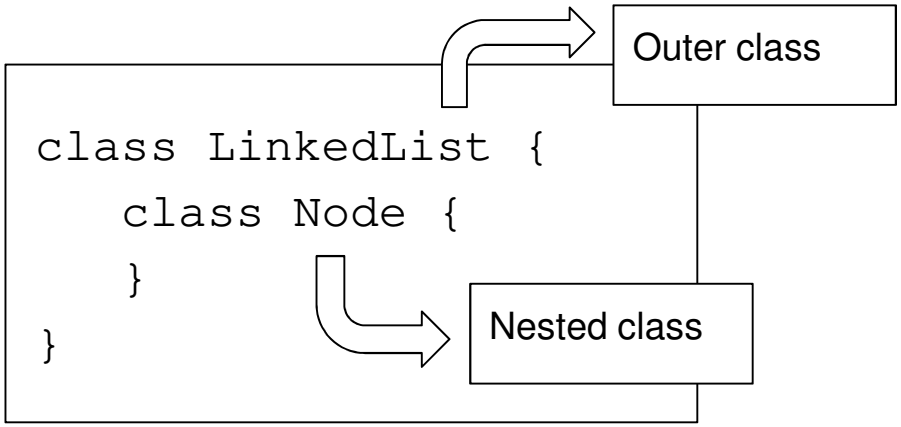
Java

Clases anidadas
Annotations
Colecciones

Clases anidadas

Una variable de una clase puede tener otra clase como miembro. Java permite definir una clase dentro de otra



- Los métodos de la clase interna tienen acceso a datos que estarían fuera de su alcance si se la define fuera de la "*Outer class*"
- La "*nested class*" puede estar oculta a otras clases en el mismo *package*

Clases anidadas: inner class

Pueden ser *private* o no. Si no son *private* se pueden instanciar desde fuera de la *outer class.*
Para poder instanciar la inner class debe existir una instancia de la outer class.

```
class OuterClass {
    int variable;
    class InnerClass {
        // puede acceder a variable
    }
}
```

```
OuterClass outerObject = new OuterClass();
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

Inner class: ejemplos

```
public static class TimerClock {
   private int interval;   boolean beep;

   public TimerClock(int interval, boolean beep) {
       this.interval = interval;       this.beep = beep;
   }
   public void start() {
       ActionListener listener = new TimePrinter();
       Timer t = new Timer(interval, listener);
       t.start();
   }
   public class TimePrinter implements ActionListener {
       @Override
       public void actionPerformed(java.awt.event.ActionEvent e) {
           System.out.println("Time: " + LocalDate.now());
           if ( beep)
                ...
       }
   }
}
```

Clases anidadas: static nested class

Pueden ser *private* o no. No necesitan una instancia de la outer class y no pueden acceder a métodos ni variables de instancia de la outer class.

```
class OuterClass {
    int count;
    static class InnerClass {
        // no puede acceder a count
    }
}
```

```
OuterClass.InnerClass innerObject = new OuterClass.InnerClass();
```

Clases anidadas: static nested class

Las clases privadas permiten el encapsulamiento.

```
class LinkedList<T> {
   private Node<T> head;

   private static class Node<E> {
       E value;
       Node<E> tail;
       ...
   }
}
```

Annotations

Surgen en JDK 1.5. En su forma más simple sirven como "marcador" de una clase, método, etc.

```
@FunctionalInterface
public interface Function<T, R>
```

```
@Override
public void method(Something param)
```

```
@Deprecated
protected int dontUseMe()
```

```
@Deprecated
public class DontUseMe
```

```
import org.hibernate.envers.Audited;

@Entity
@Audited
public abstract class PersistentClass
```

Annotations y sus posible usos

1. **Directivas al compilador:** `@Deprecated, @Override, @SuppressWarnings, @FunctionalInterface`
2. **Instrucciones para compilación:** `@Regex, @ReadOnly, @NonNull`
3. **Directivas para ejecución:** son accedidas en tiempo de ejecución por la aplicación

Algunas anotaciones pueden requerir valores asociados:

```
@Author(
        name = "John Doe",
        date = "3/27/2003"
)
public enum Rating{
```

```
@SuppressWarnings(value = "unchecked")
```

```
@SuppressWarnings("unchecked")
```

```
@Transactional(readOnly = true)
Client getClientById(Integer id);
```

@Override

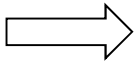
No es obligatorio usarla. Le indica al compilador que el método sobreescribe un método definido por un ancestro.

