Programmazione II
Jun Xiang Federico || 941519
Playlist: https://www.youtube.com/playlist?list=PLaunfa1yOUSLA6gFhBzIm-FD0aT3RNG4p
Zoom link:
Gitpod link: https://gitpod.io/workspaces/

Diario delle lezioni e programma: https://prog2.di.unimi.it/diario.html

PDJ 1 | Introduzione
Java è un WORM (Write Once Run Many) che risolve i problemi di portabilità tra sistemi operativi, grazie al compilatore JAVAC che traduce i sorgenti .java in bytecode .class.
La chiave del corso consiste nel concentrarsi sulla modularizzazione: suddivisone in problemi più piccoli e semplici da risolvere, che devono:
- avere lo stesso livello di dettalgio
- essere indipendenti l'un l'altro
- essere componibili

L'astrazione
Il processo di astrazione può essere vista come l'applicazione di una mappatura molti-a-uno.
Ci permette di "dimenticare" informazioni superflue e trattare elementi diversi, come se fossero gli stessi.

Vi sono 2 processi d'astrazione:
1) **Astrazione per parametrizzazione** astrae dall'identità dei dati, e li rimpiazza con parametri
Questo generalizza i moduli, permettono l'utilizzo in più situazioni
Attraverso l'introduzione di parametri possiamo includere una più vasta gamma di input, astraendo dai dettagli delle singole istanze, ma concentrandoci sulle caratteristiche generali comuni.

2) **Astrazione per specificazione** astrae dai dettagli dell'implementazione ed il suo comportamento.
Isola l'implementazione dei vari moduli uno dall'altro, e ci permette di appoggiarci solo al comportamento che c'aspettiamo dal modulo.
L'astrazione per specificazione ci permette **di astrarre dal modo in cui le procedure sono definite**, concentrandoci solo sul **comportamento finale che queste vogliono raggiungere**.
Lo facciamo definendo una **specification** che spiega qual è il **significato/effetto** della procedura nella specificazione, piuttosto che nel corpo della funzione.
Introduciamo delle **REQUIRES ASSERTIONS**, precondizioni che se rispettate ci producono effetti definiti negli **EFFECTS ASSERTION/POSTCONDITIONS**.

Questi 2 tipi di astrazione ci permettono di definire **3 tipi di astrazione**:
- **L'astrazione procedurale** (ci permette di implementare altre operazioni, nuove funzioni)
- **L'astrazione dei dati** (ci permette di implementare nuovi tipi di dato) ad esempio struct, class
- **L'astrazione iterazionale** (ci permette di iterare su oggetti in una collezione senza rivelare dettagli riguardo a come questi oggetti sono ottenuti)
Infine **la gerarchia di tipo** ci permette di astrarre da **tipi di dat**i individuali in famiglie di tipi comuni.
Tipi di dati individuali appartengono a **famiglie di tipo**, tutti i membri della famiglia di tipo hanno delle **operazioni comuni**, spesso definite dai supertipi (il tipo parente di tutti gli altri)

PDJ2 | The Java Language

JAVA è un **object-oriented language**

Ciò significa che la maggior parte dei **dati** gestiti dai programmi java sono **contenuti** in **objects**.

Gli objects hanno i loro **stati e loro le operazioni** (chiamati methods).

I programmi interagiscono con gli **objects** invocando i **methods**, che garantiscono anche **accesso al loro stato**, così come la possibilità di **osservare** lo stato corrente **dell'object**, o di **modificarlo**.

Struttura dei programmi:

I programmi JAVA sono composti da classi ed interfacce.

Le classi sono utilizzare per definire collezioni di procedures/methods; e per definire nuovi tipi di dati.

Le interfacce sono anche utilizzate per definire nuovi tipi di dato.

La maggior parte dei componenti di classi ed interfacce sono constructors e methods.

Packages

Classi ed interfacce sono suddivisi in packages

Suddividiamo in questa maniera per 2 benefici:

1) **encapsulation mechanism**: abilita la **condivisione** di informazioni solo all'interno del pacchetto, **bloccandone l'utilizzo dall'esterno**. Ogni classe ed interfaccia ha una data "**visibilità**", solo quelle definite come public possono essere utilizzate da altri pacchetti, mentre le altre possono essere utilizzate solo all'interno.

2) **naming**

Ogni pacchetto ha una **nomenclatura gerarchica** che li contraddistingue da tutti gli altri pacchetti.

Le classi e le interfacce all'interno di un pacchetto sono **relative al nome del pacchetto**.

Per richiamare elementi da altri pacchetti è possibile utilizzare il loro **"fully-qualified-name"**, che consiste nel loro nome, insieme al **nome gerarchico** del pacchetto. Eg. mathRoutines.Num

PDJ2.3 | Objects and variables

Tutti i dati presenti in JAVA sono accessibili attraverso variables.

Le local variables, risiedono nello **stack**, il loro spazio è allocato solo al momento della chiamata e vengono svuotate al momento del return.

Ogni variabile ha bisogno di una dichiarazione che indichi il suo tipo, e devono essere inizializzati prima del loro utilizzo, pena la non riuscita compilazione.

Esistono tipi di dati:

- **Primitivi**: int, boolean, char etc.

Questi tipi di dati vengono forniti da JAVA, mentre altri tipi di dati devono essere definiti dal programmatore.

- **Derivati**: array[]

Contengono riferimenti ad oggetti che risiedono all'interno **dell'heap**, e sono creati attraverso l'operatore new

Per esempio il comando per creare l'array crea uno spazio per un nuovo array di interi, allocato nell'heap, e con riferimento agli elementi caricati all'interno di a.

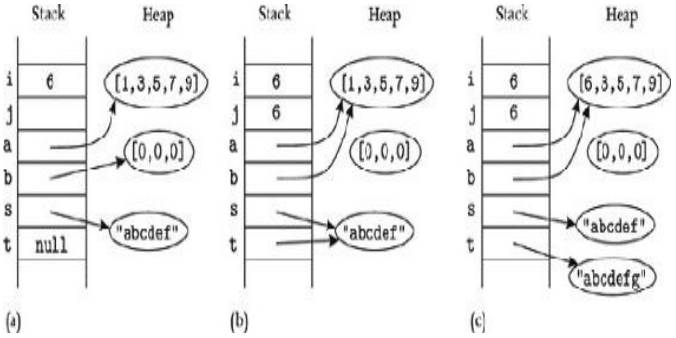
```
int[]a=new int[3];
```

Gli oggetti possono essere mutabili, o immutabili

eg. String = immutabile, vi sono operatori di concatenazione, ma non vengono modificati i loro arguments.

Un oggetto è **mutabile** se il suo **stato** può cambiare (eg. Arrays), lo **stato** di un oggetto **immutabile** non cambia mai.

Un oggetto è condiviso da 2 variabile se è accessibile da entrambi, ne consegue che modifiche su una variabile si ripercuote pure sull'altra




```
int i = 6;
int j; // uninitialized
int [ ] a = {1,3,5,7,9}; // creates a 5-element array
int [ ] b = new int[3];
String s = "abcdef"; // creates a new string
String t = null;
```

Method Call Semantics
Alla chiamata di un metodo, eg $e.m(\dots)$ il compilatore valuta prima e, cercando di ottenere la classe o l'oggetto sulla quale è stato chiamato un metodo.
Questo viene svolto per ottenere un **actual parameter**, utilizzato per creare spazio per i **parametri formali** del metodo. Gli actual parameters sono assegnati ai **parametri formali** attraverso il **passaggio by value** (call by value), ed infine il controllo è lasciato al metodo chiamato

PDJ 2.4
Type-Checking
Java è un linguaggio fortemente tipato, ciò significa che il compilatore controlla tutto il codice, assicurandosi che tutti gli assegnamenti e tutte le chiamate siano corretti. (nel caso di errore la compilazione fallisce)
Il **type**-**checking** permettere di creare un **signature** a seconda dei **tipi delle variabili**, degli header dei metodi, ed i costruttori (il tipo degli arguments ed i suoi risultati).
Il **signature** è la collezione di **arguments**, e risultati di una funzione.

Questa informazione viene utilizzata dal compilatore per estrarre il **tipo apparente** di ogni espressione.
Il tipo apparente di una variabile è il **tipo assunto dal compilatore** dalle informazioni presenti nelle sue **dichiarazioni**.
Il **tipo concreto** di un oggetto è il suo **tipo "reale"**, il tipo che riceve al momento della creazione.
Questo tipo apparente è poi utilizzato per determinare la correttezza di un **assignment**.

