Indice

1 4-02-2019: INTRO

DICHIARAZIONE ≠ ASSEGNAMENTO

L'assegnamento fa riferimento alla modifica di una variabile.

```
1
2 int n       /* Dichiarazione */
3 int m = n = 1; /* Inizializzazione */
4 n = 2         /* Assegnamento */
```

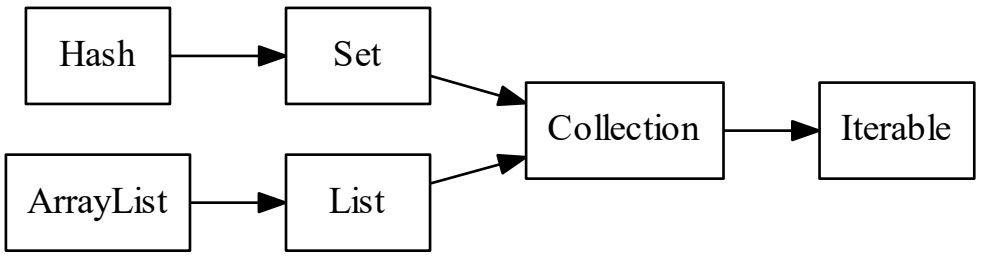
PROGRAMMAZIONE IMPEATTIVA VS FUNZIONALE

JAVA è un linguaggio *imperattivo* ad oggetti. Questo significa che si programma variando lo stato interno delle varie istanze. In contrapposizione troviamo la programmazione funzionale che si basa sul richiamare e scrivere funzioni che non variano gli stati interni delle istanze. Nella programmazione funzionale sono usati spesso metodi che ritornano copie modificate di un determinato oggetto, senza modificare l'oggetto attuale.

JAVA VIRTUAL MACHINE

Java è stato realizzato con un compilatore integrato, che non compila in assembly, questo compilatore invece produce un file sorgente che non è eseguibile direttamente dalla macchina, bensi è eseguibile da una virtual machine, la JVM(*java virtual machine*). In questo modo viene garantita la portabilità del codice in vari computer con CPU diversa (come il Phyton e .NET).
Questa separazione non conta niente per il linguaggio, si riflette solamente sul modello architetturale.

U.M.L. = unified model language ⇒ rappresentazioni di gerarchie di classi.



- Tutte le sottoclassi sono dei sottoinsiemi.
- Tutte le superclassi sono dei sovrainsiemi.

I linguaggi ad oggetti ci permettono di costruire *tipi* e di definire *valori*.

```
Animale a = new Animale();
/* in Java gli oggetti sono valori
 * dove: Animale -> tipo -> CLASSE
 * a = nome variabile
 * new Animale() ha un valore -> OGGETTO */
```

COMPILING TIME E RUNTIME
La gestione di sintassi e di errori viene fatta in due fasi. Dal compilatore(compiling time):
• Viene eseuito un controllo sintattico
• Viene eseguito controllo dei tipi, cioè che gli insiemi siano corretti
In fase di esecuzione(RunTime):
• Abbiamo a che fare con valori e non con tipi, vengono controllati i "cast"

I tipi di fatto sono una astrazione del linguaggio.
Il senso di un compilatore è quello di evitare di scrivere castonerie che a livello di esecuzione non avrebbero senso.

SOUNDNESS
Un linguaggio si dice *sound* quando il compilatore ti da la certezza che in fase di esecuzione il programma sia eseguito correttamente senza possibilità di errori.

PARAMETRO IMPLICITO
Ogni metodo dichiarato ha sempre un parametro implicito (il parametro *this*). Esso è sotto inteso e viene passato automaticamente, più eventuali parametri che vengono passati all'interno del metodo.

```
public class Animal {
  private int peso;
  ...
  public void mangia(Animali a){
    this.peso = this.peso + a.peso;
  }
}

Cane fido = new Cane();
a.mangia(fido);          /* SUBSUMPTION (assunzione) */
```

POLIMORFISMO
L'eredità è un meccanismo che garantisce il funzionamento del *polimorfismo*.
• Polimorfismo per *SUBTYPING* o anche polimorfismo per inclusione: si riferisce al fatto che una espressione il cui tipo sia descritto da una classe A può assumere valori di un qualunque tipo descritto da una classe B sottoclasse di A (Vedi codice appena sopra)
• Polimorfismo dei *GENERICS*: si riferisce al fatto che il codice del programma può ricevere un tipo come parametro invece che conoscerlo a priori (polimorfismo parametrico). In questo modo non si perde il tipo originario passato dall'oggetto al metodo

```
1   /* POLIMORFISMO SUBTYPING
2    * basato sui sottotipi ereditarietá */
3   Object ident (Object x){
4     return x;
5   }
6
7   /* POLIMORFISMO GENERICS (parametrico)
8    * non perdo informazioni sui tipi */
9   <T> T ident (T x){
10    return x;
11  }
12  /* questa funzione mi permette di riusare il
13   * metodo, in questo modo evito di fare CAST,
14   * e di sbagliare a farli */
```

2 7-02-2019: DINAMIC DISPATCHING

Nei linguaggi ad oggetti, lo strumento più potente è la classe. Quando definisco le entità che poi vado a tramutare in classi sto definendo DATI.
Le classi possono contenere dei metodi (funzioni che operano sugli oggetti della classe).
Definire sottoclassi significa definire *sottoinsiemi* nell'ambito dell'ereditarietà. Le nuove operazioni delle sottoclassi vanno inserite sapendo che le sottoclassi ereditano il set di funzioni delle superclassi. *IL POLIMORFISMO* è uno strumento molto utile perchè ci permette di scrivere codice, funzioni che posso adoperare anche con tipi diversi!
@ serve per creare delle annotazioni nel codice, serve per il compilatore (es: @ override).

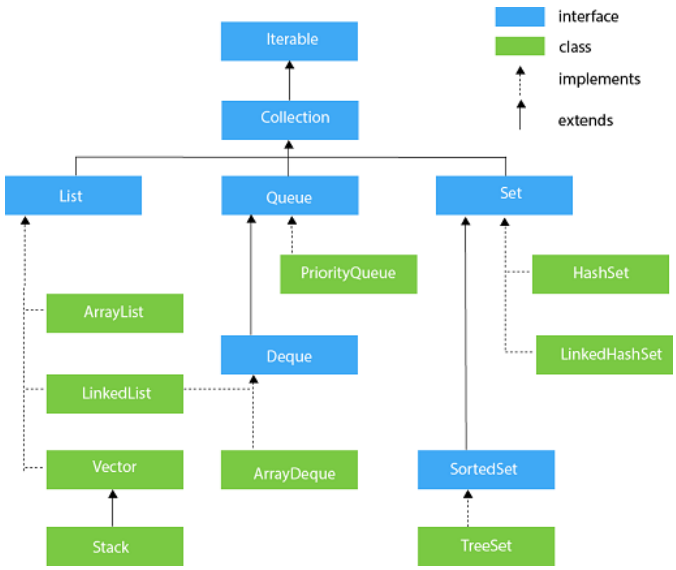
OVERRIDING
L'*Overriding* è il punto cruciale di tutta la programmazione ad oggetti. Fare overriding significa sovrascrivere un metodo ereditato dalla super classe per poterne specializzare il suo comportamento. Se non potessi farlo significa che nelle sottoclassi non posso andare a specializzare un metodo. Specializzare un metodo significa cambiare l'implementazione della super classe senza cambiarne la firma.

DINAMIC DISPATCHING
Il *Dinamic dispatching* serve in fase di runtime a scegliere la versione giusta del metodo da richiamare. Infatti se ho degli over ride nelle mie classi, sarà solo in fase di run time che Java deciderà quale metodo richiamare. Se nella mia classe non esiste il metodo richiamato, il dinamic dispatching va a prendere l'implementazione del metodo dalla superclasse. Nella memoria che contiene le informazioni degli oggetti ci sono tutti i puntatori ai metodi di una classe, in run time viene eseguito il codice del puntatore corretto. (*vedere: virtual table*). Un *OGGETTO* infatti è costituito da un insieme dei suoi campi e da puntatori ai metodi della classe ed è grazie a questo che il dispatching funziona: il compilatore controlla i tipi e garantisce che nel compiling time tutto questo funzioni.

Ogni espressione ha un tipo!

CLASSI E METODI STATICI
• I metodi statici sono quei metodi di una classe che appartengono alla classe, non alle istanze di una classe. Si possono richiamare senza creare un oggetto. I metodi statici possono quindi accedere solo a dati statici e non alle variabili di istanza della classe, possono solo richiamare altri metodi statici della classe, e sopratutto non possono usare il parametro implicito *this*.
• Le classi statiche in java possono solamente esistere se sono *innestate (nested)*. Esse possono accedere solamente dati statici della classe che le contiene. Una classe statica interna non vede il riferimento this dell'altra classe, essa può accedere solamente ai campi statici della classe che la contiene.
Le classi statiche non sono però come i membri statici, è possibile infatti instanziarne più istanze, tutte quante indipendenti e non relazionate tra di loro.

