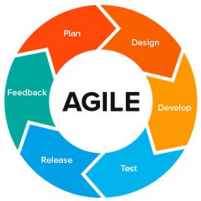
Skill Factory 2020 (2nd edition)

Java

Kotlin

git



{ REST:API }

JUnit 5

RxJava

MySQL

mongoDB.

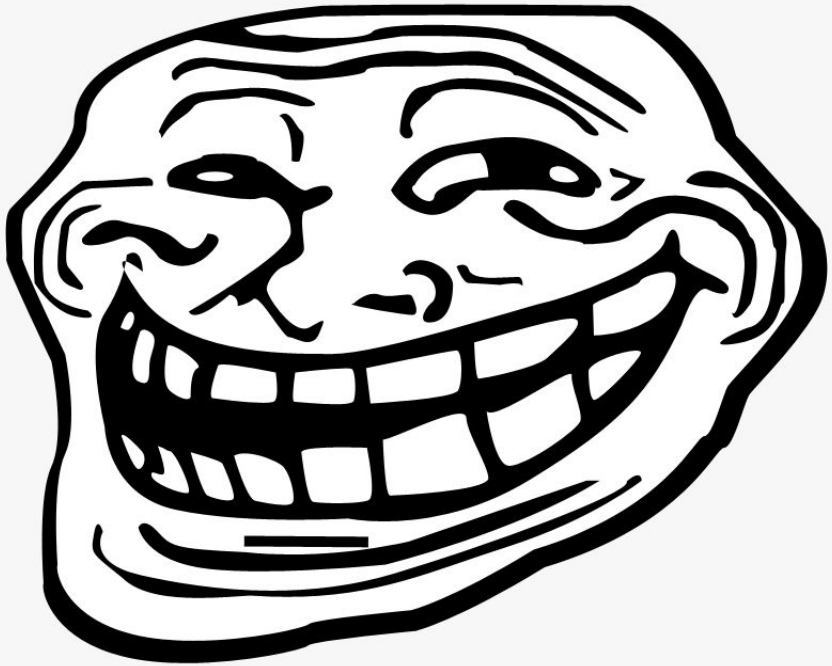
Maven

mockito

Algunas buenas prácticas en la programación

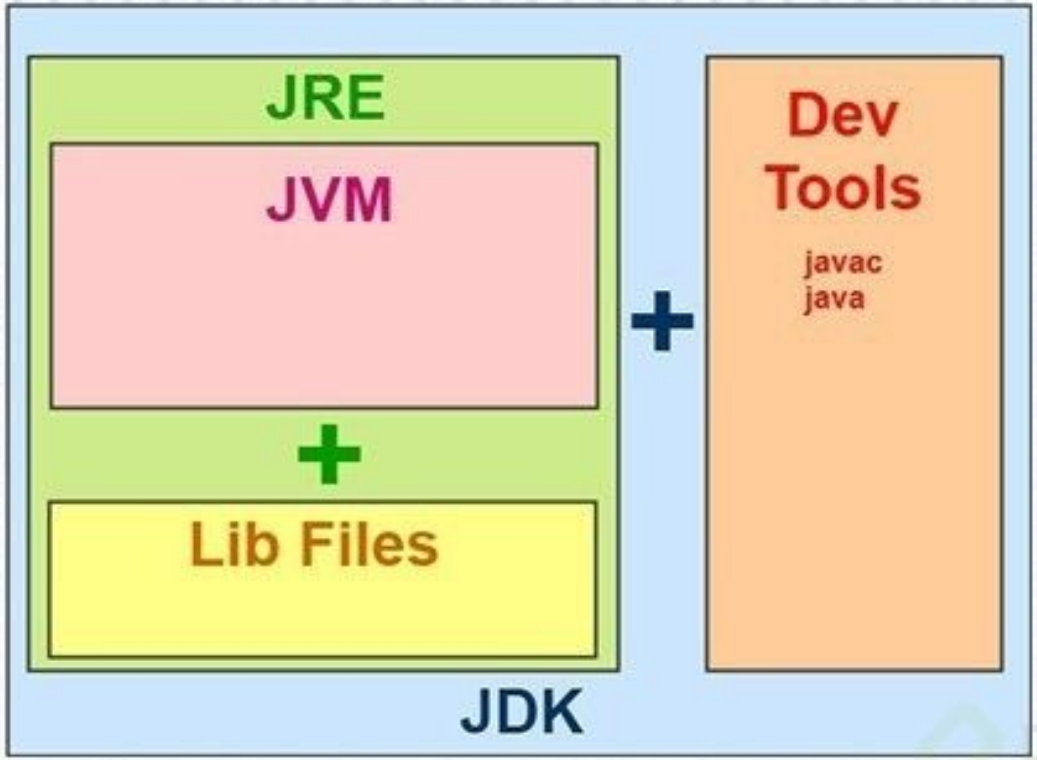
1) Usar indentación consistente (ya sea espacios o tabs, seguir siempre el mismo patrón).
2) Nombres de clases, variables, interfaces, y métodos que sean descriptivos pero concisos (NO: int x ( ) {.....}, y TAMPOCO: public void methodThatMakesAThingAndThenReturnsOther( ) {.....} .)
3) Principio DRY (don't repeat yourself): Un bloque de código no debería repetirse una y otra vez.
4) Tener una estructura de paquetes adecuada y coherente.
5) Realizar comentarios solo en caso de ser necesario y hacer que el código se explique por sí mismo (hacerlo legible).
6) Seguir convenciones según el lenguaje en el cual programemos.
7) Evitar niveles de anidamiento de código demasiado profundos.
8) Tratar de evitar lineas de codigo demasiado largas.
9) Un metodo deberia tener una sola responsabilidad.

10) Escribir el código en Inglés.
11) Tener un buen porcentaje de coverage en unit tests.
12) Manejarse con un sistema de control de versiones (ej. GIT).
13) Tener en cuenta los casos en que nuestro código puede fallar (programación defensiva).
14) Saber que todo lenguaje y tecnología es una herramienta para resolver problemas, deberíamos evaluar qué herramienta nos conviene usar para cada situación. (“Si tu única herramienta es un martillo, te sorprenderá como todo comienza a parecerse a un clavo”).
15) No hay un SIEMPRE ni un NUNCA, todo DEPENDE (pero hacer las cosas de la forma correcta es seguir las buenas prácticas (depende, jeje)).



