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

Introduzione
Il linguaggio JAVA
Astrazione procedurale
Eccezioni
Astrazione dei dati
Iterazione
Ereditarietà e composizione
Polimorfismo e generici
Specificazione, progetto e implementazione

Lezione	Data	Argomento	Materiale didattico
T01	M 29/9	Introduzione	PDJ 1
T02	V 2/10	Il linguaggio Java	PDJ 2 *1 - 3*; JT Getting Started, Language Basics, Packages
T03	M 6/10		PDJ 2 *4, 5*; JT Classes and Objects
T04	V 9/10		PDJ 2 *6 - 8*; JT Lists, IO Streams; E t04
T05	M 13/10	Astrazione procedurale	PDJ 3; [How to Write Javadoc, Javadoc Guide]; H t05
E01	V 16/10		E e01
T06	M 20/10	Eccezioni	PDJ 4; EJ 10; JT Exceptions
E02	V 23/10		E e02
T07	M 27/10	Astrazione dei dati	PDJ 5 *1 - 3*; EJ 2 *1, 2, 4*; H t07
E03	V 30/10		E e03
T08	M 3/11		PDJ 5 *4 - 6*; EJ 3 *10 - 13*; [JT Programming With Assertions]; H t08
E04	V 6/11		E e04
T09	M 10/11		PDJ 5 *7 - 9*; EJ 4 *15 - 17*
E05	V 13/11		E e05
T10	M 17/11	Iterazione	PDJ 6; EJ 4 *24*, 9 *58*; JT Access control, Nested Classes, Anonymous Classes, For-each; H t10, t11
T11	V 20/11		
T12	M 24/11	Ereditarietà e composizione	PDJ 7 *1 - 7*; EJ 4; H t12
E06	V 27/11		E *e06*
T13	M 1/12	Polimorfismo e generici	PDJ 7 *8 - 11*; JT Default Methods
T14	V 4/12		PDJ 8; JT Generics
E07	V 11/12		E *e07*; JT Collections; C. (documentation); C. (Bloch)
T15	M 15/12		EJ 5 *26 - 32*; JT Generics (Bracha)
T16	V 18/12	Specificazione, progetto e implementazione	PDJ 9, 13, 14
E08	V 8/1		
	M 12/1	Q&A / Simulazione d'esame	

PDJ 1
Il processo di astrazione può essere vista come l'applicazione di una mappatura molti-a-uno.
Ci permette di "dimenticare" informazioni 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)
- L'astrazione dei dati (ci permette di implementare nuovi tipi di dato)
- 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 dati 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
JAVA è un object-oriented language
Ciò significa che la maggior parte dei dati gestiti dai programmi java sono contenuti in objects.
Gli objects comprendono sia i loro stati, cosiccome 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 utilizzate per definire collezioni di procedure e per definire nuovi tipi di dato, 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: permettono di condividere 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.
Per richiamare elementi da altri parchetti è possibile utilizzare il loro "fully-qualified-name", che consiste nel loro nome, insieme al nome gerarchico del pacchetto. Eg. mathRoutines.Num
2) naming
Ogni pacchetto ha una gerarchia che li contraddistingue da tutti gli altri pacchetti.
Le classi e le interfacce all'interno di un pacchetto sono relative al nome del pacchetto.

PDJ2.3
Tutti i dati presenti in JAVA sono accessibili attraverso variables.
Le local variables, risiedono nel runtime stack, ed 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

Method Call Semantics
Alla chiamata di un metodo, eg e.m(…) il compilatore valuta prima, cercando di ottenere la classe o l'oggetto sulla quale è stato chiamato un metodo.
Questo viene svolto per cercare di ottenere un actual parameter, utilizzato per creare spazio per i parametri formali del metodo. Poi 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 strongly-typed-language, 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 una signature a seconda dei tipi delle variabili, degli header dei metodi, ed i

costruttori (il tipo degli arguments ed i suoi risultati).
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.
Iprogrammi 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 se stesso)
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.