COLLECTION
Le *Collection o contenitori* sono delle interfacce della libreria di JAVA e non si possono costruire.
Le *Collection* da sole non sono dei tipi, le *Collection* di un "qualcosa" sono dei tipi.I tipi parametrici vogliono infatti un *argomento*



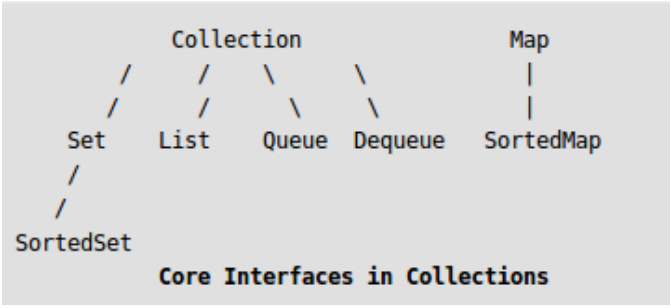
PACCHETTI JAVA
JAVA SE ⇒ Standard Edition
JAVA EE ⇒ Enterprise Edition
JAVA ME ⇒ Mobile Edition
JAVA JDK ⇒ linguaggio + tutte le librerie standard (java developement kit)
JAVA JRE ⇒ Solo a runtime, versione ridotta che serve solo a chi usa i programmi ma non al programmatore (java runtime enviroment)
File jar ⇒ Archivio di tutti i pacchetti del programma
JAVA JVM ⇒ (Java virtual machine) serve per eseguire i file .jar
La documentazione di java si trova on-line ed è diffusa in pacchetti che servono ad organizzare logicamente le classi, che sono organizzate in ordine alfabetico.

3 11-02-2019: ITERATORI E GENERICS

ITERATORE e ITERABLE
• *Iteratore*: E' un pattern, uno stile di programmazione. Il pattern degli iteratori esiste in tutti i linguaggi ad oggetti. Con iteratore intendiamo lo scorrimento di una collezione di elementi. L'iteratore serve quindi a scorrere una collection.
• *Iterable*: E' una super interfaccia, e l'interfaccia *Collection* implementa questa super interfaccia. Iterable è super tipo di tutte le interfacce.

Se una interfaccia rappresenta una super interfaccia significa che non ha un genitore, anche se in realtà estende *Object*

MAPPA
Una mappa è una struttura dati che mappa chiavi e valori, ha quindi due parametri: il tipo della chiave e il tipo del valore. Una mappa è una *collection* solo se vista come una collection di coppie. Infatti una *collection* è figlia di *iterable* (la posizione più alta nella gerarchia), ma una *mappa* NON è figlia di *iterable*



SOTTO TIPI
Ci riferiamo ad un oggetto usando la parola *sottotipo* quando esso è:
• O una sottoclasse (extends)
• O una sotto interfaccia (implements)
Ad esempio ArrayList ha come superclasse abstract arrayList. ArrayList implementa collection. Quindi ne è sottotipo ma non sottoclasse.

GENERICS
<? extends classe> e <? extends E> rappresentano un tipo.
La *SUBSUMPTION* non funziona in genere sempre tra GENERICS. Per il parametro stesso c'è subsumption, ma non per le collection. Il tipo con il ? accetta sottotipi di parametri.
• Il TIPO ESTERNO gode sempre della subsumption.(List<Int>: List è tipo esterno)

```
1 ArrayList<Int> é sottotipo di List<Int>
```

• Il TIPO INTERNO non gode mai della subsumption(cioè il tipo all'interno di <>). La subsumption si può fare solo usando i generics(quindi con <? extends E>)

```
ArrayList<Iterable> non é sottotipo di ArrayList<Collection>
```

Si dice che abbiamo quindi "*Invarianza del subtyping*": se ciò non fosse le subsumption funzionerebbero anche nel tipo interno e questo rischierebbe la totale spaccatura del linguaggio (senza l'invarianza si possono creare problemi di cast ed oggetti).
Se cosi non fosse in java non verrebbero mai rispettate le regole delle classi.

```
/* Illegal code - because otherwise life would be Bad */
List<Dog> dogs = new ArrayList<Dog>(); /* ArrayList implements List */
List<Animal> animals = dogs; /*  Awooga awooga */
animals.add(new Cat());
Dog dog = dogs.get(0); /* This should be safe, right? */
```

Java di unico ha che esiste il wildchart (?), che è un modo controllato per risolvere il problema della subsumption dei tipi interni.
Prima dei generics (2003/2004) in java si programmava tutto a typecast. Per motivi di retrocompatibilità è possibile programmare in tutti e due i modi. E' comunque consigliato usare la programmazione con i *generics*.
Metodi che ritornano un booleano iniziano con sempre come se fossero domande; es: hasNext, isEmpty etc..
Un iteratore non può essere costruito con un new perchè è un'interfaccia.

4 15-02-2019: INTERFACCE

INTERFACCE

```
1  public interface Iterator<T>{}
```

L'interfaccia è un contratto, nel senso che mette a disposizione una serie di metodi che ogni classe che estende quell'interfaccia deve obbligatoriamente implementare, pena un errore durante la fase di compilazione.In java quindi si può scrivere del codice ancora prima di sapere come si potrebbe implementare.

Facciamo un esempio: il "contratto" di iteratore è il seguente:
• boolean hasNext();
• T next();
• void remove()
Data una certa classe che può non essere sotto al nostro controllo non abbiamo bisogna sapere necessariamente come sono stati implementati i suoi metodi, ma ci ti basta sapere che esistono per poter dire se sia o meno un iteratore.
Esempio di definizione di un metodo con iteratore come input:

```
1  public statics void scorri(Iteratore <Integer> it){
2    while(it.hasnext()){
3      integer n = it.next();
4    }
5  }
```

Esempio di utilizzo

```
1  scorri(new Iterator<>(){
2    ...
3    ...
4    ...
5  });
```

Quest'ultima è un'espressione, o come meglio dire, un'oggetto fatto al volo. Questa sintassi è stata creata appositamente per le interfacce (dato che non si possono istanziare direttamente), senza dover andare a definire una classe con la classica implementazione dell'interfaccia.

ANONYMOUS CLASS meccanismo comodo per design pattern come le call-back.

Questa implementazione garantisce che la funzione sia *SOUND*, e non crasherà mai a *RunTime*

IMPLEMENTARE INTERFACCE
1) Con implements:

- controlla i metodi che hai implementato all'interno della classe
- assicura che siano implementati tutti

Tipi delle interfacce
Iterator $\Rightarrow$ non è un tipo
Iterator<T> $\Rightarrow$ è un tipo

NOTAZIONE BNS
BNS è il nome della notazione e serve per poter dare delle regole grammaticali. E' una notazione che definisce la sintassi delle espressioni
Iterator da solo, sintatticamente, sarebbe un tipo. Ma il compilatore verifica che non è un tipo e da errore.

5 18-02-2019: CLASSI ASTRATTE

CLASSE ASTRATTA

Una classe si dice astratta quando ha almeno un metodo astratto, essa serve per impedire la sua costruzione (non posso quindi istanziarla). Delle classi vengono definite astratte se anche un solo metodo è astratto. Una delle maggiori differenze tra classi astratte ed interfacce è che una classe può implementare molte interfacce ma può estendere una sola classe astratta. Una interfaccia è zucchero sintattico di una classe astratta con soli metodi astratti. Zucchero sintattico (Syntactic sugar) è un termine coniato dall'informatico inglese Peter J. Landin per definire costrutti sintattici di un linguaggio di programmazione che non hanno effetto sulla funzionalità del linguaggio, ma ne rendono più facile ("dolce") l'uso per gli esseri umani. La differenza tra classe ed interfaccia in realtà non esiste.

Un array è una struttura dati lineare, omogenea e contigua in memoria.

Per leggere una *collection* si usano gli iteratori che servono per farne il get in sequenza.

VTABLE E DIMENSIONE DI UNA CLASSE

Una virtual Table, o dispatching table, è un meccanismo utilizzato per supportare il dynamic dispatching (o anche chiamato run-time method binding).Quando una classe definisce dei metodi virtuali, il compilatore nasconde all'interno dei membri della classe una variabile che punta ad un array di puntatori a funzioni. Questi puntatori sono poi usati a runtime per invocare l'implementazione della funzione appropiata, questo perchè in compile-time non è detto che si sappia se la funzione richiamata sia quella del tipo definito o sia derivata da un'altra classe.

```
1  public static class Animale(){
2    private int peso;
3  }
4
5  public static class Cane extends Animale{
6    private String nome;
7    public void abbaia(){};
8  }
9
10 public static class PastoreTedesco extends Cane{
11
12 }
```

Se costruisco un oggetto di tipo PastoreTedesco, esso sarà grande quanto un tipo int (32 bit) ed una stringa (un puntatore). Il tutto grazie alla virtual table che tiene in memoria i puntatori dei vari campi di uno oggetto.

6 21-02-2019: REFLECTION

REFLECTION
La *reflection* è una *features* del linguaggio java che non tutti i linguaggi di programmazione posseggono (Ad esempio il C++ non la possiede). Essa ci permette di conoscere i tipi e il contenuto delle classi a runtime. Ad esempio se voglio conoscere il tipo dell'enclosing class (classe che contiene) posso fare : nome.classe.this.nome
Ad esempio se abbiamo degli oggetti che vengono passati dentro ad un metodo che come parametro ha il tipo Object, non saremo più in grado di distinguere il loro tipo di classe "originale", per superare questo problema possiamo invocare la funzione getClass() che ci ritorna il loro vero tipo.

BINDING
Esistono due tipi di BINDING: static binding () e dynamic binding (un esempio è l'override dei metodi di una classe ereditata).
BINDING avviene anche con i tipi
I parametri di una funzione sono binding nello scope della funzione.
I type argument fanno binding con i type parameter, esattamente come avviene per le funzioni tra argomenti e parametri.
Quando si programma con i generics essi si PROPAGANO.

TYPE ERASURE
La cancellazione dei tipi in java avviene quando il compilatore "butta" via i generics generando classi non anonime e li sostituisce con Object: il motivo è per mantenere la compatiblità con il vecchio codice che non aveva generics. Quindi i generics sono verificati dal compilatore e poi cancellati per eseguire. Detto questo risulta quindi importante non creare mai all'interno della classe oggetti del tipo del parametro, perchè in fase di compilazione diventeranno di tipo Object!