Il controllo dei tipi da parte del compilatore avviene:
- Letterali, a seconda del valore assegnato Java deduce il tipo: eg. $3 \rightarrow int \;\; pluto \rightarrow String$
- Durante la dichiarazione di variabili: eg. $String\ s$
- Attraverso l'analisi della segnatura dei metodi, analizzando il tipo del dominio e del codominio
Attraverso questi punti viene determinato il tipo delle espressioni grazie all'induzione strutturale

I programmi java sono type safe, comporta che non vi possono essere errori di tipo durante l'esecuzione del programma. La type safety è garantita da 3 meccanismi:
- Compile-time checking
- Automatic storage management
- Array bounds checking
Questi meccanismi garantiscono che type mismatches non possono avvenire durante il runtime.

Gerarchia di tipi:
I tipi in JAVA sono **suddivisi** in una gerarchia, ogni tipo può avere un numero arbitrario di **supertipi**, e si dice che un tipo è sottotipo di tutti i suoi supertipi. (e di sé stesso) (Transitiva e riflessiva).
Dato un $sottotipo\ a$ ed $un\ supertipo\ b$ la relazione viene indicata con $a \prec \mathrm{b}$.
La relazione di sottotipo $\prec$ deve godere del principio di sostituzione di Liskov.

La gerarchia di tipi ci permette di astrarre dalle **differenze** dei sottotipi, concentrandoci sui **comportamenti comuni** di essi, che sono definiti nel supertipo.
Ogni oggetto sottotipo ha anche **tutti i metodi definiti** nel suo supertipo (controllato dal compilatore).
Diciamo che se A è un sottotipo di B, allora a extends b.

Lo special type Object è alla vetta della gerarchia dei tipi in JAVA, tutti i tipi sono sottotipo di questo tipo.
Questo comporta che ogni tipo in JAVA eredita i metodi di Object, i più importanti sono "equals" e "toString".

```
boolean equals (Object o)
String toString ( )
```

Conversions and Overloading:
Java permette di svolgere **conversioni implicite** con i tipi primitivi (un char può essere esteso a tipi numerici).
Java inoltre permette **l'overloading**, ovvero la definizione di un numero di metodi con lo stesso nome ma con **signature** diverso.

```
static int comp(int, long) // defn. 1
static float comp(long, int) // defn. 2
static int comp(long, long) // defn. 3
```

Al momento della chiamata di comp, il compilatore deciderà a seconda dei tipi passati quale dei 3 metodi chiamare, a seconda del tipo più adatto (quello con meno conversioni).

Primitive Object Types:
I tipi primitivi come int e char **non sono sottotipi di Object.**
I tipi primitivi possono essere utilizzati in contesti in cui Objects sono necessari attraverso il processo di **wrapping**
Ogni tipo primitivo ha un **tipo object associato**.
Questo tipo permette al costruttore di creare un oggetto **incapsulando** il valore associato al tipo primitivo, ed un metodo per fare la trasformazione inversa. Questo viene fatto utilizzando:
- un costruttore $new\ T(x)$
- un metodo statico $T.valueOf(x)$

Stream I/O:
Il pacchetto java.io contiene una serie di tipi per gestire il flusso di input output.
Questo è fatto attraverso l'impiego di oggetti che appartengono al tipo "Reader" o uno dei suoi sottotipi.
Java, inoltre, fornisce una serie di oggetti predefiniti per svolgere operazioni standard di I/O, e sono contenuti nel pacchetto System del pacchetto java.lang (System.out.Println(arguments))

Java Applications:
Ci sono due tipi di applicazioni java:
- Applicazioni che funzionano da comando a command line sul terminale
Questi hanno un metodo main, che utilizza un array di strings per gestire gli arguments inseriti a command line
Per gestire valori numerici in input da command line utilizzare Integer.ParseInt(args[x])
- Applicazioni che interagiscono con un utente.

Astrazione procedurale PDJ 3 o Procedure
L'astrazione procedurale combina l'astrazione per parametrizzazione e per specificazione per permetterci di astrarre un determinato comportamento.

Un'astrazione è una mappa 1 a molti, **astrae da dettagli irrilevanti di un'istanza**, concentrandosi su **dettagli rilevanti** alla risoluzione di un problema. I benefici sono molteplici:
- L'astrazione per parametrizzazione astrae rispetto all'identità dei dati utilizzati, vengono implementati parametri formali in modo tale che il programma funzioni indipendetemente dai valori che vengono inseriti.
- L'astrazione per specificazione ci permette di concentrarci su "cosa viene fatto", permettendoci di astrarre dal "come" un determinato comportamento è implementato
Benefici:
- Comprensione o località, l'implementazione di un'astrazione può essere letta o scritta senza la necessità di analizzare le implementazioni presenti di altre astrazioni
- Modificabilità, un'astrazione può essere reimplementata senza la necessità di cambiamenti a astrazione che ne fanno già utilizzo (modularizzazione)

Specifications:
La specificazione è scritta prima della scrittura del codice, descrive il comportamento della funzione e l'input richiesto. La specificazione è scritta sotto il punto di vista:
- **Sintattico**, effettuando **un'astrazione rispetto ai dati**.
All'interno dell'header è indicato il tipo restituito ed i parametri formali
- **Semantico**, effettuando **un'astrazione per specificazione**, viene impiegato un linguaggio informale ma preciso, per descrivere il comportamento della funzione. A tale scopo vengono introdotte le 3 clausole della Liskov:

```
return_type pname (…)
   // REQUIRES: This clause states any constraints on use
   // MODIFIES: This clause identifies all modified inputs
   // EFFECTS: This clause defines the behavior
```

Le clausole della Liskov
- Clausola Requires || $@param$
definisce i limiti sotto la quale l'astrazione è definita
Cosa è necessario perché quello che c'è scritto in Effects si verifichi?
E' necessario nel caso la **procedura sia parziale**, ovvero la **procedura non è definita** per tutti gli input possibili
(se la procedura è totale, la clausola requires può essere omessa)
Indica cosa la funzione si aspetta **dall'input e dall'ambiente**, escludendo alcuni valori dal dominio.

- Clausola Effects || $@return$
descrive il comportamento della procedura per tutti gli input non esclusi dalla clausola requires.
Cosa fa questo procedure?
Definisce cosa la funzione restituisce come output, eventuali modifiche all'ambiente, e qual è l'effetto prodotto.

- Clausola Modifies
Che cosa sto modificando degli input?
Se eventuali input sono modificati, si dice che la procedura ha un side effect.
(se la procedura non modifica nessun input, la clausola modifies può essere omessa)

L'implementazione:
L'implementazione di una procedura deve produrre il comportamento definito nella sua specification.
Deve modificare gli input che compaiono nella clausola modifies, se gli input soddisfano i requirements definiti nella clausola requires, producendo risultati in linea con la clausola effects.

Se le procedure sono standalone ovvero funzionano da sole vengono dichiarate con la keyword $static$.
Se una classe è definita con $public$ essa può essere acceduta da tutte le altre classi; con la keyword $private$ indichiamo che i metodi possono essere sollevati solamente nella classe in cui sono definiti.

Linee guida per una buona implementazione:
- Minimalità dei vincoli: una specification è meno vincolata di un'altra se contiene meno limiti sui comportamenti concessi, se non rispettata possiamo incorrere in **sottospecificazione**.
- Implementazione deterministica: l'implementazione di una procedura è deterministica, per gli stessi input, produce sempre gli stessi output.
- Generalità: una specification è più generale di un'altra se è in grado di gestire una più larga classe di input.
- Semplicità: una procedure è semplice se ha un comportamento ben definito e facile da spiegare.
(un buon modo di verificarlo è provare ad assegnargli un nome)

Procedure parziali vs procedure totali:
- **Una procedura è totale** se il suo comportamento è specificato per **tutti gli input** ammessi, senò è parziale.
Le specification delle **procedure parziali** contengono sempre una clausola requires.
- **Le procedure parziali** sono **meno sicure** di quelle **totali**, dovrebbero essere impiegate **solo qualora strettamente necessario**. (casi in cui ci sia un effettivo beneficio, come il tempo/performance/costo)
- Quando possibile, le implementazioni dovrebbero verificare che gli input siano conformi ai limiti imposti dalla clausola requires, e sollevare un'eccezione qualora questi non siano soddisfatti.

