumento della dimensione dello stack ma deve essere valutata in relazione al singolo algoritmo che si sta sviluppando**. L'errore di stack overflow dovuto a descrittori ampi non si raggiunge con tanta facilità quanto l'errore di stack overflow dovuto a chiamate ricorsive infinite; dunque, **inserire uno stopping case corretto permette di non giungere ad una condizione di ricorsione infinita e di non esaurire facilmente la memoria disponibile nello stack**.

Il **descrittore di una funzione** (o **record di attivazione**) **è un'area di memoria stack adibita a conservare tutte le informazioni di una funzione** e si compone come segue:

Area dei parametri formali
Area delle variabili locali
Indirizzo di rientro
Area risultati

L'area dei risultati è posta dopo l'indirizzo di rientro con lo scopo di **essere conservata anche dopo che la funzione è terminata e la memoria stack pulita**, mentre **l'indirizzo di rientro è spesso l'indirizzo dove è avvenuta l'interruzione** (la chiamata) **al quale è aggiunta un'unità** (per poter rientrare all'indirizzo successivo la chiamata).

Il **corretto funzionamento di un algoritmo ricorsivo** è legato strettamente al concetto di **induzione matematica**, per la quale **una proposizione P_n è vera se sono vere P_1 e P_{n-1}** (**supposta P_n vera**); infatti, **durante il processo di winding e unwinding si vanno a creare una successione di proposizioni legate ognuna alla singola istanza di ricorsione e tra di loro mediante implicazioni logiche**:

$$P_{base} \Longrightarrow P_{base+1} \Longrightarrow \cdots \Longrightarrow P_{n-1} \Longrightarrow P_n$$

Ovviamente, **si parte dal caso base perché prima che esso venga raggiunto dalle varie chiamate non vengono effettuate operazioni elementari**, si fa solo una verifica della situazione per entrare

nel caso base o in un'altra chiamata ricorsiva. **Segue che un corretto algoritmo ricorsivo debba rispettare i seguenti criteri**:

1. **Non ci deve essere ricorsione infinita**, altrimenti non si raggiunge lo stopping case e non può essere instaurata una relazione di induzione matematica;
2. **Ogni caso di arresto deve eseguire le operazioni corrette per quel caso**;
3. (Per i casi che coinvolgono la ricorsione) **Se tutte le chiamate ricorsive eseguono correttamente le loro operazioni, l'intera elaborazione è corretta**.

Si può vedere come **questi criteri permettono di instaurare una relazione biunivoca tra la ricorsione e l'induzione matematica**. Nel momento in cui **uno di questi tre criteri manchi**, **l'errore va ricercato nella definizione o dello stopping case o del procedimento induttivo**; in particolare, se **mancano i primi due criteri** si andrà a **lavorare sul caso base**, se **manca il terzo** sul **procedimento induttivo**. Dal punto di vista tecnico, **per poter scrivere un buon algoritmo ricorsivo sono necessarie le seguenti condizioni**:

1. Si deve **implementare una struttura selettiva che indirizza il caso attuale verso la ricorsione o il caso base**;
2. **Uno o più branches devono condurre ad una chiamata ricorsiva**, effettuata su un campione di dati minore di quello su cui la chiamata ricorsiva attuale sta lavorando;
3. **Almeno un branch deve condurre ad una chiamata non ricorsiva**, che rappresenta il caso base.

Adesso **ci si chiede quali siano i motivi per cui un algoritmo ricorsivo è da preferire ad uno equivalente iterativo**; i motivi principali sono tre: **in primis scende notevolmente il costo computazionale**, come è stato precedentemente mostrato in termini di complessità algoritmica, **ma scende anche il rischio di errori**, dal momento in cui **non sono usati cicli** (che sono l'origine del 90% degli errori), e **aumenta la rapidità con cui si prototipizza un algoritmo**.

Si palesano, quindi, i **due principali modi con cui una stessa operazione può essere svolta**: in **maniera ricorsiva** o in **maniera iterativa**. **La relazione tra i due è di semplice implicazione logica**, cioè **un algoritmo ricorsivo può essere sempre scritto in maniera iterativa ma non vale necessariamente il contrario**. Un banale esempio di come si possono interfacciare le due metodologie è con la **potenza x^n**: per la **potenza iterativa** si può **inizializzare un ciclo di n iterazioni che moltiplica x ogni volta che è eseguito**, mentre per la **versione ricorsiva** si può **calcolare x^n come $x \cdot x^{n-1}$ fintantoché l'esponente non è nullo**. L'unico ambito in cui **un algoritmo iterativo performa meglio di uno ricorsivo** è la **memoria**: è stato mostrato come **per la ricorsione sia necessario inizializzare tanti descrittori quante siano le ipotetiche iterazioni**, andando a **consumare inevitabilmente più memoria**; anche il tempo di esecuzione può essere uno svantaggio per gli algoritmi ricorsivi, ma solo per moli di dati inferiori a quelle che si supporranno in questa sede.

`ArrayList` E LE ADT

Un **ADT** (Abstract Data Type) è un **insieme di dati per i quai sono specificate le operazioni possibili su di esse**, senza definire un modo in cui implementare o memorizzare tali dati; un ADT **può essere implementato tramite un'interfaccia**, e quindi una classe Java, **usando diverse strutture dati semplici** (un costrutto, come un array o una classe, insita in un linguaggio di programmazione).

Si parla di **strutture dati dinamiche** facendo riferimento a **quelle strutture che aumentano dimensione e cardinalità al momento della necessità**, mentre le **strutture dati statiche** sono **quelle che, una volta assegnata una dimensione, non permettono di inserire più di una tale quantità di dati**. Un esempio di struttura dati statica è l'array, in grado di memorizzare solo un numero predeterminato di elementi e nel caso in cui si volesse eccedere questo limite si incontrerebbe l'errore di stack overflow; **per poter aumentare la dimensione di un array si può procedere verso due strade**: in primis si può **creare un secondo array più capiente** e copiarne all'interno i dati già memorizzati, altrimenti **si può ricorrere a strutture dati dinamiche**. Chiaramente, **la prima opzione risulta quella più semplice ma meno elegante e pratica**; pertanto, d'ora in poi quando sarà necessario implementare una struttura vettoriale si cercherà di utilizzare sempre un alias dell'array di tipo dinamico.

Una struttura dati di questo tipo è descritta nella classe `ArrayList` del package `java.util` nella Java Class Library; **un'istanza di `ArrayList` permette di gestire i dati in maniera del tutto identica a come lo fa un semplice array**, se non per la **possibilità di gestirne dinamicamente le dimensioni**. Ci si chiede, allora, **perché non si è smesso di usare gli array in favore delle `ArrayList`**; il motivo può essere segmentato in due:

- **Un'istanza di `ArrayList` è meno efficiente di un array**;
- **Un'istanza di `ArrayList` può memorizzare solo oggetti**, non tipi primitivi come un array.

Sul primo problema non si può fare nulla in più a ponderare la scelta in funzione del caso applicativo, mentre **il secondo problema è aggirabile utilizzando le classi wrapper**. Purtroppo, **entrambi i problemi aggiungono un overhead** (sovraccarico) **al programma**, sia in termini di velocità computazionale che di leggibilità.

La causa di questi problemi va ricercata in quella che è **l'essenza delle `ArrayList`**: **le liste**; infatti, **un'istanza della classe `ArrayList` è una simulazione di un array tramite una ADT dinamica, la lista a puntatori**, che introduce intrinsecamente l'overhead menzionato. **La lista dinamica è un tipo di struttura dati che permette di legare diversi oggetti in un ordine ben preciso** (infatti è detta anche linked list), in cui **ogni elemento punta a quello successivo**, con la possibilità di **aggiungere ovunque si voglia un elemento ex novo**, senza alterare gli altri; **un'`ArrayList`, poi, raccoglie questa catena di elementi e la comunica all'utilizzatore come se fosse un array**, simulando il comportamento di un array dinamico.

Per **istanziare un `ArrayList`** è necessario in primis importare il package e la classe:

```
import java.util.ArrayList;
```

Per poi **istanziare un oggetto** come fatto finora:

```
ArrayList<tipo_base> listaStringhe = new ArrayList<tipo_base>();
```

Tuttavia, **si nota una piccola novità rispetto all'istanza di classe fatta finora: il tipo `String` tra `<>`**. Il motivo di questa sintassi sarà chiaro solo in seguito, basta pensare che **in questo caso `String` codifica il tipo di elementi da specificare nell'array** (dovranno essere sempre tipi classe). **Il costruttore di default della classe `ArrayList` costruisce una lista di 10 posizioni iniziali**, sebbene sia **possibile invocarne uno che implementi una dimensione iniziale arbitraria**. Quando si dice che **la lista ha un tot di elementi**, o che ha dimensione tot, **si intende dire che il compilatore ha allocato la memoria per quel numero di elementi**, ovvero che ci sono quel numero di allocazioni

disponibili in memoria per gli elementi di quella lista; **nel momento in cui viene dinamicamente aumentata la dimensione, l'allocatore riserverà uno spazio in più per il nuovo elemento**.

Sebbene non interferisca sul numero di elementi memorizzabili nell'array dinamico, **la dimensione di un `ArrayList` è fondamentale per limare l'impatto del problema di efficienza**: una **dimensione iniziale troppo bassa** implica che **il compilatore dovrà spesso aumentare la dimensione**, allocare memoria ed eseguire operazioni che lo sovraccaricano, ma una **dimensione iniziale troppo alta** implica **sprechi di memoria eccessivi**; come sempre, **la miglior pratica è la valutazione preliminare**.

I **metodi fondamentali** con i quali gli `ArrayList` svolgono i compiti di un array sono tre:

- **`arrrayListObject.get(indice)`**, restituisce l'elemento di `arrrayListObject` in posizione indice;
- **`arrrayListObject.set(indice, contenuto)`**, modifica l'elemento di `arrrayListObject` in posizione indice sostituendovi contenuto;
- **`arrrayListObject.add(contenuto)`**, aggiunge contenuto dopo l'ultimo elemento non nullo di `arrrayListObject`;
    - c'è anche la variante **`arrrayListObject.add(indice, contenuto)`**, con la quale l'aggiunta avviene in posizione `indice` (sempre se l'elemento precedente non è nullo) spostando di conseguenza tutti gli altri elementi;

In poche parole, **i metodi `get()` e `set()` sostituiscono l'accesso e la modifica di un array classico**, mentre l'**`add()` interviene per l'operazione di aggiunta che l'array non può eseguire** e per inizializzare la prima volta gli elementi. Si può intuire facilmente **la comodità degli `ArrayList`**: **per l'aggiunta di un elemento in coda o nel mezzo della struttura non è necessario scrivere alcun codice**, come invece sarebbe necessario in un array.

La classe `ArrayList` non implementa solo questi tre metodi, ce ne sono altri come `arrrayListObject.size()`, che restituisce la dimensione della lista (il che significa che gli indici vanno da 0 a `arrrayListObject.size() - 1`).

GENERICI E TIPI PARAMETRICI

Si riprenda il tema della sintassi aliena `ArrayList<tipo_base>`: **la classe `ArrayList` è detta classe parametrica** (o generica), cioè **una classe che prende un parametro in ingresso prima di poter essere usata per istanziare un oggetto**, mentre **`tipo_base` è detto tipo generico**; il parametro in questione è il tipo di dato che l'oggetto `ArrayList` dovrà collezionare e discrimina tutta una serie di comportamenti della lista in funzione di quale dato dovrà essere gestito.

Può essere necessario descrivere con una sola classe dei dati e dei comportamenti comuni a diversi tipi; al posto di scrivere tante classi quanti sono i tipi che condividono quell'astrazione, **si preferisce usare il tipo parametrico, un tipo che non è definito se non al momento dell'istanza**. Per convenzione, **questi tipi sono descritti da un solo carattere maiuscolo** (spesso `T`) **e vanno specificati accanto al nome della classe in cui sono utilizzati tra le parentesi <>**:

```
public class ClassName<T>{
    …
```

```
}
```

Il **tipo generico** rappresenterà un **tipo classe particolare**, quindi potrà essere usato ovunque, si può usare un tipo classe (per variabili di tipo classe, per parametri o metodi, ecc…); **ciò che non è possibile fare è istanziare un oggetto di tipo classe generico**, dal momento in cui **non sarebbe definito a priori il costruttore da invocare per costruire l'oggetto**, o una struttura dati che richiede l'allocazione di una quantità non precisa di memoria. Ad esempio:

```
T dati = new T();
T[] array = new T[20];
```

Tuttavia, non tutto è da buttare: **ciò che causa problema in queste due istruzioni è l'inizializzazione della variabile**, dal momento in cui (in entrambi i casi) si cerca di allocare memoria secondo una struttura che non è specificata; **si può evitare l'errore definendo le variabili senza inizializzarle**:

```
T dati;
T[] array;
```

Quando **una classe fa uso di tipi parametrici può essere compilata separatamente ma non può essere usata finché non vi è specificato un tipo classe effettivo da sostituire con quello parametrico**. In maniera del tutto analoga a come si fa per una funzione, specificare il tipo classe di un tipo generico è fatto tramite le parentesi <>:

```
ClassName<ClasseEffettiva>
```

Si conclude elencando alcuni vincoli sui tipi parametrici; in primis, **un costruttore non può contenere tipi parametrici se non come parametri formali**:

```
public ClassName<T>(){…} // NO
public ClassName(T parametro){…} // SI
```

Non è possibile sostituire il tipo parametrico con un tipo primitivo, solo con tipi classe:

```
ClassName<int> // NO
ClassName<Integer> // SI, Integer è una classe wrapper
```

La definizione di una classe generica può avere in sé più di un tipo parametrico, a patto che **rispetti tutti i vincoli imposti per ognuno di essi**. Può capitare che **in una classe sia definito un metodo che restituisca un oggetto di tipo parametrico che implementi un'interfaccia**; in tale caso, **non è sufficiente indicare il tipo parametrico nella definizione della classe ma bisogna anche imporre a tale tipo di implementare la stessa interfaccia del metodo**:

```
public class ClassName<T extends Interfaccia>{
    …
}
```

In questo modo **saranno ammessi solo tipi classi che implementano `Interfaccia`**. Questo vincolo **non è una comodità opzionale**, **la sua assenza porterebbe il compilatore a sollevare un errore quando sarebbero incontrate le definizioni dei metodi implementati dall'interfaccia**. Il discorso è **analogo quando il tipo parametrico deve ereditare una classe**.

Può sembrare strano che la keyword da usare per imporre ad un tipo parametrico di implementare un'interfaccia sia `extends` e non `implements`; in realtà è una scelta di comodità, potendo sia implementare un'interfaccia che ereditare una classe, si è preferito uniformare la sintassi sotto una stessa keyword:

```
public class ClassName<T extends Interfaccia, Classe>{
    …
}
```

Oltre le classi, **anche i metodi e le interfacce possono usare tipi parametrici**; in particolare, **i metodi generici possono esistere anche quando la classe non è generica o quando il tipo parametrico della rispettiva classe generica non è lo stesso di quello specificato**. Per definire un metodo generico **è adottata la stessa sintassi delle classi generiche**, se non per il fatto che **il tipo parametrico è definito dopo i modificatori** (prima del tipo di ritorno) **ed è sostituito alla chiamata prima del nome del metodo**:

```
public static <T> boolean Name(){…}
object.<Type>Name();
```

Infine, sono ammessi meccanismi di ereditarietà tra classi generiche e tra classi generiche e ordinarie.

LA GESTIONE DELLE ECCEZIONI

Per eccezione si intende un segnale che occorre in seguito all'accadere di un evento inusuale durante l'esecuzione di un programma, rappresentato da un oggetto; il processo di creazione dell'oggetto che rappresenta l'eccezione viene detto **exception throwing** e viene eseguito **in una porzione di codice ben circoscritta dal programmatore**, il quale ha bisogno di gestire l'eccezione (**exception handling** è il processo di gestione) con lo scopo di **non far fermare mai il programma**. Le eccezioni **descrivono spesso particolari condizioni**, **di errore o di output**, **che il programmatore non desidera incontrare una volta eseguito il programma**; tuttavia, **i software sono spesso progettati con lo scopo di non essere mai terminati**, se non esplicitamente desiderato, ed **un eventuale errore può preliminarmente terminare l'esecuzione e causare danni più o meno gravi** (basti pensare allo spegnimento improvviso di un software di gestione delle banche o del software di bordo della ISS). Si rende, quindi, **necessaria una ottimale gestione delle eccezioni**, in modo tale da **prevenire determinate situazioni ed agire di conseguenza prima che il programma termini** (ad esempio, una divisione per zero può essere sostituita con ∞ senza far terminare il programma). Quando si tiene in considerazione la possibilità di eventuali eventi eccezionali nella progettazione di un software, si sta adottando un tipo di programmazione detta **programmazione preventiva** (o **fault – tolerant**) ed è, in genere, una **buona pratica**.

