



UD 4 - TIPOS DE DATOS AVANZADOS

UD 4 - TIPOS DE DATOS AVANZADOS

Índice

1. El casting de referencias a objetos
2. Colecciones de datos
 1. Listas
 2. Conjuntos. Clases Wrapper
 3. Diccionarios
3. Genéricos
 1. Necesidad. Uso de colecciones con genéricos
 2. Otros usos de los genéricos

Anexo I. Fechas y horas.

UD 4 - TIPOS AVANZADOS DE DATOS

1 - El casting de referencias a objetos

► 1 – El casting de referencias a objetos

► ¿Recordáis cuando estudiamos las conversiones entre tipos primitivos? Hablábamos de:

► Las **conversiones automáticas** cuando no había pérdida de información (de int a double, por ejemplo)

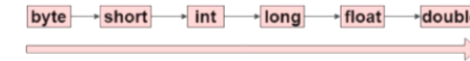
► Las **conversiones explícitas** cuando sí había pérdida de información (de double a int) y entonces el desarrollador debía indicarlo explícitamente en el código.

► Pues después de estudiar la herencia y el polimorfismo también podemos hacer “conversiones” entre las referencias a los objetos.

UD 2 - ESTRUCTURAS DE CONTROL Y OTROS ELEM. 1 - Cuarta vuelta: otros elementos

Dado que estas conversiones se realizan **SIN PÉRDIDA DE INFORMACIÓN** se realizan de forma **automática** por la máquina virtual java.

Automatic Type Conversion (Widening - implicit)



Un byte “cabe” en un short, y este en un int...

Widening es “ensanchando”

Sin embargo, en ocasiones nos puede hacer falta una conversión en “sentido contrario”, es decir, una conversión **CON PÉRDIDA DE INFORMACIÓN**. En este caso, el programador tiene que indicar explícitamente en el código que no se trata de un error y que es eso lo que desea hacer realmente.

Narrowing (explicit)



Un double “no cabe” en un float, y este tampoco cabe en un long...

Narrowing es “estrechando”