Summary:
Una procedura è un'elaborazione di input a output, con potenziali modifiche ad alcuni input.
Il comportamento, e i vari aspetti di una procedura sono definiti nella specification, che segue il seguente formato:

```
return_type pname (…)
   // REQUIRES: This clause states any constraints on use
   // MODIFIES: This clause identifies all modified inputs
   // EFFECTS: This clause defines the behavior
```

L'astrazione permette di usufruire dei benefici di locality, ovvero che ogni implementazione può essere compresa in isolamento e modifiability, ovvero che un'implementazione può essere sostituita da altre senza disturbare i programmi che ne fanno utilizzo. (manutenibilità)
La specification funziona come un "contratto" tra l'utente e l'implementatore.
L'utente può astrarre da come un comportamento è implementato, dando per scontato che questo comportamento venga svolto come descritto nella specification.

Eccezioni PDJ 4
Un'astrazione procedurale è una **mappazione da arguments a risultati**.
Gli arguments sono **membri del dominio della procedura**, e i **risultati fanno parte del suo range**.
Gli arguments, tuttavia, spesso rappresentano solo da un **sottoinsieme del dominio**, in queste situazioni si fa utilizzo di **procedure parziali**. Il chiamante di una procedura parziali deve assicurarsi che gli arguments siano elementi **permessi dal dominio**.

Quando andiamo ad implementare un programma, dobbiamo tener conto della **robustezza** del tale
Un programma robusto è un programma che è in grado di **funzionare ragionevolmente anche in presenza di errori**, e qualora effettivamente incontri un errore, sia in grado di **fornire un'approssimazione** del suo **comportamento** nella presenza di **tale errore (graceful-degradation).**
Un modo per aumentare la robustezza di un programma è quello di **usare procedure totali** (procedure per le quali il comportamento è definito per ogni input del dominio); qualora invece una procedura non sia in grado di svolgere la sua funzione per alcuni input, deve essere in grado di informare il chiamante del problema.
Introduciamo quindi il concetto di eccezioni.

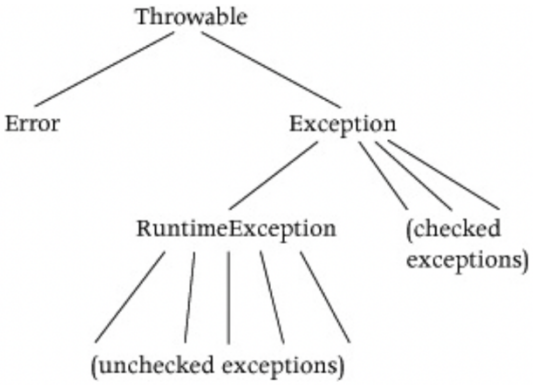
Se per esempio voglio fare la ricerca in un array, e passo un elemento che non è presente, è più appropriato sollevare un'eccezione piuttosto che fermare l'esecuzione del programma.

PDJ 4.1 Specifications
Qualora una procedura possa terminare eccezionalmente lo indicihiamo con la clausola $throws$

```
public static int fact (int n) throws NonPositiveException
```

La specifications di una procedura che chiama eccezioni deve **esplicitare** chiaramente cosa sta succedendo.
La specification deve quindi listare nell'header tutte le eccezioni che sono invocabili; così come il motivo di perché queste si possono presentare.

PDJ 4.2 The Java Exception Mechanism



Cosa sono le eccezioni?
Le eccezioni sono oggetti che appartengono ad una **gerarchia di classi particolare, chiamata Throwable**.
- Gli **Error** sono oggetti che non dipendono necessariamente dal programmatore, e sono difficilmente gestibili
- I tipi di eccezioni che interessano a noi sono **sottotipi di *Exception,* o *RunTimeException***.

Ci sono principalmente 2 tipi di eccezioni:
- **Checked exception**, sono sottotipi di *Exception,* ma non di *RunTimeException*
sono gestite esplicitamente attraverso il **catch or specify**, necessitano della definzione del throws all'interno della intestazione, pena sollevamento di un errore da parte del compilatore.
Vengono impiegate quando i metodi sono $public$ e la verifica è onerosa.
- **Unchecked exception**
Sono sottotipi di *RunTimeException*, non sono esplicitamente gestite, e non sono definite né nell'intestazione che nel corpo del codice. Fanno parte di quesot insieme gli errori e le eccezioni a runtime.

Ci sono 2 modi in cui un'eccezione checked or unchecked può essere utilizzata in JAVA:
1) Se la procedura potrebbe invocare una **checked exception**, JAVA pretende che l'eccezione sia **definita nell'header** della procedura, pena comparsa di **compile-time error**.
Le unchecked exception invece posso anche non essere incluse nell'header.
2) Se una porzione di codice chiama una procedura che potrebbe invocare una **checked exception**, il controllo viene passata alla porzione di **codice che gestisce l'eccezione**.
Nel caso di unchecked exception, non c'è bisongo che essa venga gestita nel codice chiamante.

Brutalmente: checked richiedono gestione con $try/catch$, unchecked non vengono gestite dal chiamante
Deviamo dalle regole di Java, e ci prendiamo la briga di richiedere la definizione di una lista con tutte le eccezioni che una procedure potrebbe sollevare, checked o unchecked che sia.

	Checked	Unchecked
Gestione	`catch - specify`	-
Dichiarazione	Specificazione + Throws	Specificarle solamente nella specificazione per non appesantire il codice
Quando è meglio utilizzarla?	- Recoverable Conditions: quando ci sono delle condizioni nelle quali il programma può non terminare in maniera brutale - Quando sono dovute da anomalie esterne al programma (e al programmatore) che non possono essere previste con una assoluta certezza, come un errore di input da parte dell'utilizzatore	- Usualmente per la violazione delle precondizioni - Anomalie interne al programma, come errori di implementazione - "Programming Errors": anomalie che possono essere risolte se il programmatore prestasse più attenzione
Pro (se ben utilizzate)	- Separazione dal flusso di esecuzione - Defensive Programming - Maggiore robustezza del codice - "Reminder" per il programmatore - Maggiore leggibilità	- Separazione dal flusso di esecuzione - Defensive Programming - Maggiore robustezza del codice
Contro (se si abusa)	- Non devono essere utilizzate per modificare il flusso di esecuzione (come usare le eccezioni per uscire da dei loop) - Non rendere difficile l'utilizzo del codice quando lo stiamo condividendo con altri programmatori	Non devono essere utilizzate per modificare il flusso di esecuzione (come usare le eccezioni per uscire da dei loop)

4.2.2 Defining Exception Types
Al momento della definizione di un'eccezione, dichiariamo se essa è checked or unchecked, indicando il suo supertipo:
1) Se il supertipo è *Exception*, allora è checked
Il tipo *Exception* fornisce 2 costruttori, il nome del costruttore è overloaded.
Il primo costruttore inizializza l'oggetto che contiene la stringa vuota

```
Exception e2 = new NewKindOfException( );
```

La stringa, cosiccome il tipo di eccezione può essere ottenuta chiamando la funzione toString all'oggetto dell'eccezione:

String s = e1.toString(); -> s = "NewKindOfException: this is the reason"

Il secondo costruttore inizializzza l'oggetto eccezione che contiene la stringa fornita come argument (solitamente il motivo per cui un'eccezione è stata sollevata)

```
Exception e1 = new NewKindOfException("this is the reason");
```

2) Se il supertipo è *RuntimeException*, allora è unchecked.

I tipi di eccezione devono essere definiti in un package, tale package può risiedere all'interno del codice, o in una classe esterna (solitamente si raggruppano tutte le eccezioni, in modo da permettere anche il riutilizzo).

PDJ 4.2.4 Handling Exception

Al momento della chiamata del "throw" di un'eccezione, l'esecuzione di una Java procedure termina.
Quando una Java procedure termina con un'eccezione, il controllo del flusso è passato alla porzione di codice che gestisce l'eccezione.
La porzione di codice, può gestire **l'eccezione in 2 metodi**:
1) **try-catch statement and finally**
Vengono definite porzioni di codice all'interno di vari catch, che cercano di gestire l'eccezione.
2) **propagando l'eccezione**
Avviene quando la chiamata all'interno di una data procedure P invoca un'eccezione che non è definita nella clausola try-catch; in questo caso java propaga automaticamente l'eccezione al chiamante di P a patto che:
- Il tipo dell'eccezione o uno dei suoi supertipi è definita nell'header
- L'eccezione è unchecked

Altrimenti verrà invocato un compile-time error.