Introduction to Object Oriented Programming through Java

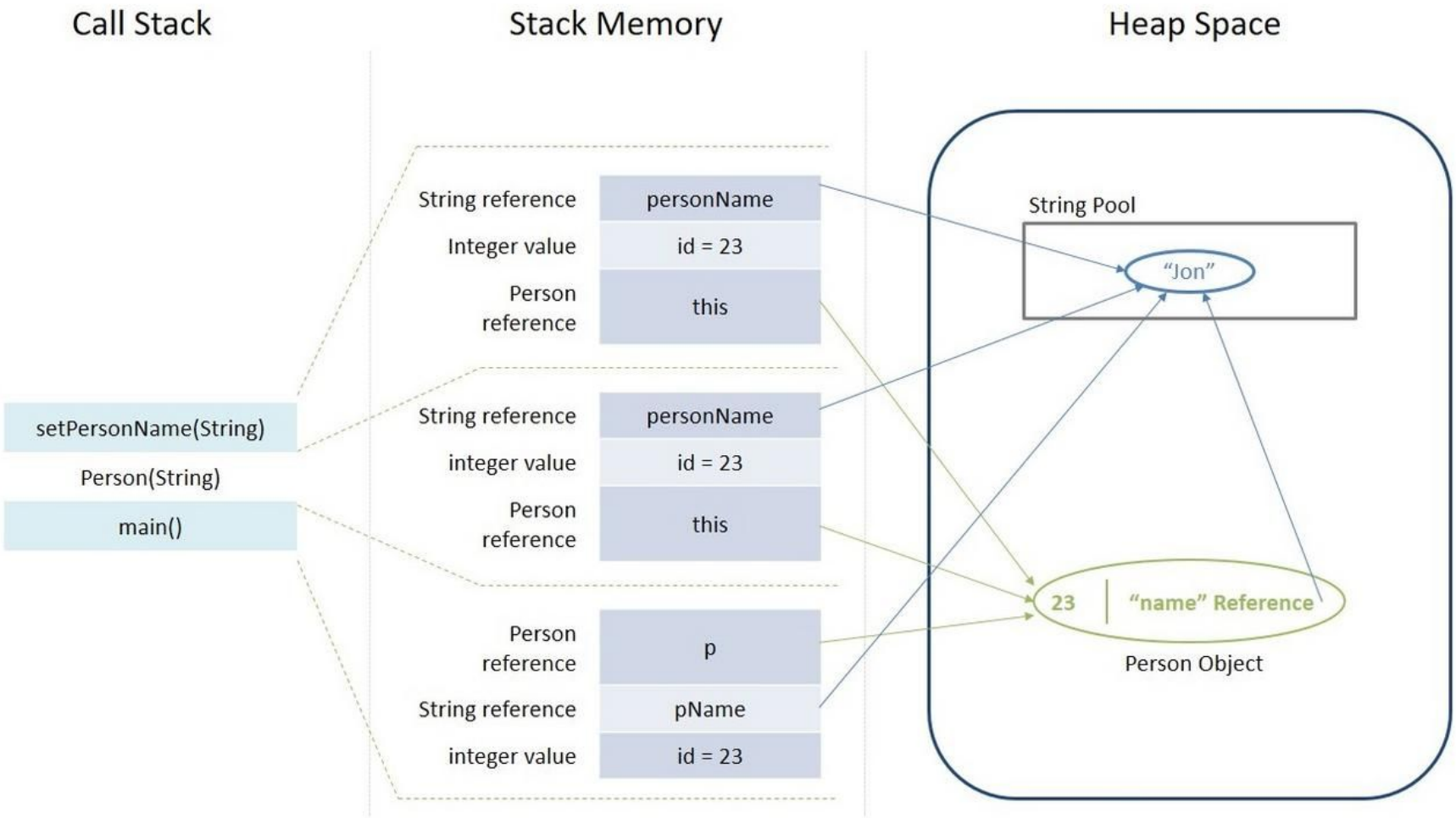
JDK vs JRE vs JVM



(ya sé qué están pensando qué esto es aburrido, cuidado, puedo leer sus mentes)

REPASO

Variables: Son usadas para almacenar datos. En Java al ser un lenguaje tipado, las hay de distintos tipos.
Operadores: Construcciones que se comportan como funciones. Los hay de asignación, lógicos, semánticos. (ej: =, ==, &&, etc.)
Condicionales: Evaluan una condición booleana para alterar el flujo de la ejecución de un programa. (ej. if, else, ? :).
Métodos: Porción de código definido dentro de una clase. Se encargan de definir el comportamiento de un objeto.
Clases: Son “fábricas” de objetos, o plantillas en base a las cuales vamos a crear objetos. En una clase definimos atributos y métodos para una entidad que va a ser creada a partir de esta.
Objetos: Son instancias de una clase.
Paquetes: Carpetas dentro de un proyecto en las cuales se agrupan clases o interfaces relacionadas según lo necesitemos (hay convenciones para esto).

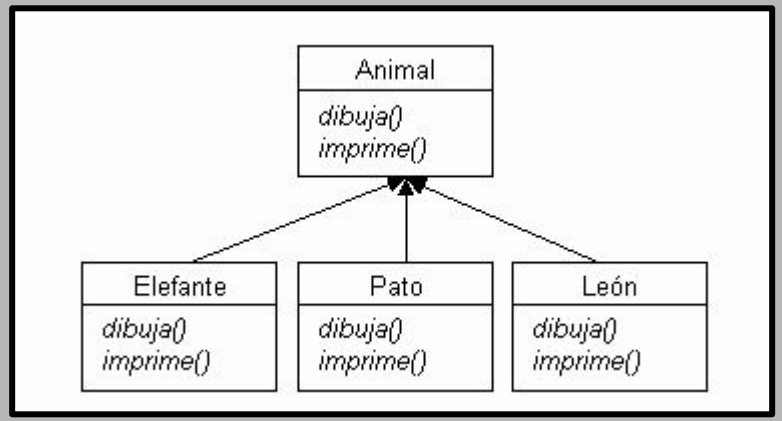

Call Stack
Stack Memory
Heap Space
setPersonName(String)
Person(String)
main()
String reference
personName
Integer value
id = 23
Person reference
this
String reference
personName
integer value
id = 23
Person reference
this
Person reference
p
String reference
pName
integer value
id = 23
String Pool
"Jon"
23
"name" Reference
Person Object

Pilares de la POO

Herencia
Polimorfismo
Abstracción
Encapsulamiento
¿Modularidad?

Interfaces (por ahora serán solo esto, pero hay más):

- Conjunto de métodos públicos y abstractos.
- Las firmas de los métodos nos dicen solo“qué” hace.
- Son implementadas por clases.
- “Contrato” para que la clase que la implementa deba definir TODOS los métodos abstractos.