7 25-02-2019: EREDITARIETA'

Se un oggetto ha dei campi esso pesa tanto quanto la dimensione dei campi.Ricordiamo che nei pc a 64 bit i puntatori pesano 8 byte.

L'ereditarietà serve anche a modificare i metodi della classe che viene ereditata. E' l'unico modo che abbiamo per modificare delle cose anche se non sappiamo cosa e chi le ha costruite. Sopratutto se non le possediamo. Un esempio è la classe ArrayList, che deve essere ereditata per implementare un motodo che ci permetta di scorrerla all'indietro.
I metodi statici non si possono override perchè non sono presenti nelle virtual table (sono funzioni sciolte).

Regola ereditarietà costruttore: se non definisco nessun costruttore nella sottoclasse è come se chiamassi il costruttore della superclasse SENZA parametro.

8 28-02-2019: COVARIANZA e CONTROVARIANZA

COVARIANZA e CONTROVARIANZA DEI TIPI
Scrivendo $C \leq A$ intendiamo che C è sottotipo di A. Definito l'operatore minore uguale passiamo alle definizioni vere e propie.

VARIANZA
$C_1 < \tau_1 > \leq C_2 < \tau_2 > \Leftrightarrow C_1 \leq C_2 \bigwedge \tau_1 \equiv \tau_2$
Questa regola del type system di *java* ci dice che il linguaggio non è COVARIANTE, in quanto i generics non cambiano.

- Esempio: ArrayList<cane> $\leq$ List<cane> (sottotipo)
- Esempio: ArrayList<cane> $\not\leq$ List<Animali>

L'ultima formula non è covariante, se fosse possibile si avrebbe una doppia subsunzione sul guscio interno e sul tipo esterno

CONTROVARIANZA
Quando eredito un metodo di una classe posso sovrascriverlo usando l'overriding. Quando lo faccio un linguaggio di programmazione si dice controvariante se mi è possibile far scendere (specializzare) il tipo di ritorno, e al contempo mi è possibile far salire (rendere più generico) l'argomento di ingresso. Vediamo un esempio (*NON VALIDO IN JAVA*):

```
1  /* Data questa gerarchia di classi: */
2    public class Cane extends Animale{
3    /* nella classe Cane faccio l'override
4     * del metodo ereditato dalla classe
5     * Animale */
6    public Cane m(Cane c){
7      return c;
8    }
9    }
10
11  public class PastoreTedesco Extends Cane{
12   @Override
13   public PastoreTedesco m(Animale c){
14     return new PastoreTedesco;
15   }
16   /* rispetto al metodo originale
17    * il tipo di ritorno del metodo
18    *    (PastoreTedesco) scende (sottoclassi)
19    * il tipo del parametro di ingresso
20    *    (Animale) sale (superclasse) */
21  }
```

Detto questo è bene specificare che Java supporta i solo i tipi di ritorno controvarianti, è possibile controvariare SOLO il tipo di ritorno del metodo, ed è possibile farlo solamente scendendo (sottotipo). Ed è possibile Fare questo nella fase di overriding, non in quella di overloading. In caso di overloading il tipo di ritorno deve essere lo stesso.

Il seguente è l'esempio corretto in java:

```
1 public Cane m (Cane c){return c;}
2
3 @Override
4 public PastoreTedesco m(Cane c){return new PastoreTedesco();}
```

SOUNDNESS IN JAVA
• SOUND : un programma che compila può essere eseguito senza errori
• SOUND JAVA: un programma compila e termina, a meno di una eccezione.
Un esempio di SOUND JAVA è il seguente:in java è possibile avvenga un segmentation fault non per un problema di casting, ma solamente se accediamo ad un indice di un array che non abbiamo allocato, quindi il programma termina a meno di una eccezione. Ci sono invece linguaggi dove non esistono gli array, quindi non accadrà mai segmentation fault e il codice terminerà sempre alla fine senza eccezioni, ovviamente senza fare i controlli di semantica.

WILDCARDS
Recentemente è stato inserito un pattern che permette anche in Java la covarianza: sono i generics con i wildcards.

```
1 Arraylist<? extends Animale> m = new List<Cane>();
```

Da questo si capisce che la covarianza può essere usata, ma solamente se esplicitata con il wildcard.
Sono molto usati perchè i wildcard non sono tipi del primo ordine, infatti il *tipo* "? extends classe" non esiste!!!Non posso definire una variabile come segue:

```
1 ? extends Animale m = new Cane();
2 /* questa sintassi si puó usare solo come type argument */
```

Significato: permettono la covarianza, sono tipi temporanei che non possono essere scritti nel codice, però possono essere subsunti con il get().
Un altro DESIGN PATTERN: callback

9 4-03-2019: CLASSI ANONIME, LAMBDA E FUNZIONI ORDINE SUPERIORE

DESIGN PATTERN

- Iteratore
- Compact o Callback o Unary function

CLASSI ANONIME

Le classi *lambda* servono per fare funzioni al "volo",senza quindi avere il bisogno di implementare delle interfacce in classi separate. Vediamone un esempio:

```
interface HelloWorld {
  public void greet();
    public void greetSomeone(String someone);
}

public static void main (String args[]) {
    HelloWorld frenchGreeting = new HelloWorld() {
        String name = "tout le monde";
        public void greet() {
            greetSomeone("tout le monde");
        }
        public void greetSomeone(String someone) {
            name = someone;
            System.out.println("Salut " + name);
        }
    };
}
```

LAMBDA ASTRAZIONI

Le espressioni *lambda* servono per fare funzioni al "volo",senza quindi avere il bisogno di implementare delle interfacce in classi separate, classi anonime e classi innestate. Si ricorda che le espressioni lambda possono essere usate solo per implementare interfacce funzionali (cioè interfacce che possiedono un solo metodo astratto). Vediamone un esempio:

```
interface MyString {
  String myStringFunction(String str);
}

public static void main (String args[]) {
  MyString reverseStr = (str) -> {
    String result = "";
    return result;
  };
  reverseStr.myStringFunction("temp");
}
```

FUNZIONI DI ORDINE SUPERIORE
Sono delle funzioni che prendono delle funzioni come parametri di ingresso

```
/* questa interfaccia é equivalente all'interfaccia java.util.Functional
 * essa infatti espone una serie di funzioni, senza implementazione, tra
 * cui anche le call-back */
public interface Func<A, B>{
  B execute(A a);

  /* questa é l'unica funzione esposta dall'interfaccia
   * un altro nome ragionevole per il metodo execute() é apply()
   * oppure call() il nome deve ricordare il fatto di richiamare
   * la funzione  */
}

/* questa funzione va ad utilizzare la funzione Func definita sopra */
public static <A,B> List<B> map(List<A> l, Func<A,B> f){
  List<B> r = new ArrayList<>();
  for(A x: l)
    r.add(f.execute(x));
  return r;

}
```

A e B sono *generics* locali al metodo(e solo al metodo)
I generics sulle classi servono per parametrizzare, non per fare polimorfismo

```
public static <A,B> List<B> map(List<A> l, Func<A,B> f){
/* dove in "public static <A,B>" dichiaro i parametri che useremo
 * mentre in "List<a> .. Func <A,B>" "uso" i parametri */
```

Funzione FILTER:

```
/* questa funzione é simile alla funzione Func definita sopra
 * solo che rende piú chiaro il suo scopo: filtrare degli elementi
 * da aggiungere in una lista */
public static <A> List<A> Filter (List<A> l, Func<A, Boolean> p){
  List<A> r = new ArrayList<>();
  for(A x : l)
    if(p.execute(x))
      r.add(x);
  return r;
}
```

La seguente *Filter2* NON funziona perché usa la *remove()* delle Collection, ma non è possibile rimuove un elemento in fase di scorrimento (è scritto nella documentazione)

```java
public static <A> void Filter2 (List<A>, Func<A, Boolean> p){
  for(A a: l)   /* il for each in Java essere zucchero sintattico */
    if(p.execute(a))
      l.remove(a);

}
```

Se non posso rimuovere come ho fatto sopra un elemento posso invece chiedere all'iteratore di rimuovere l'elemento stesso, esso rimuoverà quello a cui stiamo puntando. Quindi invece di usare un ciclo for come sopra, uso l'iteratore per scorrere la lista usando in metodo *hasNext*.

```java
/* questo funziona perché chiama la remove() dell'iteratore
 * mentre prima chiamavo il remove della lista */
public static <A> void Filter2 (List<A>l, Func<A, Boolean> p){
  Iterator<A> it = l.iterator();
  while(it.hasNext()){
    A a = it.next();
    if(!(p.execute(a)))
      it.remove();
  }
}

/* volendo posso usare le funzionalita delle nuove API FUNZIONALI
 * l.removeIf(a -> !p.execute(a)); */
```

Posso usare Function<A,B> di java come funziona Func?
Esempio di chiamata:

```java
public static void main(String argv[]){
  List<String> strings = new ArrayList<>();
  string.add("ciao");
  string.add("pippo");
  string.add("unive");
  /* voglio calcolare la lunghezza della mia lista */
  List<Integer> r = map(strings, new Func<String, Integers>{

    /* "String" é la lista, "Integers" é la funzione
         * In realtá devo passare un oggetto di tipo Func<>
         * alla funzione map, perció passo una classe anonima,
         * all'interno della quale trovo solo un metodo
         * (che ha la stessa firma del metodo dell'interfaccia Func)*/

    @Override
    public Integer execute(String a){
      return a.length();
    }
  });
```

```
}
```