PDJ 4.4 Design Issues

Una cosa importante da definire è che le eccezioni **non sono sinonimo di errori**, ma sono meccanisci che permettono ad un metodo di **fornire delle informazioni al chiamante**. Inoltre, non tutti gli errori portano ad eccezioni.

Le eccezioni dovrebbero essere utilizzate per **eliminare i vincoli imposti nella clausola requires** (dovrebbe essere impiegata solo per ragioni di efficienza, o per contesti in cui l'utilizzo è così limitato che dobbiamo garantire sia soddisfatto). Vengono utilizzate per rendere **le funzioni parziali in funzioni totali**, in modo tale da elminire i $requires$, indicando in $effects$ quali e come sono le eccezioni

Quando utilizziamo checked e quando unchecked ?

1) Le **checked exception** devono **essere gestite** nel codice del chiamante, devono essere d**efinite nella clausola throws dell'header della procedura.** (oppure vi sarà un compile-time error).
2) Le **unchecked exception**, invece, saranno **propagate implicitamente** al chiamante anche qualora esse non siano definite nell'header.
Quindi scegliere se un'eccezione deve essere checked o unchecked deve basarsi **sull'aspettativa di come questa venga utilizzata.** Se l'aspettativa è quella di utilizzare **codice per gestire l'eccezione**, allora l'eccezione dovrebbe essere **checked**; altrimenti dovrebbe essere **unchecked**.

Defensive Programming

E' la pratica di scrivere procedure in grado di **"difendersi" contro gli erro**ri; ovvero in grado di continuare a funzionare anche qualora altre procedure, l'hardware, o l'utente introducano errori.

OVERVIEW:
L'utilizzo di **unchecked exception** dovrebbe essere impiegato solo se c'è l'aspettativa che l'utente che scrive codice garantisca che **che l'eccezione non venga sollevata** è un modo conveniente e poco costoso di evitare l'eccezione. Altrimenti adottare checked exceptions.

RIASSUNTO:
Exception sono necessari per costruire programmi robusti, perché forniscono un modo per r**ispondere ad errori e situazioni particolari** (eg. arguments non previsti).
Le eccezioni sono introdotte quando le procedures sono definite, qualora le procedures non siano in grado di gestire l'intero dominio, introduciamo eccezioni.
Procedure parziali dovrebbero essere impiegate solo qualora è o troppo costoso, o non è possibile verificare una data condizione.

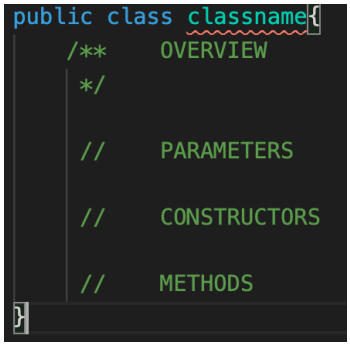
PDJ 5 Astrazione dei Dati
L'astrazione di dati ci permettere di estendere il linguaggio di programmazione che stiamo utilizzando con nuovi tipi di dati, a seconda della necessità del dominio dell'applicazione, estendendo i tipi elementari del linguaggio
Ci permette di astrarre dai dettagli di come data objects sono implementati, a come essi si comportano.
-> diamo le basi al object-oriented programming.

Quick eg.
Se vogliamo per esempio implementare un programma in grado di gestire i dati di una banca, potremmo implementare una struttura di dati più efficiente per farlo.
Partiamo da tipi più primitivi, fino a costruire una struttura dati complessa, più efficiente nel nostro contesto:
nome -> string
balance -> large-int
CF -> string

Le astrazioni di dati incorporano sia l'astrazione per parametrizzazione, che l'astrazione per specificazione.
- L'astrazione per parametrizzazione è acquisita nello stesso modo in cui lo facciamo con le procedures, astraendo i parametri quando ci è comodo farlo
- l'astrazione per specification è invece acquisita rendendo le operazioni parte del tipo, ci allontaniamo dal dettaglio implementativo, e vengono descritti i comportamenti.

Le operazioni sono incluse al tipo, formando complessivamente: $data\ abstraction = < objects, operations >$
Gli utenti, piuttosto che accedere direttamente alla rappresentazione degli oggetti chiamano le $operations$.
Quando andiamo ad implementare il tipo quindi, lo facciamo in termini della rappresentazione desiderata.
Se introduciamo abbastanza procedures, la mancanza di accesso alla rappresentazione permette all'utente di svolgere tutte le operazioni di cui potrebbe avere bisogno.

I metodi di istanza permettono di descrivere il comporatmento dell'oggetto e sono suddivisi in 4 categorie:
- Metodi di creazione, set di constructors, che servono ad inizializzare nuovi oggetti del tipo (ovvero le istanze)
- Metodi di mutazione, che vanno ad alterare lo stato degli oggetti
- Metodi di osservazione, che non alterano lo stato dell'oggetto e forniscono informazioni riguardo allo stato degli objects
- Metodi di produzione e fabbricazione, che producono altri oggetti dello stesso tipo



Per definire una nuova class si utilizza l'header *class*
Insieme al nome della class, viene associata anche la sua visibilità (public/private), la maggior parte delle classi sono definite come public, esponendo la visibilità anche all'esterno del pacchetto.

L'overview da una breve descrizione dell'astrazione di dati, così come fornire un esempio su come i dati rappresentati posso essere rappresentati, basandoci su elementi ben conosciuti
Brutalmente: faccio un esempio classico:

$$OVERVIEW: IntSet\ s\ are\ mutable, unbounded\ sets\ of\ integers$$
$$A\ typical\ intSet\ is\ S = \{x_1, \ldots, x_n\}$$

La sezione dei costruttori definisce i costruttori che inizializzano nuovi oggetti, mentre i methods forniscono all'astrazione methods per accedere a tali oggetti qualora siano stati creati.
Tutti questi elementi devono essere definiti come *public*

La parola chiave *static* non compare quando andiamo a definite methods e costructors perché appartengono agli oggetti, piuttosto che alle classi. La keyword *static* indica che l'oggetto interessato appartiene alla classe, piuttosto che ai singoli objects della classe.
Quando andiamo ad implementare all'interno di methods e constructors, l'oggetto interessato è disponibile implicitamente come argument, ed è possibile riferirvici ad esso con la keword *this*

PDJ 5.3 Implementing data abstractions
Una classe definisce ed implementa un nuovo tipo.
La specification si occupa della definizione del tipo, mentre il resto della classe provvede all'implementazione.
Quando implementiamo un'astrazione di dati, selezioniamo una representation o rep, per gli oggetti, per poi implementare methods e costructors per rappresentare/modificare correttamente la rappresentazione.

- Un record è una collezione di parametri, ognuno con un proprio nome e tipo
Possono essere comparati alle struct di C e golang. Java non provvede ad implementare questo tipo di raccolta di dati, e quindi per definirli dobbiamo utilizzare classi.

```
class Pair {
   //OVERVIEW: A record type
   int coeff;
   int exp;
```

5.4 Additional methods
Una classe di methods addizionali che tutti gli Objects hanno, sono i methods definiti da Object.
Tutte le classi definiscono sottotipi di Objects, e quindi ereditano tutte i suoi methods.
Alcuni methods degni di nota sono: equals, clone e toString

- equals, due oggetti sono equals se sono behaviorally equivalent (si comportano nello stesso modo)
Brutalmente: è impossibile distinguerli tra di loro utilizzando una qualsiasi sequenza di chiamate ai loro methods.
Objects immutabili sono equals se hanno lo stesso stato, qualora i tipi siano immutabili non è necessario implementare equals.

- clone restituisce un oggetto che ha lo stesso stato dell'oggetto interessato. Objects immutabili possono ereditare clone da Object, ma gli Objects mutabili devono implementare clone indipendentemente

- toString restituisce una stringa che illustra il tipo, e lo stato corrente dei suoi objects.
Tutti i tipi devono implementare toString indipendentemente.

Un'ulteriore method ereditato da Object è hashCode.
La specification di hashCode indica che se due oggetti sono equals secondo il metodo equals, essi dovrebbero produrre un valore hashCode equivalente.
L'implementazione di hashCode è necessaria solo qualora i tipi interessati debbano essere utilizzati come chiavi in hash tables.