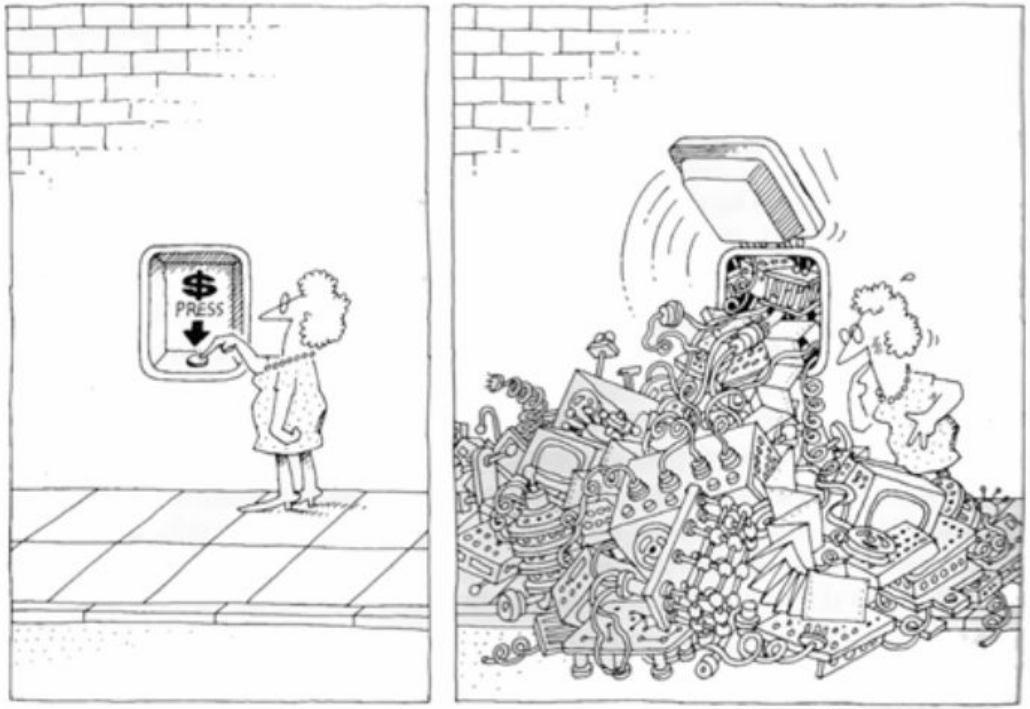
Ejemplo

```java
public interface PrestarAtencion {
    void escuchar(String palabra);
    String hablar();
    void hacerSilencio()
}

public class AlumnoDelSkillFactory implements PrestarAtencion {

    //constructores, atributos, getters y setters omitidos

    @override
    public void escuchar(String palabra) {
        //logica de metodo aqui
    }
    @override
    public String hablar() {
        //logica del metodo aqui
    }
    @override
    public void hacerSilencio() {
        //logica del metodo aqui
    }
}
```

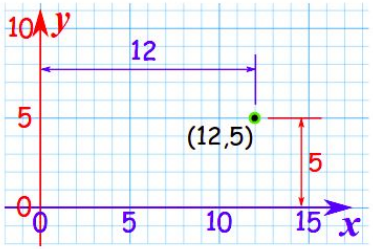


The task of the software development team
is to engineer the illusion of simplicity.

Solo un poco de contexto para el ejemplo sobre Abstracción de datos

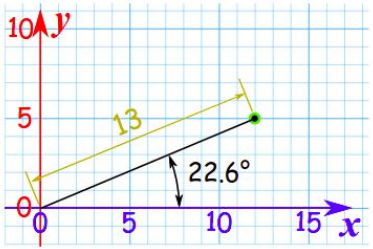
Cartesian Coordinates

Using Cartesian Coordinates we mark a point by **how far along** and **how far up** it is:



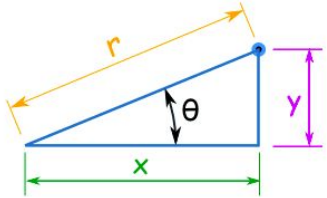
Polar Coordinates

Using Polar Coordinates we mark a point by **how far away**, and **what angle** it is:



Converting

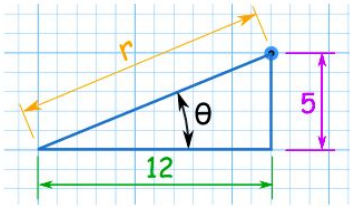
To convert from one to the other we will use this triangle:



To Convert from Cartesian to Polar

When we know a point in Cartesian Coordinates (x,y) and we want it in Polar Coordinates (r,θ) we **solve a right triangle with two known sides**.

Example: What is (12,5) in Polar Coordinates?



```java
package com.avalith.models;

public class Point {
    public Double x;
    public Double y;
}
```

```java
package com.avalith.models;

public class Point2 {
    private Double x;
    private Double y;

    public Double getX() {
        return x;
    }

    public void setX(Double x) {
        this.x = x;
    }

    public Double getY() {
        return y;
    }

    public void setY(Double y) {
        this.y = y;
    }
}
```

```java
package com.avalith.interfaces;

public interface IPoint {
    Double getX();
    Double getY();
    void setCartesian(Double x, Double y);
    Double getR();
    Double getTheta();
    void setPolar(Double r, Double theta);
}
```

Para lograr abstracción deberíamos preguntarnos cuál es la ***responsabilidad*** que debería tener un objeto

NO OCULTAN INFORMACIÓN NI IMPLEMENTACIÓN, NO HAY ABSTRACCIÓN NI ENCAPSULAMIENTO

```java
public interface IVehicle {
    Double getTankCapacity();
    Double getLitersOfGasolineInTank();
}
```