La seguente funzione data una lista di interi scarta gli elementi minori di zero: questo è il modo per non usare un for con un ciclo if innestato.

```
public static void main__filter(){
   List<Integer> interi  = new ArrayList<>();
   interi.add(89);
   interi.add(34);
   interi.add(-16);
   interi.add(560);
   interi.add(-1);
   interi.add(46);
   /* filter prende una lista e un predicato e produce una lista in uscita
      */
   List<Integer> l = Filter(ints, new Func<Integer, Boolean>(){
    @Override
    public Boolean execute (Integer a){
      return a>=0;
    }
   });
}
```

Oppure

```
  Filter2 (interi , new Func<Integer, Boolean>(){
    @Override
    public Boolean execute (Integer a)
      return a>=0;
  })
```

GENERICS LOCALI (Polimorfismo parametrico di primo ordine)

```
    public static Object ident__ugly(Object o) {
        return o;
    }
    /* con un metodo di subtyping che é POLIMORFISMO VERTICALE,
     * questa funzione NON é SOUND perché sono costretto a
     * fare un CAST di ció ció che ricevo */

    public static <X> X ident(X x) {
        return x;
    }
    /* con i generics, che é POLIMORFISMO PARAMETRICO, non
     * devo piú fare nessun cast. Sono sicuro del tipo di
     * ritorno */
```

```
1  public static void main__ident  (){
2    Cane fido = new Cane();
3    Cane c = (Cane) ident__ugly(fido); /* ritorna un cane
4                        * facendo un cast */
5    Cane c2 = ident(fido);  /* ritorna un cane senza dover
6                   * fare un cast */
7    Gatto g = ident(new Gatto());
8  }
```

10 7-03-2019: WILDCARDS, NESTED CLASS E INNER CLASS

WILDCARDS

I tre tipi possibili di wildcards sono:

- *<?> top type* (o chiamato Unbounded Wildcard)
- *<? extends nametype>* (o chiamato Upper Bounded Wildcards)
- *<? super nametype >* (o chiamato Lower Bounded Wildcards)

TOP TYPE

Il tipo "?" non può essere usato come tipo per una variabile quindi dichiarare"? x" non si può fare. Posso però dichiarare questo: "Object x = l1.get(..)"

```
List<?> l1 = new ArrayList<Cane>();
/* Il "?" da solo indica un tipo che gerarchicamente
 * e piú in alto di Object, viene detto top type.
 * In poche parole indica che non ci sono
 * restrizioni sul tipo di parametro passato,
 * e che quindi la lista contiene un tipo non
 * conosciuto */

l1.get(int index) /* il compilatore ritorna l'errore in
          * compile- time: " capture of ? "
          * qualcosa che sia figlio del top type */

/* List<?> significa che creo una lista di un tipo
 * sconosciuto. In conseguenza di ció il compilatore
 * mi segnala errore quando cerco di richiamare il
 * metodo l1.add("qualcosa") perché non sa come
 * proteggere la lista! Infatti una lista puó
 * avere un solo tipo per volta, se con il top
 * type potessi aggiungere di tutto mi ritroverei una
 * lista con elementi tutti diversi tra di loro. Non
 * potendo peró verificare di che tipo sia la lista,
 * che potrebbe essere una lista di qualsiasi tipo,
 * non mi lascia aggiungere niente.
/*
```

UPPER BOUNDED WILDCARDS

"? extends Animale" indica che come parametro posso passare un qualsiasi tipo che sia figlio di Animale.
l2.get(0) -> ritorna un capture of ? extends Animale -> qualsiasi cosa figlia di animale (posso però fare Binding di qualcosa che sia al massimo Animale)

```
  List<Animale> t3 = new ArrayList<Gatto>();
```

```
2   /* é errato, si puó subscrivere solamente il guscio
3    * esterno,per farlo con il guscio (tipo) interno
4    * la versione corretta é fatta con il wildcard */
5
6   List<? extends Animale> t3 = new ArrayList<Gatto>();
```

Vediamo ora un'altro esempio:

```
1   List<?> l1 = new ArrayList<Cane>();
2   List<? extends Animale> l2 = new ArrayList<Gatto>();
3   l1 = l2 /* é assegnazione valida */
4   l2 = l1 /* NON é assegnazione valida */
```

Continua a valere il discorso del top type. Data una lista di un upper bounded type il compilatore non mi lascia aggiungerci elementi perchè semplicemente non può verificare la loro correttezza! Un esempio è il seguente:

```
1   List<Gatto> l1 = new ArrayList<Gatto>();
2   l1.add(new Gatto("bianco"));
3   List<? extends Animale> l2 = l1; /* operazione valida */
4   l2.add(new Cane("nero"));
5   /* il compilatore mi segnalerá un errore quando uso add, se cosi
6    * non fosse avrei grossi problemi, avrei aggiunto alla
7    * lista di gatti un cane !!! */
```

Questo tipo di liste vanno bene per poter schematizzare una funzione che riceve liste che poi userà per riempirne un'altra.

LOWER BOUNDED WILDCARDS
"? super Animale" indica che come parametro posso passare qualsiasi tipo che sia più su di Animale (più generale).
In questo caso posso passare animale a tutto quello che sta sopra.
l2.add(new Animale) -> ?? non compila perchè...
Questo genere di wildcards sono molto utili, perchè permettono di essere passati ad una funzione, la quale poi andrà ad aggiungerci dati. Vediamone un esempio:

```
1   public static aFunc(List<? super Cane> l){
2     l.add(new Cane("nero")); /* valido */
3     l.add(new PastoreTedesco(nero)); /* valido */
4     l.add(new Animale()); /* NON VALIDO */
5   }
6   /* perché é valido? perché é una lista di un tipo che
7    * non conosco di preciso, ma so che é super tipo
8    * di cane. Quindi ogni oggetto di sotto tipo di Cane
9    * é sicuramente subsubimile al super tipo */
```

ESEMPI
Riprenderemo la map vista nella lezione scorsa.

```
public static <A,B> List<B> map(List<A> I, Func<A,B> f)();
```

Ad esempio, per trasformare animali in piante:

```
public static class Vegetale{}

public static void main_map(){
  List<cani> l1 = new ArrayList<>();
  List<Vegetali> l2 = map(l1, new func<Animale, Vegetale>(){
    @Override
    public Vegetale execute(Animale a){
      return null;
    }

  });
}
```

Questa funzione riportata sopra non compila in quanto i *generics* non sono soggetti alla subsumption. Per farla compilare modifichiamo la funzione map come segue:

```
public static  <x, y> List<x> map(List<x>, Func(? super <x, y> f)){
...

}
```

Riporto anche una versione il più generale possibile:

```
    public static <X, Y> List<Y> map(List<X> l, Func<? super X, ? extends
        Y> f) {
        List<Y> r = new ArrayList<>();
        for (X x : l) {
            r.add(f.execute(x));
        }
        return r;
    }
```

NESTED CLASS

```
public class Main__Functional {
  public static void main__filter() {
    ...
  }
}
```

In java esistono due tipi di nested class (classi innestate): quelle statiche (nested class) e quelle non statiche(inner class).
• Le *nested class* sono senza nessuna relazione con la outer class, o enclosing class, e quindi non hanno riferimento al this. Di esse posso istanziarne più oggetti senza che venga istanziata nessuna istanza dell'enclosing class.
• Le *inner class* invece vedono i campi della enclosing class, compreso il parametro implicito this. Di esse posso crearne più istanze solo ed esclusivamente se sono istanziate con un riferimento ad una istanza della outer class.
Le due tipologie di classi (Inner e Outer) non servono a creare gerarchie, ma solo a creare ordine nel codice.
Se una classe innestata non ha relazioni con la enclosing class è meglio farla statica, risulta un codice più pulito e meno pesante, perchè non serve mantenere il riferimento al this ogni volta.

OVERLOADING
Il meccanismo dell'overloading permette di definire metodi con stesso nome ma firma diversa.

```
public static class c{
  public int m(){
    return 1;
  }
  public int m(int x){
    return x+1;
  }
  public int m(float x){
    return (int)(x-1.0f);
  }
  public int m(int x, int y){
    return x+y;
  }
}
```

L'overloading non è permesso cambiando il tipo di ritorno e lasciando il resto inalterato. Devono essere diversi i solo i parametri!
-ordine
-tipi
-numeri

L'overloading è del tutto gestito dal compilatore, e **non** viene quindi fatto durante la run-time.

```
public Number m(Number x){
  return x;
}
```

Vediamo un esempio:

```
  int foo(){...};
  float foo(){...};
  ...
  ... = foo();
  /* come fa il compilatore a sapere
   * quale foo deve richiamare se
   * differiscono per il parametro di ritorno?
   * Non puó saperlo, ecco perché
   * l'overloading richiede i tipi di ritorno
   * uguali */
```

11 11-03-2019: WILDCARDS, NESTED CLASS E INNER CLASS

WAITING FOR SOME DATA
Sembrerebbe che nessuno abbia preso appunti questo giorno

12 18-03-2019: ECCEZIONI

ECCEZIONI

Quando si lanciano eccezioni è bene ricordasi la differenza tra *throw* e *throws*

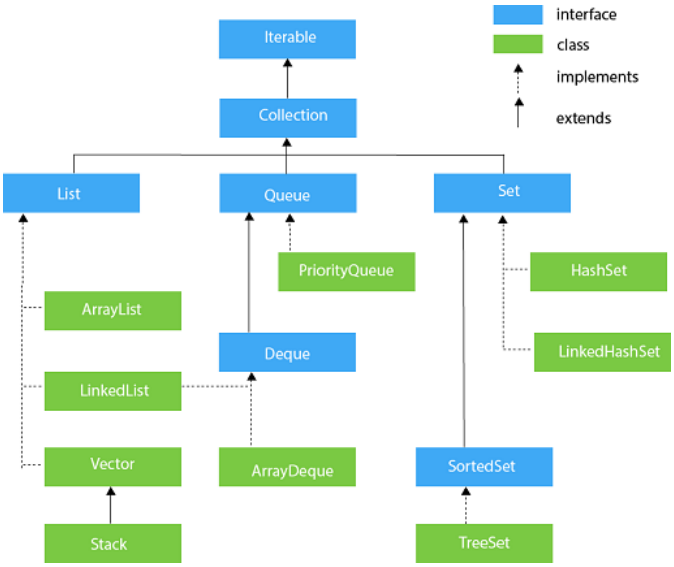
```
1 public void writeList() throws IOException {
2   if(true)
3       throw new IOException("demo");
4 }
```