PDJ 5.5 La funzione d'astrazione AF e l'invariante di rappresentazione IR

- La funzione d'astrazione rappresenta l'intenzione dell'implementatore nello scegliere una particolar rappresentazione; definisce come le istanze di variabili vengono utilizzate, e come si relazionano all'oggetto astratto che vogliono rappresentare.
- L'invariante di rappresentazione è definita durante l'implementazione dei costruttori e dei methods. Rappresenta l'insieme di proprietà che le istanze di variabili devono mantenere per mantenere valida la rappresentazione.

La funzione d'astrazione

Ogni implementazione d'astrazione di dati deve definire **come** gli **oggetti appartenenti al tipo sono rappresentati**. Questa relazione può essere definita con **la funzione d'astrazione**, che collega le **istanze di oggetti all'oggetto astratto** che essi vogliono rappresentare:

$$AF: C \rightarrow A$$

La funzione d'astrazione quindi mappa **da stati concreti a stati astratti**.
La funzione d'astrazione è quindi un potente strumento che ci permette di definire il **significato di una rappresentazione**, di come gli oggetti di una **classe dovrebbero implementare l'oggetto astratto**.
La descrizione avviene via **commento nell'implementazione**.

Incontriamo tuttavia un problema, quando andiamo a descrivere l'elemento, solitamente utilizziamo un **linguaggio informale.** Per raggirarlo, diamo una descrizione di un tipico **oggetto astratto**.
Questo ci permette di definire la funzione d'astrazione in termini di **oggetti tipici**.
Brutalmente: facciamo un esempio sull'oggetto che vogliamo astrarre descrivendo un tipico oggetto.
Per esempio, per l'IntSet scriviamo

$$A\ typical\ IntSet\ is\ \{x_1; \dots; x_n\}$$

Quando andiamo a definire la funzione d'astrazione allora scriviamo

$$The\ AF\ is\ AF(c) = \{c.else[i].intValue \mid 0 \leq i < c.els.size\}$$

Quando andiamo a definire l'invariante di rappresentazione possiamo utilizzare le notazioni di insiemi:
- Set: $\{x_1; \dots .; x_n\}$ ove gli x sono gli elementi univoci dell'insieme
- Set union: $t = s_1 + s_2$ ovvero *t* contiene l'unione degli insiemi s1 e s2, se un elemento è presente in entrambi, nell'union compare una sola volta.
- Set difference: $t = s_1 - s_2$, ovvero t contiene tutti gli elementi di s1 – gli elementi di s2
- Set Intersection $t = s_1 \& s_2$, ovvero t contiene gli elementi che sono presenti sia in s_1 che in s_2
- Cardinality: $|s|$ rappresenta la grandezza del set *s*
- Set membership: x è $in\ s$ è $vera\ sse\ x$ è $un\ elemento\ di\ s$
- Set former: $t = \{x \mid p(x)\}$, è l'insieme degli elementi di x t.c. p(x) è vera

Se ho una serie di attributi $\boldsymbol{a_o}, \boldsymbol{a_1}, \dots, \boldsymbol{a_n}$ che possono assumere un qualsiasi valore dei loro domini $A_o, A_1, \dots, A_n$, facendo il **prodotto cartesiano** dei domini si ottiene lo spazio di tutti i valori che può assumere lo stato di un'oggetto:

$$\boldsymbol{A}: \boldsymbol{A_o} * \boldsymbol{A_1} * \dots * \boldsymbol{A_n}\ \boldsymbol{insieme\ di\ tutti\ gli\ stati}$$

La funzione di astrazione **associa un certo possibile insieme di valori del dominio ad un certo $\boldsymbol{A}$**, un oggetto dell'astrazione.

L'invariante di rappresentazione

Diciamo invariante, perché è valido per tutte le **rappresentazioni valide di un oggetto astratto.**
In JAVA, il type checking garantisce che quando un method o un costruttore è chiamato, l'object *this* appartenga alla classe. Tuttavia, è possibile che non tutti gli oggetti della classe siano **valide rappresentazioni dell'oggetto astratto**.
Una dichiarazione di proprietà che garantisce che tutti gli elementi siano validi è chiamata invariante di rappresentazione, o rep invariante.

$$J: C \rightarrow boolean$$

Ovvero, è un booleano che ci garantisce che un dato oggetto sia una valida rappresentazione dell'oggetto astratto.
Può succedere che l'invariante di rappresentazione sia violato per effettuare delle operazioni, ma quando il controllo viene tornato al chiamate, l'IR deve essere verificato.
Per esempio per l'IntSet abbiamo bisogno che non vi siano 0 elementi, che gli elementi siano integers, che gli

elementi siano univoci, e che gli elementi non siano più di quelli ammessi dal size
L'invariante di rappresentazione può essere scritto anche con un linguaggio più informale

$$The\ rep\ invariant\ is\colon c.else\ != null\ \&\&\ all\ elements\ of\ c.els\ are\ integers\ \&\&\ there\ are\ no\ duplicates$$

Anche in questo caso abbiamo una serie di notazioni di predicati che ci aiutano:
- && per le congiunzione, se $p\&\&q$ è vera, sia p che q sono vere
- || per le disgiunzioni, $p||q$ è vera, se o p o q è vera
- → per le implicazioni, $p \rightarrow q$ significa che se p è vera, anche q lo è
- iff, sse

L'invariante di rappresentazione non è verificata di default, va attivata con $java - ea\ test$ con $ea = Enable\ Assertions$. Le asserzioni sono un meccanismo linguistico che permettono di verificare controlli di validità.

Implementazione di repOk e AF
- Il metodo *toString* è utilizzato per implementare la funzione d'astrazione
Serve a rappresentare lo stato di tutti gli stati validi di un oggetto astratto.
- il metodo $boolean\ repOk()$ è utilizzato per implementare l'invariante di rappresentazione
Definisce tutte i requisiti che devono essere rispettati perché l'oggetto interessato sia una valida rappresentazione.

$$EFFECS\colon returns\ true\ if\ rep\ invariants\ hold\ for\ this, false\ otherwise$$

Abbiamo quindi visto che la funzione d'astrazione definisce le interpretazioni della rappresentazione, mappa gli stati legali delle istanze ad oggetti astratti che essi vogliono rappresentare. È implementato con la funzione $toString$.
L'invariante di rappresentazione invece definisce l'insieme delle assunzioni che una qualsiasi istanza di una rappresentazione deve adibire. Definisce quali rappresentazioni sono ammesse, mappando ogni oggetto ad un valore booleano vero o falso; è implementato con la funzione repOk

Benevolent Side Effects
Qualora un'implementazione va a modificare la rep senza influire sullo stato astratto dell'oggetto diciamo che sta facendo un *benevolent side effect*. Questi sono possibili solo quando la funzione d'astrazione è *uno a molti.*
Una funzione "espone la rep" se fornisce agli utenti un meccanismo che gli permette di accedere a componenti mutabili della rep.

Local reasoning, private vs non private
Per implementare la data abstraction è necessario ottenere il **local reasoning**, ovvero essere in grado di garantire che una classe è corretta solamente esaminando il suo codice. Il local reasoning è valido solamente se la rappresentazione degli oggetti astratti non può essere modificata al di fuori della loro implementazione. Se non c'è il local reasoning la rappresentazione si dice esposta, quindi i componenti della rappresentazione sono accessibili dall'esterno della classe.

L'esposizione si verifica quando effettuiamo una dichiarazione delle variabili $non\ private$, in questi casi l'esposizione della rappresentazione risulta errata perché:
- se la rappresentazione è mutabile è possibile alterarla dall'esterno
- se espongo la rappresentazione sto esponendo anche il dettaglio implementativo, sto perdendo l'astrazione per specificazione

PDJ 5.7 Ragionamento sull'astrazione di dati.
Quando andiamo a scrivere un programma, ragionaniamo in una maniera informale.
Quando andiamo a "convincerci" che una implementazione è valida "accettiamo" le pre-condizioni e verifichiamo che il codice effettui effettivamente quello che vogliamo.
Quando tuttavia parliamo di astrazioni di dati diventa un po' più difficile, dato che dobiamo considerare l'intera classe.

Data type induction, induzione sull'invocazione delle procedure per produrre lo stato corrente dell'oggetto.