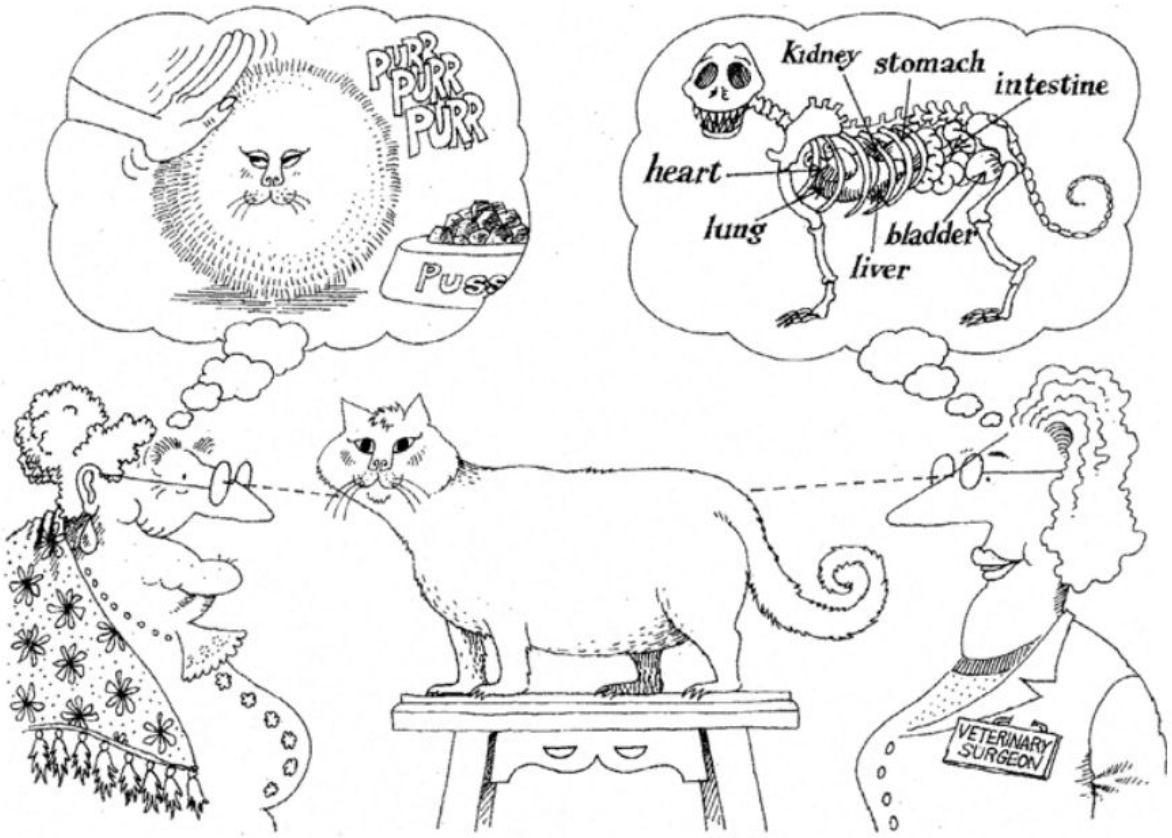
```java
public interface IVehicleAbstraction {
    Double getPercentageFuelRemaining();
}
```

Usa términos concretos para indicar el nivel de combustible (NO HAY ABSTRACCIÓN).
En este caso estamos seguros que se trata de métodos de acceso a variables (conocemos su estructura interna)

Utiliza la ***abstracción*** del porcentaje para mostrar lo que se necesita. Oculta su implementación e información interna (ENCAPSULAMIENTO).

En resumen:

- *Representar estructuras de datos, no sus detalles.*
- *Reforzar política de acceso.*
- *No queremos mostrar los detalles de los datos, sino expresarlos en términos abstractos.*
- *La abstracción y modularidad no se consiguen simplemente usando interfaces o métodos de acceso y mutación, hay que meditar la forma óptima de representar un objeto y los datos que contiene.*
- *La peor opción es agregar getters y setters a ciegas.*



Abstraction focuses on the essential characteristics of some object, relative to the perspective of the viewer.

Collections

AVALITH.

¿Qué es una colección?

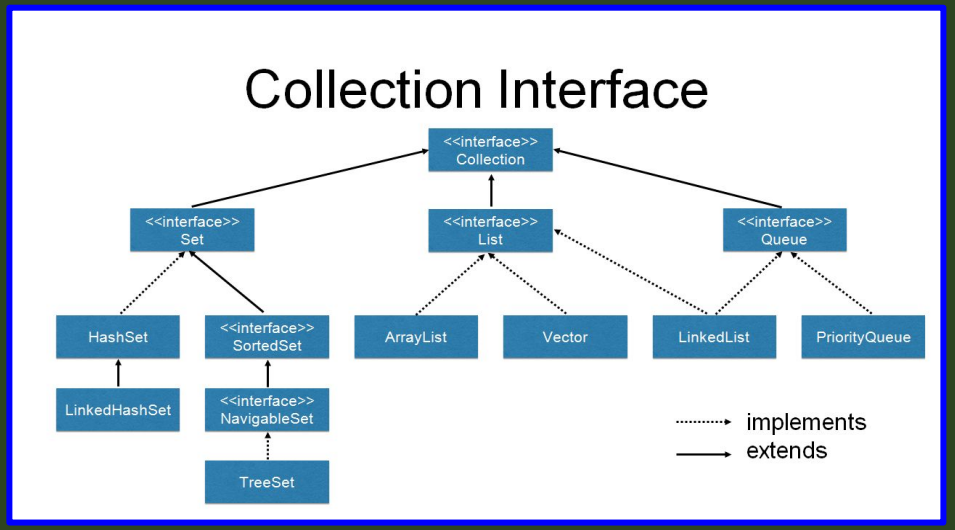
colección (kolek'θjon)

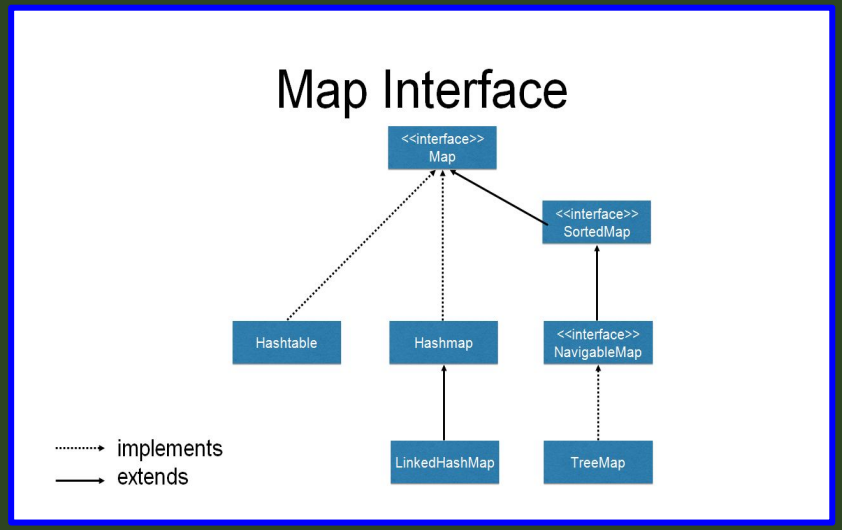
sustantivo femenino

1. conjunto de cosas de la misma clase reunidas y clasificadas *una colección de monedas*
2. *fashion* conjunto de modelos creados por un diseñador para una temporada *Se presentó la colección primavera-verano en un desfile.*
3. gran número de personas o cosas *Este libro guarda una colección de sinsentidos.*

Bueno si... pero me dijeron qué aca ibamos a programar, entonces, ¿ que sería una colección ?