Per poter **prevenire un errore** è dapprima necessario **individuarlo**; tuttavia, **individuare un errore senza aver mai scritto una riga di codice risulta difficile e perciò il processo di gestione delle eccezioni è successivo alla progettazione e all'implementazione del programma**. Nel processo di sviluppo di un software, **una volta arrivati al momento in cui gestire le eccezioni** bisogna dapprima **chiedersi dove è possibile trovare eventuali errori** (non è efficiente gestire gli errori su tutto il codice perché determinate porzioni possono non contenerne, sprecando inutilmente risorse); una volta **individuato il blocco di codice che potenzialmente contiene eccezioni**, si procede a **racchiudere tale intorno in un try – catch block**, un particolare costrutto che Java mette a disposizione per individuare e risolvere eventuali eccezioni. Un costrutto di questo tipo è **composto da due parti**:

- **Blocco try**, in cui vengono racchiuse le righe di codice individuate e in cui verrà sollevata un'eventuale eccezione;
- **Blocco catch**, in cui viene raccolta l'eccezione (lanciata con l'istruzione `throw`) e vengono eseguite delle righe di codice risolutive o palliative, che non terminano l'esecuzione e rimandano il flusso del programma verso altre direzioni.

Il rapporto tra i due blocchi è lo stesso che c'è tra un lanciatore e un battitore in una partita di baseball: una volta **individuata e accorsa l'eccezione**, **il lanciatore** (blocco try) **lancia** (`throw`) **l'eccezione che viene raccolta** (`catch`) **dal battitore** (blocco catch). Dal punto di vista pratico, il blocco try – catch può essere implementato come segue:

```
try{
    Blocco che contiene l'eccezione
    throw new Exception("Messaggio");
    Altro codice (non eseguito se si solleva l'eccezione)
} catch (Exception e){
    Gestione dell'eccezione
}
```

Sia **lo statement `catch` `(Exception e)` che `throw new Exception("Messaggio")` possono sembrare chiamate a funzioni con dei parametri**; tuttavia, esse **non sono nulla di simile**. L'espressione **`throw new Exception("Messaggio")` non fa altro che creare un oggetto istanza della classe `Exception`**, ovvero un eccezione, **che verrà raccolta e gestita nel blocco catch utilizzando il riferimento `e` di tipo `Exception`**. La necessità di specificare `Exception` non è ridondante, dal momento in cui **esistono diverse tipologie di eccezioni ed ognuna viene gestita da una classe diversa**; quindi, **il blocco catch gestirà solo le eccezioni dello stesso tipo di quella individuata nella propria espressione**.

Dal momento in cui **l'accorrere di un eccezione indirizza il flusso di esecuzione verso il catch** e la sua assenza, invece, permette al flusso di permanere nel try, **può sembrare che il costrutto try – catch possa essere un equivalente del if – else**. I due costrutti **sono molto simili**, con la sola (ed importante, per quanto verrà mostrato di seguito) **differenza che il try – catch permette la comunicazione di un messaggio** (tramite la creazione di un oggetto eccezione) **tra i due blocchi**.

Determinate operazioni e determinati metodi sono stati costruiti in modo tale che, sotto determinate ipotesi di errore, **sollevino automaticamente un'eccezione senza la necessità dell'istruzione `throw`**. Spesso, **le eccezioni così sollevate sono di un tipo diverso dal tipo `Exception`** mostrato precedentemente; in genere, **il nome dell'eccezione è autoesplicativo** ma accedendo al messaggio di errore (tramite l'istruzione `e.getMessage()`) si può avere un quadro più chiaro di che evento ha causato l'eccezione. Alcuni tipi di eccezioni predefiniti sono:

- `BadStringOperationException`;
- `ClassNotFoundException`;
- `IOException`;
- `NoSuchMethodException`;
- …

In apertura del capitolo, è stato detto che **le eccezioni non sono necessariamente errori** ma, generalmente, **eventi eccezionali**; segue, quindi, che **un'eccezione può essere anche sollevata da**

condizioni sintatticamente e semanticamente corrette ma logicamente sbagliate. Questo tipo di eccezioni, nella maggior parte dei casi, **non hanno una classe eccezione dedicata** e andrebbero **gestite con la generale `Exception`, che non sempre è adeguata**; per risolvere questo problema si può **implementare una classe eccezione ad hoc alla propria necessità**.

Per implementare **eccezioni custom** è necessario **dichiarare una classe che eredita dalla classe generica `Exception`** (o da una classe eccezione qualsiasi) **in modo da non dover ridefinire tutti gli strumenti già a disposizione** che permettono la gestione dell'eccezione; ciò che, nella maggior parte dei casi, andranno **modificati e riadattati** alla nuova classe saranno i **costruttori**, spesso **modificati per creare un messaggio custom da memorizzare come attributo**. Le eccezioni custom, non essendo predefinite nel linguaggio, **non prevedono eventi che le sollevano automaticamente** e, quindi, **andranno sempre sollevate manualmente con l'istruzione `throw`** all'occorrere delle situazioni previste all'atto della loro progettazione; in realtà, **il processo logico che porta alla creazione di un'eccezione custom è inverso**: **ogni qualvolta è necessaria l'istruzione `throw` è necessaria una classe eccezione custom**.

Praticamente, nella forma più generale, un'eccezione custom è definita come segue:

```
public class CustomException extends Exception{
    public CustomException(){
        super("Eccezione custom!");
    }
    public CustomException(String message){
        super(message);
    }
}
```

I costruttori creati sono due (per convenzione): uno in cui si passa un messaggio di default al costruttore della classe `Exception`, e uno in cui il messaggio è inserito all'atto dell'istanza dall'utilizzatore; il messaggio, in ogni caso, è accessibile con il metodo `getMessage()`, ereditato da `Exception` e che, pertanto, non va ridefinito. **Non è illecito ridefinire gli altri metodi della classe da cui si eredita ma è buona pratica non farlo finché non è necessario**.

Si menziona anche **l'impossibilità di implementare una classe eccezione generica (`<T>`)**, dal momento in cui **al lancio non si è in grado di sapere qual è la classe di cui si fa istanza e di che tipo sono i suoi metodi**.

È sempre meglio **implementare classi eccezioni custom invece che usare la generica `Exception`**; implementando **in uno stesso blocco try del codice che può sollevare due eccezioni**, una **`DivisionByZeroException`** e una custom, e un unico catch del tipo **`Exception`**, **quando occorrerà una divisione per zero**, questa **non verrà gestita con un blocco catch adeguato alla `DivisionByZeroException` ma con lo stesso blocco catch con cui viene gestita quella custom**, incorrendo in una **errata gestione dell'eccezione**. Quindi, **essendo ogni eccezione predefinita una `Exception`**, conviene **limitare l'uso di questa classe al minimo** (al massimo solo per casi di eccezione default, come eccezioni non previste) e **preferire le implementazioni custom**.

Ci sono determinate situazioni in cui **è ragionevole pensare di far cogliere un eccezione ad un blocco di codice che non l'ha sollevata**, ovvero di **delegare la raccolta di un'eccezione sollevata**. Ad esempio, **un metodo solleva una `DivisionByZeroException` ma delega al programma**

chiamante la sua collezione e gestione; in questo caso, **l'ipotetica istruzione `throw`** (o l'istruzione che solleva direttamente l'eccezione) **è nel metodo**, mentre **il costrutto try – catch è nel programma chiamante**. Ovviamente, **è necessario che il metodo comunichi al chiamante che la gestione dell'eccezione viene delegata ad esso e di quale tipo di eccezione si parla**; tutto ciò è fatto **tramite la direttiva `throws`**, da inserire nella definizione del metodo e seguita dal tipo di eccezione da delegare:

```
public void sampleMethod() throws ExceptionType{…}
```

Riprendendo l'analogia con il battitore e il ricevitore, **è come se, una volta lanciata la palla, il battitore la colpisse per reindirizzarla ad un secondo battitore, che la gestisce**; infatti, l'operazione di delega delle operazioni assomiglia ad un ri – lancio e perciò alcuni linguaggi di programmazione (come C++) usano l'istruzione `rethrow`.

Con quanto appena detto, **si individuano due modi con cui la maggior parte delle eccezioni sollevate in un metodo sono gestite**:

- **Con un blocco catch** nel corpo del metodo stesso;
- **Mediante un meccanismo di delega** al chiamante.

Nel caso in cui in uno stesso metodo si possa dover gestire due (o più) eccezioni, non è vietato gestirle in modi diversi ma nel caso in cui esse fossero **di tipi diversi e le si volesse entrambe delegare si dovrebbero inserire entrambi i tipi nella direttiva `throws`**:

```
public void sampleMethod() throws ExceptionType1, ExceptionType2 {…}
```

Se **il metodo chiamante è chiamato a sua volta da un terzo metodo**, è possibile **delegare a quest'ultimo le eccezioni che possono essere sollevate dal primo metodo**, sebbene questa pratica sia **fortemente sconsigliata**; infatti, **già il meccanismo di delega è da evitare il più possibile** (se c'è un problema non lo si rimanda, lo si affronta in loco), **lo è ancora di più la sua reiterazione**. Infine, **supponendo di avere una classe in cui un metodo delega la gestione delle eccezioni**, nell'eventualità che **un metodo di una classe figlia ridefinisse tale metodo**, **è necessario conservare il tipo di eccezione da delegare**.

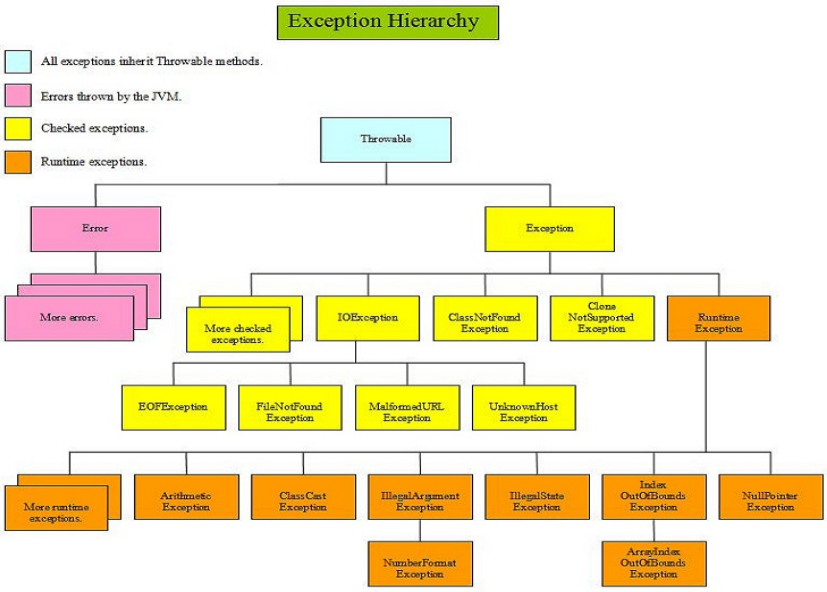
Finora è solo stato introdotto il tema dei tipi di eccezione, non sono stati elencati tutti questi tipi e nemmeno la loro suddivisione. **Java individua due categorie di eccezioni**:

- **Checked exceptions**, devono essere raccolte in un blocco catch o delegate con la direttiva `throws`;
- **Unchecked exceptions**, possono essere sollevate a runtime ma non necessitano di essere raccolte in un catch (o delegate con `throws`).

Quando viene **sollevata un'eccezione unchecked**, nella maggior parte dei casi, significa che **c'è un problema a livello del codice che è meglio gestire andando a modificare il codice stesso**, evitando che ad una seconda esecuzione del programma venga risollevata l'eccezione. In genere questo tipo di eccezioni **non sono previste e non vengono sollevate con l'istruzione `throw`** (originano da valutazioni di espressioni o da particolari operazioni come accessi in memoria) e **conducono sempre alla terminazione del programma**.

Per **verificare che un'eccezione sia checked o unchecked** si può ricorrere alla **documentazione ufficiale** per la **Java Class Library**, che elenca e suddivide le eccezioni in base a quella da cui derivano. Volendo ricostruire i rapporti tra le varie eccezioni, si deve tenere in conto della classe da

cui originano tutte, `Exception`, per poi ridurre il campo di studi (almeno per questa sede) alle `Throwable`, le eccezioni che possono essere sollevate, e così via… Graficamente si individua il seguente schema:



Si può erroneamente pensare che un'eccezione sia un errore. Volendo essere specifici, **un errore è un oggetto della classe `Error`**, che eredita da `Throwable`, **e rappresenta delle particolari condizioni** (molto simili alle unchecked exception per il fatto che non è necessario il blocco catch) **che**, a differenza delle eccezioni, **sono più o meno fuori dal controllo del programmatore**; di conseguenza, **quando occorre un errore, il programma termina senza che sia data possibilità** (utile) **di reindirizzare il flusso di esecuzione** e, per evitare che ciò accada, **si deve agire prevalentemente modificando il codice**.

Nella sua definizione, **un blocco try non preclude un numero preciso di eccezioni sollevabili**, il che significa che **si possono sollevare tutte le eccezioni possibili**; poiché **ogni eccezione necessita di almeno un blocco catch per essere gestita** (in genere è buona pratica implementarne uno per ognuna), **dopo uno stesso blocco try possono essere implementati tanti blocchi catch quante sono le possibili eccezioni che il codice circoscritto può sollevare**, con altrettanti tipi di eccezione. Tuttavia, **è importante valutare attentamente l'ordine dei diversi blocchi catch**, visto che **al sollevamento di un'eccezione essi vengono scrutati in ordine di riga**; poiché un eccezione è sia del tipo classe da cui è istanziata che del tipo eccezione da cui tale classe eredita, **può capitare che sia invocato un catch sbagliato al momento sbagliato** (ecco perché il tipo `Exception` è sconsigliato). Per evitare situazioni non desiderate, **la miglior pratica consiste nell'ordinare i blocchi catch in modo da gestire sempre prima le eccezioni più specifiche e di non implementare mai due blocchi catch che gestiscono lo stesso tipo di eccezione**.

Dopo una sequenza di blocchi catch, è possibile inserire un **blocco finally, il cui codice verrà eseguito indipendentemente dal fatto che sia sollevata un'eccezione o meno** e può essere utile per **concludere alcune operazioni lasciate in sospeso quando viene sollevata un'eccezione** (come la chiusura di un file o di un terminale di input). **Implementando un costrutto try – catch – finally**, sono possibili tre situazioni:

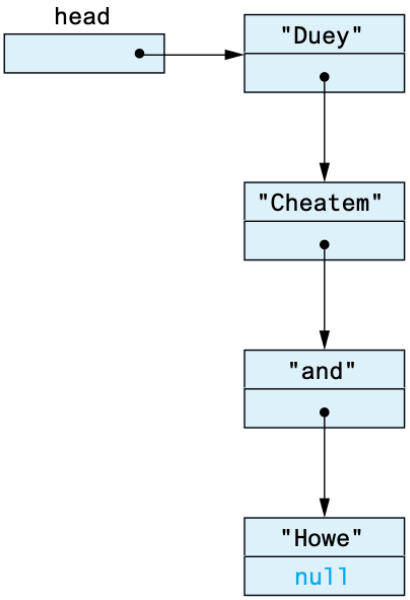
1. **Il blocco try viene eseguito fino alla fine**, non sollevando alcuna eccezione ed **eseguendo in coda il blocco finally**;

2. **Il blocco try solleva un'eccezione**, che viene **gestita da uno dei blocchi catch** implementati, ed **eseguendo in coda il blocco finally**;
3. **Il blocco try solleva un'eccezione**, di cui **non è prevista la gestione con uno dei blocchi catch** implementati, ed **eseguendo in coda il blocco finally** (in cui si può anche gestire tale eccezione "anomala", violando le convenzioni del buon codice).

Per concludere la trattazione delle eccezioni, si menziona la possibilità di **incorrere in un'eccezione all'interno del codice inserito in un blocco catch**; tuttavia, **la gestione di questa eccezione con un costrutto try – catch innestato è fortemente sconsigliata**.

LE LISTE A PUNTATORI

Le **liste a puntatori** sono delle **ADT dinamiche** che **descrivono una collezione di dati in maniera del tutto analoga ad un array**, con la sola differenza che **non ci si deve preoccupare della quantità di spazio da allocare** perché sono in grado di gestire alla necessità lo spazio da usare. Come il nome inglese suggerisce, linked list, **questo tipo di struttura è caratterizzata da una serie di oggetti** (che rappresentano i dati da collezionare), detti **nodi**, **collegati tra di loro attraverso i link**, **puntatori** con i quali è possibile raggiungere un altro nodo.



Il link `head` non è un nodo della lista ma è un semplice puntatore con cui è possibile accedere al primo elemento della collezione che, come sarà più chiaro in seguito, rappresenta il **punto di accesso all'intera struttura**.

In Java le liste a puntatori sfruttano una particolare proprietà degli oggetti, quella per cui **un oggetto istanziato è accessibile solo attraverso un riferimento** (un puntatore) **all'area di memoria in cui il record di attivazione del relativo oggetto è memorizzato**. Di conseguenza, **implementando i nodi come oggetti, si è in grado di accedere al dato senza necessitare variabili aggiuntive**.

In pratica, **un nodo sarà schematizzato attraverso una classe dotata di (almeno) due attributi**: un **dato** e un **riferimento**; **il dato sarà l'informazione da registrare in un elemento della collezione mentre il riferimento sarà quello del prossimo nodo**, anch'esso un oggetto dotato di dato e riferimento, che a sua volta sarà un oggetto, e così via.