```
public class IntegerWrapper {
   private Integer value;

   public IntegerWrapper(int value) {
      this.value = value;
   }

   public boolean equals(IntegerWrapper obj) {
       return value == obj.value;
   }
}
```

¿Qué imprime?



```
public static void main(String[] args) {
   Object n1 = new IntegerWrapper(15);
   Object n2 = new IntegerWrapper(15);

   System.out.println(n1.equals(n2));
}
```

Ejemplos

```
public class Example {

   @SuppressWarnings("deprecation")
   private Date date = new Date(2019, 3, 12);
```

```
@SuppressWarnings("deprecation")
public class Example {

   private Date date = new Date(2019, 3, 12);
```

Creando nuestras propias Annotations

Ejemplo: Queremos agregar validaciones a ciertas propiedades antes de persistirlas, en principio que no sea `null`.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface NotNull {

}
```

Creando nuestras propias Annotations

```
public abstract class Entity {
    @NotNull
    private String id;
}
```

```
public class Professor extends Entity {
    @NotNull
    private String description;
    @NotNull
    private String password;
    private boolean active = true;
    private String remarks;
}
```

Creando nuestras propias Annotations

Para crear una annotation se deben definir dos cosas:

- **Una política de retención** (*retention policy*) especifica cuánto perdura la annotation en el ciclo de vida del programa
  - SOURCE: es descartada por el compilador
  - CLASS: se mantiene hasta generar el .class. Pero no es requerida por la JVM que procesa el .class
  - RUNTIME: son usadas en tiempo de ejecución
- **Un objetivo** (*target)*:
  - CONSTRUCTOR
  - FIELD
  - METHOD
  - PARAMETER
  - etc.

Creando nuestras propias Annotations

La annotation puede ser simplemente una marca o tener uno o más parámetros limitados a:

- tipos primitivos (`int`, `double`, etc.)
- String
- class
- enum
- annotation
- array de los anteriores

Creando nuestras propias Annotations

```
public static void store(Entity entity) throws IllegalAccessException {
   for(Field field : entity.getClass().getDeclaredFields()){
       Class type = field.getType();
       String name = field.getName();
       field.setAccessible(true);
       Annotation[] annotations = field.getDeclaredAnnotations();
       for (Annotation ann : annotations) {
           if (ann instanceof NotNull && field.get(entity) == null) {
               throw new RuntimeException(name + " can't be null");
           }
       }
   }
   // OK, ya podemos persistir la entidad
   ...
}
```

Creando nuestras propias Annotations

Agregamos ahora la posibilidad, para atributos numéricos, acotar sus valores

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Range {
   boolean allowNull() default false;
   double min();
   double max();
}
```

Creando nuestras propias Annotations

```
public class Professor extends Entity {
    @NotNull
    private String description;
    @NotNull
    private String password;
    private boolean active = true;
    private String remarks;
    @Range(min = MIN_AGE, max = MAX_AGE)
    private Integer age;

    @Range(min = 5.0, max = 10.0, allowNull = true)
    private Double extra;
}
```

Creando nuestras propias Annotations