Una colección en Java, es objeto que representa un conjunto o secuencia de objetos agrupados según un tipo de dato.
Para poder trabajar con grupos de elementos, Java nos brinda distintas colecciones, las cuales forman parte del **Collections Framework**.





Collections Framework:

Provee lo necesario para almacenar y manipular grupos de objetos.

- Define clases e interfaces para representar grupos de objetos como una unidad.
- Mediante el uso de Collections podemos insertar, realizar búsquedas, ordenamientos, eliminaciones en distintas estructuras de datos.
- Típicamente en una colección representamos datos que forman un grupo “natural”.

Es una arquitectura unificada para representar y manipular distintas colecciones.

Está compuesto de :

- *Interfaces:* Tipos abstractos que representan cada tipo de colección.
- *Implementaciones:* Son los tipos concretos que implementamos. Son estructuras reutilizables.
- *Algoritmos:* Conjunto de métodos para realizar operaciones sobre los objetos contenidos en una implementación.

Colecciones

- Contenedores que son usados para almacenar, manipular u obtener datos.
- Implementación de interfaces (distintos comportamientos).
- Brindan versatilidad al trabajar con muchos objetos.
- Primero deberíamos evaluar qué uso van a tener. Por ejemplo, realizar más búsquedas qué inserciones (o viceversa), no almacenar elementos duplicados, usar métodos de ordenamiento ya implementados, etc.

Collection class	Thread-safe alternative	Your data				Operations on your collections						
		Individual elements	Key-value pairs	Duplicate element support	Primitive support	Order of iteration			Performant 'contains' check	Random access		
						FIFO	Sorted	LIFO		By key	By value	By index
HashMap	ConcurrentHashMap	✗	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗
HashBiMap (Guava)	Maps.synchronizedBiMap (new HashBiMap())	✗	✓	✗	✗	✗	✗	✗	✓	✓	✓	✗
ArrayListMultimap (Guava)	Maps.synchronizedMultiMap (new ArrayListMultimap())	✗	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗
LinkedHashMap	Collections.synchronizedMap (new LinkedHashMap())	✗	✓	✗	✗	✓	✗	✗	✓	✓	✗	✗
TreeMap	ConcurrentSkipListMap	✗	✓	✗	✗	✗	✓	✗	✓*	✓*	✗	✗
Int2IntMap (Fastutil)		✗	✓	✗	✓	✗	✗	✗	✓	✓	✗	✓
ArrayList	CopyOnWriteArrayList	✓	✗	✓	✗	✓	✗	✓	✗	✗	✗	✓
HashSet	Collections.newSetFromMap (new ConcurrentHashMap<>())	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗
IntArrayList (Fastutil)		✓	✗	✓	✓	✓	✗	✓	✗	✗	✗	✓
PriorityQueue	PriorityBlockingQueue	✓	✗	✓	✗	✗	✓**	✗	✗	✗	✗	✗
ArrayDeque	ArrayBlockingQueue	✓	✗	✓	✗	✓**	✗	✓**	✗	✗	✗	✗

Implementaciones de Interface List <E>:

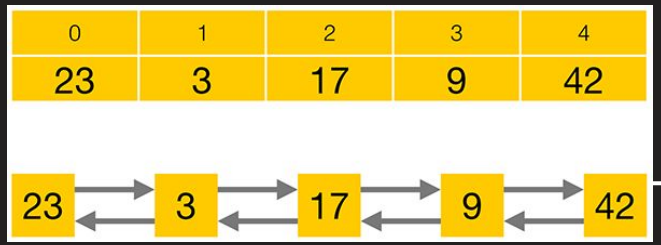
- Redimensionable

- Puede contener elementos repetidos

ArrayList: *- Elementos indexados.*
- Rápida para búsquedas (acceso arbitrario).

LinkedList: *- Lista doblemente enlazada.*
- Sus elementos no están indexados.
- Rápida para agregar o eliminar al final de la lista.

ArrayList ---->



LinkedList ---->

Implementaciones de Interface Set <E>:

* ninguna permite elementos repetidos.

HashSet: - Elementos NO ordenados, ni indexados.
- Rápida para acceso directo a un elemento.

TreeSet: - Elementos ordenados, pero NO indexados.
- Rápida para búsquedas mediante iteraciones.

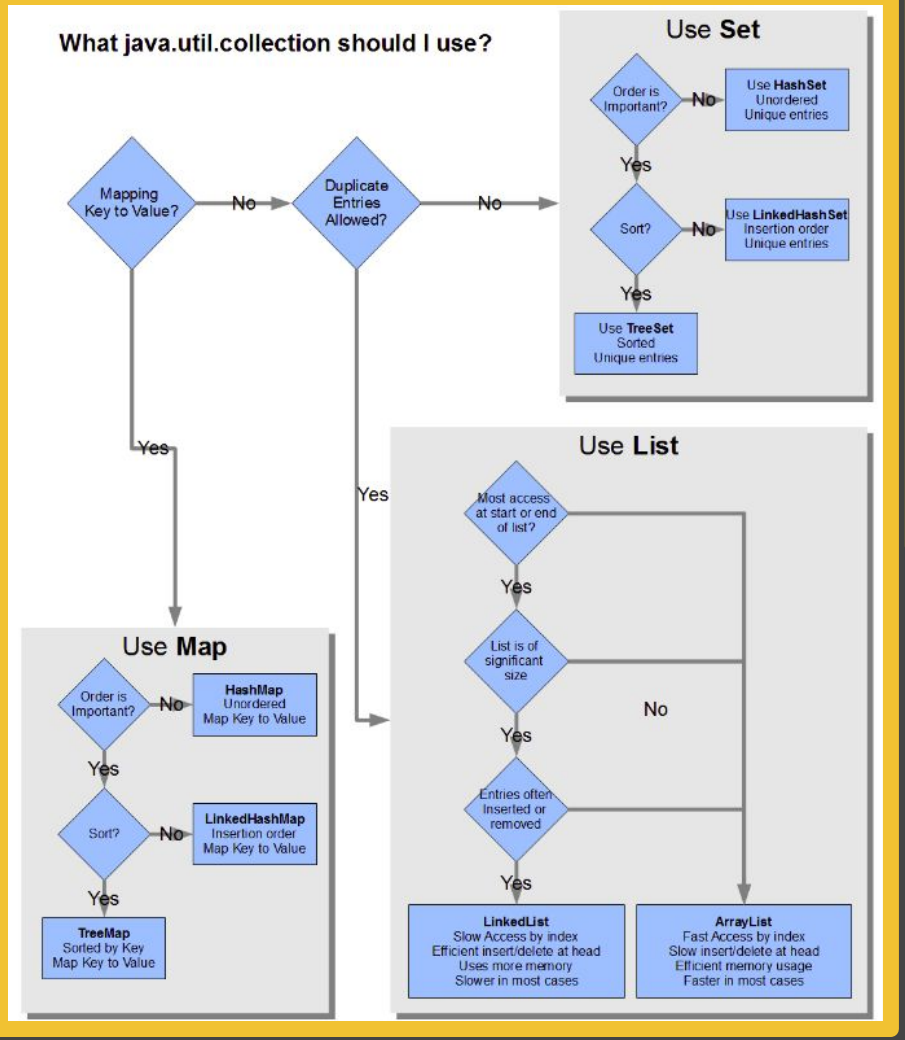
LinkedHashSet: - Permite “recordar” el orden en el cual fueron insertados los objetos.