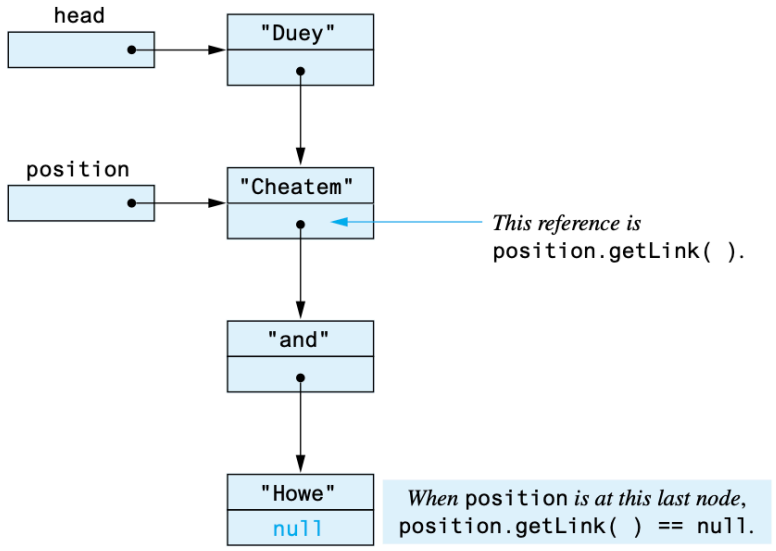
```
public class Node{
    String dato;
    Node riferimento;
}
```

Si noti la **definizione ricorsiva** della struttura, che permette di **ampliare la ricorsione da prerogativa dei metodi operativi alla definizione di ADT**.

Un oggetto nodo sarà istanziato o per essere un elemento della collezione generico (e quindi sarà puntato da un altro nodo e punterà ad un altro nodo, quindi riferimento non è nullo) **o per essere la testa della collezione** (e quindi non sarà puntato ma punterà ad un altro nodo) **o per essere l'elemento di coda di una lista** (e quindi sarà puntato ma non punterà ad un altro nodo). Tutte queste tre condizioni sono **rappresentabili dai due seguenti costruttori**:

```
public Node(){
    dato = "";
    riferimento = null;
}
public Node(String dato, Node riferimento){
    this.dato = dato;
    this.riferimento = riferimento;
}
```

Ciò che differenzia i due primi tipi di nodo è il tipo di oggetto che li punta: **se si parla di un nodo nel mezzo della lista, esso sarà puntato da un altro nodo, mentre se si parla di un nodo di testa della struttura, esso non sarà puntato da un nodo ma dall'head pointer**. Questo ultimo tipo di nodo è detto **nodo di head** ed è il **punto di accesso per l'intera struttura**; infatti, non disponendo dell'indirizzo di memoria di un elemento generico, **per accedervi si dovrà utilizzare un puntatore di scorrimento** che, a partire dal nodo di testa, preleva il riferimento contenuto nel nodo attuale e cambia il suo valore a quell'indirizzo, spostandosi nel nodo successivo fintantoché non si è giunti nella posizione desiderata.



Ad esempio, se si volesse accedere all'n – esimo elemento della lista, bisognerebbe eseguire n volte le seguenti operazioni:

```
scorrimento = scorrimento.riferimento;
```

In modo da modificare il valore del puntatore, contenente l'indirizzo a cui si sta puntando, con l'indirizzo contenuto nella struttura dati a cui si sta attualmente accedendo.

Si può già notare una **dicotomia tra gli array e le liste a puntatori**, individuando vantaggi e svantaggi per entrambe le strutture: **per gli array è scomodo gestire strutture dati dinamiche ma è comodo e rapido accedere ad un elemento generico della collezione** (basta sapere la sua posizione), mentre **per le liste è comodo gestire strutture dinamiche** (la dimensione è aggiornata alla necessità) ma **è scomodo accedere ad un elemento generico a causa del numero di operazioni da eseguire**. Soffermandosi sulle **funzioni di accesso**, **si noti che per gli array sono sempre costanti** ma **per le liste a puntatori sono lineari**, il tempo necessario per accedere ad un elemento della struttura è **dipendente dal numero di elementi totali della lista**; ciò è dovuto al fatto che **in un array i dati sono contigui**, disponendo del puntatore al primo elemento ho anche il puntatore a tutti gli altri (incremento di n per accedere all'n – esimo elemento), mentre **nella lista a puntatori i dati sono sparpagliati secondo criteri non disponibili all'esterno** e l'accesso ad un elemento dipende dal contenuto del nodo precedente.

Non bisogna usare il puntatore head come puntatore di scorrimento perché altrimenti si andrebbe a perdere il punto di accesso all'intera lista, perdendo anche i dati all'interno. L'operazione appena descritta è detta **operazione di scorrimento** e può essere eseguita **anche per scorrere tutta la lista**, come nel metodo seguente:

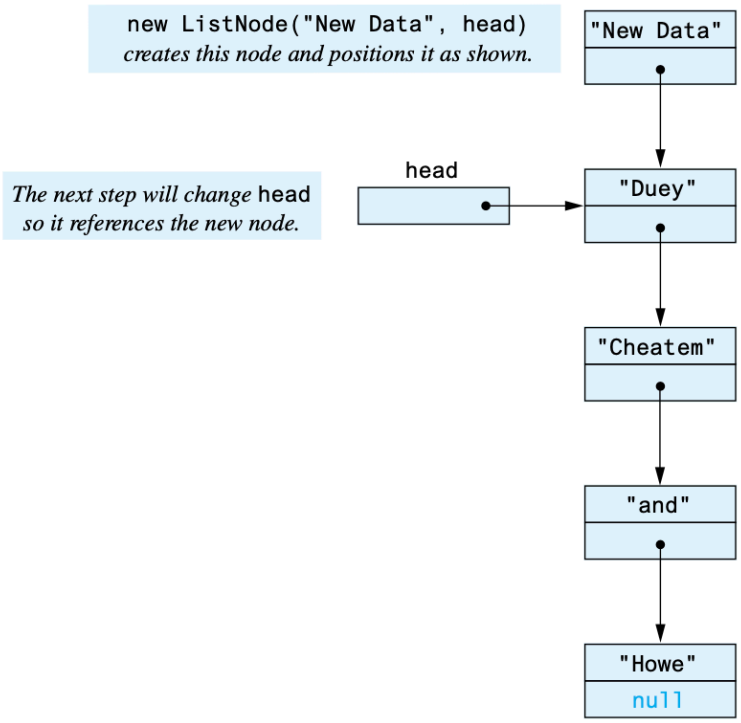
```
public void showList()
{
    ListNode position = head;
    while (position != null)
    {
        System.out.println(position.getData());
        position = position.getLink();
    }
}
```

In questo modo **si sono stampati tutti gli elementi della lista a schermo**. Ciò è possibile perché **si sa che il nodo di coda conterrà un puntatore nullo**. Per cercare un elemento contenente uno specifico dato all'interno della collezione si sfrutta un principio simile:

```
private ListNode find(String target)
{
    boolean found = false;
    ListNode position = head;
    while ((position != null) && !found)
    {
        String dataAtPosition = position.getData();
        if (dataAtPosition.equals(target))
            found = true;
        else
            position = position.getLink();
    }
    return position
}
```

Una particolare operazione che si può fare con le liste a puntatori (e che sarebbe troppo meccanica con gli array) è **l'inserimento in testa di un dato**. Per eseguire questa operazione, è necessario

istanziare un nuovo nodo, **inserirvi come riferimento il riferimento al primo elemento della lista** (quindi **`head`**) e **cambiare il valore dell'head reference**:



In maniera del tutto analoga, **per rimuovere la testa di una lista basta semplicemente modificare il contenuto del puntatore `head`**, inserendovi il riferimento al secondo elemento della lista, contenuto a sua volta nel nodo da puntato da `head`:

```
head = head.getCollegamento();
```

Non è necessario, a differenza di altri linguaggi di programmazione, **deallocare l'elemento che è stato eliminato** perché **Java dispone di un garbage collector** che osserva se una struttura in memoria è puntata da qualche puntatore; se la risposta è negativa, la struttura viene automaticamente eliminata.

Nell'usare le linked list si fa un estensivo uso dei puntatori. Tra i puntatori **è comune l'eccezione `NullPointerException`**, che occorre quan**do si sta cercando di accedere ad un'area di memoria puntata dal puntatore `null`** (detto null pointer). Un esempio di sollevamento di questa eccezione si ha **quando si vuole fare lo scorrimento oltre l'elemento di coda della lista** (quindi non inserendo il `while` nel blocco di scorrimento).

Per **privacy leak** si intende un fenomeno che **occorre al mancato rispetto dei principi di incapsulamento**, in seguito al quale **un dato che non dovrebbe essere accessibile è modificabile dall'esterno**. È stata proposta la soluzione del modificatore di accesso `private`, in accoppiamento ai getters e setters per l'accesso e la modifica sicura; quando si parla di **liste a puntatori**, e quindi di riferimenti e di memoria, **il privacy leak non può essere risolto semplicemente con i modificatori d'accesso**, dal momento in cui **i dati sono comunque modificabili da un oggetto esterno alla lista proprio attraverso i setters**. In particolare, **è necessario fare in modo che il riferimento al prossimo elemento non sia mai modificato se non dalla lista stessa**; altrimenti potrebbe accadere che l'oggetto che gestisce la lista modifichi per sbaglio il riferimento di un nodo e corrompa la

collezione di dati, mentre se è la lista stessa che raccoglie l'esigenza dell'oggetto che vuole gestirla si ha più controllo sulla struttura e si evitano situazioni scomode.

Nella pratica, **una soluzione potrebbe essere quella di usare una inner class**: si implementa **dapprima una classe `LinkedList`**, che **gestisce le operazioni da fare sull'intera struttura dati e le sue variabili di istanza**, per poi **inserirvi all'interno un'ulteriore classe**, **`Nodo`**, che **contiene le variabili di istanza e i metodi che servono per gestire il singolo nodo**. Un **oggetto esterno alla lista può gestire la lista**, accedendo alle sue variabili di istanza private con i getters e setters e ai suoi metodi con la relativa firma, **ma non può modificare in alcun modo il riferimento o il dato di un nodo se non passa attraverso la lista**, unica struttura che può accedere al nodo; in questo modo, **si escludono modifiche accidentali o corruzioni volontarie dei nodi di una lista**.

```
public class OuterClass
{
    Declarations_of_OuterClass_Instance_Variables
    Definitions_of_OuterClass_Methods
    private class InnerClass
    {
        Declarations_of_InnerClass_Instance_Variables
        Definitions_of_InnerClass_Methods
    }
}
```

Spesso, **lavorando con le liste a puntatori, è necessario accedere a specifici elementi nella collezione**, andando di conseguenza a **scorrere tutta la lista per arrivare in quella specifica posizione** e poi lavorare con i dati in essa contenuti; **un iteratore è una qualsiasi entità che permette operazioni di questo tipo**. Si procederà osservando prima un iteratore per un array, dal momento in cui è più semplice da realizzare e comprendere rispetto ad un iteratore per le liste a puntatori; infatti, gli iteratori per gli array sono già stati utilizzati e sono già familiari: **per scorrere un array non si fa altro che istanziare una variabile di tipo `int`** (`index`), **incrementarne il valore di uno ad ogni iterazione e definirne il massimo valore assumibile**, per poi **accedere all'elemento indicato dall'iteratore ad ogni iterazione** di un ciclo. Questo comportamento può essere codificato come segue:

```
for (index = 0; index < anArray.length; index++)
    Process anArray[index]
```

La variabile `index` è l'iteratore in questione e permette di **iterare lungo l'array**, ovvero di **andare da elemento a elemento incrementando la posizione dell'accesso** ad ogni iterazione di un ciclo.

Per quanto riguarda le linked list, **può essere utile anche qui andare ad iterare lungo la collezione per accedere ai singoli dati in essa contenuti**; un modo più semplice **per fare un'operazione di questo tipo è prelevare tutti i dati della lista a puntatori e portarli in un array**, di più semplice lavorazione e comprensione. **Il metodo responsabile di questa operazione potrà essere chiamato `toArray()` e sarà codificato come segue**:

```
public String[] toArray()
{
    String[] anArray = new String[length()];
    ListNode position = head;
    int i = 0;
    while (position != null)
    {
```

```
            anArray[i] = position.data;
            i++;
            position = position.link;
        }
        return anArray;
    }
```

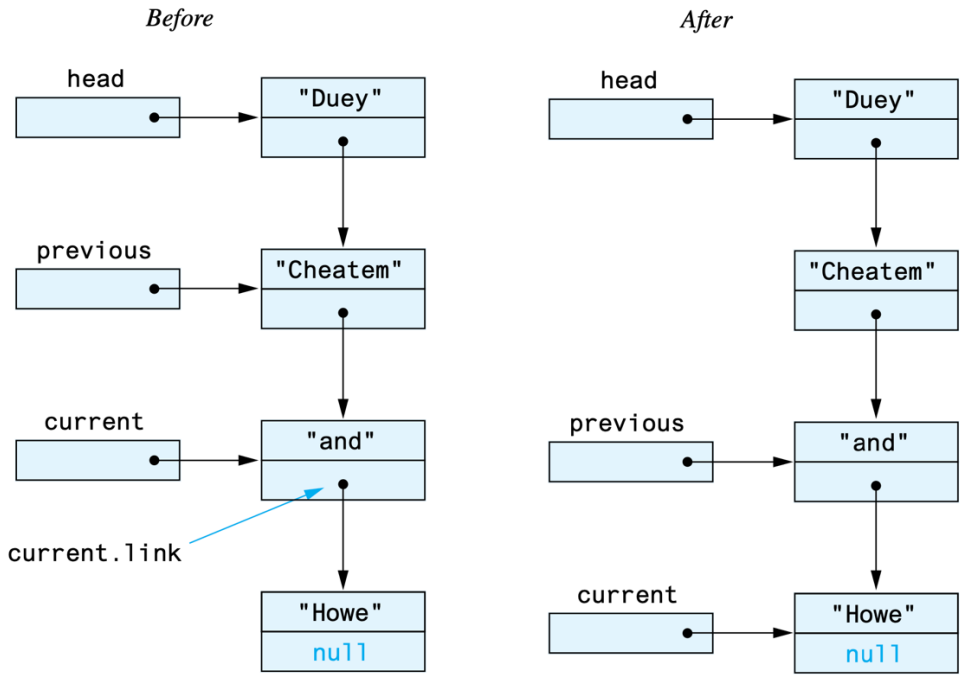
Tuttavia, **questa soluzione è adatta solo in caso si vogliano effettuare operazioni di lettura**, per scrittura o modifica dei nodi della lista non è sufficiente; in realtà gli iteratori degli array suggeriscono automaticamente il modo in cui procedere, poiché **sussiste una similitudine tra il rapporto di un indice e l'elemento a cui esso indica con il rapporto di un riferimento e il nodo a cui esso punta**. Pertanto, implementando nella classe che gestisce la lista a puntatori una variabile di tipo riferimento, sarà possibile realizzare un iteratore proprio come fatto per gli array.

Nel realizzare gli iteratori per le liste a puntatori, **è importante andare ad indicare quale verso gli iteratori stessi stanno percorrendo lungo la lista**; infatti, spesso risulta comodo e conveniente implementare **due iteratori per i due versi possibili della lista**, che sono convenzionalmente chiamati **`current` e `previous`**. Il modo in cui tali iteratori funzionano è leggermente diverso rispetto agli array ma tutto sommato vicino; **l'iteratore non fa altro che puntare ad una locazione di memoria in cui l'elemento della lista in esame è conservato e l'accesso al dato sarà realizzato andando a puntare con il suo riferimento** (che è l'iteratore) **la porzione del record di attivazione dedicata al dato**:

```
current.getDato();
```

Per spostarsi all'elemento successivo, con gli array si incrementava di un'unità l'iteratore; questo meccanismo non è più efficace nelle liste a puntatori, perché gli elementi di una stessa lista non sono contigui come in un array. Tuttavia, viene in aiuto la seconda variabile di istanza di un oggetto Nodo: il riferimento; **cambiando l'indirizzo puntato dall'iteratore al valore contenuto nella variabile `riferimento` del nodo puntato dall'iteratore stesso**, si permette a quest'ultimo di puntare all'elemento successivo della collezione:

```
current = current.getRiferimento();
```



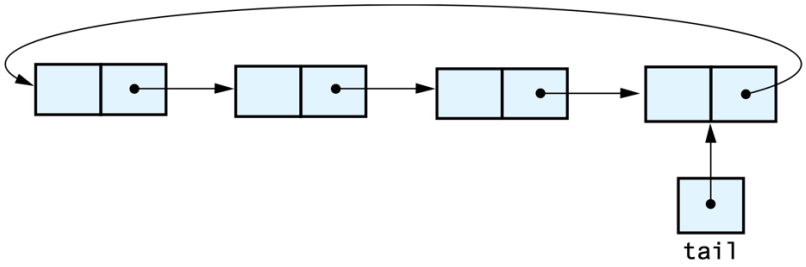
Esistono **diversi modi con cui gestire l'iteratore `previous`**, uno di questi prevede di **cambiargli il valore a quello di `current` prima che quest'ultimo sia cambiato**:

```
previous = current;
current = current.getRiferimento();
```

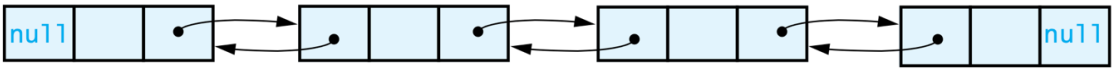
Java considera gli iteratori non come semplici variabili ma **come oggetti**, i cui metodi e le cui variabili di istanza sono raccolte nella **classe `Iterators`**; alcuni di questi metodi sono:

- `hasNext`, ritorna true se l'iterazione ha un altro elemento da ritornare;
- `next`, ritorna il prossimo elemento dell'iterazione;
- `remove`, rimuove dalla collezione l'ultimo elemento ritornato dal metodo `next()`.