```
if (ann instanceof Range) {
   Number value;
   try {
       value = (Number) field.get(entity);
   } catch (Exception e) {
       // Si no es el tipo adecuado, ignoramos la anotación
       continue;
   }
   Range r = (Range) ann;
   if (value == null) {
       if (!r.allowNull())
           throw new RuntimeException(name + " can't be null");
   } else if (r.max() < value.doubleValue() ||
              r.min() > value.doubleValue()) {
       throw new RuntimeException(name + " must be between " +
              r.min() + " and " + r.max());
   }
}
```

Java Collections Framework

Una colección es un objeto que agrupa a un conjunto de elementos compatibles.

Permite almacenar, eliminar, recuperar elementos

Java Collections Framework es un conjunto de interfaces, clases y métodos para manipular colecciones, abstrayéndonos de su implementación.

Collections: Interfaces

- **Iterable**
  - **Collection**
    - **Set**
      - **SortedSet**
        - **NavigableSet**
    - **List**
    - **Queue**
      - **Deque**
- **Map**
  - **SortedMap**
    - **NavigableMap**

https://docs.oracle.com/javase/8/docs/api/java/util/package-tree.html

Iterator

```
public interface Iterator<E> {

    boolean hasNext();

    E next();

    default void remove() {
                throw new
UnsupportedOperationException("remove");
    }

    ...
}
```

Iterator: ejemplo

```
public class Fibonacci implements
Iterator<Integer> {

    private int result = 0;
    private int next = 1;

    @Override
    public boolean hasNext() {
        return true;
    }

    @Override
    public Integer next() {
        int res = result;
        int aux = result + next;
        result = next;
        next = aux;
        return res;
    }

    @Override
    public void remove() {
        // nada
    }
}
```

Ejercicio

Agregar a la implementación del ejercicio 7.3 un método que retorne un iterador sobre la lista.
No implementar remove().

```
package ar.edu.itba.poo.tp7.list;

public class LinearListImpl<T> implements LinearList<T> {

   private Node<T> first;

   @Override
   public void add(T obj) {
       Node<T> current = first;
       if (first == null) {
           first = new Node<>(obj, null);
       } else {
           while (current.tail != null) {
               current = current.tail;
           }
           current.tail = new Node<>(obj, null);
       }
   }
```

Iterator

```
List<Integer> l = new ArrayList<>();

Iterator<Integer> it = l.iterator();

while ( it.hasNext()) {
   Integer i = it.next();
   ...
}
```

```
for (Integer i : l) {
   ...
}
```

```
while ( it.hasNext()) {
   Integer i = it.next();
   if ( i % 2 == 1)
       it.remove();
}
```

```
while ( it.hasNext()) {
   Integer i = it.next();
   if ( i % 2 == 1)
       l.remove(i);
}
```

ListIterator

Un iterador que además permite recorrer una lista en ambos sentidos y reemplazar un elemento

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void set(E e);
    void add(E e);
}
```

ListIterator

```
List<Integer> l = new ArrayList<>();

for ( int i = 1; i <= 10; i++)  {
   l.add(i);
}
```

```
ListIterator<Integer> li = l.listIterator();
Integer n = li.previous();
```

ListIterator

```
List<Integer> l = new ArrayList<>();

for ( int i = 0; i < 10; i++)  {
   l.add(i);
}
```

```
ListIterator<Integer> li = l.listIterator();
Integer n = li.next();          // 0
n = li.next();                  // 1
li.set(33);
if ( li.hasPrevious())
   n = li.previous();           // 33
n = li.next();                  // 33
li.remove();
n = li.nextIndex();             // 1
li.set(55);                     // IllegalStateException
```

Iterable

Una única funcionalidad: ofrecer un iterador

```
public interface Iterable<T> {
    Iterator<T> iterator();

    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }
    ...
}
```

`Iterable`: ejemplo

```
public class Range implements Iterable<Integer> {
   private int start, end;

   public Range(int start, int end) {
       this.start = start;
       this.end = end;
   }

   public Iterator<Integer> iterator() {
       return new RangeIterator();
   }
```