Implementaciones de Interface Map <K,V>:

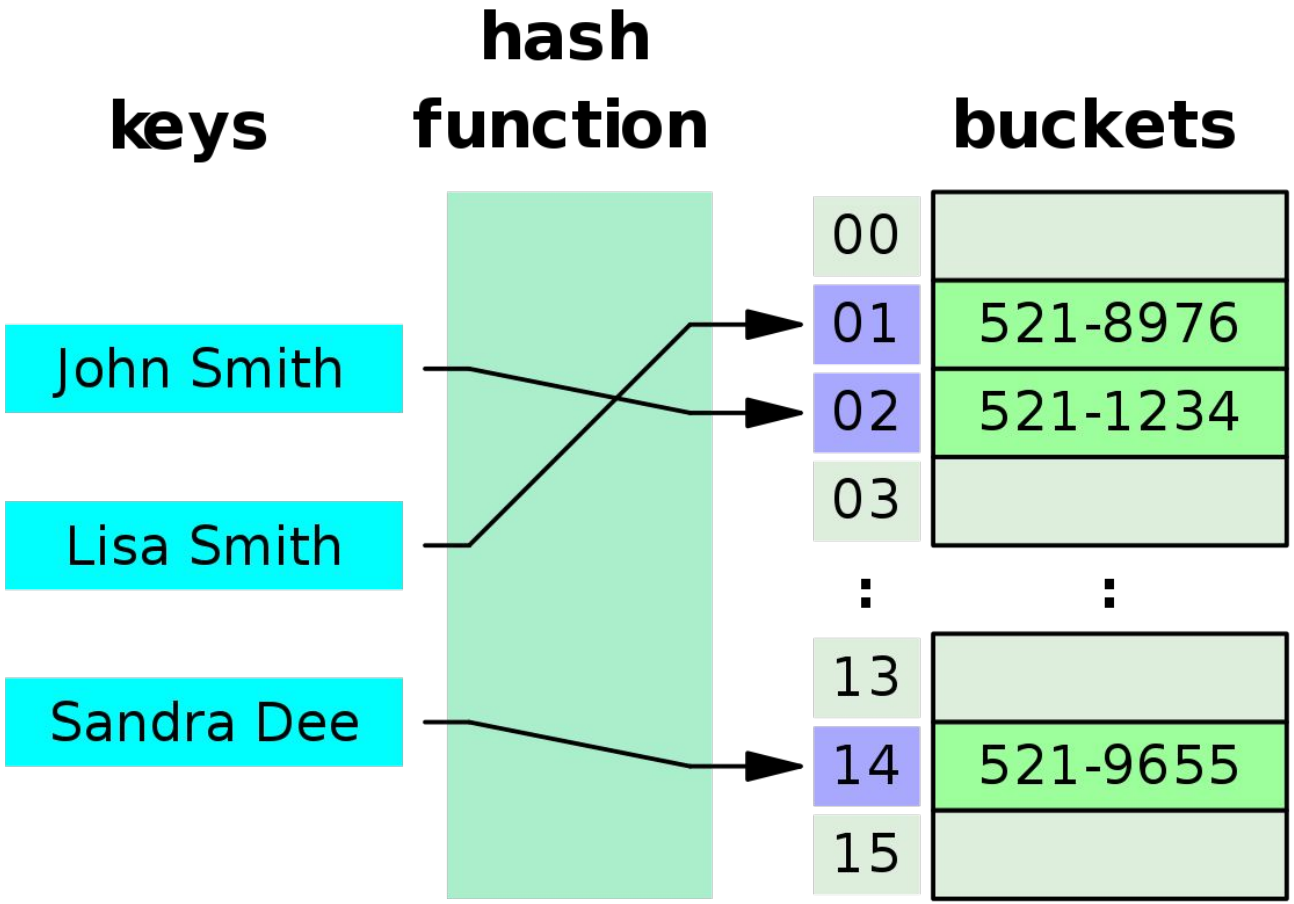
*Estructuras que mapean una clave con un valor

HashMap:
- Los elementos no están ordenados.
- No puede contener claves duplicadas, pero si valores duplicados.
- Permite clave (una sola obviamente) y valores null.
- MUY rápida para acceso directo a un elemento.

TreeMap:
- Claves ordenadas.
- No puede contener clave null pero si valores null.
- Rápida para búsqueda mediante iteración de claves.



Al analizar un problema vamos a necesitar de cierta funcionalidad para manipular los datos. Si conocemos las particularidades de cada coleccion, vamos a poder decidir cuál nos conviene.


keys
hash function
buckets
John Smith
Lisa Smith
Sandra Dee
00
01
02
03
13
14
15
521-8976
521-1234
521-9655

Bueno, todo muy bonito pero... ¡quiero codear!

Declaración de un Arraylist de Integer:


```
ArrayList<Integer> myArrayList = new ArrayList<>();
```

Las colecciones deben declararse del tipo estático de su interfaz para poder hacer uso del POLIMORFISMO.

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

List vs. ArrayList

Interfaz

Define comportamiento

Tiene varias implementaciones

Clase concreta

Implementa funcionalidad

Implementa List
(todos los métodos)

Cuando usamos una colección, los métodos que se exponen son los de la interfaz, y se ejecutan los definidos en la clase concreta.