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.
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 -in objects-.
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.

```
public Integer(int x) // the constructor int n = Integer.parseInt(s);
public int intValue( ) // the method
```

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))

```
System.in      // Standard input to the program.
System.out     // Standard output from the program.
System.err     // Error output from the program.
```

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

```
public static void main(String[ ])
```

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:
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.
spunto:
- L'astrazione per parametrizzazione astrae rispetto all'identità dei dati utilizzati.
- L'astrazione per specificazione ci permette di concentrarci su "cosa viene fatto", permettendoci di astrarre dal "come" un determinato comportamento è implementato
Benefici:
- Locality, l'implementazione di un'astrazione può essere letta o scritta senza la necessità di analizzare le implementazioni presenti di altre astrazioni
- Modificability, un'astrazione può essere reimplementata senza la necessità di cambiamenti a astrazione che ne fanno già utilizzo (modularizzazione)

Specificazione:
Le specificazioni possono essere scritte in un linguaggio formale, o informale (formali per precise)

1) Header, fornisce il nome alla procedura, il numero, l'ordine, il tipo di parametri, e il tipo di risultato
   Vengono anche fornite eventuali eccezioni, se presenti

```
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:
- Requires, 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)
- Modifies, definisce i nomi degli input che vengono modificati dalla procedura
Che cosa sto modificando ?
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)
- Effects, descrive il comportamento della procedura per tutti gli input non esclusi dalla clausola requires.
Cosa fa questo procedure?
Definisce inoltre quali output vengono prodotti, e quali modifiche vengono svolte sugli input come definito nella clausola modifies.

L'implementazione:
L'implementazione di una procedura dovrebbe produrre il comportamento definito nella sua specification.
Dovrebbe 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.

Linee guida nella implementazione:
- Minimality: una specificazione è più minimale di un'altra se contiene meno limiti sui comportamenti concessi
- Undertermined Behavior: una procedura è indeterminata per alcuni input se la sua specificazione ammette più di un risultato

- Deterministic Implementation: l'implementazione di una procedura è deterministica, se per gli stessi input, produce sempre gli stessi output.
- Generality: una specification è più generale di un'altra se è in grado di gestire una più larga classe di input.

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 sollevale 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 sono spesso formati 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

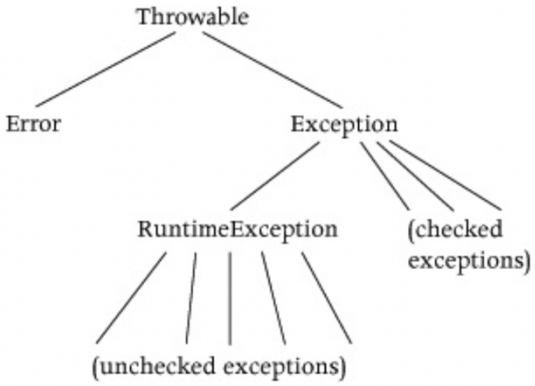
Introduciamo quindi il concetto di eccezioni.
Specifications 4.1
Con la clausola *throws* una procedura può terminare eccezionalmente.

```
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; cosiccome il motivo di perché queste si possono presentare.

4.2

I tipi di eccezioni sono o sottotipi di *Exception* o *RunTimeException*, che sono entrambi sottotipo di *Throwable*.



Ci sono principalmente 2 tipi di eccezioni:

1) Checked exception

Sono sottotipi di *Exception,* ma non di *RunTimeException*

2) Unchecked exception

Sono sottotipi di *RunTimeException*

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.

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).

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, gestisce l'eccezione principalmente in 2 metodi:

1) try-catch statement

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.

4.4 Design Issues
Una cosa importante da definire è che le eccezioi 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).

Quando utilizziamo checked e quando unchecked ?
1) Le checked exception devono o essere gestite nel codice del chiamante, o devono essere definite 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 id come questa venga utilizzata.
Se l’aspettativa è quella di utilizzare codice per evitare chiamate che possano invocare l’eccezione, allora l’eccezione dovrebbe essere unchecked; altrimenti dovrebbe essere checked.

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
Perché è un modo conveniente e poco costoso di evitare l’eccezione, ed il contesto d’utilizzo è locale.
Altrimenti adottare checked exceptions.

Defensive Programming
E’ la pratica di scrivere procedure in grado di “difendersi” contro gli errori; ovvero in grado di continuare a funzionare anche qulora altre procedure, l’hardware, o l’utente introducano errori.

RIASSUNTO:
Exception sono necessari per costruire programmi robusti, perché forniscono un modo per rispondere 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.