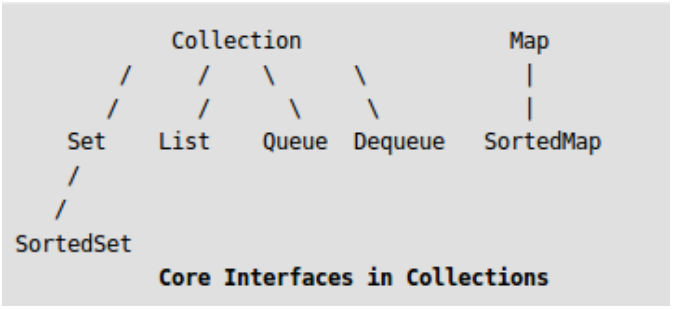
• throws viene messa affianco alla firma del metodo, seguita da una lista delle eccezioni, e serve per dichiarare quali *TIPI* di eccezioni possono essere lanciate da un determinato metodo.
• throw seguito da un oggetto di tipo eccezione, serve per lanciare l'eccezione.
Si noti quindi che con throws si dichiarano i tipi che verranno lanciati ma poi lanciamo oggetti: questo rappresenta un tipo di subtyping.
Si possono lanciare solo eccezione figlie del tipo *Throwable*. Throwable è un super tipo di Exception e di tutte le classi lanciabili.
Come regola generale quindi throw ha bisogno di essere seguito da un'espressione con tipo compatibile per poter essere lanciata.
Il catch è lo strumento di binding per il throw.
Le eccezioni non ritornano per forza al chiamante se ritornano a chi se le prende. Infatti quando avviene una chiamata ad un metodo, questa appartiene ad una catena di chiamanti. Le eccezzioni possono quindi restituire qualcosa o al chiamante della funzione o ritornare qualcosa ad uno dei chiamanti della catena. Se risalendo questa catena l'eccezione non viene catturata nemmeno dal main questa passa direttamente alla *Java Virtual Machine*
Il sistema try-catch è stato ideato per evitare delle forti anomalie del programma.
Nel canale ufficiale del return vengo solamente ritornati i risultati "giusti", in caso contrario verrà lanciata un'eccezione: questo è lo stile richiesto per i linguaggi evoluti. Invece di complicare il tipo di ritorno usiamo le eccezioni.
Al posto delle eccezioni possiamo definire un tipo di ritorno che codifica il fatto che hai trovato o meno quello che cercavi. Questa tecnica non è molto leggibile per chi non ha scritto il codice, sarebbe meglio usare il design pattern " tipo-eccezione". Esso è utile anche perchè in questo modo non si può scrivere codice che non funzioni, mentre definendo un nuovo tipo è possibile.

I TRE TIPI DI ECCEZIONE

• Checked exception: Sono le condizioni di errore che l'utente deve poi obbligatoriamente gestire. Esse si usano per situazioni eccezionali in cui l'utente può ragionevolmente gestire il tutto con i try catch (ad esempio una selezione invalida in una interfaccia grafica).Questi tipi di errori devono necessariamente usare la clausola throws.
• Runtime exception: Sono le condizioni di errore a gestione non obbligatoria, che se non gestite, possono arrivare fino al main e chiudere il programma. Sono usate per quegli errori non recuperabili con la gestione try e catch e che si possono evitare prestando attenzione quando si scrive il codice (es: index out of bound exception). Questi tipi di errori NON devono necessariamente usare la clausola throws.
• Errors: Sono quegli errori irrecuperabili che l'utente non può gestire (memoria finita)

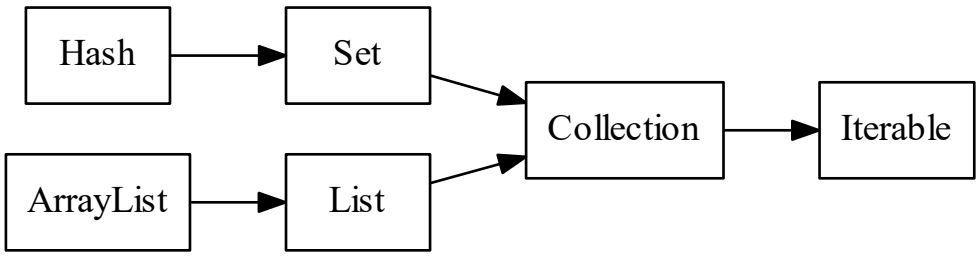
13 21-03-2019: COLLECTION, LIST E SET





- -> ITERABLE:Posso solo scorrere
- —> COLLECTION: Posso scorrere, aggiungere e togliere
- ——> LIST: Posso scorrere e aggiungere o togliere con un indice
- ———-> ARRAYLIST

ArrayList è sottotipo di list, in quanto è una classe che implementa list!



Caratteristiche dell'interfaccia set:
• Gli elementi non sono duplicati
• Gli elementi non sono ordinati in base a come sono inseriti
• Non vengono inseriti metodi nuovi, eredita solo quelli del padre
• I metodi nuovi vengono messi nella classe che implementa l'interfaccia

Per evitare di riprodurre codice si usano le classi astratte dalle quali poi si erediterà.

Relazione di ordinamento: operatore binario che permette di mappare elementi di due insiemi diversi.

Una classe con metodi tutti statici non si può costruire. Rappresenta dunque un contenitore di metodi (è un pezzo di libreria).

SET
Il professore questo giorno ha caricato su github del codice chiamato: TinyJDK. Lo riporto per completezza:

```
1 /* Classe: MyIterable.java */
2 public interface MyIterable<E> {
3
4     MyIterator<E> iterator();
5     int find(E x) throws Exception;
6
7 }
```

```
1 /* Classe: MyIterator.java */
2 public interface MyIterator<E> {
3     boolean hasNext();
```

```
    E next();
}
```

```
/* Classe: MyCollection.java */
import java.util.Collection;
import java.util.function.Function;

public interface MyCollection<T> extends MyIterable<T> {
    void add(T x);
    void clear();
    void remove(T x);   // da decidere se ci piace o no
    boolean contains(T x);
    boolean contains(Function<T, Boolean> p);
    int size();


}
```

```
/* Classe: MyList.java */

public interface MyList<T> extends MyCollection<T> {
    void add(int i, T x);
    T get(int i);
    void set(int i, T x);
}
```

```
/* Classe: MySet.java */
public interface MySet<T> extends MyCollection<T> {
}
```

```
/* Classe: MyArrayList.java */
import java.util.Collection;
import java.util.function.Function;

public class MyArrayList<T> implements MyList<T> {

    private Object[] a;
    private int actualSize;

    public static class MyException extends Exception {
        public MyException(String s) {
            super(s);
        }
    }
```

```
15
16 /*    T[] toArray() {
17         return (T[]) a;
18     }*/
19
20 /*    public static Exception returnNow() {
21         return new Exception("msg");
22     }
23
24     public static void throwNow() throws Exception {
25         throw new Exception("msg");
26     }
27
28     public static void caller() throws Exception {
29         Exception e = returnNow();
30         throwNow();
31     }
32
33     public static void caller2() {
34         try {
35             caller();
36         }
37         catch (Exception e2) {
38             // fai qualcosa con e2
39         }
40
41     }
42
43     public void m(int x) throws Exception {
44         MyException e = new MyException("error messagge");
45         if (x < 0) throw e;
46     }
47   */
48
49     public MyArrayList() {
50         clear();
51     }
52
53     public static class NotFoundException extends Exception {
54     }
55
56     @Override
57     public int find(T x) throws NotFoundException {
58         int cnt = 0;
59         MyIterator<T> it = iterator();
60         while (it.hasNext())
61         {
62             T y = it.next();
63             if (x.equals(y)) return cnt;
64             ++cnt;
65         }
66         throw new NotFoundException();
67     }
```

```
68
69
70
71
72      public static void main3() {
73          MyArrayList<Integer> c = new MyArrayList<>();
74          try {
75              int index = c.find(6);
76              System.out.println("found at index = " + index);
77          } catch (NotFoundException e) {
78              try {
79                  int index = c.find(7);
80              } catch (NotFoundException e1) {
81
82              }
83
84          }
85      }
86
87      @Override
88      public boolean contains(T x) {
89          for (int i = 0; i < actualSize; ++i) {
90              Object o = a[i];
91              if (o.equals(x)) return true;
92          }
93          return false;
94      }
95
96
97      @Override
98      public boolean contains(Function<T, Boolean> p) {
99          return false;
100     }
101
102     @Override
103     public int size() {
104         return actualSize;
105     }
106
107
108     @Override
109     public void clear() {
110         a = new Object[100];
111         actualSize = 0;
112     }
113
114     @Override
115     public void add(T o) {
116         a[actualSize++] = o;
117         if (actualSize >= a.length) {
118             Object[] u = new Object[a.length * 2];
119             for (int j = 0; j < a.length; ++j)
120                 u[j] = a[j];
```

```
            a = u;
        }
    }

    @Override
    public MyIterator<T> iterator() {
        return new MyIterator<T>() {
            private int pos = 0;

            @Override
            public boolean hasNext() {
                return pos <= actualSize;
            }

            @Override
            public T next() {
                return (T) MyArrayList.this.a[pos++];
            }
        };
    }

    @Override
    public void add(int i, T x) {

    }

    @Override
    public T get(int i) {
        return (T) a[i];
    }

    @Override
    public void set(int i, T x) {
        a[i] = x;
    }

    @Override
    public void remove(T x) {

    }

}
```

```
/* Classe: MyArrayListSet.java */
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.function.Function;

public class MyArrayListSet<T extends Comparable<T>> implements MySet<T> {
```

```
9       private final Comparator<T> p;
10      private final ArrayList<T> a;
11
12      public MyArrayListSet(Comparator<T> p) {
13          this.a = new ArrayList<T>();
14          this.p = p;
15      }
16
17      @Override
18      public void add(T x) {
19          if (!contains(x)) {
20              a.add(x);
21              sort();
22          }
23      }
24
25      private void sort() {
26          Collections.sort(a, p);
27      }
28
29      @Override
30      public void clear() {
31          a.clear();
32      }
33
34      @Override
35      public void remove(T x) {
36          a.remove(x);
37      }
38
39      @Override
40      public boolean contains(T x) {
41          return a.contains(x);
42      }
43
44      @Override
45      public boolean contains(Function<T, Boolean> p) {
46          return a.contains(p);
47      }
48
49      @Override
50      public int size() {
51          return a.size();
52      }
53
54      @Override
55      public MyIterator<T> iterator() {
56          return a.iterator();
57      }
58
59      @Override
60      public int find(T x) throws Exception {
61          return a.find(x);
```

```
62          }
63  }
```