Algunas operaciones específicas de Map:

```java
Map<String,String> mySuperMap = new HashMap<String,String>();

// podemos omitir los tipos concretos de la implementacion...
Map<String,String> mySuperMap = new HashMap<>();

mySuperMap = new TreeMap<>(); // cambiamos de implementacion

mySuperMap.put("avalith", "skill factory"); // agregamos clave y su valor

String avalith = mySuperMap.put("12345", "67890"); // agregamos clave y su valor retornandolo

String otherAvalith = mySuperMap.get("avalith"); // retorna valor mapeado a la clave

Integer mapSize = mySuperMap.size() // retorna tamaño del map (int)

Boolean isSomeoneThere = mySuperMap.isEmpty(); // map vacio retorna true, sino false
```

Formas de iterar un map:

```java
// bucle ForEach sobre el entry set
Map<Integer, Integer> map = new HashMap<Integer, Integer>();
for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
    System.out.println("Key = " + entry.getKey() + ", Value = " + entry.getValue());
}

// solo sobre las claves
for (Integer key : map.keySet()) {
    System.out.println("Key = " + key);
}

// solo los valores
for (Integer value : map.values()) {
    System.out.println("Value = " + value);
}
```

```java
// iterator
Iterator<Map.Entry<Integer, Integer>> entries = map.entrySet().iterator();
while (entries.hasNext()) {
    Map.Entry<Integer, Integer> entry = entries.next();
    System.out.println("Key = " + entry.getKey() + ", Value = " + entry.getValue());
}

// iterar claves y buscar valores
for (Integer key : map.keySet()) {
    Integer value = map.get(key);
    System.out.println("Key = " + key + ", Value = " + value);
}

// metodo forEach (esta -> magia la veremos luego)
items.forEach((k,v) -> System.out.println("Item : " + k + " Count : " + v));
```

```java
// streams (proximamente en cartelera)
map.entrySet().stream()
.forEach(e -> System.out.println("Key = " + e.getKey() + ", Value = " + e.getValue));
```

Material de lectura y guia de practica

Métodos comunes a todos los objetos:

Si bien Object es una clase concreta, está diseñada para su extensión. Todos sus métodos que no son final (equals, hashcode, tostring, clone y finalize) tienen contratos generales explícitos ya que están diseñados para ser sobreescritos. Al cumplir con estos contratos podemos asegurar que otras clases que dependen de los contratos (ej. HashMap, HashSet) funcionen correctamente en conjunto con la clase.

Obedecer contrato general al sobreescribir equals( ):

Parece sencillo sobreescribir este método pero hay muchas formas de hacerlo mal, y sus consecuencias son malas.
La forma más fácil de evitar problemas es no sobrescribir el método, por lo cual cada instancia de la clase seria solo igual a si misma.
Esto sería lo correcto en situaciones muy puntuales como:

1) Cada instancia es única: Podría ser aplicable a la clase Thread que representa una entidad más que valores.
2) Cuando no hay necesidad de proveer igualdad “lógica”.
3) Una superclase ya ha sobreescrito el equals y el comportamiento de la superclase es apropiado para la subclase. Ejemplo, Las implementaciones de Set heredan la implementación de AbstractSet o List de AbstractList.
4) La clase es privada o package-private y sabemos que el equals no va a ser invocado

Cuando es apropiado sobreescribir el equals()?

Cuando tenemos la necesidad de comparar valores. Un ejemplo para esto son las clases que representan valores (Integer, String etc), en las cuales el programador quiere saber si son iguales en cuanto a valores, no si hacen referencia al mismo objeto.
No solo satisface las necesidades del programador sino también permite que las instancias sean usadas como claves en maps o elementos dentro de un Set con un comportamiento predecible.
El método equals() implementa una relación de equivalencia y tiene las siguientes propiedades:

- Reflexivo: Para cualquier referencia no null (value x) , x.equals(x) debe retornar true.
- Simétrico: Para cualquier referencia no null (value x, value y), x.equals(y) debe retornar true si y sólo si y.equals(x) retorna true.
- Transitivo: Para referencias no null values a, b, c, a.equals(b) retorna true , y b.equals(c) retorna true entonces a.equals(c) debe retornar true.
- Consistente: Ante múltiples invocaciones de x.equals(y) debe retornar consistentemente true o false.

- Para cualquier referencia no null (value x) x.equals(null) debe retornar false.

Así como dice una frase que “ninguna clase es una isla”, frecuentemente instancias de una clase son pasadas a otras. Entonces muchas clases incluyendo las clases de colecciones requieren que los objetos que les son pasados obedezcan el contrato.

Receta para escribir un equals de calidad:

1) Usar el operador **==** para ver si el argumento es una referencia a this: Si es así retornar true. Esto es solo para optimización de performance que es necesaria cuando la comparación sea costosa.
2) Usar el operador **instanceof** para chequear si el argumento tiene el tipo correcto. De no ser así, retornar false. Típicamente, el tipo correcto el la clase en donde el método está definido. Ocasionalmente, es alguna interfaz implementada por la clase. Usar una interfaz si la clase que implementa la interfaz va a ser comparada con otras que implementen la interfaz.
3) Castear el argumento al tipo correcto. Como el casteo fue precedido por un instanceof, es seguro que va a ser exitoso.
4) Para cada campo “significativo” de la clase, chequear si ese campo del argumento matchea con el campo que corresponda en el objeto. Si todos estos pasos tienen éxito, retornar true, sino, retornar false. La mejor forma de hacer esto es comparando todos los casos y utilizar el operador **&&** entre caso y caso, así cuando uno sea false ya la condición se hace false entonces retorna sin necesidad de seguir evaluando. Si el caso 2 se da que son de tipo de una interfaz, debemos acceder a los campos del parámetro a través de métodos de la interfaz. Si el tipo es una clase, podemos acceder a ellos directamente, siempre dependiendo de su accesibilidad.
   Para tipos primitivos usar el operador == para comparar. Para referencias a objetos , usar el equals() recursivamente, para campos float usar el método estático Float.compare(float,float) al igual para double con Double.compare(double,double). Para arrays , aplicar esta guía para cada elemento.
   Algunas referencias a objetos podrían contener null legítimamente. Para evitar un NullPointerException usar el método estatico Objects.equals(Object,Object).
   Si bien los IDE como el intelliJ nos generan una versión óptima del equals(), es conveniente saber qué hacer cuando los casos son más particulares y en qué orden deseamos evaluar la igualdad.
   Hay que tener en cuenta que para mejor performance deberíamos comparar primero los campos que suponemos que van a tender a diferir. Por ejemplo si tenemos una clase Polígono y guardamos el área, sus bordes y sus vértices. Si dos polígonos tienen áreas distintas, no necesitamos comparar sus vértices y bordes.