La linked list è solo la più semplice delle strutture dati dinamiche che fanno uso del concetto di collegamento; una variazione utile in alcuni casi è quella di una **lista a puntatori circolare**, in cui **l'ultimo elemento della collezione** (la coda) **non punta a `null` ma a `head`** (la testa):



Come si può notare, **il puntatore `head` è coadiuvato da quello che punta alla coda della lista, detto `tail`**. Il puntatore alla coda non è una prerogativa di questa struttura, può essere necessario anche in una semplice linked list ma lo è ancora di più in una **doppia lista a puntatori**:



Questa struttura è caratterizzata da un **doppio verso di percorrenza**, **gli iteratori possono andare sia da destra verso sinistra che viceversa**; questa doppia faccia rende **puramente convenzionale le nomenclature `head` e `tail`** e prevede sempre **l'esistenza di almeno due iteratori**, uno `next` che scorra verso destra e uno `previous` verso sinistra. Infine, **di queste due ultime strutture particolari esiste una loro combinazione**, la **doppia lista a puntatori circolare**.

STRUTTURE DATI BASATE SULLE LISTE A PUNTATORI

Di seguito sono approfondite **strutture dati più complesse che fanno uso ampio delle liste a puntatori**.

PILA

Una pila (o stack in inglese) **è una ADT non necessariamente legata alle liste a puntatori ma spesso implementata come tale**. Essa è **gestita tramite una filosofia LIFO**, Last In First Out, in modo che **gli ultimi elementi inseriti nella collezione sono i primi ad essere rimossi**. Le **operazioni principali** che possono essere eseguite su una pila sono:

- `pop()`, rimuove l'elemento in coda dalla collezione;
- `push(dato)`, aggiunge in coda alla collezione un uovo elemento.

CODA

La coda (o queue in inglese) **è una ADT non necessariamente legata alle lista a puntatori ma spesso implementata come tale**. È molto **simile alla pila**, l'implementazione è quasi uguale, solo che **la filosofia con cui i dati sono gestiti è FIFO**, First In First Out, in modo che **gli ultimi elementi inseriti nella collezione siano gli ultimi ad essere rimossi**. Le operazioni principali che possono essere eseguite su una pila sono:

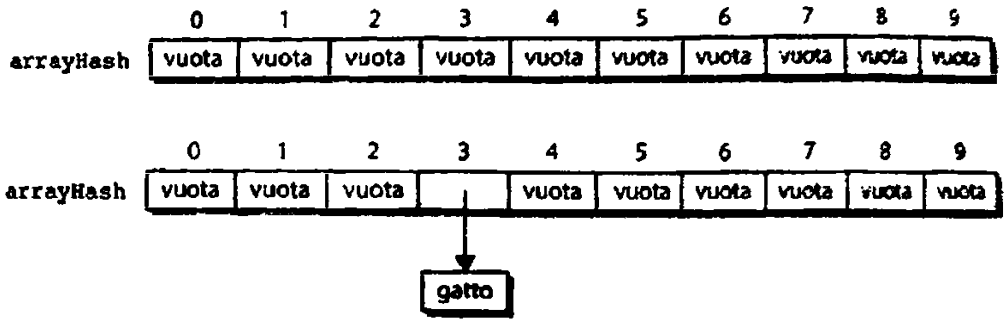
- `pop()`, rimuove il primo elemento in testa dalla collezione;
- `push(dato)`, aggiunge in testa alla collezione un uovo elemento.

TABELLE DI HASH

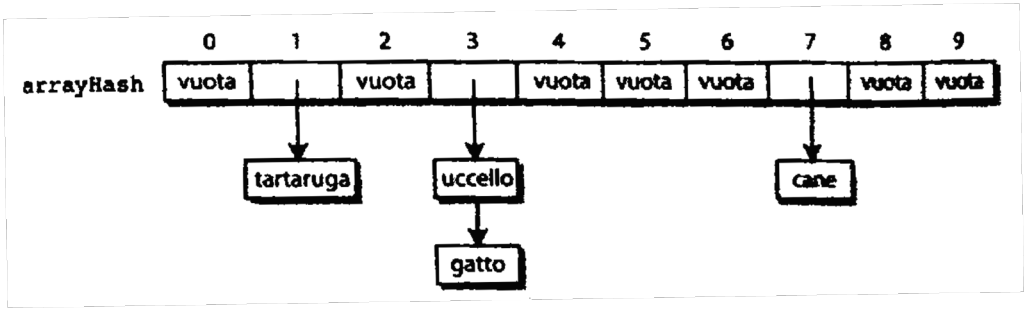
Una mappa (o tabella) **di hash è una struttura dati che organizza in maniera efficiente i dati in memoria**; in questa sede ne verrà affrontata una sola implementazione, quella che fa uso di liste a puntatori e array classici. **La comodità delle tabelle di hash risiede nel fatto che il tempo di accesso ad un dato non è più dipendente dal numero di elementi della collezione**, come per gli array e le liste a puntatori, ma potenzialmente costante e fissato ad un numero finito di operazioni (sono poche le situazioni in cui il tempo di accesso è lineare).

Per **memorizzare un dato in una tabella di hash** gli si **assegna una chiave**, **nota la quale è nota la posizione dell'elemento associato**; idealmente, **una chiave identifica univocamente un dato** e **nel caso in cui questo non possegga intrinsecamente una chiave**, glie ne sarà **associata una** tramite una **funzione di hash**.

Per comprendere il funzionamento di questa struttura dati, si faccia l'esempio di un dizionario: **ogni parola possibile sarà inserita nella mappa di hash**, mentre **ogni parola non contenuta nella struttura sarà scritta in maniera scorretta**. Per realizzare la mappa di hash del dizionario in esame, **si consideri un array statico di dieci posizioni**, ognuna delle quali **contenenti un riferimento ad una lista a puntatori, inizialmente vuota**. Quando si vuole **aggiungere un elemento**, **si calcola tramite la funzione di hash la sua chiave** (detta anche valore di hash), la quale andrà ad **indirizzare uno degli indici dell'array** e permetterà **il posizionamento del dato associato nella rispettiva lista concatenata**.



Si procede così per gli altri elementi, finché **la funzione di hash non restituisce una chiave già utilizzata da un altro elemento**; in questa situazione si verifica una **collisione**: **i due elementi dotati dello stesso valore di hash sono concatenati nella stessa lista**.



Per **cercare un elemento** specifico nella tabella di hash, **si produce il suo valore di hash**, che **indirizza ad una delle liste concatenate dell'array**, contenente tutti gli elementi con quel valore di hash; si andrà, poi, a **cercare l'elemento in quella lista concatenata** ma, nel caso in cui **la ricerca restituisca esito negativo**, **l'elemento non sarà presente nella tabella di hash**.

Un modo semplice per calcolare i valori di hash numerici di una stringa consiste nel sommare i codici ASCII di ogni carattere e calcolare il resto della divisione della somma per la lunghezza dell'array:

```
private int calcolaHash(String s){
    int hash = 0;
    for(int i = 0; i < s.lenght(); i++){
        hash +=s.charAt(i);
    }
    return hash % dimensione;
}
```

Nella pratica, **la lunghezza dell'array viene scelta pari ad un numero primo maggiore del numero di elementi che saranno immagazzinati nella tabella di hash**, evitando così che compaiano fattori comuni ed eventuali collisioni).

L'efficienza di una struttura dati del genere è da determinare sulla base di diversi fattori; in primis, **si individua il caso peggiore in quello corrispondente ad una tabella di hash dove tutti gli elementi sono associati allo stesso valore di hash**, andando a generare **sempre delle collisioni**. In tale situazione, tutti gli elementi della collezione si andranno a condensare in una sola lista concatenata, che non è facile da scorrere per trovare una corrispondenza. Viceversa, **il caso migliore occorre quando tutti gli elementi sono associati a chiavi diverse e l'operazione di ricerca si riduce ad un semplice accesso all'array**, perché non essendoci conflitti non ci sono liste concatenate a più di un nodo.

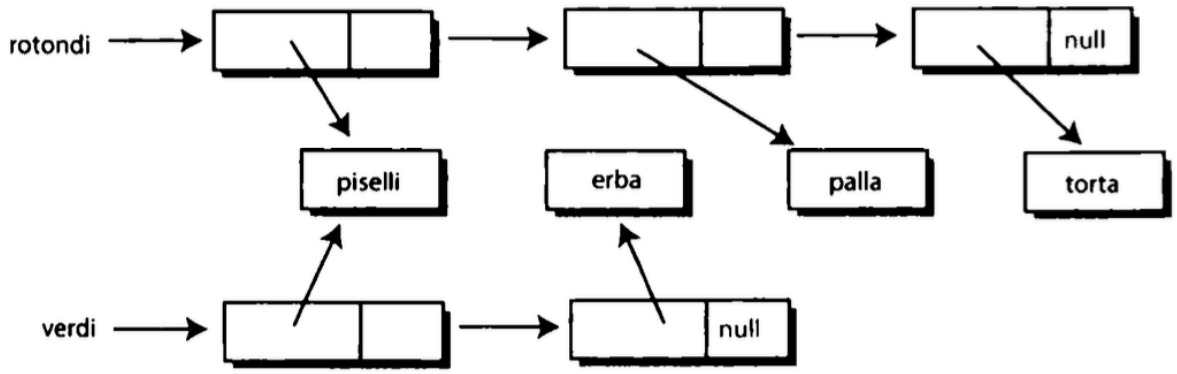
Si individua quindi una **correlazione tra numero di collisioni ed efficienza della struttura**; per **ridurre le collisioni** è necessario **migliorare la funzione di hash**, potendo impiegare diverse soluzioni, come quella riportata di seguito:

```
for(int i = 0; i < s.lenght(); i++){
    hash += 31* hash + s.charAt(i);
}
```

In modo da ridurre la probabilità che due stringhe abbiano lo stesso valore di hash. **La probabilità di collisioni è diminuita anche semplicemente aumentando la dimensione della tabella di hash**, quindi dell'array statico; tuttavia, una soluzione di questo tipo è poco conveniente per numeri relativamente bassi perché si va a sprecare molta memoria.

INSIEMI

Un insieme è una collezione di elementi dei quali si ignorano ordine e molteplicità. Un'implementazione di questa struttura **può partire da una lista concatenata ordinata** (ereditandone le operazioni fondamentali) che **rappresenta un insieme**; quando **vi si aggiunge un elemento che**, concettualmente **appartiene a più insiemi**, **esso viene referenziato da tutti gli insiemi in cui fa virtualmente parte**:



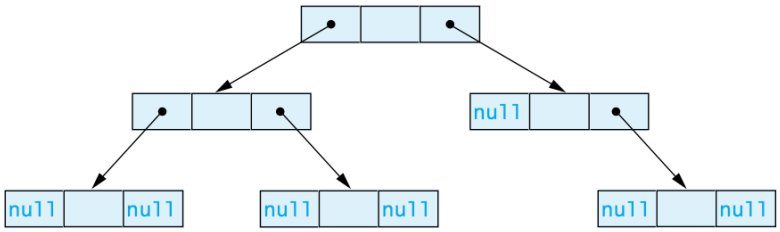
Su un insieme sono possibili le seguenti **operazioni**:

- **Aggiunta di un elemento**, con la verifica che esso non sia già presente nella collezione;
- **Verifica di appartenenza** ad un insieme;
- **Unione**, a partire da un insieme vuoto;
- **Intersezione**, a partire da un insieme vuoto.

Sia l'operazione di aggiunta che di verifica richiedono un numero di operazioni proporzionale alla cardinalità della struttura, mentre **l'unione e l'intersezione un numero di operazioni proporzionale alla somma delle dimensioni degli insiemi** presi in esame (nonostante ci siano diverse chiamate nascoste che portano la complessità a divenire quadratica); pertanto, si preferiranno altre strutture dati quando il numero di elementi della collezione sarà maggiore.

ALBERO

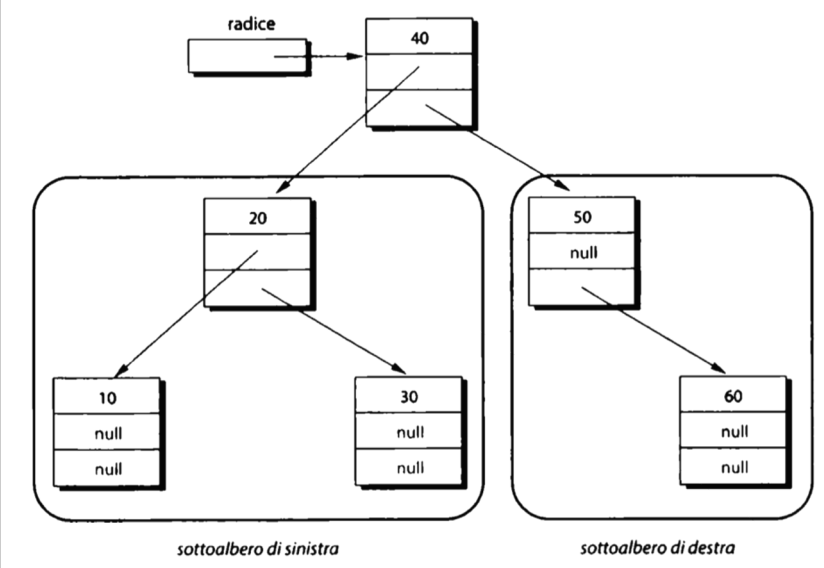
Un albero è una struttura dati concatenata di tipo più complesso rispetto a quelle viste finora. Si tratta di una **struttura molto utilizzata** e, pertanto, verrà approfondita in dettaglio, a partire dalla sua organizzazione:



In un albero, **qualunque nodo può essere raggiunto partendo dalla radice (root) lungo un percorso costituito da una sequenza di collegamenti (rami)**. Si noti che in un albero **non sono mai ammessi percorsi chiusi (cicli)**; seguendo i collegamenti, **si arriverà necessariamente ad una fine**, identificata da un **nodo diverso dalla radice (foglia)**. Un **albero** è detto **binario se i collegamenti possibili tra un nodo ed un altro sono sempre e solo due**.

Il nodo di riferimento, come è possibile intuire, **è la radice** e ne sarà creata una **variabile puntatore con un ruolo simile a quello di `head`** per le liste concatenate ordinate, mentre **i nodi foglia alla fine della collezione sono caratterizzati da un riferimento a `null` per ogni possibile collegamento**; quando **il nodo root e il nodo foglia coincidono**, si parla di **albero vuoto**, dal momento in cui laddove partono le diramazioni, lì finiscono e la struttura sarà caratterizzata da un solo nodo.

Gli **alberi** possono essere visti come delle **strutture ricorsive**: **ogni albero binario contiene due sottoalberi il cui root è collegato al root dell'albero originale**, ma anch'essi hanno a loro volta due sottoalberi ciascuno, e così via. Graficamente, questa relazione può essere apprezzata nella figura seguente:



La struttura ricorsiva di un albero lo rende **particolarmente incline ad algoritmi ricorsivi**, **limitandone la complessità** e mostrandolo come una s**truttura indicata per la gestione efficiente dei dati**. Si pensi, ad esempio, al problema della ricerca in un albero:

- **Elaborazione pre – ordine:**
  - Si elaborano i dati nel nodo radice;
  - Si elabora il sottoalbero sinistro;
  - Si elabora il sottoalbero destro.
- **Elaborazione in – ordine:**
  - Si elabora il sottoalbero sinistro;
  - Si elaborano i dati nel nodo radice
  - Si elabora il sottoalbero destro;
- **Elaborazione post – ordine:**
  - Si elabora il sottoalbero sinistro;
  - Si elabora il sottoalbero destro;
  - Si elaborano i dati nel nodo radice.

L'albero mostrato nell'ultima figura contiene i numeri immagazzinati in un modo particolare, che segue la **regola di immagazzinamento per la ricerca in alberi binari** (binary search tree storage rule), così riassunta:

1. **Tutti i valori nel sottoalbero sinistro sono minori del valore nel nodo radice**;
2. **Tutti i valori nel sottoalbero destro sono maggiori o uguali al valore nel nodo radice**;
3. La regola si applica ricorsivamente a ognuno dei due sottoalberi.

Un albero che segue la regola suddetta viene detto **albero di ricerca binaria**, o **albero binario ordinato**. In tutti questi algoritmi ricorsivi, **il caso base consiste in un albero vuoto**.

Si parlerà di **profondità** (o altezza) **dell'albero** per indicare **il massimo numero di nodi che intercorrono tra la radice ed una foglia**, tenendo in considerazione che **il root ha profondità 0 e un albero vuoto -1**; mentre **un albero binario è detto pieno o bilanciato** (full o balanced) quando **tutte le foglie hanno la stessa profondità** ed **ogni non – foglia ha due figli**.

In un albero più corto possibile (le lunghezze dei percorsi dal nodo radice a qualunque nodo foglia differiscono al più di una unità), **il metodo di ricerca binaria sono tanto efficienti quanto l'algoritmo di ricerca binaria in un array ordinato** ma ciò non dovrebbe sorprendere, visto che i due algoritmi sono molto simili. Nel **caso peggiore**, **l'algoritmo avrà complessità proporzionale al logaritmo del numero di nodi dell'albero** (visto che in un albero binario di profondità n il numero di nodi è pari a 2^n); ciò significa che **la ricerca in un albero binario corto è molto efficiente**. **L'efficienza si riduce man mano che l'albero diventa meno corto e largo e sempre più lungo e sottile**, finché nel **caso estremo** diventa **pari a quella della ricerca in una lista concatenata con lo stesso numero di nodi dell'albero**. Le **tecniche di gestione di un albero** volte a **mantenerlo corto e largo** (bilanciato) quando vengono aggiunti nuovi elementi vanno al di là della trattazione; qui ci si limiterà a segnalare il fatto che se gli elementi da immagazzinare nell'albero si presentano in maniera casuale, l'albero risulterà naturalmente sufficientemente bilanciato da presentare le proprietà di massima efficienza appena discusse.