14 25-03-2019: LIST E SET

Ha continuato quello che ha fatto la scorsa lezione... in più ha spiegato le interfacce *comparable* e *comparator*
• *Comparable*: ha un unico metodo compareTo(T obj) che compara se stesso con un oggetto di tipo T
• *Comparator*: ha molti metodi, il metodo compare(T a, T b) ci permette di realizzare una classe che compara oggetti tra di loro.
Vediamo un esempio della classe Sort:

```
Sort(T a) /* Posso usare questo metodo quando
       gli oggetti sono giá comparabili */
Sort(T a, Comparator<T>) /* Uso questo metodo
         quando voglio rendere gli oggetti
         comparabili al momento */
```

Inserisco il codice caricato dal professore quel giorno, esso o modifica classi già definite la lezione precedente o ne aggiunge di nuove:

```
/* Classe: MySortedSet.java */
public interface MySortedSet<T extends Comparable<T>> extends MySet<T> {

}
```

```
/* Classe: MyArrayListSortedSet.java */
import java.util.*;
import java.util.function.Function;

public class MyArrayListSortedSet<T extends Comparable<T>>
        extends MyAbstractArrayListSet<T> {

    public MyArrayListSortedSet() {
        super();
    }

    @Override
    protected void sort() {
        Collections.sort(a);
    }

}
```

```java
/* Classe: MyArrayListSet.java */
/* é stata modificata la classe dell'altra volta*/

import java.util.*;
import java.util.function.Function;

public class MyArrayListSet<T> extends MyAbstractArrayListSet<T> {

    private final Comparator<T> p;

    public MyArrayListSet(Comparator<T> p) {
        super();
        this.p = p;
    }

    @Override
    protected void sort() {
        Collections.sort(a, p);
    }

}
```

```java
/* Classe: MyAbstractArrayListSet.java */
import java.util.*;
import java.util.function.Function;

public abstract class MyAbstractArrayListSet<T> implements MySet<T> {
    protected final ArrayList<T> a;

    protected MyAbstractArrayListSet() {
        this.a = new ArrayList<T>();
    }

    @Override
    public void add(T x) {
        if (!contains(x)) {
            a.add(x);
            sort();
        }
    }

    protected abstract void sort();

    @Override
    public void clear() {
        a.clear();
    }

    @Override
    public void remove(T x) {
        a.remove(x);
```

```
30      }
31
32      @Override
33      public boolean contains(T x) {
34          return a.contains(x);
35      }
36
37      @Override
38      public boolean contains(Function<T, Boolean> p) {
39          return a.contains(p);
40      }
41
42      @Override
43      public int size() {
44          return a.size();
45      }
46
47      @Override
48      public MyIterator<T> iterator() {
49          Iterator<T> it = a.iterator();
50          return new MyIterator<T>() {
51
52              @Override
53              public boolean hasNext() {
54                  return it.hasNext();
55              }
56
57              @Override
58              public T next() {
59                  return it.next();
60              }
61          };
62      }
63
64      @Override
65      public int find(T x) throws Exception {
66          int r = a.indexOf(x);
67          if (r < 0) throw new Exception("not found");
68          return r;
69      }
70  }
```

```
1   /* Classe: Main.java */
2   import java.util.*;
3
4   public class Main {
5
6       public static class Plant {
7           private int height;
8
9       }
```

```
10
11      public static class Animal implements Comparable<Animal> {
12          private int weight;
13          private String name;
14
15          public Animal(String name, int w) {
16              this.name = name;
17              this.weight = w;
18          }
19
20          public int getWeight() { return weight; }
21
22          public String getName() {
23              return name;
24          }
25
26          @Override
27          public int compareTo(Animal o) {
28              return this.weight - o.weight;
29              /*if (this.weight == o.weight) return 0;
30              else if (this.weight > o.weight) return 1;
31              else return -1;*/
32          }
33      }
34
35      public static class Dog extends Animal {
36
37          public Dog(String name, int w) {
38              super(name, w);
39          }
40      }
41
42      public static void main(String[] args) {
43
44          List<Animal> a = new ArrayList<>();
45          Collections.sort(a);
46
47          List<Plant> b = new ArrayList<>();
48          Collections.sort(b, new Comparator<Plant>() {
49              @Override
50              public int compare(Plant x, Plant y) {
51                  return x.height - y.height;
52              }
53          });
54
55      }
56  }
```

15 28-03-2019: MAP

Affinchè due classi siano confrontabili devono implementare il metodo compare to
Solitamente si usa il modificatore protected invece che private in modo tale che da fuori non possano essere modificate, ma possano essere viste quando vengono ereditate.

MAPPE
Una mappa è un oggetto che mappa chiavi con valori. Non ci possono essere chiavi duplicate e ogni chiave può mappare al più un valore.
Una mappa rappresenta una collection associativa.
Una mappa è parametrica su due tipi di argomenti. Uno rappresenta il dominio (chiave), e l'altro rappresenta il codominio (valore). La gerarchia delle mappe è slegata dalla gerarchia delle collection.
Per la documentazione java sulle mappe consultare questo SITO
Una mappa è una collection associativa, con coppie associate a valori.

Il foreach di JAVA funziona solo con le librerie del JDK originale di JAVA.
Super può essere usato per il costruttore o per metodi che eredito, non è di perse un metodo e non ha un tipo.

Durante la lezione il professore ha aggiunto nel repository di github il seguente codice:

```java
/* Classe: Pair.java */
public class Pair<A, B> {
    private A a;
    private B b;

    public Pair(A a, B b) {
        this.a = a;
        this.b = b;
    }

    public A getFirst() { return a; }
    public B getSecond() { return b; }

}
```

```java
/* Classe: NotFoundException.java */
public class NotFoundException extends Exception {
}
```

```java
/* Classe: MyMap.java */
public interface MyMap<K, V> extends MyCollection<Pair<K, V>> {
  /*K = sono le chiavi */
```

```
  /*V = sono i valori  */
    void put(K key, V value);
    V get(K key) throws NotFoundException;

}
```

Decidiamo di implementare la nostra mappa in due maniere diverse!

```
/* Classe: MyListMap__old.java */
import java.util.function.Function;

public class MyListMap__old<K, V> implements MyMap<K, V> {

    private MyList<Pair<K, V>> l;

    public MyListMap__old() {
        this.l = new MyArrayList<>();
    }

    @Override
    public void put(K key, V value) {
        l.add(new Pair<K, V>(key, value));
    }

    @Override
    public V get(K key) throws NotFoundException {
        MyIterator<Pair<K, V>> it = l.iterator();
        while (it.hasNext()) {
            Pair<K, V> p = it.next();
            if (key.equals(p.getFirst())) return p.getSecond();
        }
        throw new NotFoundException();
    }

    @Override
    public void add(Pair<K, V> x) {
        put(x.getFirst(), x.getSecond());
    }

    @Override
    public void clear() {
        l.clear();
    }

    @Override
    public void remove(Pair<K, V> x) {
        l.remove(x);
    }

    @Override
    public boolean contains(Pair<K, V> x) {
```

```
44          return l.contains(x);
45      }
46
47      @Override
48      public boolean contains(Function<Pair<K, V>, Boolean> p) {
49          return l.contains(p);
50      }
51
52      @Override
53      public int size() {
54          return l.size();
55      }
56
57      @Override
58      public MyIterator<Pair<K, V>> iterator() {
59          return l.iterator();
60      }
61
62      @Override
63      public int find(Pair<K, V> x) throws Exception {
64          return l.find(x);
65      }
66  }
```

```
1   /* Classe: MyListMap.java */
2   public class MyListMap<K, V> extends MyArrayList<Pair<K, V>>
3           implements MyMap<K, V> {
4
5       @Override
6       public void put(K key, V value) {
7           add(new Pair<K, V>(key, value));
8       }
9
10      @Override
11      public V get(K key) throws NotFoundException {
12          MyIterator<Pair<K, V>> it = iterator();
13          while (it.hasNext()) {
14              Pair<K, V> p = it.next();
15              if (key.equals(p.getFirst())) return p.getSecond();
16          }
17          throw new NotFoundException();
18      }
19  }
```

```
1   /* Classe: MyArrayListSortedSet.java */
2   /* é stata modificata la classe dell'altra volta*/
3   import java.util.*;
4   import java.util.function.Function;
5
```

```
6  public class MyArrayListSortedSet<T extends Comparable<T>>
7          extends MyAbstractArrayListSet<T>
8          implements MySortedSet<T> {
9
10     public MyArrayListSortedSet() {
11         super();
12     }
13
14     @Override
15     protected void sort() {
16         Collections.sort(a);
17     }
18
19 }
```

```
1  /* Classe: Main.java */
2  /* é stata modificata la classe dell'altra volta*/
3  /* riporto solo la modifica: ha aggiunto main2 */
4
5      public static void main2() {
6          MyCollection<Pair<String, Integer>> rubrica = new MyListMap<>();
7          rubrica.add(new Pair<>("Alvise", 34712345));
8          rubrica.add(new Pair<>("Diego", 987654321));
9      }
```