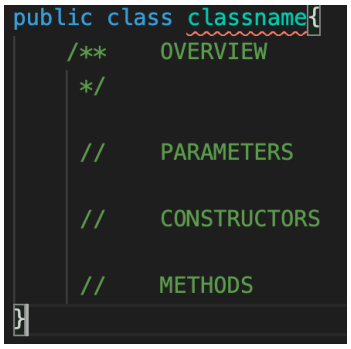
Astrazione dei Dati PDJ 5
Ci permette di astrarre dai dettagli di come data objects sono implementati, e di come si comportano.
-> diamo le basi al object-oriented programming.
L’astrazione di dati ci permettere di estendere il linguaggio di programmazione che stiamo utilizzando con nuovi tipi di dati,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

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.

Ogni classe definisce un tipo definendo una serie di oggetti:
- un set di constructors, che servono ad inizializzare nuovi oggetti del tipo (ovvero le istanze)
- un set di istance methods/methods
Una volta creato un'istanza, l'utente può accede all'oggetto attraverso la chiamata dei suoi methods.

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, in modo tale da vere visibilità anche all'esterno del pacchetto.



L'overview da una breve descrizione dell'astrazione di dati, cosi 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 il method 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 riferirici ad esso con la keword *this*

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.

Con la parola chiave *static* dichiariamo che le istanze di variabili appartengono ad objects, che sono distinti per ogni istanza di object.

- 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
- Due oggetti sono equals, se sono behaviorally equivalent (si comportano nello stesso modo)
Brutalmente: è impossiible 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 indipendetemente.
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 devono essere utilizzati come chiavi in hash tables.

5.5 La funzione d'astrazione AF e l'invariante di rappresentaizone IR
- La funzione d'astrazione rappresenta l'intenzione dell'implementatore nello scegliere una particolar erappresentazione; 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 quinid 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, solitament eutilizziamo un linguaggio informale. Per raggirarlo, diamo una descrizione di un tipico oggetto astratto.
Questo ci permette di definire la funzione d'astraizone in termini di oggetti tipici.
Brutalmente: facciamo un esempio sull'oggetto che vogliamo astrarre descrivendo un tipico oggetto.
Per esempio, per l'IntSet scriviamo

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

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 inserimi:
- Set: $\{x_1; \ldots; x_n\}$ 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

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 unn 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

Implementazione di repOk e AF
- Il method *toString* è utilizzato per implementare la funzione d'astrazione
Serve a rappresentare lo stato di tutti gli stati validi di un oggetto astratto.
- il methods $boolean\ repOk()$ è utilizzato per implementare l'invariante di rappresentazione
Definisce tutte i requisiti che devono essere rispettati perché l'oggetto interessato sia una valida rappresentazione.
$EFFECS\colon returns\ true\ if\ rep\ invariants\ hold\ for\ this, false\ otherwise$

Benevolent Side Effects
Qualora un'implementazione va a modificare la rep senza influisce 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.*

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.

5.8.1
Le astrazioni di dati possono essere mutaibli (quando i valori degli object possono cambiare), o immutaibli.
- Una AD è mutaible se ha metodi mutazionali, altrimenti la AD è immutabile.
- Ci sono 4 tipi di operazioni:
1) methods creazionali, che creano nuovi object (costruttori)
2) methods producers, che producono nuovi object a partire da object esistenti come arguments
3) methods mutazionali, che modificano lo stato degli object
4) methods osservazionali, che forniscono informazioni riguardo allo stato degli objects

Iterazione PDJ 6
Sono una generalizzazione dei meccanismi di iterazione.
Permettono all'utente di iterare sui dati in maniera efficiente.

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

In Java per implementare un iterator utilizziamo una *procedure* speciale, chiamata iterator.
Un iteratore è una procedure che restituisce un generator; un'astrazione di dati può avere più methods che vanno ad implementare differenti iterators. 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.
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.

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

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.

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; cosi 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

PDJ7 8 – 11:

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

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 fornisce l'astrazione per specificazione per famiglie di tipo, garantendo 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 sottotipo del supertipo.

"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 gerachia di dati impone che gli elementi di una data famiglia abbiano un comportamento legato.
Il comportamento del supertipo deve essere garantito da tutti i suoi supertipi.

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 dai tipi, ci riferiamo all'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.
Questo tipo è 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

Definizione di una gerarchia di tipi
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.
Entrambi i tipi di classi contengono methods normali e finals methods.
I final methods 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, cosiccome permettere di definire nuovi methods.
Nel caso volessimo reimplementare dei methods della superclasse, usiamo la parola chiave override.