UD 4 - TIPOS AVANZADOS DE DATOS

1 - Casting de referencias a objetos

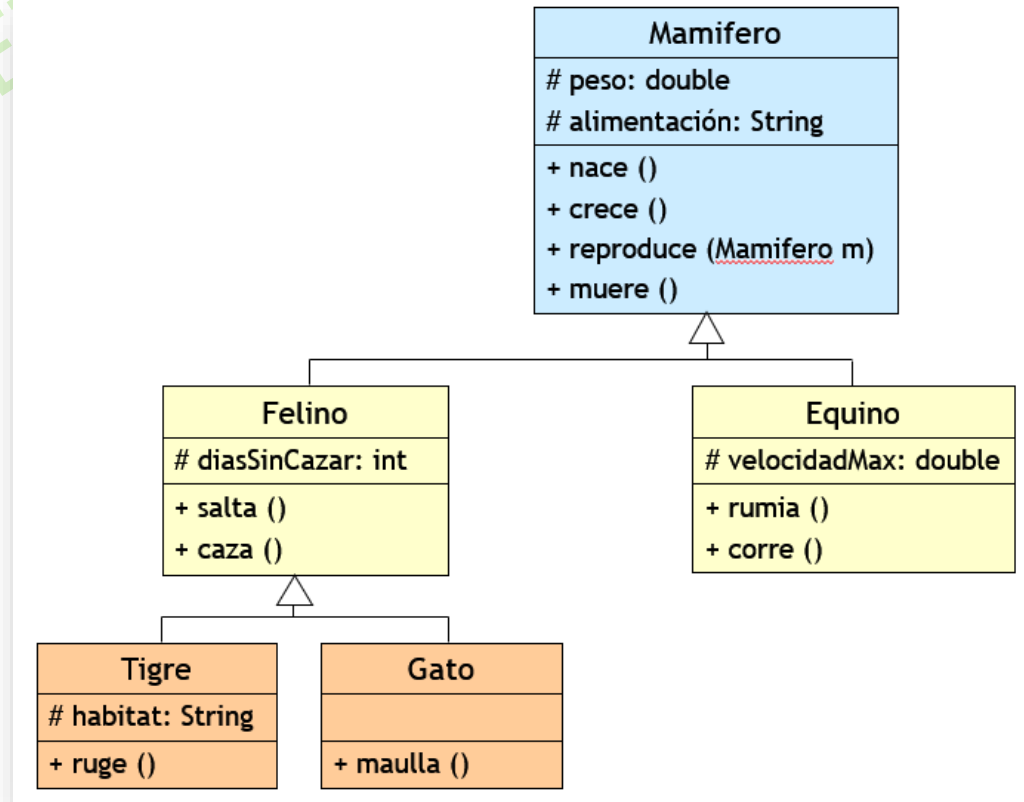
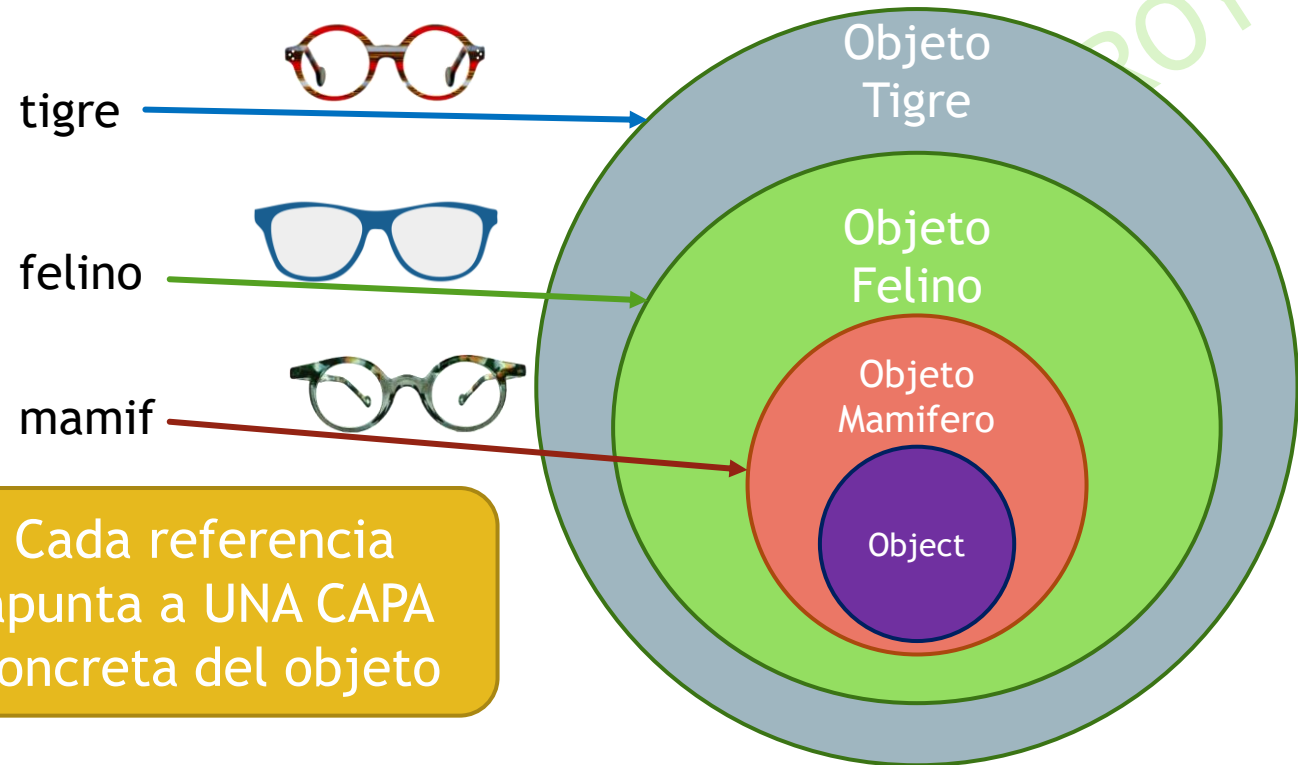
- Para verlo, estudiamos el siguiente trozo de código:

```
Tigre tigre = new Tigre();
```

```
Felino felino = tigre;
```

```
Mamifero mamif = tigre;
```

- Si miramos en la memoria lo que está ocurriendo es:



UD 4 - TIPOS AVANZADOS DE DATOS

1 - Casting de referencias a objetos

- **IMPORTANTE:** al objeto no le estamos cambiando nada. Lo que hacemos es “convertir” referencias para que apunten a las distintas capas internas que conforman el objeto tigre.
- Estas referencias se comportan del siguiente modo:

```
Tigre tigre = new Tigre();
```

```
Felino felino = tigre;
```