Iterable: ejemplo

```
private class RangeIterator implements Iterator<Integer> {
        private int cursor;
        public RangeIterator() {
            this.cursor = Range.this.start; // o solo start
        }
        @Override
        public boolean hasNext() {
            return this.cursor < Range.this.end;
        }
        @Override
        public Integer next() {
            if (this.hasNext()) {
                int current = cursor;
                cursor++;
                return current;
            }
            throw new NoSuchElementException();
        }
    }
}
```

`Iterable`: ejemplo

```
public static void main(String[] args) {
   Range range = new Range(1, 10);

   Iterator<Integer> it = range.iterator();
   while (it.hasNext()) {
       int cur = it.next();
       System.out.println(cur);
   }

   for (Integer cur : range) {
       System.out.println(cur);
   }
}
```

Java Collections framework

La interface `Collection` es el mínimo denominador común entre todas las colecciones implementadas. Algunas colecciones aceptan duplicados, otras no. Java no provee ninguna implementación directa de esta interface.

`Set` representa una colección que no puede tener repetidos. Los métodos son los mismos que los de `Collection`.

`List` representa una colección con orden en sus elementos (una secuencia).

Interface `Collection`

```
public interface Collection<E> extends Iterable<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Object[] toArray();
    <T> T[] toArray(T[] a);
    boolean add(E e);
    boolean remove(Object o);
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
    default Stream<E> stream() {
                return StreamSupport.stream(spliterator(), false);
    }
```

Interface `Collection`

```
default boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);
    boolean removed = false;
    final Iterator<E> each = iterator();
    while (each.hasNext()) {
        if (filter.test(each.next())) {
            each.remove();
            removed = true;
        }
    }
    return removed;
}
```

Ejemplos

```
public static void showAll(L
    for(Object o: c) {
        System.out.print(o +
    }
    System.out.print('\n');
}

public class Point {
    private double x, y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}
```

```
public static void showAll(Iterable<?> c) {
    c.forEach(e -> System.out.println(e + ", "));
    System.out.print('\n');
}
```

Ejemplos

```
Collection<Point> points = new ArrayList<>();
points.add(new Point(1.5, 2.0));
points.add(new Point(2.0, 2.5));
points.add(new Point(2.5, 3.0));

showAll(points);
```

```
Point aPoint = new Point(2.0, 2.5);
System.out.println(
    "Contains returns " + points.contains(aPoint));
```

Agregamos `equals` y `hashCode` a Point

```
@Override
public boolean equals(Object o) {
   if (this == o) return true;
   if (o == null || getClass() != o.getClass()) return false;

   Point point = (Point) o;

   return new EqualsBuilder()
           .append(x, point.x).append(y, point.y).isEquals();
}

@Override
public int hashCode() {
   return new HashCodeBuilder(17, 37)
           .append(x).append(y).toHashCode();
}
```

Set

Colecciones que no aceptan elementos repetidos. Algunas implementaciones son:

- `HashSet`: internamente con una tabla de hashing.
- `LinkedHashSet`: los elementos preservan el orden de inserción
- `TreeSet`: ordenados de acuerdo al orden natural o un comparador
- `EnumSet`: los elementos son un enumerativo del mismo tipo

Interface `SortedSet`

```
public interface SortedSet<E> extends Set<E> {
    Comparator<? super E> comparator();
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);
    E first();
    E last();
}
```

Implementaciones: `ConcurrentSkipListSet, TreeSet`

SortedSet: ejemplo

```
Set<Point> set = new TreeSet<>();

set.add(new Point(1.5, 2.0));
set.add(new Point(2.0, 2.5));
set.add(new Point(2.5, 3.0));
set.add(new Point(2.5, 3.0));
set.add(new Point(2.0, 2.5));

System.out.println("Size: " + set.size());

Point aPoint = new Point(2.0, 2.5);
System.out.println("Contains return " + points.contains(aPoint));

showAll(set);
```