HEAP

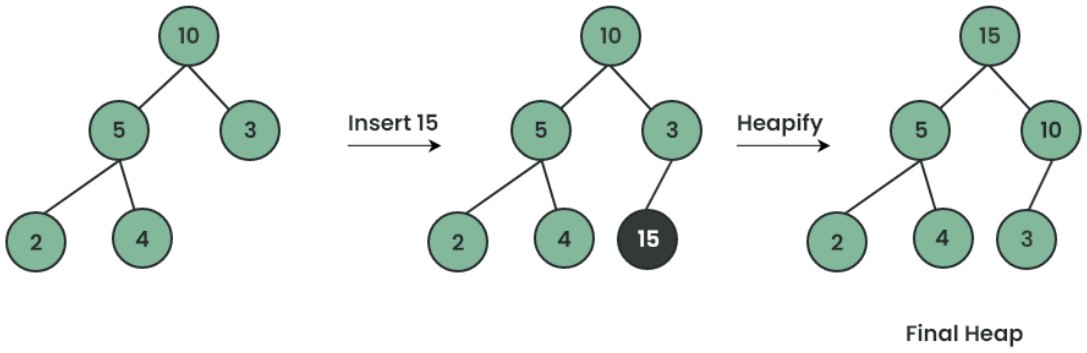
Una heap è un particolare tipo di albero binario, molto utilizzato per la gestione delle **strutture dati dotate di un certo grado di priorità** (quindi task del sistema operativo o ordinazioni per un ristorante) e perciò vengono chiamate anche **code con priorità**. In particolare, **per realizzare una heap è necessario costruire un albero binario completo**, quindi aggiungendo i nodi figli da sinistra verso destra fintantoché tutte le foglie non abbiano la stessa profondità.

In una heap, **i nodi non devono più essere descritti solo da un dato e dai riferimenti ai prossimi elementi ma è necessario che essi contengano anche una chiave di priorità** che può e dovrà essere comparata a quelle degli altri nodi. La chiave in questione, detta **"heap property"**, andrà a specificare **l'importanza che quel dato ha nella struttura in relazione agli altri**; pertanto, è necessario che sia **codificabile numericamente**, in modo tale da poter instaurare una relazione d'ordine con le altre chiavi di una stessa collezione. Si noti, però, che **la relazione d'ordine deve sussistere** (come per il collegamento negli alberi) **tra nodi padre – figlio, non tra nodi fratelli**.

Le operazioni che possono essere svolte su una heap non differiscono molto rispetto a quelle di un albero, sebbene occorrono delle modifiche legate alla strutturazione del concetto di priorità; come si è già avuto modo di intuire, **l'aggiunta di un nuovo elemento alla heap avviene seguendo il criterio degli alberi binari completi**: il nuovo nodo verrà aggiunto nel primo spazio disponibile, ovvero in modo tale che, essendo una foglia, sia alla stessa profondità (n) delle altre foglie, a meno ché non ci siano già 2^n foglie (in tal caso si aumenta la profondità e il nodo viene aggiunto in un nuovo livello).

Dopo l'aggiunta alla collezione, è necessario **ordinare il nuovo nodo in base al proprio livello di priorità**: sulla base della heap property, **il nuovo nodo è fatto risalire fintantoché il padre non ha una chiave numerica maggiore e il figlio una chiave numerica minore della propria**; il **caso peggiore** occorre quando **la chiave del nuovo nodo è maggiore della chiave della radice**, a quel punto **il nodo dovrà risalire tutto l'albero**, mentre il **caso migliore** occorre quando **la chiave del nuovo nodo è minore della chiave del padre** a cui è inizialmente legato. L'operazione appena

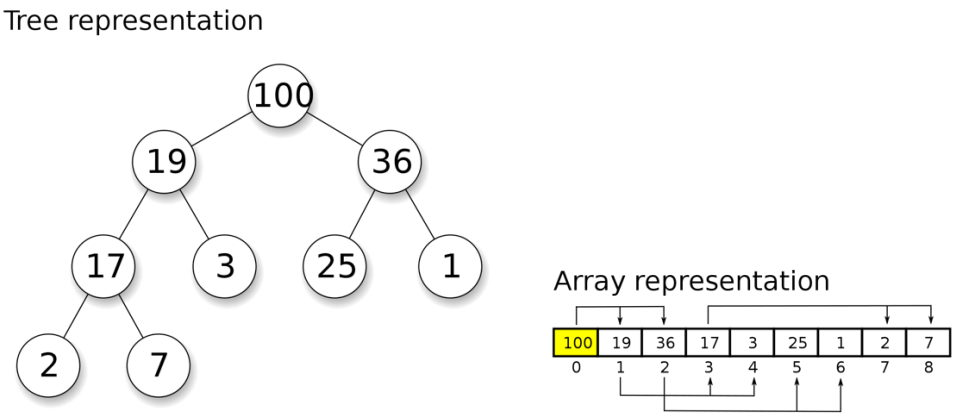
descritta è detta **ordinamento della heap** (o **reheapification upward**) e non è esclusiva per l'aggiunta di un elemento; infatti, si suppone che alla creazione di una heap, questa sia già ordinata ma, nel caso non lo fosse, sarebbe possibile eseguire in qualsiasi momento l'ordinamento della struttura per ogni singolo elemento.



In aggiunta alle due operazioni descritte (che sono abbastanza classiche, nonostante prevedano qualche accorgimento in più rispetto a quelle di un albero), **le heap presentano un'operazione diegetica**: la **rimozione della testa** (o reheapification downward). Poiché **le heap rappresentano strutture dotate del concetto di priorità**, esse andranno a **descrivere una situazione per la quale una determinata task va eseguita prima di un'altra**; una volta **assolto il compito di un nodo**, è inutile che esso venga conservato nella collezione e, pertanto, **può essere rimosso**, facendo attenzione a **porre in cima alla struttura l'elemento che ha la priorità più alta** dopo quella del nodo appena eseguito.

Per eseguire il minor numero di operazioni possibili, **non si pone in cima alla lista già il nuovo elemento a priorità maggiore** (si dovrebbe prima determinare e poi spostare ogni singolo nodo più in alto facendo attenzione al completamento dell'albero) ma si pone, invece, **l'ultimo aggiunto** (visto che non ha figli e visto che la sua rimozione non intacca la completezza dell'albero). Poiché **quasi mai questa nuova testa sarà il nodo a maggior priorità**, si effettua un'operazione di **ordinamento inverso** per la **quale il nodo in questione verrà fatto scendere fintantoché non avrà un padre a priorità più grande e un figlio a priorità minore della sua**. Automaticamente, **alla testa della nuova struttura si troverà l'elemento a maggior priorità** ed essa rappresenterà nuovamente una **heap corretta**.

Spesso **conviene rappresentare le heap anche in strutture monodimensionali**, come un array contenente tutte le priorità dei nodi corrispondenti. In primis, **si determina la massima profondità della heap** (n), in modo da poter **inizializzare un array di** $\sum_{i=1}^{n} 2^i$ **elementi** (essendo la heap un albero binario completo, si suppone che a profondità n il numero massimo di foglie sia 2^n e il minimo $2^{n-1} + 1$; per ridondanza, si preferisce sempre allocare un po' di spazio in più). Nell'array, **gli elementi vengono aggiunti non seguendo nessuno dei tre criteri di elaborazione degli alberi**, bensì un tipo di **elaborazione a livelli**: **viene aggiunta la radice**, poi vengono **aggiunti tutti i nodi del primo livello** (da **sinistra verso destra**), **poi quelli del secondo livello, e così via**… **L'array completo dovrà contenere nelle prime** 2^0 **posizioni gli elementi del livello zero**, **nelle successive** 2^1 **quelli del primo**, **e così via** fino ad avere **nelle ultime** 2^n **tutte le foglie della heap**.



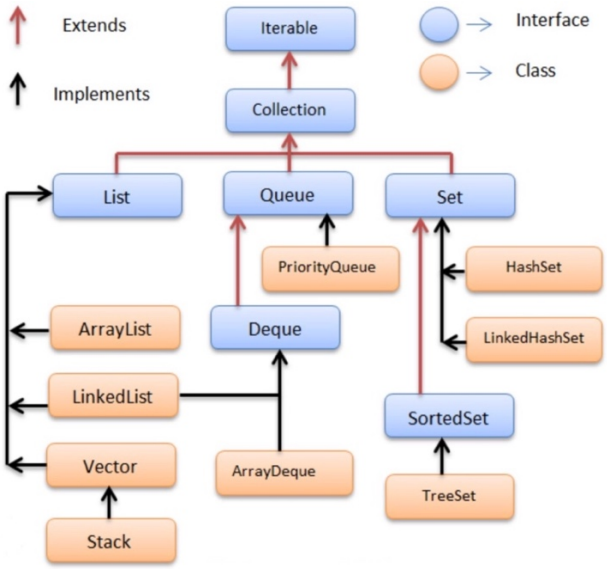
Tra una heap e l'array delle priorità sussiste una relazione biunivoca proprio in seno a questo tipo di ordinamento; infatti, **conoscendo un array si può ricostruire la relativa heap semplicemente ripercorrendo i procedimenti appena descritti al contrario**. Se si fosse adottato un ordinamento pre – order, in – order o post – order sarebbe stato molto più difficile risalire alla struttura della heap senza specificare i collegamenti tra i vari nodi.

COLLEZIONI, MAPPE E ITERATORI

Una collezione è una particolare struttura dati che contiene degli elementi, come un oggetto `ArrayList<T>`; Java dispone di una grande quantità di interfacce e classi che coprono in modo esteso il tema delle collezioni

In particolare, **una collezione è una qualsiasi classe che implementi l'interfaccia `Collection<T>`** la quale, come sarà più chiaro a breve, **può essere anche usata come struttura dati predefinita**; infatti, **quelle definite nei precedenti capitoli sono una particolarizzazione di questo concetto**. L'interfaccia `Collection<T>` consente di scrivere codice che può essere applicato a tutte le collezioni Java, così che non debba essere necessariamente riscritto nulla per una specifica collezione.

Sono molte le collezioni predefinite che già implementano l'interfaccia `Collection<T>`, di seguito elencate:



Il concetto di collezione è strettamente legato a quello di iteratore, essendo questo il mezzo con cui ci si districa in una struttura dati, ma il tema verrà approfondito a tempo debito. Prima di discutere in maniera più accurata della collezione `Collection<T>`, **si faccia una piccola digressione sul modo di specificare il tipo dei parametri**.

Le classi e le interfacce della libreria delle collezioni Java utilizzano una modalità di specifica del tipo di parametro particolare, con la quale è possibile specificare cose come **"L'argomento deve essere un `ArrayList<T>` ma può avere qualunque tipo base"**; più in generale, questa modalità di specifica del tipo di parametri **usa i tipi generici senza specificare completamente il tipo di oggetto da sostituire al tipo di parametro** e sono **note come wildcards**. Il wildcard più semplice è **`<?>`** ed **indica che può essere usato qualunque tipo al posto del tipo di parametro**:

```
public void metodoEsempio(String arg1, ArrayList<?> arg2)
```

Il metodo in questione viene invocato passando due argomenti: il primo deve essere di tipo String, mentre **il secondo deve essere un `ArrayList<T>` con qualunque tipo base**.

Si noti che **`ArrayList<?>` e `ArrayList<Object>` sono diversi**, il primo permette di passare come argomento un oggetto `ArrayList<String>`, mentre il secondo richiede necessariamente un `ArrayList<Object>` (nonostante si sappia che `String` è un `Object`). **È possibile limitare una wildcard al passaggio di un tipo che sia antenato o discendente di una qualsiasi classe o interfaccia**:

```
ArrayList<? extends/super className/interfaccia>
```

Essenzialmente, l**a differenza tra wildcards e generici risiede nel fatto questi ultimi sono utilizzati per definire classi, interfacce e metodi con tipi di parametri**, mentre **le wildcards sono utilizzate per dichiarare il tipo di variabili che possono contenere tipi generici sconosciuti**.

`Collection<T>` costituisce l'interfaccia più generica della libreria Java che contiene le classi per le collezioni e descrive le **operazioni di base** che devono essere implementate da tutte le classi di tipo collezione (si faccia riferimento alla documentazione ufficiale per approfondimenti). Poiché **un'interfaccia è un tipo**, è possibile **definire metodi con parametri di tipo `Collection<T>`**, con la possibilità di essere sostituiti con argomenti che siano istanza di qualunque classe collezione. La proprietà appena enunciata risulta molto potente, sapendo che **si possono implementare metodi in una collezione che lavorano su collezioni diverse**, purché implementino la stessa interfaccia `Collection<T>`; a**d esempio, nella classe `ArrayList<T>` può essere implementato il seguente metodo**:

```
public boolean containsAll(Collection<?> collezioneDiObiettivi)
```

E può essere **usato da due oggetti `ArrayList<T>`** (senza che il tipo base sia necessariamente lo stesso) **o da un `ArrayList<T>` e un'istanza di una qualsiasi classe che implementi `Collection<T>`**.

Come è possibile notare dalla figura precedente, **le principali interfacce che implementano `Collection<T>` sono due**: **`Set<T>` e `List<T>`**; **le classi che implementano `Set<T>` non ammettono elementi ripetuti**, mentre **quelle che implementano `List<T>`** (come `ArrayList<T>`) **ordinano i loro elementi in una lista** (così che esista un indice 0, un indice 1, e così via…) **ammettendo la possibilità di elementi ripetuti**. L'interfaccia **`Set<T>` ha le stesse intestazioni dei metodi dell'interfaccia `Collection<T>` ma in alcuni casi la semantica è**

diversa (ad esempio, i metodi per l'aggiunta di nuovi elementi all'insieme non consente duplicati); **`List<T>`**, invece, **ha più metodi dell'interfaccia `Collection<T>` ed alcuni di quelli in comune hanno una semantica diversa** (ad esempio, i metodi per l'aggiunta di nuovi elementi all'insieme devono disporre di eventuali criteri per individuare quale tra due duplicati mantenere e quale rimuovere).

Per semplificare l'implementazione di queste due interfacce, **sono fornite le classi astratte `AbstractSet<T>` e `AbstractList<T>`**, **con** (quasi esclusivamente) **i metodi richiesti dalle interfacce che implementano**. Nonostante siano pochi i metodi astratti in esse contenuti, gli altri hanno implementazioni pressoché inutili e richiedono generalmente la ridefinizione; pertanto, **spesso ha più senso utilizzare `HashSet<T>`, `ArrayList<T>` o `Vector<T>`**, comunque **implementate a partire da `AbstractSet<T>` e `AbstractList<T>`** (rappresentano, infatti, **implementazioni complete delle interfacce `Set<T>` e `List<T>`**). In maniera del tutto analoga, **si può considerare l'uso della classe astratta `AbstractCollection<T>`,** implementazione di `Collection<T>`, sebbene **raramente sarà necessario ricorrere a tale strumento**.

Se si necessitasse di una classe che implementi `Set<T>` senza alcun metodo oltre quelli dell'interfaccia, si potrebbe **utilizzare la classe concreta `HashSet<T>`** precedentemente menzionata; la parola hash si riferisce al fatto che la classe in questione è implementata utilizzando una tabella di hash. Per approfondire le implementazioni dei metodi di questa classe si può ricorrere alla documentazione ufficiale. Chiaramente, **se si utilizza una classe propria al posto del parametro `T`, è necessario ridefinire i due metodi seguenti**:

```
public int hashCode();

public boolean equals(Object obj);
```

Il primo restituirà una chiave numerica che, idealmente, **rappresenta un identificativo univoco per un oggetto della classe**, mentre per quanto riguarda il secondo, **la ridefinizione qui non rappresenta più una buona pratica ma una necessità**, dal momento in cui tale metodo **è usato per verificare se l'oggetto che si sta cercando di inserire è già nella collezione** (operazione essenziale per un `Set<T>`). Nel caso in cui due oggetti venissero associati alla stessa chiave di hash, infatti, bisognerebbe comunque verificarne l'uguaglianza, in modo tale da discriminarne l'appartenenza alla collezione sotto forma di collisione o no.

L'utilità di un'interfaccia è quella di specificare quali metodi possano essere utilizzati su un oggetto del tipo dell'interfaccia, in modo da poter **scrivere codice valido per oggetti arbitrari di quel tipo**; l'idea alla base dei **metodi opzionali** è che **normalmente questi vengano implementati**, anche se in situazioni particolari il programmatore potrebbe **lasciarli come "non supportati"** gestendone un eccezione **`UnsupportedOperationException`** (essendo derivata da `RuntimeException`, è un'eccezione non controllata). Quindi, **si può ben intuire come i metodi opzionali non siano davvero opzionali**, è comunque necessario specificare un corpo per quel metodo, sebbene **possa essere scelta un'implementazione banale o eccezionale**.