- Un supertipo è definito o dalla classe o da un'interfaccia, che provvedono a fornice 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).

7.5 Exception types
I tipi delle eccezioni sono sottotipi di *Throwable*, l'implementazione di tale provvede methods che permettono l'accesso all'interno dell'oggetto eccezione.

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.

7.7 Interfaces
Una classe è utilizzata per definire un tipo, e per provvedere alla sua parziale o completa implementazione.
L'interfaccia, invece definisce solo 1 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.

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.

E' necessario quindi che le specifications dei sottotipi supportino il ragionamento sulla quale si basa la specifications del supertipo

Queste 3 regole implicano il mantenimento del principio di sostituzione della LISKOV:

1) Regola per 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
Ovviamente i metodi devono essere compatibile, 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)

2) Regola metodi/Methods Rules
Le chiamate dei methods sottotipo devono comportarsi come le chiamate dei corrispondenti methods del supertipo
Brutalmente: Ciascun metodo del sottotipo si comporta come si comporta il metodo del supertipo

3) Regola proprietà/Properties Rule(provabile con induzione/sostituzione)
(la rottura di questa proprietà può avvenire con la ricerca di un controesempio)
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

Signature Rule
La signature rule garantisce che se un programma è corretto, sulla base della specification del supertipo, allora è corretto anche rispetto alla specification del sottotipo

Method Rule / Specificazione
La methods rule si concentra sulle chiamate su 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
- Indebolimento delle precondizioni,
i methods del sottotipo necessitano di meno dal chiamate, di quello richiesto dai methods del supertipo

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

- *Precondition Rule:* $pre_{super} => pre_{sub}$
- *Postcondition Rule:* $(pre_{super} \&\& post_{sub}) => post_{super}$

Le precondizioni del tipo implicano le precondizioni del sottotipo
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
Qualora valgono le precondizioni di T e valgono le postcondizioni del sottotipo, devono essere vere le postcondizioni di T

Regola dei metodi + LSP
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.

Properties rule:
- La properties rule garantisce che il ragionamento sulle proprietà di un oggetto basato sulla specificazione del supertipo sia valido anche quando quell'oggetto garantisce ad un 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 InteSet al contrario etc, il numero di elemento deve sempre essere maggior o uguale a 0.

Tipi di supertipo:
- Supertipi incompleti, che stabiliscono convenzione di nomenclatura per methods dei sottotipi, ma non provvedono fornire le specification per quei metodi.
- Supertipi completi provvedono a fornire l'intera astrazione di dati, con specification applicabili a tutti i methods
- Snippets, provvedono a fornire solo alcuni methods, non possono essere qualificati come un'intera astrazione di dati. Questi methods, tuttavia, sono definiti in un modo che garantisce loro la possibilità di scrivere codice in termini del loro supertipo.

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 estensionabilità, permettendoci di aggiungere sottotipi qualora necessario, o estendere il comportamento di methods già presenti.
- Questi benefici pososno 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.
In java il polimorfismo è espresso attraverso la gerarchia. Solitamente il supertipo per eccellenza è Object.

Questo concetto può essere implementato generalizzando gli arguments in entrata dei metodi/costruttori a Object.
Ciò implica che gli oggetti 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())

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

8.1 to 8.6 contengono tecniche da adottare quando si implementano astrazioni polimorfe.

8.7
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:
- La maggior parte delle astrazioni polimorfe utilizzano methods sui propri parametri, ma solitamente solo i methods di Object sono necessari
- Per le astrazioni polimorfe che necessitano più methods di quelli forniti da Object, utilizziamo un'interfaccia per definire i loro requirements
- 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.

Polimorfismo parametrico
Effettuato in JAVA con i generici.

Prendiamo come soggetto un set (bounded) un set di interi con il limite (renderle impiegabili con molteplici casi)
Possiamo implementarli 2 modi:
- Polimorfa [PDJ8] "old style" alla LISKOV
- Generici JT (tipi parametrici)

Collections (interfacce) sono una struttura dati utilizzati solitamente da algoritmi:
- List Array + Linked Lists
- Set Hash/Tree
- Map Hash/Tree
Bisogna stare attenti a implementare equals e hashCode

Polimorfismo e generici
Tipi generici, astraiamo rispetto al tipo utilizzato all'interno di una collezione o tipo.
Tipo generico: *class C<T>{…T…} ove T = Variabile/Parametro*
Per passare da un tipo generico ad un tipo parametrizzato utilizziamo un processo di istanziazione
C<String> e = new C<>(); ove <> avviene un inferenza di tipo, il tipo viene dedotto automaticamente dato che in precedenza compare String

Metodi generici
Non sono metodi che vengono impiegati per tipi generici, ma sono metodi che nella cui intestazione compare un parametro generico. Funzionano quindi per una famiglia di tipi.