Ordenamiento

Todas las clases que implementan ordenamiento se basan en el orden natural de los objetos (`Comparable`), o bien deben recibir un criterio de ordenamiento (`Comparator`).

```
Set<Point> set = new TreeSet<>(
    (a,b) -> a.getX() > b.getX() ? 1 :
       a.getX() < b.getX() ? -1 : 0 );
```

o bien

```
class Point implements Comparable<Point> {
@Override
public int compareTo(Point o) {
   ...
}
```

SortedSet: ejemplo

```
Set<Point> set = new TreeSet<>(
    (a,b) -> a.getX() > b.getX() ? 1 :
       a.getX() < b.getX() ? -1 : 0 );
set.add(new Point(1.5, 2.0));
set.add(new Point(2.0, 2.5));
set.add(new Point(2.5, 3.0));
set.add(new Point(2.5, 3.0));
set.add(new Point(2.0, 2.5));
System.out.println("Size: " + set.size());

Point aPoint = new Point(2.0, 2.5);
System.out.println("Contains return " + set.contains(aPoint));

showAll(set);
```

Interface `List`

```
public interface List<E> extends Collection<E> {

    default void sort(Comparator<? super E> c) {
        Object[] a = this.toArray();
        Arrays.sort(a, (Comparator) c);
        ListIterator<E> i = this.listIterator();
        for (Object e : a) {
                i.next();
                i.set((E) e);
        }
    }
```

Interface `List`

```
default void replaceAll(UnaryOperator<E> operator) {
    Objects.requireNonNull(operator);
    final ListIterator<E> li = this.listIterator();
    while (li.hasNext()) {
        li.set(operator.apply(li.next()));
    }
}
```

Interface `List`

También provee funcionalidad de arrays

```
E get(int index);
E set(int index, E element);
void add(int index, E element);
E remove(int index);
int indexOf(Object o);
int lastIndexOf(Object o);
ListIterator<E> listIterator();
ListIterator<E> listIterator(int index);
List<E> subList(int fromIndex, int toIndex);
```

`List`: ejemplo

```
List<Point> list = new ArrayList<>(50);
list.add(new Point(1.5, 2.0));
list.add(new Point(2.0, 2.5));
list.add(new Point(2.5, 3.0));
list.add(new Point(2.5, 3.0));
list.add(new Point(2.0, 2.5));

System.out.println("Size: " + list.size());
showAll(list);
aPoint = new Point(2.0, 2.5);
System.out.println("Contains returns " + list.contains(aPoint));
System.out.println(list.get(0));
System.out.println(list.get(1));
System.out.println(list.get(3));
System.out.println(list.get(5));
```

List

Algunas implementaciones de List

- `ArrayList`
- `LinkedList`
- `Vector`
- `Stack`
- `CopyOnWriteArrayList`

Ejemplos

```
List<String> alpha = new ArrayList<>();
alpha.add("a");    // retorna true o false
alpha.add("b");
alpha.add("c");
alpha.add("d");
```

```
List<String> alpha = new ArrayList<>();
String aux[] = new String[]{"a", "b", "c", "d"};
for(String s : aux)
  alpha.add(s);
```

```
List<String> alpha = Arrays.asList("a", "b", "c", "d");
```

Queue

Ofrece además la posibilidad de encolar y desencolar elementos

```
public interface Queue<E> extends Collection<E> {

        // Encolar un elemento
    boolean add(E e);              // Excepción si la cola está llena
    boolean offer(E e);   // Retorna false si la cola está llena

    // Desencolar un elemento
    E remove();           // Excepción si la cola está vacía
    E poll();             // null si la cola está vacía

    // Consultar el primer elemento
    E element();          // Excepción si está vacía
    E peek();             // null si está vacía
```

Algunas implementaciones: `LinkedList, PriorityQueue`

Ejemplo

```
Queue<Point> pq = new PriorityQueue<>((a,b) ->
       a.getX() > b.getX() ? 1 : a.getX() < b.getX() ? -1 : 0);
pq.offer(new Point(1.5, 2.0));
pq.offer(new Point(2.0, 2.5));
pq.offer(new Point(2.5, 3.0));
pq.offer(new Point(3.5, 3.0));
pq.offer(new Point(2.0, 2.5));

System.out.println("Size: " + pq.size());
showAll(pq);
System.out.println(pq.poll());
System.out.println(pq.poll());
System.out.println(pq.poll());
System.out.println(pq.poll());
System.out.println(pq.poll());
```