Links para ampliar el tema (ver también contrato de hash code):
https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html
https://www.baeldung.com/java-equals-hashcode-contracts

An equals method constructed according to the previous recipe is shown in this simplistic PhoneNumber class:

```
// Class with a typical equals method
public final class PhoneNumber {
    private final short areaCode, prefix, lineNum;

    public PhoneNumber(int areaCode, int prefix, int lineNum) {
        this.areaCode = rangeCheck(areaCode, 999, "area code");
        this.prefix   = rangeCheck(prefix,   999, "prefix");
        this.lineNum  = rangeCheck(lineNum, 9999, "line num");
    }

    private static short rangeCheck(int val, int max, String arg) {
        if (val < 0 || val > max)
           throw new IllegalArgumentException(arg + ": " + val);
        return (short) val;
    }

    @Override public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof PhoneNumber))
            return false;
        PhoneNumber pn = (PhoneNumber)o;
        return pn.lineNum == lineNum && pn.prefix == prefix
                && pn.areaCode == areaCode;
    }
    ... // Remainder omitted
}
```

Here are a few final caveats:

- **Always override hashCode when you override equals** (Item 11).
- **Don't try to be too clever.** If you simply test fields for equality, it's not hard to adhere to the equals contract. If you are overly aggressive in searching for equivalence, it's easy to get into trouble. It is generally a bad idea to take any form of aliasing into account. For example, the File class shouldn't attempt to equate symbolic links referring to the same file. Thankfully, it doesn't.
- **Don't substitute another type for Object in the equals declaration.** It is not uncommon for a programmer to write an equals method that looks like this and then spend hours puzzling over why it doesn't work properly:

```
// Broken - parameter type must be Object!
public boolean equals(MyClass o) {
    ...
}
```

The problem is that this method does not *override* Object.equals, whose argument is of type Object, but *overloads* it instead (Item 52). It is unacceptable to provide such a "strongly typed" equals method even in addition to the normal one, because it can cause Override annotations in subclasses to generate false positives and provide a false sense of security.

Consistent use of the Override annotation, as illustrated throughout this item, will prevent you from making this mistake (Item 40). This equals method won't compile, and the error message will tell you exactly what is wrong:

```
// Still broken, but won't compile
@Override public boolean equals(MyClass o) {
    ...
}
```

Writing and testing equals (and hashCode) methods is tedious, and the resulting code is mundane. An excellent alternative to writing and testing these methods manually is to use Google's open source AutoValue framework, which automatically generates these methods for you, triggered by a single annotation on the class . In most cases, the methods generated by AutoValue are essentially identical to those you'd write yourself.

IDEs, too, have facilities to generate equals and hashCode methods, but the resulting source code is more verbose and less readable than code that uses AutoValue, does not track changes in the class automatically, and therefore requires testing. That said, having IDEs generate equals (and hashCode) methods is generally preferable to implementing them manually because IDEs do not make careless mistakes, and humans do.

Composición vs herencia:

https://www.thoughtworks.com/insights/blog/composition-vs-inheritance-how-choose
https://www.journaldev.com/12086/composition-vs-inheritance
https://icodemag.com/favor-composition-over-inheritance/

POO según Alan Kay (Smalltalk):

https://wiki.c2.com/?AlanKaysDefinitionOfObjectOriented
https://softwareengineering.stackexchange.com/questions/46592/so-what-did-alan-kay-really-mean-by-the-term-object-oriented

Java Collection documentación:

https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

Ejercicios Collections && Interfaces

1) Crear 3 clases. Cada una contiene una implementación (a elección) de las interfaces `List`, `Map` y `Set`. Cargarlas con objetos(`String`, `Integer` u objetos propios si gusta) y luego realizar las operaciones básicas en cada una (obtener, eliminar, leer e imprimir, iterar sobre la colección e imprimir todos sus elementos uno a uno (iterar de 2 formas diferentes al menos)). También ordenar e invertir el orden de los elementos en la/s colección/es que sea posible.
2) Se va a realizar una carrera entre distintas "especies". Crear las clases `Human`, `Dog` y `Robot`. Cada objeto debe tener una velocidad diferente como atributo (podría ser un random, pero en este caso debemos definir un rango para cada especie) . Las 3 clases deben implementar la interfaz `Sprintable`, la cual tiene un método qué es ***public Double run(Double distance)*** (el método recibe la distancia de la carrera (en kilómetros) que van a correr y devuelve el tiempo que tardó dicho competidor, calculado con el "coeficiente" de velocidad de cada objeto) . Debemos crear una clase que se llame `Marathon` en la cual debemos cargar los objetos en una `List` o `Set` y luego debemos iterar sobre dicha colección e ir imprimiendo el tiempo que realiza cada competidor.
3) Escribir un método `removeEvenLength()` que reciba un `ArrayList` de `String` y elimine todos los `String` de longitud par.
4) Escribir un método que reciba n cantidad de `Integer` (no un número fijo) y una `List`, los ordene dentro de un `ArrayList` e imprimir por pantalla.
   Luego convertir este Arraylist en LinkedList y repetir el proceso.
5) Llenar un `ArrayList` con objetos `Car` (`String model, Integer prize`), cargar algunos varias veces, luego eliminar los repetidos.
6) Crear una clase `ClubMember` con los siguientes atributos:

```
name: String
id: UUID
phone: String
age: Integer
```

   redefinir `equals()`, `hashCode()` y `toString()`

- Los socios deben tener la posibilidad de ser comparados por su nombre
- Crear una clase `Club` en donde los `ClubMember` van a tener la posibilidad de hacerse socios, votar y renunciar.
- Entre los mismos socios se votan entre ellos para elegir presidente, la unica condicion es que no se pueden votar a ellos mismos.
- Por reglamento del club tampoco se puede votar 2 veces, si esto ocurre se descalifica al socio quitándole la posibilidad de ser votado por el resto.
- Cada voto debe ir acompañado del id del socio que votó, si no ingresa su id, a este socio que votó, se le resta un voto, siempre y cuando tenga votos, sino, no se le resta.
- El club mes a mes va publicando una lista actualizada de sus socios, ordenada por orden alfabético (investigar cómo ayudar a la colección a comparar objetos).
- Si desea, puede agregar más funcionalidades y reglas.