Inferenza
E' un meccanismo che avviene a compile time, in base al tipo apparente il compilatore assume automaticamente il tipo da assegnare a date variabili
Avviene nelle seguenti situazioni:

- Istanziazione (con il diamond)
- Nei metodi/costruttori (witness)
- Target type

List<String> l = List.of(…)
Il tipo viene assunto dallo *String*

Come funzionano
Devono funzionare mescolati anche a codice non generici, si basano quindi sul principio di Type Erasure.
Durante il compile-time, gli elementi generici vengono sostituiti con il tipo reale.
C<T> ----> Durante il runtime: --→ C (Raw Type)
1) tutti i parametri di tipo diventano Object T -> Object
2) aggiungono i cast dove sono necessari -> Cast
3) metodi "bridge", che garantisco il funzionamento dell'overloading etc.

Conseguenza sui generici:
Positivi:
1) Sono backwards compatible
2) Sono performanti
Negativi:
1) Non è possiible istanziare un tipo generico: new T() - class literal
2) non si può fare instanceOf eg. List<String>, List<Integer> perché per il compilatore sono tutti List<Object>
3) non si possono creare array di tipo generico Si potrebbe fare un supress dell'unchecked warning

Generici, subtyping
Dato un S < T S è un sottotipo di T
Che legame c'è tra G<S> G<T> ? Il primo è un sottotipo del secondo? Focus sull'interno
No, G<T> non è un supertipo
Con H < G e G<T> ? che tipo di legame c'è ? Focus sul bordo
Si, H<T> < G<T>

Testing e debugging PDJ 10
Quando andiamo a verificare se un programma funziona, e se funziona nel modo in cui vogliamo possiamo utilizzare una serie di usi che ci facilitano il prcesso.
Parliamo di validation quando ci riferiamo al processo volto ad aumentare la confidence nel fatto che il nostro programma lavori nel modo corretto.
Lo facciamo utilizzando una combinazione di testing, e ragionamento che ci porta a crede che l'implementazione sia corretta. Il debugging si riferisce al processo di identificazione del problema che porta il programma a non funzionare correttamente; il defensive programming è la pratica di scrivere programmi in modo tale che sia facilmente validabile e facile da debuggare.

Validation (costruiamo la cosa giusta?) and verification (costruiamo nel modo giusto?)
Per validare che un progetto sia stato implementato correttamente entra in gioco il processo di verificazione (programmatore e validatore); basato sul *principio di correttezza*.
Per effettuare il processo di verification effettuiamo il *testing*, provando l'implementazione.
Qualora sorgano problemi si effettua il debug, che consiste nella rimozione dei problemi inaspettati per processo di implementazione.

Testing: (Actual vs Expected result)
E' il processo di esecuzione del programma su un set di casi prova, con conseguente confronto dei risultati che ci aspettiamo, ed i risultati che effettivamente riceviamo.
Il testing non ci porta a trovare gli errori, ma solo alla conferma della presenza di essi. (la correzione avviene col debugging)
Gli input da provare dovrebbero essere:

- “piccolo”
- “rappresentativo” eg. se calcolo lo zero su una matrice 4x4 dovrebbe andar bene anche su 10x10 in teoria

Black box testing vs glass box testing
Il glass box testing si fa dopo, conoscendo l’implementazione andiamo a verificare anche i casi più limite

Black box testing (specifiche) adoperazione a “scatola chiusa”
Forma di testing che si basa sulle specifiche (analizzo javadoc), senza analizzare il codice in sé.
Analizzo a prima mano cosa c’è scritto nella specification, e faccio dei test rispettando le clausole.
Analizzando le specifiche posso fare dei:
- boundary case / casi limite (requires) provo a lavorare sui requires
- path (effects) provo i vari effetti descritti nelle effects

Glass box testing (implementazione)
Questo tipo di testing prende in considerazione il codice che si sta andando a testare.
Andiamo a provare principalmente:
- path (selezione, iterazione, ricorsione)
analizzando la coverage, costruendo dei test case che vanno a provare a profondità i casi (in teoria completezza, ma molto spesso ci accontentiamo di un’approssimazione)
Il glass-box testing supplementa il black-box testing; insieme garantiscono che i tests siano path-complete: ovvero che ogni path possibile nel codice sia esercitato almeno 1 volte.
- Nel caso ci siano dei loop di iterazione, i tests dovrebbere includere uno o due iterazioni per loop.
Qualora vi sia un numero variabile di iterazioni, i tests dovrebbero includere zero, uno e due iterazioni del loop.
Nel caso di ricorsione i tests dovrebbero includere nessuna ricorsione, e una ricorsione.