Deque

Funcionalidad de pila y cola

```
public interface Deque<E> extends Queue<E> {

    void push(E e);
    E pop();

    ...
}
```

Algunas implementaciones: `LinkedList, ArrayDeque`

Collections

```
public class Collections {
   private Collections() {   }
   public static <T extends Comparable<? super T>> void sort(List<T> list) {
      list.sort(null);
   }

   public static <T> void sort(List<T> list, Comparator<? super T> c) {
         list.sort(c);
   }

   public static void reverse(List<?> list) { ... }

   public static void shuffle(List<?> list) { ... }
```

Collections

```
(cont)
   public static <T>
   int binarySearch(List<? extends Comparable<? super T>> list, T key) {
      if (list instanceof RandomAccess ||
        list.size()<BINARYSEARCH_THRESHOLD)
          return Collections.indexedBinarySearch(list, key);
      else
          return Collections.iteratorBinarySearch(list, key);
   }

   ...
```

Ejemplo

```
List<Point> list = new ArrayList<>(50);
...

// Orden natural
Collections.sort(list);

// Orden arbitrario
Collections.sort(list, (a,b) -> a.getY() > b.getY() ? 1 :
    a.getY() < b.getY() ? -1 : 0);

// Mezclamos
Collections.shuffle(list);
```

Map

```
public interface Map<K,V> {
   int size();
   boolean isEmpty();
   boolean containsKey(Object key);
   boolean containsValue(Object value);
   V get(Object key);
   V put(K key, V value);
   V remove(Object key);
   void putAll(Map<? extends K, ? extends V> m);
   void clear();
   Set<K> keySet();
   Collection<V> values();
   ...
}
```

`Map`: algunas implementaciones

- `HashMap:` acepta un null como clave y valor
- `HashTable:` no acepta un null como clave ni valor
- `EnumMap:` la clave es un Enum
- `LinkedHashMap` extiende `HashMap`
- `Properties:` clave y valor String
- `TreeMap:` internamente usa un red-black tree y mantiene los datos ordenados por el orden natural o un `Comparator`

Ejemplos

```
class Point implements Comparable<Point> {
   @Override
   public boolean equals(Object o) {
       if (this == o) return true;
       if (o == null || getClass() != o.getClass()) return false;
       Point point = (Point) o;
       return Double.compare(point.getX(), getX()) == 0
             && Double.compare(point.getY(), getY()) == 0;
   }

   @Override
   public int hashCode() {
       return Objects.hash(getX());
   }
```

Ejemplos

```
Map<Point, String> map = new HashMap<>(10);
map.put(new Point(1.5, 2.0), "1.5, 2.0");
map.put(new Point(1.5, 1.5), "1.5, 1.5");
map.put(new Point(1.5, 2.0), "1.5, 2.0 bis");
map.put(new Point(2.5, 2.0), "2.5, 2.0");
map.put(new Point(2.5, 1.5), "2.5, 1.5");

System.out.println(map.size());
System.out.println(map.containsKey(new Point(1.5, 2.5)));

Point point = new Point(3.5, 4.0);
map.put(point, "3.5, 4.0");
point.setY(4.5);
System.out.println(map.get(point));
point.setX(4.5);
System.out.println(map.get(point));
```

Ejercicio

Se tiene una lista de productos vendidos (cantidad e importe) por fecha. Se desea obtener una colección ordenada por producto sumarizando el total de unidades y monto vendido.

```
class Sale {
    String description;
    Date date;
    int qty;
    double amount;

    public Sale(String description, Date date, int qty, double amount) {
        this.description = description;
        this.date = date;
        this.qty = qty;
        this.amount = amount;
    }
}
```