PDJ 6 Iteration Abstraction

L'astrazione iterazionale è l'ultimo meccanismo di astrazione che andiamo a vedere, rappresentano una generalizzazione dei meccanismi di iterazione.
Permettono all'utente di iterare sui dati in maniera efficiente.

$$for\ all\ elements\ of\ the\ set$$
$$do\ action$$

Quello che vogliamo andare ad implementare sono meccanismi generali d'iterazione che siano efficienti e convenienti da usare, e che preservino l'astrazione. **L'iteratore** è lo strumento che ci serve, è una procedura speciale che permette l'iterazione graduale su un gruppo di oggetti che appartengono ad un gruppo.
Per implementare un iterator utilizziamo una *procedure* speciale, chiamata iterator.

Un iteratore ($implements\ iterator$) è un metodo che restituisce un generator. Il generatore è un elemento che estrae gli elementi utilizzati nell'iterazione. Il generator tiene conto dello stato dell'iterazioni all'interno del suo rep; ha method hasNext per verificare la presenta di un successivo elemento, e method next per restituire il prossimo elemento. Un'astrazione di dati può avere più methods che vanno ad implementare differenti iterators.
Tutti gli iterator sono sottotipi dell'interfaccia Iterator.

```
public interface Iterator{
    public boolean hasNext(){
        // EFFECTS: Returns true if there are more elements to yield
        // else returns false
    }
    public Object next ( ) throws NoSuchElementException;
        // MODIFIES: this
        // EFFECTS: If there are more results to yield, returns the next result
        // and modifies the state of this to record the yield.
        // Otherwise, throws NoSuchElementException
}
}
```

Qualora non vi sia un next, viene sollevata un'eccezione incontrollata NoSuchElementException.

La specification di un iterator descrive come l'iterator utilizza gli arguments per produrre un generator, e come questo si comporta.

$$class\ elements\ implements\ Iterator < Integer > \{\dots\}$$

PDJ 6.4 Implementing iterators

Quando andiamo ad implementare un iterator, lo facciamo con riferimento all'elemento alla quale si riferisce l'iterator. Supponiamo di dover iterare su una lista di interi

$$List < Integer > lst = List.of(1,2,3,4)$$

Per creare un iteratore definiamo:

$$Iterator < Integer > \ it = lst.iterator();$$

Utilizzando i methods hashNext() e next possiamo lavorare sui singoli elementi dell'iterator.
- L'implementazione di un iterator è legata all'implementazione di una data class
- Il generator è una *static inner class*, è implementata all'interno della class.
- Dobbiamo definire la AF e la IR anche per i generator.

Un iterator è una procedura che restituisce un generator.
Il generator è l'oggetto che produce gli elementi utilizzati in un'iterazione, è dotato di methods per ricavare il prossimo elemento, e per verificare la presenza di tali. Tutti i generator sono sottotipo di Iterator.

Dobbiamo andare a definire funzioni di astrazione e invarianti di rappresentazione anche per gli iterator.
- L'invariante di rappresentazione per i generatori è definito in maniera simile ai tipi astratti, la differenza è che non viene fornita un method verificarli, l'invariante di rappresentazione è espresso attraverso istanze dell'oggetto (ne

consegue che l'oggetto non può essere null).
- La funzione di astrazione di un generatore è definito come segue: tutti i generatori hanno lo stesso stato astratto, una sequenza di oggetti rimanenti che deve essere generata. Ne consegue che la funzione di astrazione deve mappare la rappresentazione di questi oggetti alla sequenza.

SUMMARY
Siccome spesso dobbiamo svolgere elementi su un'intera collezione di elementi, abbiamo bisogno di un modo efficiente, conveniente e sicuro per accedere a questi elementi.
Gli iterator restituiscono un tipo speciale di oggetto, i generator, che restituiscono elementi di una collezione uno alla volta. Questa gradualità garantisce che qualora un elemento desiderato sia trovato, il generator può essere fermato; così come rendere superflua l'utilizzo di spazio esterno.
Gli iterator astraggono per specification incapsulando il modo in cui gli elementi vengono prodotti.

PDJ 7 Type Hierarchy
L'ereditarietà si basa sul concetto d'estensione dei tipi attraverso la creazione di sottotipi:
Le ragioni concettuali sul perché utilizzare l'ereditarietà sono:
- Ragioni ontologiche, quando un elemento è interno ad un altro elemento
Ci sono casi specifici in cui semanticamente l'ereditarietà è necessaria (is-a)
- Introduzione di comportamento specializzati:
Intenzione di aggiungere determinati comportamenti
Introduzione di comportamenti specifici
- Provvedere differenti implementazioni: (poly-sparse)
Per migliorare sotto il punto di vista di efficienza, spazio, tempo etc. a seconda delle necessità

Gerarchia dei tipi/Ereditarietà e composizione
- Una **gerarchia di tipi** è utilizzata per definire **"famiglie" di tipi**, formate da **supertipi ed i loro sottotipi**.
La gerarchia può estendersi anche su molti livelli
- In generale: i sottotipi **estendono il comportamento del supertipo** (eg. fornendo ulteriori methods)
- LSP: LISKOV SUBSTITUTION PRINCIPLE **garantisce che il comportamento del sottotipo sia garantito in concomitanza della specificazione del suo supertipo.**
Una volta definito il comportamento per un **dato supertipo**, il codice continua a funzionare anche se a **questo sottotipo è sostituito un suo supertipo**.
Il comportamento del supertipo deve essere garantito da tutti i suoi sottotipi.

"Se S è un sottotipo di T, oggetti di tipo S devono comportarsi come oggetti del tipo T, qualora vengano trattati come oggetti di tipo T."

Una gerarchia di dati impone che gli elementi di una data famiglia abbiano un **comportamento legato.**
Questo ci permette di **scrivere codice nei termini** della **specificazione del supertipo**, garantendo tuttavia il corretto funzionamento anche quando queste vengano applicate ad oggetti del sottotipo.
Eg. codice scritto in termini del $Reader$ funzionano anche con $BufferedReader$

Questo principio ci permette di acquisire **un'astrazione per specificazione** per una famiglia di tipi. Astraiamo dalle **differenze tra i sottotipi**, concentrandoci sulle loro somiglianze, che sono catturate dalla **specificazione del supertipo.**

PDJ 7.1 Assignment e Dispatching:
Qualora S sia un sottotipo di T, gli oggetti S possono essere assegnati a valori di variabili di tipo T, e possono essere utilizzati come arguments ove sono previsti risultati di tipo T.
eg. DensePoly e SparsePoly sono sottotipi di Poly, siccome questi sono sottotipi di Poly, possiamo assegnare a loro un tipo Poly senza rompere l'astrazione.
Per distinguere i tipi, sfruttiamo l'apparent type e l'actual type:
- L'apparent type/tipo apparente è il tipo che il compilatore deduce dalle informazioni presenti nelle dichiarazioni
- L'actual type/tipo reale è il tipo che l'oggetto effettivamente ha.

L'actual type è sempre un sottotipo del tipo apparente (consideriamo inoltre un tipo essere sottotipo di sé stesso)
Il compilatore per fare i vari controlli utilizza il tipo apparente.

Dispatching
Il codice eseguito dipende dal tipo reale degli oggetti, tuttavia il compilatore conosce solo il tipo apparente.
- Il compilatore deduce il tipo apparente di ogni oggetto utilizzando le operazioni nelle variabili, e i metodi di dichiarazione.
- Ogni object ha un tipo reale, che riceve quando esso viene creato, definito dalla classe che lo costruisce.
- Il compilatore garantisce che il tipo apparente sia sempre un supertipo del tipo reale di un oggetto.
- Il compilatore determina quali operazioni sono ammissibili a seconda del tipo apparente dell'oggetto

PDJ 7.2 Defining a Type Hierarchy
La prima cosa da definire quando andiamo a creare una gerachia di tipi è **definire il supertipo in cima**.
La specification di questo supertipo può essere **incompleta**, per esempio priva di costruttori.
Quando andiamo a definire **sottotipi di questo supertipo** lo facciamo in modo **relativo**, specificando come questa implementazione va ad estendere il **comportamento del supertipo.**

Le gerarchie di dati definite in Java utilizzando il **meccanismo di ereditarietà.**
Questo permette ad una classe la possibilità di essere una **sottoclasse** di una o più classi.
I supertipi in JAVA sono definiti dalle loro **classi e dalle loro interfacce.**
Ci sono 2 tipi di classi, **classi concrete e classi astratte**
- Le **classi concrete** provvedono alla **completa implementazione del tipo**.
- Le **classi astratte** provvedono implementare **parzialmente il tipo**: non hanno objects, ed il loro codice non può chiamare i suoi costruttori. Classi astratte, inoltre, possono avere $abstract\ methods$, methods che non sono implementati dalla superclasse, e quindi devono essere implementati dalla $subclass$
Entrambi i tipi di classi contengono methods normali e finals methods, tuttavia quest'ultimi non possono essere reimplementati dalle sottoclassi.
(per garantire che il comportamento non sia modificabile da una qualsiasi sottoclasse).