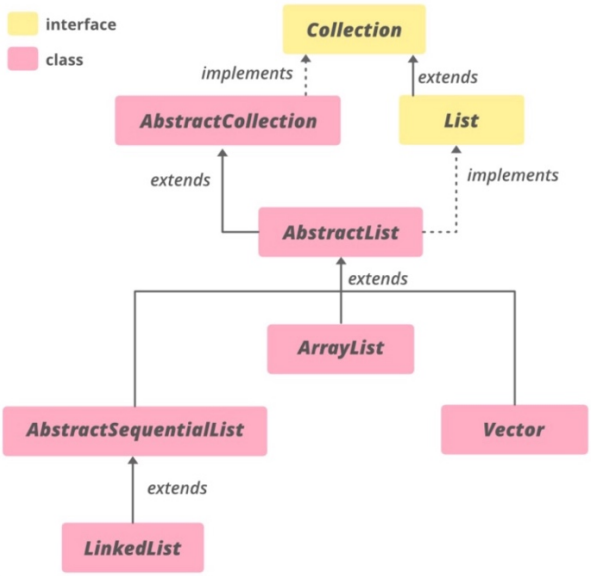
Generalmente, **è sempre preferibile usare le classi predefinite di tipo collezione**, salvo che queste non forniscano tutte le funzionalità necessarie. Per esempio, si supponga di volere che ogni elemento aggiunto ad un insieme contenga un riferimento all'insieme stesso; ciò potrebbe essere utile per determinare, ad esempio, se due elementi appartengono allo stesso insieme: basterebbe confrontare i riferimenti agli insiemi contenuti negli elementi stessi. Senza un implementazione di questo tipo, andrebbe chiamato ogni volta il metodo `contains`, con il quale sarebbero sprecate molte più risorse. In questi casi **sarebbe più comodo e pratico progettare una classe personalizzata**, invece di usare

quelle predefinite; in caso contrario, **si prediliga la scelta delle classi predefinite e non si reinventi la ruota**.

In maniera del tutto analoga, **se si necessitasse di una classe che implementi `List<T>` senza alcun metodo oltre quelli dell'interfaccia, si potrebbe utilizzare o la classe concreta `ArrayList<T>` o `Vector<T>`**. Inoltre, si può ricorrere alla classe astratta `AbstractSequentialList<T>`, derivata dalla classe `AbstractList<T>` (sebbene non aggiunga nuovi metodi), e pensata con lo scopo di **fornire un'implementazione efficiente degli spostamenti sequenziali lungo una lista** ma **al costo di un inefficiente accesso diretto agli elementi** (metodo `get()`); da questa, deriva anche la classe concreta `LinkedList<T>`, ereditandone tutti i pregi e i difetti. In definitiva, **quando è necessario un accesso diretto efficiente agli elementi, conviene l'utilizzo di `ArrayList<T>` o di `Vector<T>`**, mentre **se è necessario convogliare l'efficienza sullo spostamento sequenziale** lungo la lista, **conviene `LinkedList<T>` o una sua derivata**.

Si vuole, adesso, fare una digressione sulle differenze tra `ArrayList<T>` e `Vector<T>`, che all'apparenza possono sembrare uguali, soprattutto per molti utilizzi; infatti, **i punti di distacco** tra le due classi sono limitati a **pochi metodi non presenti in `ArrayList<T>` ma solo in `Vector<T>`**. La maggior parte di questi metodi, tuttavia, **corrisponde principalmente a nomi alternativi per metodi presenti in entrambe le classi**. In genere, però, **si preferisce usare `ArrayList<T>`**, visto che è **considerata più efficiente** della sua controparte gemella; il motivo risiede nel fatto che **è stata creata più recentemente** e come parte della libreria Java delle collezioni, **mentre `Vector<T>` è più vecchia** (sebbene sia stata riadattata con l'aggiunta di nuovi metodi per renderla compatibile con la libreria di collezioni). **Il motivo per cui `Vector<T>` non è ancora in disuso è dovuto al fatto che molto del codice già esistente non è stato scritto con `ArrayList<T>`, perché non ancora creato**.

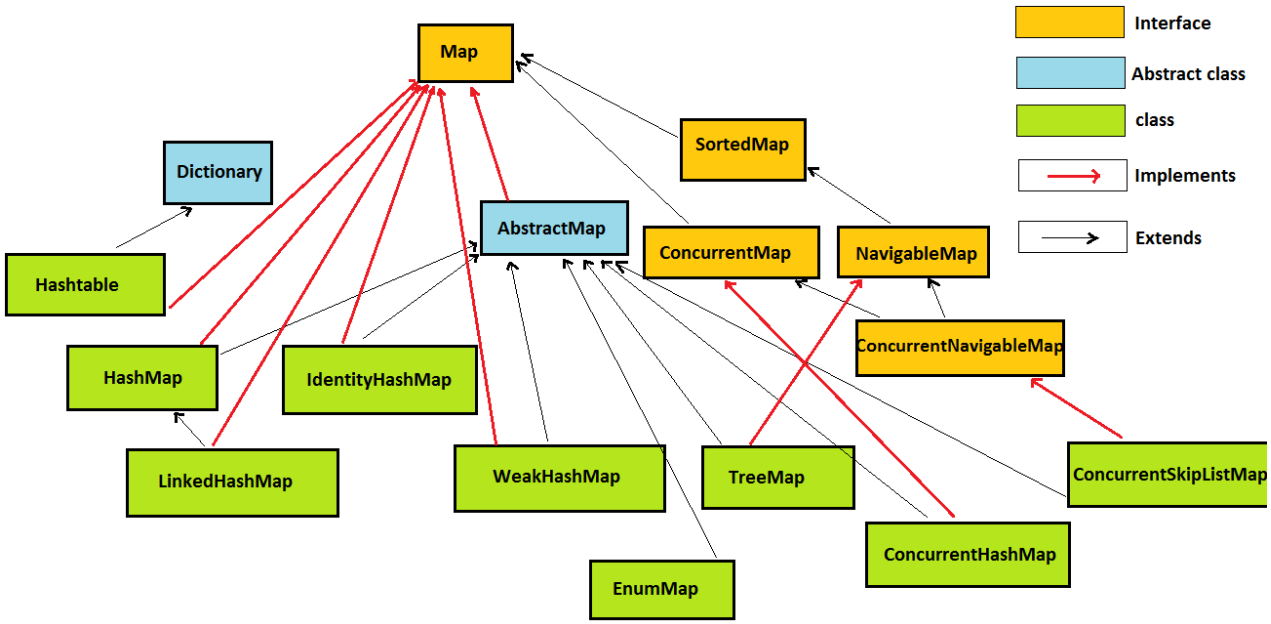
Infine, **si menzioni la possibilità di implementare `Set<T>` in modo da disporre di un'efficiente estrazione degli elementi** con l'interfaccia `SortedSet<T>` o con la classe concreta `TreeSet<T>`.



Prima della versione 5.0, Java non disponeva di tipi parametrici e la libreria delle collezioni consisteva di classi ed interfacce ordinarie, come `Collection`, `List`, `ArrayList` e così via… Nonostante questa vecchia libreria sia stata soppiantata dalla nuova versione, **le classi e le interfacce**

in questione sono ancora presenti nelle librerie standard e in molto del codice già esistente. Poiché non si dispone di uno strumento in grado di differenziare la singola classe o interfaccia sulla base del tipo di dato su cui si sta lavorando, **la libreria delle collezioni priva di tipi parametrici risulta molto scomoda e difficile da usare** e, pertanto, **è sconsigliabile approcciarla**; nel caso in cui la si incontri, **si tenga in considerazione che non è troppo sbagliato approssimare `Collection` a `Collection<T>` (e così via…)**. Proprio perché si sta avviando al disuso, **i moderni compilatori di Java potrebbero interpretare l'omissione del generico `<T>` come un errore di compilazione**; per evitare errori di questo tipo, **possono venire in aiuto i warning segnalati dal compilatore stesso, andando poi a sostituire il codice obsoleto con quello relativo alla libreria moderna**.

La libreria Java delle mappe è molto simile a quella delle collezioni, ad eccezione del fatto che **lavora su collezioni di coppie ordinate**; pertanto, **gli oggetti della libreria delle mappe possono implementare funzioni e relazioni matematiche e**, quindi, **costruire classi per la gestione di basi di dati** (si pensi ad una relazione chiave – valore associato di una matricola ed il relativo studente). Di seguito sono elencate le principali interfacce e classi di questa libreria, sapendo che ci si concentrerà perlopiù su `Map<K,V>`, `AbstractMap<K,V>` e `HashMap<K,V>`:



Come si è potuto intuire, **in seno all'associazione chiave – valore, per la libreria delle mappe è necessario specificare due tipi di parametro** (per le collezioni ne era sufficiente uno): **`K` per la chiave (key) e `V` per il valore (value)**.

La classe astratta **`AbstractMap<K,V>` è comoda se si vuole implementare l'interfaccia `Map<K,V>`**, esattamente come `AbstractSet<K,V>` lo era per `Set<K,V>`. Definendo una classe derivata da `AbstractMap<K,V>`, **sarà necessario definire non solo i metodi astratti, ma anche tutti gli altri metodi che si intendono utilizzare**; di solito, **ha più senso usare** (o definire classi derivate da) **`HashMap<K,V>` o `TreeMap<K,V>`**, derivate sempre da `AbstractMap<K,V>` ma **implementazioni complete dell'interfaccia `Map<K,V>`** (sempre che non si voglia una struttura completamente personalizzata).

In questa sede ci si concentrerà perlopiù sulla classe **`HashMap<K,V>`**, che, come già detto, è un'implementazione completa dell'interfaccia `Map<K,V>`. **La struttura indotta da questa classe non fornisce alcuna garanzia sull'ordine degli elementi contenuti nella mappa**; se si necessita di una **struttura ordinata** è preferibile **ricorrere alla classe `TreeMap<K,V>`** (che organizza

internamente i propri elementi in una struttura ad albero) **o a `LinkedHashMap<K,V>`** (che usa una lista a doppio concatenamento per mantenere l'ordine in un oggetto `HashMap<K,V>`, da cui è derivata). Come si può facilmente intuire, **la classe `HashMap<K,V>` utilizza internamente una tabella di hash**, della quale è utile conoscere il criterio operativo in modo da ottimizzarne l'utilizzo; infatti, in maniera del tutto analoga a quanto detto riguardo le tabelle di hash nei precedenti capitoli, **andrebbe ricercato un simile compromesso tra dimensione della tabella e numero di elementi inseriti anche quando si utilizza la classe `HashMap<K,V>`**. Uno dei **costruttori della classe** permette di **specificare anche una capacità iniziale ed un fattore di carico**: il primo parametro indica **quanti "blocchi" esistono nella tabella di hash**, mentre il secondo è un **numero compreso tra 0 e 1 che specifica una percentuale tale per cui**, se il numero di elementi aggiunti alla tabella di hash supera questo fattore di carico, **la capacità della tabella viene incrementata automaticamente**. Ad esempio, se il fattore di carico è 0.75 e la capacità iniziale è 16, ogni 12 elementi aggiunti alla mappa verrà automaticamente incrementata (all'incirca del doppio) la capacità della tabella; questo processo è detto **rehashing** e **può richiedere anche un tempo non trascurabile** per mappe con molti elementi. Nonostante l'incremento della capacità sia automatico, **un programma verrà eseguito in modo più efficiente se la capacità iniziale è impostata al numero di elementi che ci si aspetta vengano aggiunti alla mappa**, in modo che **il numero di questi incrementi sia minimizzato**.

Come già ripetuto, la classe `HashMap<K,V>` implementa tutti i metodi dell'interfaccia `Map<K,V>`, non aggiungendone di nuovi se non per i costruttori e per il metodo `clone()`. Proprio come detto per `HashSet<T>`, **se si utilizza una classe propria come tipo parametrico `K` in `HashMap<K,V>`, la classe dovrà ridefinire i due seguenti metodi** (necessari per indicizzare e confrontare le chiavi):

```
public int hashCode();

public boolean equals(Object obj);
```

Un iteratore è un oggetto utilizzato insieme ad una collezione per fornire un accesso sequenziale agli elementi della collezione stessa, con la possibilità di esaminare ed eventualmente modificare il dato al quale si è fatto l'accesso. Si può, quindi, intuire che **l'utilizzo degli iteratori si limita ad un ordinamento degli elementi della collezione, anche quando questa non impone alcun ordine sui propri elementi**. A rigore, non si è ancora incontrato alcun iteratore, sebbene sia possibile **individuare nelle variabili di scorrimento indice degli array uno strumento il cui obbiettivo soddisfa l'idea intuitiva di iteratore** (visto che permette l'accesso sequenziale agli elementi di un array).

Java formalizza il concetto di iteratore per mezzo dell'interfaccia `Iterator<T>` e, pertanto, qualunque oggetto di qualunque classe che implementi questa interfaccia è un iteratore (per questo motivo **le variabili di scorrimento indice degli array non sono iteratori in sé e per sé**). Ovviamente, **un iteratore in quanto tale non ha senso se non è associato ad una collezione**; questa associazione **avviene tramite il metodo `iterator()`**, **necessario per qualunque classe che implementi l'interfaccia `Collection<T>`**, e **restituisce un `Iterator<T>`**.

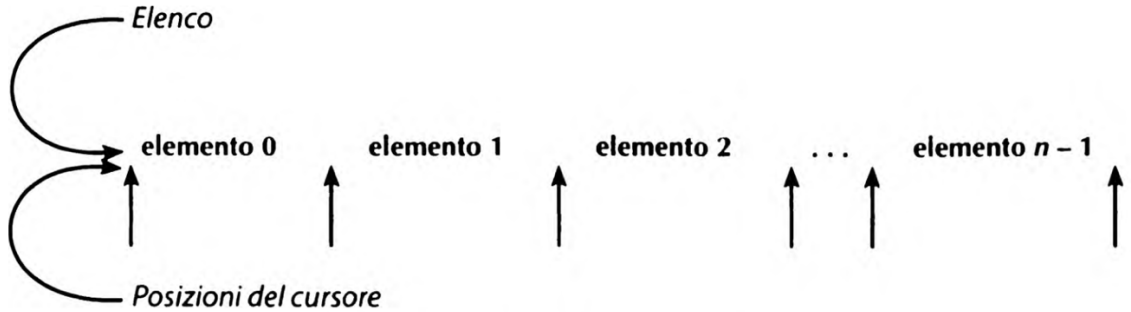
Si supponga di **lavorare con una collezione che non impone ai propri elementi un particolare ordine**, come `HashSet<T>`; **un iteratore associato ad un oggetto istanza di questa collezione andrà ad imporre internamente un ordine alla struttura**, sulla base dell'implementazione del proprio metodo `next()`, senza alcun vincolo sull'ordinamento. **Non è un errore se lo stesso iteratore in due esecuzioni diverse dello stesso programma restituisce gli elementi di una stessa collezione in due ordini diversi**; tuttavia, **se la collezione dispone intrinsecamente di un**

ordinamento, l'iteratore non ne può prescindere e dovrà restituire gli elementi sempre seguendo quello stesso ordine.

Non sempre gli iteratori sono la scelta migliore, ovviamente ci sono situazioni in cui questi sono indicati (se non necessari) ma **può accadere piuttosto spesso che sia più efficiente la gestione della collezione con un semplice for – each** (questa particolarità è alla base del motivo per cui si studiano ancora e sono ancora utilizzati i finti iteratori degli array); tuttavia, **quando si lavora con un ciclo for – each al posto di un iteratore, è necessario adattare lo strumento a tutte le operazioni che un iteratore proprio può eseguire su quella collezione**.

La libreria delle collezioni dispone di due interfacce di tipo iteratore: quella già menzionata **`Iterator<T>`**, che può essere **usata con qualunque classe implementi l'interfaccia `Collection<T>`**, e **`ListIterator<T>`**, progettata per **lavorare con le collezioni che implementano l'interfaccia `List<T>`**. Come si può facilmente intuire, **`ListIterator<T>` estende `Iterator<T>` ed ha tutti i suoi metodi, più altri che abilitano nuove funzionalità**: un `ListIterator<T>` può muoversi lungo la lista degli elementi della collezione in entrambe le direzioni e offre metodi (come `set()` e `add()`) che possono essere utilizzati per modificare gli elementi della collezione.

La libreria delle mappe non supporta direttamente le interfacce per gli iteratori, sebbene possano essere usati i **metodi `keySet()`, `values()` e `entrySet()` per ottenere collezioni iterabili** contenenti, rispettivamente, le chiavi, i valori e le coppie (chiave, valore) di una mappa. L'idea appena mostrata è molto efficace ma deve essere approfondita per poter, poi, comprendere il funzionamento dei **metodi `next()` e `previous()` dell'interfaccia `ListIterator<T>`**; ogni iteratore di questo tipo ha un **indicatore di posizione nella lista noto come cursore**: se la lista contiene **n elementi**, questi sono **numerati da 0 a $n-1$ ma ci sono $n+1$ posizioni possibili per il cursore**. Infatti:



La posizione iniziale di default per il cursore è quella più a sinistra.

Quando viene chiamato `next()`, viene **restituito l'elemento immediatamente successivo** alla posizione del cursore e quest'ultimo è **spostato in avanti nella posizione successiva**; **quando**, invece, **viene chiamato `previous()`**, viene **restituito l'elemento immediatamente precedente** alla posizione del cursore e quest'ultimo è **spostato all'indietro nella posizione precedente**. **Restituire l'elemento**, come si è già abituati a pensare, **può significare due cose**:

1. **È restituita una copia** dell'elemento della collezione;
2. **È restituito un riferimento** all'elemento della collezione.