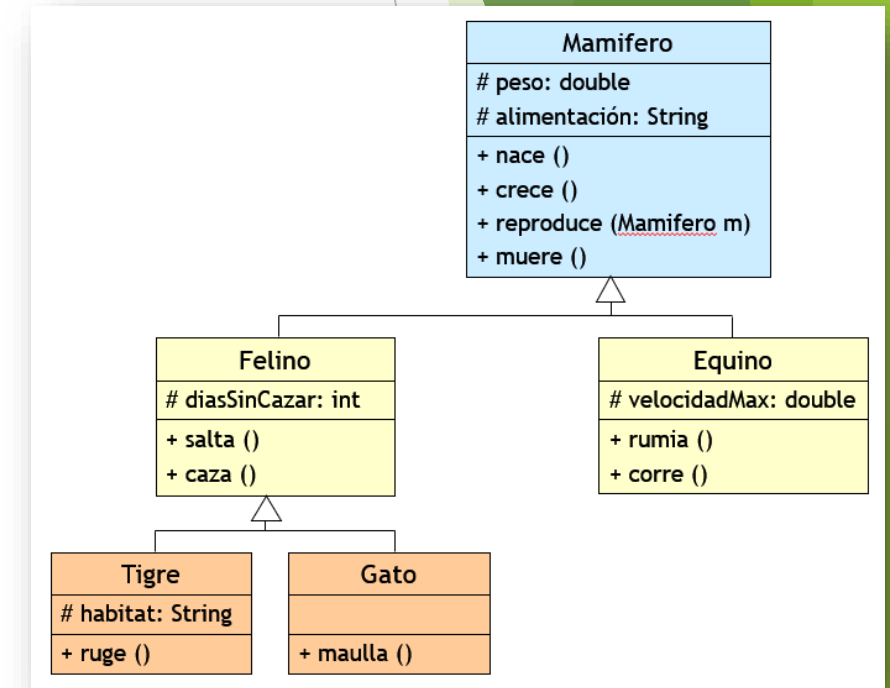
```
felino.caza(); // Bien, todos los felinos cazan
```

```
felino.salta(); // Bien, los felinos saltan
```

```
felino.ruge(); // Error de compilación, NO todos los felinos rugen
```

```
tigre.ruge(); // Bien, los tigres sí rugen
```

- Cuando hacemos que una referencia apunte a una determinada capa del objeto, solo se podrán usar las propiedades y métodos que estén en esa capa o alguna de sus capas más internas.



UD 4 - TIPOS AVANZADOS DE DATOS

1 - Casting de referencias a objetos

- ▶ Usaremos las conversiones de referencias a objetos para **obtener los beneficios del polimorfismo**, es decir, poder tratar a un conjunto de objetos como un grupo con ciertas características comunes y obviando las diferencias entre ellos.
- ▶ Fíjate, además, que estamos **subiendo en la jerarquía**, es decir, de un tigre pasamos a un felino que es su superclase. Este tipo de casting hacia arriba se llama **upcasting** y no conlleva pérdida de información, así que Java lo hace sin quejarse en absoluto.
- ▶ Pero y ¿qué pasa si queremos hacer el camino inverso? Es decir, pasar de un Mamifero a un Felino o de un Felino a un Tigre? En este caso estamos **bajando en la jerarquía (downcasting)** y aquí puede haber problemas porque un Mamifero también podría ser un Equino y un Felino también podría ser un Gato... **Es decir podríamos estar perdiendo o corrompiendo la información del objeto.**



UD 4 - TIPOS AVANZADOS DE DATOS

1 - Casting de referencias a objetos

- Fíjate en el siguiente código:

```
Tigre tigre = new Tigre();  
Felino felino = tigre;  
Tigre tigre2 = (Tigre) felino;  
// Esta conversión se realizaría con éxito
```

Aquí Java quiere que el desarrollador asuma el riesgo explícitamente y le obliga a poner el operador de conversión de tipo (Tigre)

// Pero ¿qué pasaría con esta otra?

```
Object objeto = new Object();  
Tigre tigre = (Tigre) objeto;
```

Sintácticamente es correcto ya que el desarrollador asume el riesgo poniendo el operador de conversión de tipo (Tigre)

// Esta conversión provocaría el
// siguiente fallo en tiempo de ejecución

java.lang.ClassCastException: class java.lang.Object cannot be cast to class Tigre



UD 4 - TIPOS AVANZADOS DE DATOS

1 - Casting de referencias a objetos

En caso de duda, podemos usar el operador **instanceof** para preguntar si un objeto pertenece a una determinada clase antes de hacer la conversión y que se pueda producir una **ClassCastException**.

Ejemplo:

```
Animal a = zoo.getAnimalAleatorio();
```

```
if (a instanceof Tigre) {
```

```
    Tigre t = (Tigre) a;
```

```
    t.ruge();
```

```
}
```

```
else if (a instanceof Gato) {
```

```
    Gato g = (Gato) a;
```

```
    g.maulla();
```

```
}
```