Per rendere un testing valido e completo, dobbiamo svolgerlo su questi componenti:
- Procedure
- Iteratori
- Tipi di dati, dobbiamo quindi andare ad analizzare lo stato
-> dobbiamo analizzare : - i costruttori
- metodi mutazionali/produttori
- metodi osservazionali
Dobbiamo studiarli con approccio black-box e glass box, e l’interplay tra di loro
Quando entra in gioco il polimorfismo, andiamo a fare l’analisi con un caso generico.
Quando invece entra il gioco il SubTyping andiamo a fare l’analisi del LSP, faccio analisi sul supertipo, ma anche sul sottotipo. Faccio una sostituzione del supertipo al sottotipo.
Faccio una sostituzione anche su tutti i metodi a cui ho cambiato la specifica (molto critico), ma anche sui metodi nuovi (perché solitamente sottotipi vanno a estendere il comportamento dei loro supertiti)

I test svolti sono:
- test unitari
Vanno a testare un singolo modulo in isolamento dagli altri
Che si concentrano sull’implementazione sulla quale vengono in ausilio alcuni strumenti
- test di integrazione
Vanno a testare un gruppo di moduli insieme
che si concentra sulla specificazione, ovviamente cresce con la scala, e la complessità
- test di regressione che consiste nel riprovare tutti i test qualora sorge un errore.
per diverse versioni

Sui test unitari (unit test):
Vogliamo andare a verificare il comportamento di un’unità, subject under test/SUT ||
Dependent on component/DOC
Siccome alcune volte abbiamo bisogno di alcuni elementi prima di poter fare i test possiamo dividerlo in
- setup

- test
- teardown

Quando costruiamo dei test case vogliamo scegliere casi i casi rappresentativi.
Per l'esecuzione dei test è possibile implementare dell'automatizzazione,
Cosi come implementare strumenti di reporting (che contiene gli esiti, e i coverage)

SUMMARY:
Il testing è il processo con al quale validiamo la correttezza di un programma. Il testing può essere svolto concentrandoci su singole unità di codice, o integrando più unita di codice.
E' possibile che durante il testing sorgano dei bug. Il debugging è il processo che consiste nella comprensione e correzione di un problema.
Il debugging è reso più semplice se pratichiamo il defensive programming, che consiste nell'inserimento di controlli nel programma che vadano a segnalare errori qualora si presentino.
- Il testing è il processo di validazione della correttezza di un programma
- Il debugging è il processo di ricerca e rimozione di bugs
- Defensive programming consiste nell'inserimento di controlli in grado di trovare errori all'interno del programma, in modo tale da rendere il debugging più facile

Design PDJ13 mettere assieme le astrazioni (software engineering)
Il design dovrebbe rispettare una serie di obbiettivi:
1) Soddisfare le richieste dal punto di vista delle funzionalità
Considerare le prestazioni, che cambiano in base all'implementazione
2) Riflessione sulla struttura di come sono fatti i moduli
- "I pezzi sono tutti buoni"
- Semplicità, l'implementazione dovrebbe essere semplice da costruire e da manutenere (fixare il software)
- Deve essere ragionevole modificare il software (estendere il software)

Introducing the Design Notebook
Mantenere un quaderno degli appunti al fine di descrivere il design ad alto livello del programma, così come il design delle sezioni per ogni astrazione
Dovrebbe essere strutturato nel seguente modo:
- Un diagramma di dipendenza dei moduli

Se un modulo è dipendente da un altro modulo, dovrebbe essere specificato esplicitamente nel Design Notebook.
- Una sezione per ogni astrazione che contiene:
    1) Le specification
    2) I requisiti di performance
    3) Uno sketch dell'implementazione
    4) Informazioni additive, accenando anche ad alternative e giustificazioni