Finita la lezione il professore ha deprecato la classe: MyListMap_ _ old.java

16 1-04-2019: HASH MAP

HASH MAP
Le hash Map sono sempre delle mappe, solo che il tipo di una chiave viene identificato con un interno. Questo può rendere a ricerca più veloce!.
Per cercare si esegue l'hash di quello che voglio cercare (per esempio hashare una stringa), poi vado a cercare in un array indicizzato (questa operazione costa O(1) costante).
Le l'hash map sono però inefficienti in termini di memoria.

Il costruttore dell' hash map è il seguente:

```
1  HashMap(int initalcapacity, float loadFactor)
```

Esso costruisce una hash map vuota con una capacità iniziale uguale a quella specificatta e con un fattore di carico. Il fattore di carico (load factor) specifica di quanto si deve moltiplicare quando si ingrandisce la mappa.

```
1  HashMap(Map<? extends k, ? extends v> m)
```

Costruisce una nuova hash map, popolandola con i valori della mappa passata. Viene chiamato costruttore per copia.

Si subsume fino al punto che serve (un buon compromesso è subsumere fino all'interfaccia). Se però mi servono dei metodi specifici, non possono subsumere.

La classe Object ha un metodo hashcode che tutti gli oggetti ereditano. Serve per hashare this.

Il professore ha aggiunto il seguente codice nel suo repository github quel giorno:

```
1  /* Classe: Main.java */
2  /* é stata modificata la classe dell'altra volta */
3  /* Ha aggiunto solo l'hashCode e il main3()      */
4
5  import java.util.*;
6
7  public class Main {
8
9      public static class Plant {
10         private int height;
11
12     }
13
14     public static class Animal implements Comparable<Animal> {
15         private int weight;
16         private String name;
17
```

```
18          public Animal(String name, int w) {
19              this.name = name;
20              this.weight = w;
21          }
22
23          public int getWeight() { return weight; }
24
25          public String getName() {
26              return name;
27          }
28
29          @Override
30          public int compareTo(Animal o) {
31              return this.weight - o.weight;
32          }
33
34          @Override
35          public int hashCode() {
36              return weight * name.hashCode();
37          }
38
39      }
40
41      public static class Dog extends Animal {
42
43          public Dog(String name, int w) {
44              super(name, w);
45          }
46
47          @Override
48          public int hashCode() {
49              return super.hashCode();
50          }
51      }
52
53      public static void main3() {
54
55          Map<Dog, String> m = new HashMap<>();
56          Dog emma = new Dog("emma", 10);
57          Dog toby = new Dog("toby", 10);
58          Dog bob = new Dog("bob", 20);
59
60          m.put(emma, "cecilia");
61          m.put(toby, "mihail");
62          m.put(bob, "alex");
63      }
64
65
66
67
68
69      public static void main2() {
70          MyCollection<Pair<String, Integer>> rubrica = new MyListMap<>();
```

```
71          rubrica.add(new Pair<>("Alvise", 34712345));
72          rubrica.add(new Pair<>("Diego", 987654321));
73      }
74
75
76      public static void main(String[] args) {
77
78          List<Animal> a = new ArrayList<>();
79          Collections.sort(a);
80
81          List<Plant> b = new ArrayList<>();
82          Collections.sort(b, new Comparator<Plant>() {
83              @Override
84              public int compare(Plant x, Plant y) {
85                  return x.height - y.height;
86              }
87          });
88
89      }
90  }
```

17 4-04-2019: SINGLETON E THREAD

DESIGN PATTERN SINGLETON
Si limita l'istanzazione di una classe a uan sola istanza.
Può servire per creare oggetti statici (per esempio un campo statico esiste anche prima dell'istanza dell'oggetto del tipo della classe a cui appartiene).
Un costruttore privato può essere accessibile solo dalla classe, e non dall'esterno.
I metodi statici che ritornano lo stesso tipo del costruttore della classe sono costruttori travestiti. i costruttori sono metodi d'istanza. Non serve il modificatore di accesso static, anche se sono simili a metodi statici.
I metodi statici sono pseudo costruttori, se usati nel modo opportuno.
Un metodo statico è un metodo che non ha bisogno di istanza. un campo statico si può vedere anche se non ha istanza (da un metodo statico si può accedere solo a campi statici, poichè non ha riferimento al this).
In questo corso non andremo oltre con questo design pattner perchè sempre di più in programmazione si sta andando verso la programmazione funzionale (in contrapposizione a quella imperattiva).

```java
/* Classe: Main.java */
package patterns.singleton;

public class Main {

    public static void main(String[] args) {
        Singleton single1 = Singleton.getInstance();
        Singleton single2 = Singleton.getInstance();
        System.out.println("is it the same object? " + (single1 ==
            single2));
    }

}
```

```java
/* Classe: Singleton.java */
package patterns.singleton;

class Singleton {
    private static Singleton instance = null;

    // costruttore privato per non permettere a nessuno di costruire
        questa classe
    private Singleton() {}

    // metodo statico che funge da pseudo-costruttore
    public static Singleton getInstance() {
        if (instance == null)
            instance = new Singleton();
        return instance;
    }
```

```
16 }
```

INTERFACCIA GRAFICA
Un widget è un supertipo dell'interfaccia grafica.
Busy-Loop: ciclo che cicla a vuoto, non restituendo il controllo al S.O.
Le interfacce grafiche non si programmano a busy-loop. Si chiede al sistema operativo di "svegliarmi quando accade un evento". Dunque si programmano a suon di call-back.
un esempio di call-back è il ctr-c per interrompere un programma in una shell linux. Questa combinazione di tasti manda un segnale, ma in realtà viene invocata una call-back.
programmazion ad eventi: si programma con la call-back che vengono chaimate quando opportuno.

THREAD IN JAVA
Esiste una classe Thread che si eredita e si ovverrida.

```
1  /* Classe: Main.java */
2  package threads;
3
4  public class Main {
5
6      private static void loop(int n, int ms) {
7          for (int i = 0; i < n; ++i) {
8              System.out.println(String.format("thread[%d]: #%d",
                   Thread.currentThread().getId(), i));
9              try {
10                 Thread.sleep(ms);
11             } catch (InterruptedException e) {
12                 e.printStackTrace();
13             }
14         }
15     }
16
17     public static class MyThread extends Thread {
18         private final int n;
19
20         public MyThread(int n) {
21             this.n = n;
22         }
23
24         @Override
25         public void run() {
26             Main.loop(n, 300);
27         }
28
29     }
30
31
32     public static void main(String[] args) {
33         MyThread th = new MyThread(23);
34         /* thread start comincia un nuovo thread
35          * e poi invoca il run */
```

```
36          th.start();
37
38          /* Richiamare il run cosi avvia semplicemente
39           * il metodo run, ma senza cominciare un nuovo
40           * thread. infatti le chiamate di questo run
41           * si sovrappongono a quelle dello start */
42          th.run();
43
44          /* Questa procedura parte dopo che é stata
45           * richiamata e terminata quella invocata
46           * da th.run() */
47          loop(11, 500);
48      }
49
50  }
```

18 8-04-2019: THREAD SYNCHRONIZED

Ai fini didattici andremo ad implementare una pool di thread, ovvero una coda dove sono tenuti dei thread "fermi" da un semaforo mutex. Quando avremo bisogno di un thread andremo a rendere verde il suo mutex e nel momento in cui non ci serve più, al posto di eliminarlo, andremo a rimetterlo in coda.
Una coda è una struttura FIFO, che nel nostro caso deve essere bloccante (blocking queue). Infatti si tratta di una coda che viene blocccata nel momento in cui è vuota e viene richiesto un elemento.Cosi facendo una get() non potrà mai fallire, dato che aspetta sempre che ci sia qualcosa.
Nella nostra versione: se il thread è presente nella coda lo usiamo e lo facciamo andare, se invece non è presente lo creiamo al momento. La pool di thread quindi crea thread al momento del bisogno, ed implica che la coda può ingigantirsi all'infinito.
L'interfaccia che più si avvicina alle nostre esigenze è: *Queue<E>*