Nel primo caso, una modifica al risultato dell'operazione (`next()` o `previous()`) non modificherà l'elemento della collezione a patto che la copia non sia in profondità; nel secondo caso,

invece, una modifica al risultato dell'operazione cambierà l'elemento della collezione. `Iterator<T>` e `ListIterator<T>` non specificano a priori quale politica adottare ma si tenga in considerazione che gli iteratori delle classi `ArrayList<T>` e `HashSet<T>` restituiscono dei riferimenti.

Raramente sarà necessario definire nuove classi che implementino `Iterator<T>` o `ListIterator<T>`; nel caso lo fosse, **il modo più comune e semplice per definire una nuova collezione consiste nell'estensione di una classe predefinita** (`ArrayList<T>` o `HashSet<T>`, ad esempio), **in modo tale da ereditare metodi come `iterator()` o `listIterator()` e risolvere facilmente il problema degli iteratori**. Se proprio **non si potesse fare altro che definire una collezione ad hoc**, il modo migliore per **definire la classe per gli iteratori** è come **inner class della collezione**.

Si vuole concludere menzionando la **differenza tra deep copy e shallow copy** di un oggetto:

- **Deep copy** (o **copia in profondità**), è una copia che non ha alcun riferimento in comune con l'oggetto originale tranne che per oggetti immutabili (oggetti che non espongono metodi `set()` per i propri attributi);

```
public Animale getOld(){
    return new Animale(old.getNome(), old.getEta(), old.getPeso);
}
```

- **Shallow copy** (o **copia superficiale**), è una qualsiasi copia che non risulti essere una deep copy.

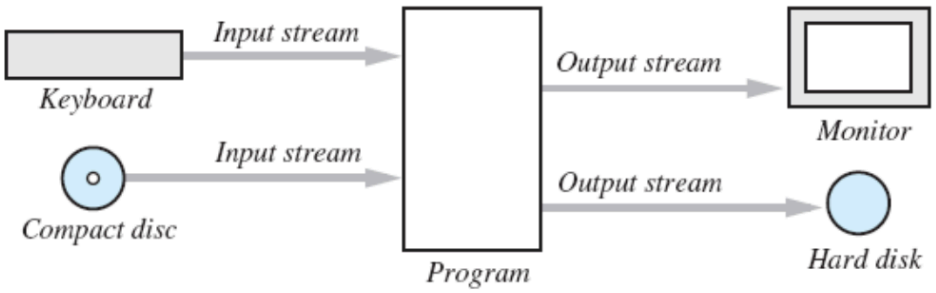
```
public Animale getOld(){
    return old;
}
```

La principale differenza tra i due tipi di copia consiste nella **possibilità di far permanere eventuali modifiche nell'oggetto da cui si copia** per la shallow copy, cosa che non accade per la deep copy se non utilizzando un secondo oggetto di appoggio (che non sarà `old`).

STREAM E I/O DA FILE DI TESTO

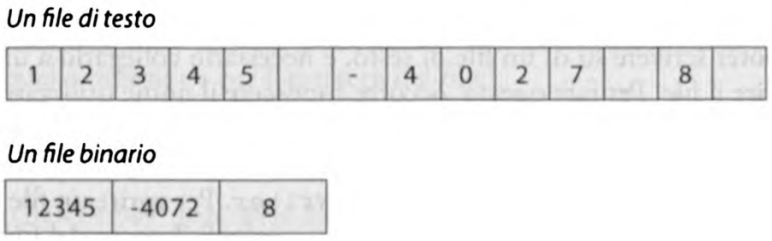
L'unico tipo di input e output al quale ora è stata prestata attenzione è quello da terminale; tuttavia, **l'utilizzo dei file rappresenta un'alternativa molto utilizzata in ambiente professionale**. I file sono molto versatili: **permettono di salvare classi Java, programmi, musica, video, ma anche di immagazzinare i dati di input a un programma o il suo output**.

Java gestisce i dati da e verso un file (così come per il terminale) **in termini di flussi di dati**, **stream**; uno stream **può essere costituito da caratteri, numeri o generici byte** e si distingue in **stream di input**, se i dati fluiscono nel programma, e in **stream di output**, se i dati fluiscono dal programma.



Il motivo per cui è necessario **evolversi dallo stream su terminale in uno stream avanzato su file** risiede nel **modo in cui i dati vengono gestiti**; **su terminale, i dati sono temporanei e disponibili solo finché il programma è in esecuzione**, mentre i file sono uno strumento utile ad immagazzinarli in modo permanente. Infatti, **a differenza del terminale, il contenuto di un file è accessibile anche asincronamente al programma** che lo gestisce e **rimane in memoria fintantoché un utente o un altro programma non ne decida il futuro**; inoltre, **uno stesso file di input può essere utilizzato da più programmi, per più volte e può contenere molti più dati senza che l'utente debba mettere la propria mano**. Si pensi ad un software che gestisce una banca e che aggiorna il proprio sistema una volta al giorno: senza l'utilizzo dei file e della loro automazione, un impiegato dovrebbe manualmente aggiornare ogni giorno una grande quantità di dati, sprecando una quantità di tempo immensa.

Qualunque tipo di file si utilizzi, **i dati sono sempre memorizzati in bit**, ovvero sottoforma di **informazioni binarie**; **ciò che distingue i vari tipi di file è il modo in cui una stessa sequenza è interpretata**, ad esempio **in forma di testo o in forma decimale**. In questa ottica, si distinguono **file di testo** (i cui dati sono interpretati come una sequenza di caratteri) **e file binari** (i cui dati sono interpretati come una sequenza numerica, decimale o binaria, non è dato saperlo a questo livello di astrazione). **I programmi Java (`.java`) sono file di testo**, interpretabili da qualsiasi computer allo stesso modo (a patto che sia installato il compilatore Java), mentre le foto o i video sono tipi particolari di file binari. **Gli stessi programmi Java appena menzionati possono, poi, operare sia su file di testo che binari**, dal momento in cui **sia lettura che scrittura sono uguali sui due tipi di file**; tuttavia, il tipo di file discrimina quali classi debbano essere utilizzate per l'input e quali per l'output.



Il vantaggio principale dei file di testo è la facilità con cui possono essere creati, visualizzati e modificati con un editor di testo, mentre **per un file binario** le operazioni di lettura e scrittura sono **da affidare necessariamente ad un programma di interpretazione esterno apposito**. Sebbene per un file binario possano essere utilizzate diverse codifiche, interpretabili o meno da alcuni sistemi piuttosto che da altri, i file binari Java sono indipendenti dalla piattaforma e sarà possibile spostarli su altri sistemi senza perdere la possibilità di utilizzarli.

Generalmente, **in un file di testo ogni carattere è rappresentato per mezzo di uno o due byte, a seconda del tipo di codifica in uso** (ASCII o Unicode), ed il risultato di un eventuale operazione (come la stampa) su questi bit è lo stesso che si avrebbe se si utilizzasse il terminale, impiegando anche la stessa quantità di dati. Di contro, **i file binari immagazzinano tutti i valori dello stesso**

tipo primitivo nello stesso formato e come una sequenza dello stesso numero di byte; questa proprietà rende **la gestione dei file binari più efficiente di quella dei file di testo**, visto che il modo con cui un programma Java interpreta questi byte è molto simile a quanto fa con i dati nella memoria principale.

La classe `PrintWriter` nella Java Class Library (appartenente al package `java.io`) **definisce i metodi per creare e scrivere file di testo** ed è lo strumento **da utilizzare preferibilmente per l'output su file di testo**. Prima di poter **scrivere su un file di testo**, supposto già creato, è necessario **collegarlo ad uno stream di output**, operazione detta di **apertura del file**; per aprire un file **occorre conoscerne il nome utilizzato dal sistema** (ad esempio, `file.txt`) e **dichiarare una variabile di stream da utilizzare come riferimento allo stream che gestisce il file** tramite il costruttore della classe `PrintWriter`, passandogli il nome del file come argomento (l'operazione può generare un'eccezione, è meglio includerla in un blocco try). Riassumendo:

```
String nomeFile = "file.txt"; // Potrebbe essere chiesto all'utente
PrintWriter oStream = null;
try{
    oStream = PrintWriter(nomeFile);
} catch (FileNotFoundException e) {
    System.out.println("Errore nell'apertura del file " + nomeFile;
    System.out.exit(0);
}
```

L'eccezione eventualmente sollevata è `FileNotFoundException` ed è inclusa nel package `java.io`. Come si è potuto notare, **il nome dei file è indicato sottoforma di stringa** e, probabilmente, sarà letto da qualche altro stream, non inserito come nell'esempio direttamente nel codice. Collegando in questo modo un file ad uno stream, **il programma partirà sempre da un file vuoto**: **se contiene delle informazioni, queste andranno perse**, e se non esistesse ne verrebbe creato uno vuoto. **L'eccezione `FileNotFoundException` viene sollevata**, non solo se il file non è stato trovato, ma **anche quando non è possibile crearne uno nuovo** (ad esempio, il nome può esser già stato usato per nominare una directory).

Aperto un file, è possibile scrivervi dei dati riferendosi allo stream tramite la variabile di stream (perciò la si dichiara fuori dal blocco try); per una questione di continuità, **il metodo `println()`** (analogamente per `print()`) **della classe `PrintWriter` consente di scrivere dati in un file di testo esattamente come l'omonimo metodo `System.out.println()` permette di scrivere sul terminale**. **L'output** prodotto da una chiamata a `println()` **non viene inviato al file immediatamente**, **è prima salvato in una memoria di buffer** la quale, **una volta riempita**, **trasferisce il proprio contenuto al file**; il motivo di questa operazione intermedia, detta buffering, va ricercato in un'**elaborazione più veloce dei dati**. **Nonostante Java disponga di un garbage collector, è importante chiudere appositamente** (con l'istruzione `close()`) **un file** una volta che ha assolto al proprio compito; infatti, per come è pensato lo stream, **il garbage collector interverrebbe sul file solo in fase di terminazione del programma**, lasciando per tutta la durata dell'esecuzione delle risorse dedicate ad uno stream che, magari, non è più in uso. La chiusura di un file è utile anche perché, **in caso di interruzione inaspettata del programma, si avrebbero meno probabilità di corrompere il file e di perdere i dati**.

Se un programma scrive dati in un file e successivamente legge dallo stesso file, sarà comunque necessario **chiudere il file per la scrittura e aprire lo stesso per la lettura** per mezzo di **due stream**

diversi (nonostante Java disponga di una classe che offre sia la possibilità di scrivere che di leggere con lo stesso stream, ma questa opzione non verrà discussa in questa sede). In genere, in aggiunta all'operazione di chiusura, **è buona pratica avvisare l'utente che le operazioni sono state completate**, inserendo anche un semplice messaggio di testo da terminale; **nel caso in cui questa prassi non fosse seguita, si otterrebbe un cosiddetto programma silenzioso** e l'utente potrebbe risultare confuso sul completamento delle operazioni da lui richieste.

Per i file menzionati finora e per quelli che verranno in futuro menzionati, è necessario adoperare una dovuta **distinzione tra file reale e file logico**: **il file reale risulta essere l'insieme di dati che sono o saranno effettivamente in memoria**, mentre p**er file logico si intende lo stream con il quale è scritto o letto un file reale**; **il file logico non esisterà più alla terminazione del programma**, a differenza del file fisico, ma esso costituisce **l'unica via per connettere il programma** (e quindi l'utente) **al file fisico**.

Proprio come è stato mostrato per `System.out.println()`, **anche `println()` di `PrintWriter` invocherà automaticamente il metodo `toString()` quando vi è passato un oggetto come argomento** e lo stesso accade con gli omonimi `print()`. Si può ancora di più intuire come sia importante la definizione del metodo `toString()` in una classe.

Chiaramente, non sempre risulta comoda la situazione mostrata in precedenza: **può capitare che un file non sia vuoto e che sia necessario preservare i dati al suo interno prima di scriverne altri**. Per poter aggiungere l'output di un programma a un file di testo, **la connessione tra il file e lo stream dovrà essere effettuata come segue**:

```
oStream = new PrintWriter(new FileOutputStream(nomeFile, true));
```

La classe `PrintWriter` non offre direttamente un costruttore che consenta l'operazione di aggiunta e, pertanto, **si ricorre alla classe `FileOutputStream`** (anch'essa da importare dal package `java.io`). **Il secondo argomento passato al costruttore di questa classe** (`true`) **indica che si vogliono aggiungere dati al file se questo esiste già**, conservando eventualmente il contenuto originale e scrivendo solo dopo di esso; **se, però, il file non esistesse, Java creerà un file nuovo vuoto** e vi scriverà normalmente. L'utilizzo dei blocchi try – catch non subisce alcuna variazione se si adotta questo procedimento.

Per leggere un file, le classi più utilizzate sono `Scanner` e `BufferedReader`:

- **`Scanner`**

La classe `Scanner` è la stessa usata finora per prendere l'input da tastiera, invocando il costruttore con il parametro `System.in`. **Per leggere un file con `Scanner`, non è possibile passare al costruttore il nome di un file**, visto che una **stringa** verrà interpretata come una **sequenza di dati** (e non come il nome di un file); tuttavia, **è possibile utilizzare il costruttore che prende in ingresso un'istanza della classe `File`**, la quale **ha un costruttore che accetta il nome di un file**:

```
Scanner iStream = new Scanner(new File(nomeFile));
```

Se il file che si sta cercando di aprire in lettura non esiste verrà sollevata una `FileNotFoundException`, da gestire in un apposito blocco try – catch proprio come fatto per la scrittura. Per leggere praticamente il contenuto di un file si può operare riga per riga finché il file aperto ha una "prossima riga":

```
while(iStream.hasNextLine()){
    System.out.println(iStream.nextLine());
}
```

I metodi usati per l'input da tastiera sono gli stessi ed operano allo stesso modo anche per la lettura da file. Il metodo **`hasNextLine()`**, mai visto finora, **restituisce `true` se nel file è presente ancora almeno un'altra riga da leggere**; proprio come i metodi `nextLine()`, `nextInt()`, `nextDouble()` e `next()`, anche per `hasNextLine()` esistono le varianti `hasNextInt()`, `hasNextDouble()` e `hasNext()` ed assolvono alle omonime funzioni.

- **`BufferedReader`**

Prima dell'introduzione della classe `Scanner` nella versione 5.0 di Java, **la classe `BufferedReader` monopolizzava la lettura su file di testo** (anche ora è piuttosto utilizzata). Per aprire un file con questa classe **si procede in maniera analoga a quanto mostrato per `Scanner`**; infatti, **non è presente alcun costruttore che accetti il nome di un file come parametro ed è necessario ricorrere alla classe `File`**. **L'omonimo di `nextLine()`** per la classe `BufferedReader` **è il metodo `readLine()`** ma nel caso **si cercasse di leggere oltre la fine** del file non sarebbe sollevata alcuna eccezione, bensì **verrebbe ritornato il valore sentinella `null`**.

A differenza di `Scanner`, **la classe `BufferedReader` dispone solo di due metodi per la lettura**: **`readLine()`**, già menzionato, e **`read()`**, che lavora in maniera singolare; il metodo in questione **legge un carattere alla volta e restituisce un valore di tipo `int` relativo al carattere letto**, che **non è il carattere stesso**. Si può, quindi, intuire che per utilizzare questo metodo **è necessaria una conversione del tipo**:

```
char prossimo = (char)(iStream.read());
```

Poiché, però, `read()` restituisce un `int`, **non può essere restituito `null` in caso di fine del programma**; **il valore sentinella associato a questo metodo è -1**, dal momento in cui i caratteri sono mappati su valori interi sempre positivi. **Per la lettura di valori numerici non è possibile ricorrere, come per la classe `Scanner`, a metodi specifici**: bisogna **dapprima leggere dei caratteri e poi**, in secondo luogo, **convertirli in valori numerici attraverso le classi wrapper**.

L'eccezione `FileNotFoundException`, sollevabile anche per la classe `BufferedReader`, **è una specializzazione dell'eccezione `IOException`**, che può occorrere con i metodi di lettura `read()` e `readLine()`; pertanto, **è teoricamente possibile gestire tutte le eccezioni di `BufferedReader` con un unico catch relativo a `IOException`**, rischiando però di limitare le informazioni sulle possibili cause dell'eccezione (non si sa se il sollevamento avviene aprendo il file o leggendolo).

LA GESTIONE DEI FILE E I/O DA FILE BINARI

Di seguito verranno approfondite alcune tecniche, utilizzabili sia con i file di testo che con i file binari, per ottimizzare la gestione dei file, a partire dalla già menzionata classe `File`.

La classe `File` consente una gestione omogenea dei file a partire dal loro nome; infatti, come è già stato possibile apprezzare, **il nome di un file è interpretato da Java come un'informazione**,

mentre **la stessa stringa assume un significato diverso se contestualizzata dalla classe `File`. Passando al costruttore di questa classe una stringa** (che teoricamente deve rappresentare il nome di un file), **l'oggetto associato sarà interpretato da Java come identificativo di un file reale**; in altre parole, **si tratta di un'astrazione indipendente dalla piattaforma**, anziché di un vero file.

Ad esempio, l'oggetto:

```
new File("file.txt");
```

Non è una semplice stringa ma è un oggetto che sa di essere interpretato come l'identificativo di un file.