Una sottoclasse dichiara la sua superclasse inserendo nell'header la parola chiave $extends\ class$.
Questo permette alla sottoclasse di **ereditare tutti i methods** della superclasse.
Nel caso volessimo reimplementare dei methods della superclasse, usiamo la parola chiave override.
Ogni metodo che andiamo a **sovrascrivere** deve avere una $\boldsymbol{signature}$ **identica** a quella del supertipo.

- **Un supertipo** è definito o dalla classe o da **un'interfaccia**, che provvedono a fornire la sua specification, e nel caso della classe, una parziale o completa implementazione.
- **Una classe astratta** provvede a fornire solo una **parziale implementazione**, non contiene objects, e non ha costruttori che l'utente può chiamare.
- Una sottoclasse può ereditare l'implementazione della sua superclasse, ma può anche sovrascrivere (con il comando override) alcune implementazioni (non finali).

Implementazione:
- (Grado 0) Il supertipo che non implementa nessun comportamento si chiama **interfaccia** (questo serve quando voglio fare diverse implementazione dello stesso tipo). E' una collezione di metodi con **segnature** che non contengono **nessuna implementazione, non ha attributi, non ha rappresentazione** e contiene solo metodi pubblici. Le interfacce sono utilizzate per definire una base comune di **comportamenti** a tutte le classi che la implementano.
- (Grado intermedio) Nelle classi **astratte** ci sono alcuni metodi che sono implementate e ci sono alcuni metodi che specificano il comportamento e basta come nelle interfacce.
- Una classe **concreta** e una raccolta di metodi e di attributi che realizza concretamente un certo insieme di comportamenti
- Le classi concrete e astratte estendono i loro supertipi. Le interfacce sono implementate dai loro sottotipi.

PDJ 7.5 Concrete Classes
Sono classi in cui gli attributi sono ereditati dal supertipo (qualora siano public); ed in cui ne sono definiti di nuovi.
RI e AF: l'RI dovrà essere estesa per parlare anche dei nuovi attributi, oltre a quelli ereditati dal supertipo.
Contiene nuovi metodi d'istanza, oltre a quelli ereditati. I methods possono essere finals se necessari.
eg. MaxIntSet < IntSet

PDJ 7.6 Abstract Classes
Una classe astratta provvede solo all'implementazione parziale di un tipo.
I costruttori non possono essere chiamati dall'utente. Le classi astratte servono a definire dei comportamenti.
Può contenere metodi astratti $abstract\ T\ f()$; i sottotipi provvederanno a fornire un override di f.

PDJ 7.7 Interfaces
Nelle interfacce definisco solo comportamenti (ad esempio LIST); ho più implementazioni di LIST (LinkedList, ArrayList), con stessi comportamenti, ma efficienza ed implementazioni diverse.
Una classe è utilizzata per definire un tipo, e per provvedere alla sua parziale o completa implementazione.
L'interfaccia, invece, si limita a definire il tipo.
Contiene solo nonstatic, public methods, e tutti i suoi metodi sono astratti.
L'interfaccia non provvede ad implementare niente, l'implementazione è fornita da una classe che ha una clausa implements nell'header.
Le interfaces consentono di specificare i comportamenti (contratti), ma non ci impegna nella scelta della rappresentazione.
La gerarchia ci permette di implementare in più modi un tipo. Per esempio, nonostante DensePoly e SparsePoly siano 2 implementazioni differenti di polinomiali, entrambi appartengono alla stessa rappresentazione Poly.

PDJ 7.9 The meaning of Subtypes
Ereditarietà e composizione - Ragionamento sui sottotipi S < T
Tutti i sottotipi in java devono soddisfare il principio di sostituzione di LISKOV:

Il principio di sostituzione = Il principio di sostituzione LISKOV
Dato un comportamento definito nel tipo, qualora sostituissi un sottotipo, questo comportamento deve essere garantito anche in quel contesto.

Quando andiamo a creare dei sottotipi dobbiamo tener conto di alcune considerazioni:
- LSP: il tipo concreto può sempre essere quello di un supertipo
- Regola delle signature:
Tutti gli oggetti sottotipo devono possedere i methods del supertipo, e la signature del sottotipo devono essere compatibili con la signature dei corrispondenti methods del supertipo.
- Regola dei metodi:
Ciascun metodo del sottotipo deve comportarsi semanticamente come quello del supertipo.
- Regola delle proprietà:
Se delle proprietà valgono per un supertipo, esse devono valere anche per tutti i suoi supertipi

Il principio di sostituzione della LISKOV è garantito da queste 3 regole:
1) Regola delle signature/Signature Rule(gestita automaticamente dal compilatore)
Tutti gli oggetti sottotipo devono possedere i methods del supertipo, e la signature del sottotipo devono essere compatibili con la signature dei corrispondenti methods del supertipo.
Brutalmente: Un comportamento garantito nel tipo, deve essere garantito anche nel sottotipo, tuttavia il comportamento può essere esteso. Un sottotipo deve presentare tutti i methods di un supertipo con una signature compatibile
Ovviamente i metodi devono essere compatibili, possono cambiare le eccezioni (con tipi più specifici), e i tipi di covariante return (il sottotipo essendo più specifico, anche il return può essere più specifico)
Il meccanismo che va a garantire la validità di questo concetto è l'ereditarietà, un sottotipo eredita tutti i tipi del supertipo quindi questa regola è sempre vera

La signature rule garantisce che ogni chiamata sollevata che è corretta secondo la specification del supertipo è valida anche per tutti i sottotipi.

2) Regola metodi/Methods Rules
La semantica dei methods di un sottotipo deve essere simile a chiamate dei corrispondenti methods del supertipo.
Brutalmente: Tutti i metodi si devono comportare come i metodi del supertipo

La methods rule si concentra sulle chiamate sui methods definiti del supertipo
Al momento della chiamata, gli oggetti interessati che appartengono al sottotipo, vanno a cercare il codice che viene definito ed implementato dal sottotipo.
Tuttavia qualora un method venga sovrascritto, c'è potenzialmente spazio per incontrare un errore.

(slip-in: precondizioni: definita dalla clausola requires, definisce cosa deve essere garantito dal chiamante, per effettuare la chiamata)
post-condizioni: definita dalla clausola effects, definisce cosa deve essere garantito subito dopo la chiamata)

Un metodo del sottotipo può indebolire le pre-condizioni e rafforzare le post-condizioni
Entrambe queste condizioni devono essere soddisfatte per garantire la compatibilità tra i methods coinvolti

A subtype method can *weaken* the precondition and can *strengthen* the postcondition.

- *Precondition Rule:* $\text{pre}_{super} => \text{pre}_{sub}$
- *Postcondition Rule:* $(\text{pre}_{super} \&\& \text{post}_{sub}) => \text{post}_{super}$

- Indebolimento delle precondizioni
indebolire le precondizioni significa che i methods del sottotipo sono meno stringenti rispetto a quelli dei methods del supertipo. Questo funziona, in quanto quando andiamo a scrivere il codice del sottotipo, lo facciamo in termini della specification del supertipo, che devono quindi essere soddisfatte. Convinti di questa precondizione forzata sul sottotipo possiamo essere sicuri che una chiamata ad un method del sottotipo sia legale, se la chiamata al method super tipo è legale.