Synchronized
• *Synchronized come firma del metodo:* é una keyword di java che inserita nella firma di un metodo garantisce la mutua esclusione del metodo in modo tale che venga eseguito da un thread alla volta. Il Sync di sotto è uguale a questo solo che usa this come oggetto per il mutex, perciò garantisce la mutua esclusione per tutti i metodi dello stesso oggetto (anche metodi differenti della stessa istanza sono sincronizzati sul this e quindi si aspettano a vicenda), oggetti differenti funzionano senza "sentire il mutex".
• *Synchronized come mutex di un oggetto:* la stessa keyword può essere usata anche come segue:*Synchronized(Object)azioni da sincronizzare* questo implica che ogni oggetto può essere usato come un mutex (Object ha infatti i metodi "wait" e "notify").
Il miglior metodo per gestire la concorrenza è usare Synchronized all'iterno di un metodo e non nella sua firma.
Di sbagliato infatti c'è che non va mai bene proteggere tutto un metodo, ma andrebbe protetta soltanto la sezione critica! Notiamo che Synchronized è uno statement e non un'espressione, quindi al posto di un for useremo un while.

```
1  /* Classe: Main.java */
2  /* é stata modificata la classe dell'altra volta  */
3  /* Ha rimosso il run ed aggiunto i due syncronize */
4  package threads;
5
6  public class Main {
7    /* avendolo messo static, se non sincronizzo
8     * puó essere che due thread si pestino i
9     * piedi sul valore di "i" */
10     public static Integer i = 0;
11
12     private static void loop(int n, int ms) {
13
14         synchronized (i) { i = 0; }
15         while (i < n) {
16             System.out.println(String.format("thread[%d]: #%d",
                   Thread.currentThread().getId(), i));
17             try {
```

```
18                  Thread.sleep(ms);
19              } catch (InterruptedException e) {
20                  e.printStackTrace();
21              }
22              synchronized (i) { ++i; }
23          }
24
25      }
26
27      public static class MyThread extends Thread {
28          private final int n;
29
30          public MyThread(int n) {
31              this.n = n;
32          }
33
34          @Override
35          public void run() {
36              Main.loop(n, 300);
37          }
38
39      }
40
41
42      public static void main(String[] args) {
43          MyThread th = new MyThread(23);
44          th.start();
45          /* ha rimosso il run perché é giá richiamato
46           * dalla th.start(); */
47          loop(11, 500);
48      }
49
50  }
```

Vediamo un modo migliore di sincronizzare il thread:

```
1   /* Classe: SynchronizedMain.java */
2   package threads;
3
4   public class SynchronizedMain {
5       public static final Counter counter = new Counter();
6
7       private static class Counter {
8           public static final int MAX = 30;
9
10          private int value = 0;
11
12          public synchronized int get() {
13              return value;
14          }
15
```

```
16          public synchronized void set(int x) {
17              value = x;
18          }
19
20          // questo metodo fa la lettura e il post-incremento in maniera
                ATOMICA
21          public synchronized int getAndIncrement() {
22              return value++;
23          }
24      }
25
26      private static void printAndSleep(int counter, int delay) {
27          System.out.println(String.format("thread[%d]: #%d",
                Thread.currentThread().getId(), counter));
28          try {
29              Thread.sleep(delay);
30          } catch (InterruptedException e) {
31              e.printStackTrace();
32          }
33      }
34
35      // usando i metodi get() e set() di Counter, non é garantita
            l'atomicitá dell'operazione di incremento
36      // non c'é corruzione di memoria perché sia le letture (get) sia le
            scritture (set) sono atomiche
37      // tuttavia il comportamento non é quello atteso e produce in output
            piú thread che contemporaneamente stampano lo stesso counter
38      private static void loop__BAD(int delay) {
39          while (counter.get() < Counter.MAX) {
40              printAndSleep(counter.get(), delay);
41              counter.set(counter.get() + 1);
42          }
43      }
44
45      // qui invece usiamo un metodo apposito che legge e incrementa in modo
            ATOMICO
46      // in questo modo il comportamento é corretto
47      private static void loop__GOOD(int delay) {
48          int i;  // occorre una variabile di appoggio: é un binding nello
                scope locale di una copia dell'intero post-incrementato
49          while ((i = counter.getAndIncrement()) < Counter.MAX) {
50              printAndSleep(i, delay);    // se al posto della variabile di
                    appoggio i usassimo direttamente counter.get() qui,
                    potremmo ottenere un valore sbagliato       }
51          }
52      }
53
54      private static void loop(int delay) {
55          // cambiare la chiamata per provare entrambe le versioni
56          //loop__BAD(delay);
57          loop__GOOD(delay);
58      }
59
```

```
60      private static class MyThread extends Thread {
61          private final int delay;
62
63          public MyThread(int delay) {
64              this.delay = delay;
65          }
66
67          @Override
68          public void run() {
69              SynchronizedMain.loop(delay);
70          }
71
72      }
73
74      public static void main(String[] args) {
75          MyThread th = new MyThread(500);
76          th.start();
77          loop(300);
78      }
79
80  }
```

19 11-04-2019: THREAD POOL

ConcurrentLinkedQueue
Aggiunto appunti miei... ConcurrentLinkedQueue è semplicemente una coda di tipo FIFO, che può essere usata da più thread contemporaneamente.Essa ci garantisce che se un thread aggiunge un elemento mentre un'altro thread ne sta aggiungendo un'altro non si verifichino problemi di memoria. Nulla vieta però che se la coda ha un elemento e cerchiamo di leggerlo, un'altro thread non l'abbia già letto e ci ritroviamo a leggere da una coda vuota.

wait() and notify()
• *wait():* Forza il thread corrente ad aspettare. Il thread viene sospeso fino a che qualcuno non chiama notify
• *notify():* Risveglia e mette in run i thread che erano stati fermati con wait.

Gli appunti dell'altra volta sono gli stessi di questa lezione, il professore ha aggiunto un po di codice.

```
1  /* Classe: SynchronizedMain.java */
2  /* ha modificato il codice della scorsa volta */
3  /* riporto solo la modifica */
4      public static void main(String[] args) {
5  //         MyThread th = new MyThread(500);
6  //         th.start();
7          new MyThread(500).start();
8
9          new Thread(() -> loop(400)).start();
10
11         loop(300);
12     }
```

Sucecssivamente è tornato a concentrarsi sui thread pool.

```
1  /* Classe: ThreadPool.java */
2  package threads;
3
4  public class ThreadPool implements Queue<Thread> {
5
6
7
8  }
```

```
1  /* Classe: ThreadPool.java */
2  /* Questa classe non é ancora completa
3   * é una bozza, verrá completata nelle prossime
4   * lezioni */
5  import java.util.Queue;
```

```
6  import java.util.concurrent.ConcurrentLinkedQueue;
7
8  public class ThreadPool {
9      private Queue<PooledThread> q = new ConcurrentLinkedQueue<>();
10
11     public static class PooledThread extends Thread {
12         private Runnable r;
13     }
14
15     public PooledThread acquire(Runnable cb) {
16         if (q.isEmpty()) {
17             return new PooledThread();
18         }
19         else {
20             PooledThread p = q.poll();
21             p.notify();
22         }
23     }
24
25     public void release(PooledThread t) {
26         q.add(t);
27     }
28
29     public static void threadMain(PooledThread p) {
30         while (true) {
31             try {
32                 p.wait();
33                 p.r.run();  // TODO: potrebbe essere null
34             } catch (InterruptedException e) {
35                 e.printStackTrace();
36             }
37         }
38     }
39
40     public static void main(String[] args) {
41         ThreadPool pool = new ThreadPool();
42         Thread t1 = pool.acquire();
43         t1.
44     }
45 }
```

Succcssivamente ha modificato ancora la synconized main

```
1  /* Classe: SynchronizedMain.java */
2  /* é stata modificata la classe di prima    */
3  /* ne riporto solamente la modifica al main */
4      public static void main(String[] args) {
5          MyThread th1 = new MyThread(500);
6          th1.start();     // spawn thread 1
7
```

```
8           // non serve conservare l'oggetto thread in una variabile se non é
                necessario
9           new Thread(() -> loop(400)).start();     // spawn thread 2 con un
                runnable in costruzione
10
11          // esegue lo stesso codice anche col main thread
12          loop(300);
13      }
```

20 15-04-2019

Di questa lezione al momento ho solo il codice del professore Non ho aggiunto tutto il codice perchè una parte penso sia solo il risultato finale e lo farà prossimamente

```java
/* Classe: ThreadPool.java */
/* Ha modificato la classe */
/* In modo da completarla, la riporto tutta */

package threads.work_in_progress;

import org.jetbrains.annotations.NotNull;
import org.jetbrains.annotations.Nullable;

import java.util.Objects;
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;

public class ThreadPool {
    private Queue<PooledThread> q = new ConcurrentLinkedQueue<>();

    public static class PooledThread extends Thread {
        @Nullable
        private Runnable currentRunnable;

        @Override
        public void run() {
            while (true) {
                try {
                    synchronized (this) {
                        print("waiting...");
                        wait();
                        print("awaken");
                    }
                    Objects.requireNonNull(currentRunnable).run();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }

        @Override
        public String toString() {
            return String.format("[%s:%d]", this.getName(), this.getId());
        }

    }

    public PooledThread acquire(Runnable cb) {
        @NotNull final PooledThread p;
        if (q.isEmpty()) {
```

```
47              System.out.println("spawning");
48              p = new PooledThread();
49              p.start();
50          } else {
51              p = q.poll();
52          }
53          synchronized (p) {
54              print("notify");
55              p.currentRunnable = cb;
56              p.notify();
57              return p;
58          }
59      }
60
61      public void release(PooledThread t) {
62          q.add(t);
63      }
64
65
66      public static void main(String[] args) {
67          ThreadPool pool = new ThreadPool();
68          Thread t1 = pool.acquire(() -> {
69              try {
70                  Thread.sleep(300);
71              } catch (InterruptedException e) {
72                  e.printStackTrace();
73              }
74              print("bye");
75          });
76      }
77
78      private static void print(String s) {
79          System.out.println(String.format("%s: %s", Thread.currentThread(),
                s));
80      }
81  }
```

21 18-04-2019

PLACE HOLDER

22 29-04-2019

PLACE HOLDER

23 2-05-2019

PLACE HOLDER

24 6-05-2019

PLACE HOLDER