Quando si utilizza il nome di un file per aprirlo in uno dei modi visti in precedenza, **si presuppone che il file si trovi nella stessa directory nella quale viene eseguito il programma**; tuttavia, **in caso contrario, si può specificare il percorso da fare per giungere alla stessa directory del file** specificando il percorso (o path name). Un **percorso** è detto **assoluto** (**absolute path**) se fornisce le indicazioni per **individuare il file a partire dalla directory radice** (root), mentre è detto **relativo** (**relative path**) se fornisce il percorso per **raggiungere il file relativamente alla directory di esecuzione del programma**. **Il modo con cui i percorsi vanno specificati dipende dal sistema operativo** e ne verranno discussi solo alcuni esempi, come **per i sistemi UNIX**:

```
/user/luca/lavoro/dati.txt
```

E per creare un oggetto `File` da questo file:

```
new File("/user/luca/lavoro/dati.txt");
```

Windows preferisce il backslash (\) al posto dello slash (/) e la directory root è (spesso) `C:`. Un tipico **percorso Windows**, quindi, è:

```
C:\lavoro\dati.txt
```

E l'oggetto File diventa:

```
new File("C:\\lavoro\\dati.txt");
```

Il doppio backslash senza spazio serve per impedire al compilatore Java di considerare backslash + altro carattere come un carattere non stampabile da ignorare, visto che il backslash da solo ignora eventuali combinazioni. Questo accorgimento **non va seguito se il percorso è inserito dall'utente**, dal momento in cui **il compilatore ignora i caratteri speciali dal buffer del terminale** quando è in atto un inserimento da tastiera; infatti, **con il doppio backslash** verrebbe memorizzato un percorso del tipo

```
C:\\lavoro\\dati.txt
```

Che **non corrisponde ad alcun file**. Infine, la differenza tra slash e backslash può essere ignorata perché un programma Java accetterà sia percorsi scritti in formato UNIX che in formato Windows.

I metodi della classe `File` sono utilizzati per analizzare le proprietà dei file; ad esempio, è possibile verificare se un file ha un nome specificato o se è leggibile. Si supponga di creare un'istanza della classe `File` associata ad un file `"dati.txt"`:

```
file oggettoFile = new File("dati.txt");
```

Il metodo `exists()` permette di verificare se esiste un file con il nome specificato e, quindi, di condizionare il flusso del programma per evitare errori o eccezioni:

```
if(!oggettoFile.exists()){
    System.out.println("Errore nell'apertura del file ";
    System.out.exit(0);
}
```

Se il file esiste, **è possibile verificare se il sistema operativo è in grado di leggere il file** con il metodo **`canRead()`**:

```
if(!oggettoFile.canRead()){
    System.out.println("Errore nella lettura del file ";
    System.out.exit(0);
}
```

Questo metodo è una **buona soluzione per verificare se un file è stato reso**, volontariamente o non, **illeggibile**. In maniera del tutto analoga si può trattare del metodo `canWrite()`, consentendo di verificare la possibilità di scrivere nel file da parte del sistema operativo.

I metodi `canRead()` e `canWrite()` sono necessari perché alcuni sistemi operativi consentono di indicare determinati file (come i file di sistema o file particolarmente importanti per l'utente) **come non leggibili e/o non modificabili o come modificabili e/o leggibili solo da utenti con determinati privilegi**.

Per l'I/O di un file binario sono utili le classi `ObjectInputStream` e `ObjectOutputStream`, con le quali **il file viene letto byte per byte** ed interpretato sulla base di ciò che è necessario fare; è anche possibile **convertire un dato di tipo primitivo o di tipo classe e salvarli in un file come se fosse costituito non da byte ma da valori di uno dei tipi primitivi/classe specificati**.

Per **creare un file binario** si può usare la classe **`ObjectOutputStream`**, tenendo bene in considerazione che **qualsiasi istruzione per l'I/O con i file binari può sollevare una `IOException`**; di conseguenza, **è buona prassi inserire, non più parte del codice, ma tutto il codice di gestione dell'I/O in un blocco try – catch**. Un file binario sarà un file denominato con l'estensione `.dat` e lo si potrà creare con la seguente istruzione:

```
ObjectOutputStream oStream = new ObjectOutputStream(
    new FileOutputStream("file.dat")
);
```

Come nel caso dei file, **questa è detta operazione di apertura del file** (visto che però file.dat non è nel sistema, Java lo crea); anche per i file binari **l'apertura semplice implica l'eliminazione del contenuto preesistente**. Il comportamento di questa classe è **equivalente a quello delle classi per l'I/O su file di testo**, solo che si sta usando una classe diversa. **Anche il comportamento di `FileOutputstream` è analogo a quanto detto per i file di testo**; infatti, proprio come `File` è passato al costruttore di `Scanner` per leggere un file di testo, `FileOutputStream` è passato al costruttore di `ObjectOutputStream` per leggere un file binario. Per quanto riguarda le eccezioni,

il costruttore di `ObjectOutputStream` può sollevare una `IOException`, mentre **quello di `FileOutputStream` una `FileNotFoundException`**.

A differenza delle classi per la scrittura su schermo o in file di testo, **la classe `ObjectOutputStream` ha bisogno di differenziare il tipo del dato che va in scrittura sul file binario**; quindi, **non esisterà un metodo `println()`, ma un metodo `writeInt()`** con cui si comunica al compilatore che il dato in scrittura è un intero:

```
oStream.writeInt(integerValue);
```

Anch'esso **può generare una `IOException`**. In maniera del tutto analoga, **esisteranno i metodi `writeLong()`, `writeDouble()`, `writeFloat()` e `writeChar()`**; quest'ultimo può essere **usato per scrivere un solo carattere**, sebbene ci si aspetta che **il valore passato come parametro sia un intero**. Il motivo di questa stranezza è lo stesso che c'è alla base del metodo `read()` nella classe `BufferedStream`:

```
oStream.writeChar((int)'A');
```

Così come per i file di testo, **dopo aver lavorato su un file binario è necessario chiudere il corrispettivo logico tramite il metodo `close()`**:

```
oStream.close();
```

Come si può notare, **non è stato menzionato alcun metodo per la scrittura delle stringhe**; in realtà, per un'operazione di questo tipo **viene in aiuto il metodo `writeUTF()`**, così utilizzato:

```
oStream.writeUTF("Paolone nazionale");
```

UTF sta per Unicode Text Format ed è la **codifica universale** attualmente in uso per rappresentare caratteri e stringhe. Agli albori **l'informatica era dominata dal linguaggio anglosassone** ed era necessario dapprima **codificare le lettere e i simboli dell'alfabeto inglese**; pertanto, le prime codifiche, **ASCII a 7 bit e ASCII a 8 bit**, erano **concentrate sulla codifica delle lettere dell'alfabeto latino**, in aggiunta a **caratteri speciali per le lingue occidentali** (come l'italiano, il francese e lo spagnolo) **e ai caratteri non stampabili**. Soprattutto con l'avvicinamento della fine della guerra fredda, **anche il mondo orientale ha iniziato ad approcciare all'informatica**, richiedendo con sempre più insistenza una codifica dei propri alfabeti; tuttavia, **la codifica ASCII a 8 bit era già satura** dagli alfabeti occidentali e, pertanto, **si decise di inglobare questa codifica in una più universale, la codifica Unicode**. La **codifica UTF** prevede l'utilizzo di **minimo 1 byte** (fino ad un massimo di 4 byte) **per rappresentare un simbolo**: **i primi 128 valori** (quindi **il primo byte**, corrispondente ai **numeri binari che hanno l'ottavo bit abbassato**) **è usato per codificare il già esistente alfabeto ASCII a 8 bit** (di cui, quindi, è un sottoinsieme), **i successivi 2 byte** (corrispondenti ai numeri binari che hanno l'ottavo bit alzato) **sono usati per la codifica degli altri alfabeti** e, infine, **gli ultimi 3 – 4 byte per la codifica di caratteri speciali** (come le emoji). **Lo standard UTF**, oggi universalmente consolidato, risulta **inefficiente nei paesi di lingua inglese** ma lo si continua ad usare per una maggior globalizzazione dell'informatica.

Scendendo più nel dettaglio e volendo analizzare più da vicino come funziona il metodo `writeUTF()` **si deve considerare che i metodi di scrittura su file binario finora visti impiegano un numero di bit** (o byte) **costanti e pari alla dimensione massima dell'informazione ad essi associata** (ad esempio, `writeInt()` scriverà sempre lo stesso numero di bit, pari al numero di bit necessari per codificare un `int`). Il metodo **`writeUTF()` funziona in maniera singolare e non si**

adatta a questa schematizzazione; infatti, **per rappresentare una stringa**, a differenza di come accade per un `char` o un `float`, **non è richiesta una lunghezza fissa**, essa può variare sulla base dell'informazione da scrivere: **stringhe più lunghe richiederanno più byte e stringhe più corte meno byte**. Questa proprietà delle stringhe **può rappresentare un problema per Java**, dato che in un file binario **non esistono separatori tra i singoli dati** e sarebbe piuttosto **difficile cercare di risalire al numero di byte della stringa senza specificarlo in alcun modo**; perciò, **il metodo `writeUTF()` non solo scrive sul file binario la stringa ma ne fa anticipare il dato relativo al numero di byte necessari** per codificare quella particolare informazione, in modo che **un eventuale metodo `readUTF()` sappia quanti byte deve leggere prima di comunicare al programma** il contenuto del file. In realtà, **il comportamento del metodo `writeUTF()` è leggermente più complesso**, visto che **il numero di caratteri della stringa ed il numero di byte per rappresentarla non sempre coincidono** (un carattere può essere codificato con più o meno byte di un altro); tuttavia, **se si utilizzano solo caratteri ASCII**, che sono sempre e solo di un byte, **questa distinzione diventa puramente teorica** e il funzionamento del metodo può essere modellato come in precedenza.

In uno stesso file binario **possono essere scritti dati di diverso tipo**; tuttavia, mescolare in questo modo le informazioni **richiede particolare attenzione** perché **il file** (così come un'eventuale programmatore esterno che interviene sul codice) **non è conscio a priori dell'ordine in cui questi dati spezzettati sono inseriti** e, pertanto, **potrebbero essere letti ed interpretati erroneamente**. In particolare, **occorre tenere traccia dell'ordine nel quale i dati sono stati scritti nel file**, dato che, come sarà più chiaro in seguito, **si utilizza un metodo diverso per leggere ogni tipo di dato**.

Se **un file binario** è stato creato e vi si è scritto sopra tramite la classe `ObjectOutputStream`, lo **si può leggere utilizzando la classe `ObjectInputStream`**. Poiché **le due classi devono essere complementari**, si può osservare una dualità tra i metodi elencati per `ObjectOutputStream` e quelli della classe `ObjectInputStream`; ad esempio, al metodo `writeInt()` corrisponderà un `readInt()` e così via… Anche l'apertura del file per la lettura è simile (cambiano solo le classi in `ObjectInputStream` e `FileInputStream`):

```
ObjectInputStream iStream = new ObjectInputStream(
    new FileInputStream("file.dat")
);
```

Proprio come mostrato per la scrittura, **anche per la classe `ObjectInputStream` il relativo costruttore non ammette una stringa per il percorso del file ma un oggetto istanza di `FileInputStream`**, che rappresenta il **file logico**; inoltre, **anche le eccezioni sollevate sono esattamente le stesse** delle classi `ObjectOutputStream` (`IOException`) e `FileOutputStream` (`FileNotFoundException`).

Se per la scrittura era stata ammessa la **possibilità di scrivere su uno stesso file dati di tipi diversi**, **per la lettura è ammessa l'operazione speculare**; tuttavia, **per leggere file binari su cui sono stati scritti dati di diversi tipi, è necessario conoscere l'ordine e le dimensioni di questi vari dati**, in modo da non interpretare (erroneamente) una sequenza di byte numerica quando in realtà in quella posizione è stata scritta una stringa.

La distinzione tra lettura/scrittura per file binari e di testo è necessaria perché i metodi e i principi di funzionamento delle relative classi non sono trasversali: **non è possibile leggere/scrivere su un tipo di file utilizzando la classe dedicata all'altro**. **Molti metodi per la lettura da file binari sollevano una `EOFException`** (EOF sta per End Of File, è derivata da `IOException`) **quando si cerca di leggere oltre la fine del file**; questa proprietà permette di **semplificare la lettura di un file** perché

non rende necessaria la definizione di un metodo apposito che verifica se si è raggiunti la fine del file (come, invece, è necessario fare per i file di testo), semplicemente si gestisce un'eccezione `EOFException`:

```
try{
    while (true){
        // Leggi il file
    }
} catch (EOFException e) {
    System.out.println("Si è raggiunti la fine del file");
    System.out.exit(0);
}
```

Si può pensare che **un codice così scritto generi un loop infinito** (`while(true)`) ma in realtà **tale loop è infinito finché il blocco try – catch non rileva e gestisce l'eccezione `EOFException`**, che corrisponde alla fine del file. Come si può notare, **non è necessario impiegare nessun metodo che rilevi valori sentinella** (come `null` o -1 nel caso di file di testo) e **si può scrivere l'algoritmo non dovendo pensare ad eventuali costrutti di selezione o variabili di appoggio**, **evitando di controllare in modi sbagliati la fine del file** (e quindi di entrare in un loop infinito o di far terminare inavvertitamente il programma). Purtroppo, **questa comodità si limita ai file binari** e non può essere trasposta nella lettura dei file di testo.

Si vuole concludere la trattazione dell I/O su file binari menzionando **come procedere per oggetti e array utilizzando `ObjectInputStream` e `ObjectOutputStream`**. Per salvare un oggetto in un file binario **si può pensare di estrapolare le variabili di istanza, memorizzarle come variabili di tipo primitivo e, in fase di lettura, ricostruire l'oggetto originale istanziandone uno nuovo** e inserendovi le variabili di istanza registrate nel file. Questo approccio **non è sbagliato ma non è il migliore**, soprattutto quando le variabili di istanza di un oggetto sono anch'esse oggetti (ad esempio nelle strutture dati dinamiche); **Java offre un modo semplice per la rappresentazione di un oggetto sottoforma di sequenza di byte**, chiamato **serializzazione degli oggetti**. Affinché un oggetto sia rappresentabile mediante serializzazione, **è necessario che sia serializzabile**; rendere serializzabile una classe significa **aggiungere la specifica `implements Serializable` all'intestazione della definizione della classe**:

```
public class className implements Serializable {}
```

L'interfaccia `Serializable` fa parte della libreria standard Java, che appartiene al package `java.io`. Essa è vuota, **non ci sono metodi aggiuntivi che debbano essere implementati**, ma, sebbene possa sembrare un qualcosa di completamente inutile, **serve a Java per permettergli di serializzare la classe**. Per scrivere un oggetto in un file binario già aperto si utilizza il metodo `writeObject()` della classe `ObjectOutputStream` e, per leggerlo, il metodo `readObject()` della classe `ObjectInputStream`; per quanto riguarda quest'ultimo, vanno fatti alcuni accorgimenti: **dal momento in cui né il metodo `readObject()` né il file sono consci a priori della struttura e del tipo classe dell'oggetto che si sta leggendo, ciò che verrà restituito sarà un oggetto di tipo `Object`** e sarà necessario un downcasting per conformare l'oggetto al tipo classe opportuno. Tutto ciò non è necessario quando l'oggetto lo si scrive.

```
className oggettoLetto = (className)iStream.readObject();
```

È stata menzionata la possibilità, da parte di una **classe**, di avere una **variabile di istanza che è un oggetto**; in questa situazione, **la classe è serializzabile non solo se implementa l'interfaccia `Serializable`, ma se anche la classe dell'oggetto variabile di istanza è serializzabile**. In definitiva, una classe è serializzabile se verifica le seguenti condizioni:

- **La classe implementa l'interfaccia `Serializable`**;
- **Tutte le variabili di istanza di tipo classe sono istanze di classi serializzabili**;
- **La superclasse diretta della classe**, se esiste, **è serializzabile** o definisce un costruttore di default.

Rendere una classe serializzabile non produce alcun effetto diretto sulla classe ma solo su come Java gestisce l'I/O su file con oggetti di quella classe; infatti, **Java assegna un numero di serie ad ogni oggetto della classe serializzabile che viene scritto in uno stream `ObjectOutputStream`**, in modo che, **se lo stesso oggetto viene scritto nello stream più volte, Java non ricopia l'intero oggetto ma solo il suo numero di serie**. In questo modo **sono ridotte le dimensioni del file** su cui si scrive e viene reso l'I/O molto più efficiente; **quando, poi, il file viene letto, i numeri di serie duplicati vengono restituiti come riferimenti allo stesso oggetto**.

Si può pensare che, **essendo la serializzazione una proprietà così utile, sia comodo rendere di default tutte le classi serializzabili**; in realtà, **in certi casi per motivi di sicurezza si preferisce non serializzare una classe**, visto che il sistema di serializzazione **rende semplice l'accesso agli oggetti salvati su memorie secondarie**, **oppure non è necessario salvare gli oggetti su una memoria esterna** (e quindi renderli disponibili per un secondo momento) perché **descrivono informazioni utili solo quando osservate in tempo reale** (come le informazioni sullo stato di un sistema).

Per quanto riguarda gli array, **visto che Java li considera dei veri e propri oggetti, è sufficiente quanto detto finora, sia per la scrittura che per la lettura**; bisogna solo appuntare che, proprio come precedentemente detto per le variabili di istanza, **se il tipo base degli elementi di un array è una classe, è necessario che essa sia serializzabile**. Ad esempio, per la lettura:

```
className[] arrayLetto = (className[])iStream.readObject();
```