Siccome se sono vere le precondizioni del supertipo, sono vere anche le precondizioni del sottotipo, possiamo essere sicuri che le precondizioni del sottotipo siano valide, se sono soddisfatte quelle del supertipo
Date le precondizioni di T, non possiamo garantire che siano valide le postcondizioni di S

- Rafforzamento delle postcondizioni
Tuttavia, solo soddisfare le precondizioni non è sufficiente a garantire che la specification di un method di un sottotipo sia corretto, dobbiamo infatti tener conto anche degli effetti di quest'ultima. Questi effetti sono definiti nella post-condition rule.
Questa garantisce che la method di un sottotipo fornisce sempre tutto quello che fornirebbe la method supertipo, più eventualmente qualcosa di più. $method\ subtype \geq method\ supertype$

3) Regola proprietà/Properties Rule(provabile con induzione/sostituzione)
(la rottura di questa proprietà può avvenire con la ricerca di un controesempio)
Oltre a ragionare sugli effetti delle singole chiamate, dobbiamo anche soffermarci sulle proprietà degli oggetti.
Alcuni oggetti sono invarianti, valgono per tutti gli oggetti legali.
Per mostrare che un sottotipo sia coerente con la properties rule dobbiamo provare che preserva tutte le proprietà dei supertipi.
- La properties rule garantisce che le proprietà che valgono su un supertipo, valgano anche su un suo sottotipo.
Le proprietà devono essere specificate nella sezione specification del supertipo.

Nonostante implementazioni differenti in sottotipo, una serie di proprietà degli oggetti che deve rimanere valida è solitamente presente. Parliamo di proprietà del' RI che sono valide trasversalmente.
Per esempio quando parliamo di IntSet, che sia un IntSet ordinato, un IntSet al contrario etc, il numero di elemento deve sempre essere maggior o uguale a 0.
Il sottotipo deve mantenere tutte le proprietà che possono essere provate dagli oggetti del supertipo.
Brutalmente: data una qualsiasi proprietà che funziona nel programma per un determinato tipo T, questa proprietà deve valere anche per tutti i sottotipi S di T.

Tutte queste regole si riferiscono solo alla specification
Le regole 2 + 3 non sono computabili, non possono essere gestite dal compilatore

Overview:
- La signature rule garantisce che se un programma è type-correct rispetto alla specification del supertipo, allora è type-correct anche rispetto alla specification del sottotipo.
- la methods rule garantisce che i comportamenti definiti da un method del supertipo sia valido anche quando esso è implementato da un suo sottotipo
- la properties rule garantisce che le proprietà degli oggetti definiti nella specification del supertipo sia validi anche qualora essi appartengano a oggetti di sottotipi.

Metodo equals
(è possibile progettare un metodo equals che sia soddisfi il contratto di equals = soddisfa transitività e simmetria ?)
Brutalmente: è praticamente impossibile mantenere il metodo equals valido
(fornire un equals che sia valido sia per S che per T)
Quando implementiamo un sottotipo, e il sottotipo aumenta lo stato, il metodo equals non è implementabile.
Non è possibile garantire la simmetria e la transitività.
La soluzione è di dividere i metodi equals per ogni tipo.

PDJ 7.11 SUMMARY:
L'ereditarietà è utilizzata per definire tipi di famiglie, e multiple implementazioni.
Questa modalità ci permette di definire nuovi tipi di astrazione nella quale il programmatore astrae dalle proprietà di un dato gruppi di tipi legati, per identificare quali proprietà questi tipi hanno in comune.
Questo viene fatto per:
- Facilitare la comprensione
- La gerarchia ci permette di definire astrazioni che funzionano su un'intera famiglia di tipi.
(per esempio, un methods che funziona su un supertipo, funziona anche per un suo sottotipo)
Codice scritto in termini di un supertipo, può funzionare anche per ulteriori tipi (tipi sottotipo del supertipo)
- Ci fornisce un tipo di estensionalità, permettendoci di aggiungere sottotipi qualora necessario, o estendere il comportamento di methods già presenti.
- Questi benefici possono essere ottenuti sse i sottotipi rispettano il LSP

Polimorfismo e generici (generici di JAVA)
Concetto di astrazione polimorfa
eg. Avendo introdotto collezioni come ints (IntSet), se volessimo impiegarle per contenere un set di Stringhe dovremmo implementare una collezione completamente nuova.
Dover implementare una nuova astrazione di collezione ogni volta è poco efficiente, introduciamo quindi il concetto di Astrazione Polimorfa. (polimorfa perché sono in grado di funzionare per molti tipi.

Le astrazioni polimorfe generalizzano il livello su cui si basa l'astrazione, permettendo loro di funzionare con un numero arbitrario di tipi.
Ci risparmiano la necessità di ridefinire l'astrazione per ogni situazione in cui vogliamo impiegarle, permettendoci di impiegare una singola astrazione che diventa più dinamicamente impiegabile.
Una procedura o un iteratore può essere polimorfa rispetto ai tipi di uno o più arguments.
Un'astrazione di dati può essere polimorfa rispetto a tutti gli elementi che contiene.

Questo concetto può essere implementato generalizzando gli arguments in entrata dei metodi/costruttori a Object.
Ciò implica che gli oggetti primitivi in collezioni devono essere Object, e alcuni devono essere wrappati nel tipo corrispondente.
eg.
Se voglio inserire un integer, devo prima wrapparlo (dato che non è sottotipo di Object), e al momento del return devo unwrapparlo.

Polimorfismo “ad hoc”, polimorfismo che avviene adottando 2 tecniche:
- Overloading, ci permette di fornire una forma più specifica del metodo/comportamento.

$$x.function(something), overloading\ depends\ on\ something$$

- Overriding, ci permette di fornire ad un sottotipo un comportamento differente.

Ogni sottotipo può implementare un comportamento differente, a seconda della necessità (per esempio toString())

$$x.function(something), overriding\ depends\ on\ x$$

Le astrazioni polimorfe sono desiderabili perché ci permettono di astrarre dal tipo dei parametri, permettendoci di creare un’astrazione più dinamica, in grado di funzionare per più di un singolo tipo.

Questo concetto può applicarsi a procedures, iterators e data abstractions.
Solitamente un’astrazione polimorfa necessità di determinati methods per accedere ai propri parametri.

Solitamente questi methods sono quelli che derivano dal tipo Object.
Nel caso in cui ulteriori methods siano necessari, l’astrazione polimorfa impiega l’utilizzo di interfacce.
Ci sono 2 metodi per definire le interfacce:
1) Implementare un’interfaccia che può essere intesa come un supertipo del tipo dell’elemento
(element-subtype approach) ogni potenziale tipo dell’elemento deve essere definito come sottotipo dell’interfaccia
- Necessità di preplanning

2) Implementare l’interfaccia in modo tale che sia un supertipo dei tipi che sono legati al tipo appartenente dagli oggetti (related subtype approach)
Deve essere implementata un’interfaccia per ogni sottotipo legato

OVERVIEW:
- Nell’approcio element-subtype, tutti i potenziali tipi di elementi devono essere sottotipo dell’interfaccia associata.
- Nell’approccio related-subtype, un sottotipo dell’interfaccia deve essere definita per ogni potenziale tipo di elemento.

PDJ 9 Specifications and Specificand Sets

Lo scopo di una specification è quello di definire il comportamento di un’astrazione.
Un’implementazione che fornisce il comportamento descritto si dice che “soddisfa” la specification.
La specificand set della specification è l’insieme dei moduli dei programmi che soddisfano una data specification fornita.
Consideriamo una funzione:

```
static int p (int y)
   // REQUIRES: y > 0
   // EFFECTS: Returns x such that x > y.
```

Lo specificand set di questa specification è l’insieme di tutte le procedure, che quando chiamate con un valore maggiore di 0 restituiscono un valore maggiore dell’argument.
In questo caso la specificand set è infinito. (l’insieme di tutte le funzioni che restituiscono $x > y$

Le specification devono tener conto di una serie di principi per essere considerate valide:
- devono essere sufficientemente restrittive, per escludere le implementazioni che non sono adatte all’astrazione richiesta, ma sufficientemente generali e chiare
- devono essere abbastanza generali, per garantire che il numero minore possibile di implementazioni siano escluse.
- devono essere abbastanza chiare, per facilitare la comunicazione delle intenzioni tra utenti, non essere eccessivamente verbose

La specification costituisce l’accordo tra utente e programmatore, il programmatore scrive un modulo che appartenga allo specificand set, e l’utente accetta l’implementazione senza sapere come avviene l’implementazione.
Un’astrazione è intangibile, senza una descrizione non ha significato, e la specification fornisce questa descrizione.
Questa descrizione fornisce informazioni riguardo a come l’astrazione è acquisita, permettendone l’utilizzo appropriato.