De este modo podremos descender por la jerarquía pero tomando precauciones

Realizamos el ejercicio 1 del boletín



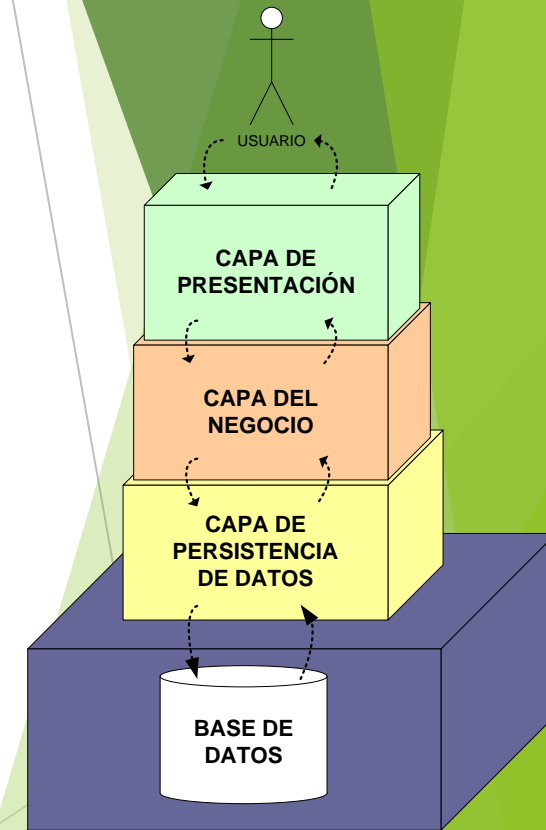
Durante este tema vamos a usar a menudo las conversiones explícitas entre objetos

UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

► 2 – Colecciones de datos

- Gran parte del trabajo de una aplicación consiste en **mover y transformar datos** de un punto a otro del código.
- Hasta ahora hemos usado los arrays para manejar una colección de elementos. Son estructuras que permiten acceder a los elementos a mucha velocidad pero presentan una serie de problemas:
 - Pensar en posiciones de la 0 a la TAM-1 no es cómodo.
 - Sintaxis “engorrosa” por los []
 - Si un array se me queda “pequeño” hay que redimensionarlo.
 - Si borro un elemento tengo que manejar el concepto de “hueco” o bien desplazar los elementos para “tapar” el hueco...
- Necesitamos colecciones de datos más potentes y versátiles



UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

- ▶ **Tipos de colecciones de datos**
- ▶ Usualmente en un aplicación de software nos podemos encontrar con las siguientes colecciones de datos:
- ▶ **LISTAS:** son colecciones que cumplen con dos criterios:
 - ▶ La **posición de los elementos es importante** de algún modo. Es decir, hay un criterio de orden.
 - ▶ Se permite almacenar **elementos duplicados** (pero no es obligatorio).
- ▶ Ejemplos:
 - ▶ Una lista de participantes de una carrera que se va “llenando” con los nombres de los corredores según su orden de llegada a la meta.
 - ▶ Una lista de palabras de un documento de texto. En este caso necesitamos mantener el orden de las palabras para preservar la coherencia del texto y se admiten palabras duplicadas.
 - ▶ Una lista de contactos de teléfono que está ordenada alfabéticamente.

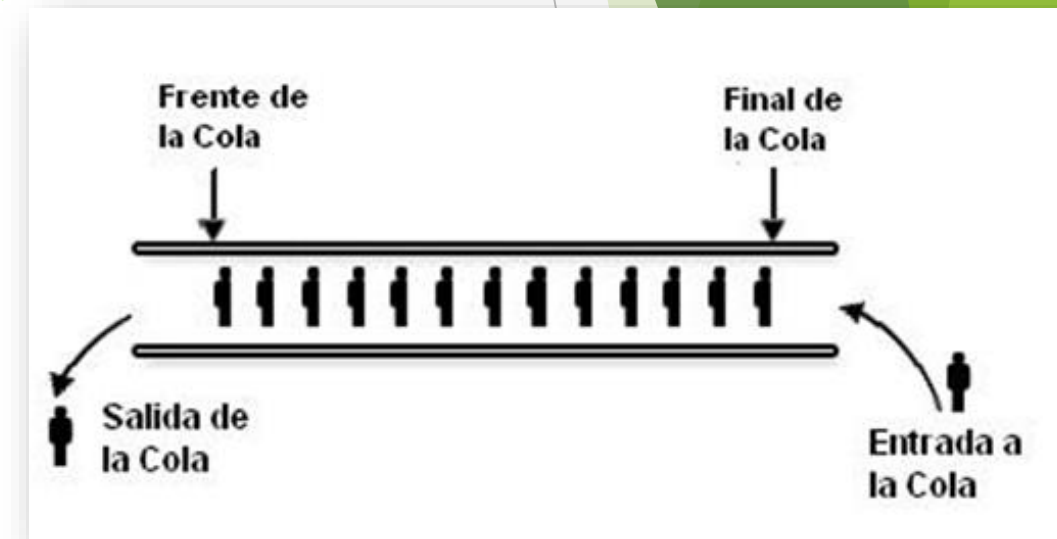


Los arrays que hemos usado en el tema 2 son un ejemplo de listas

UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

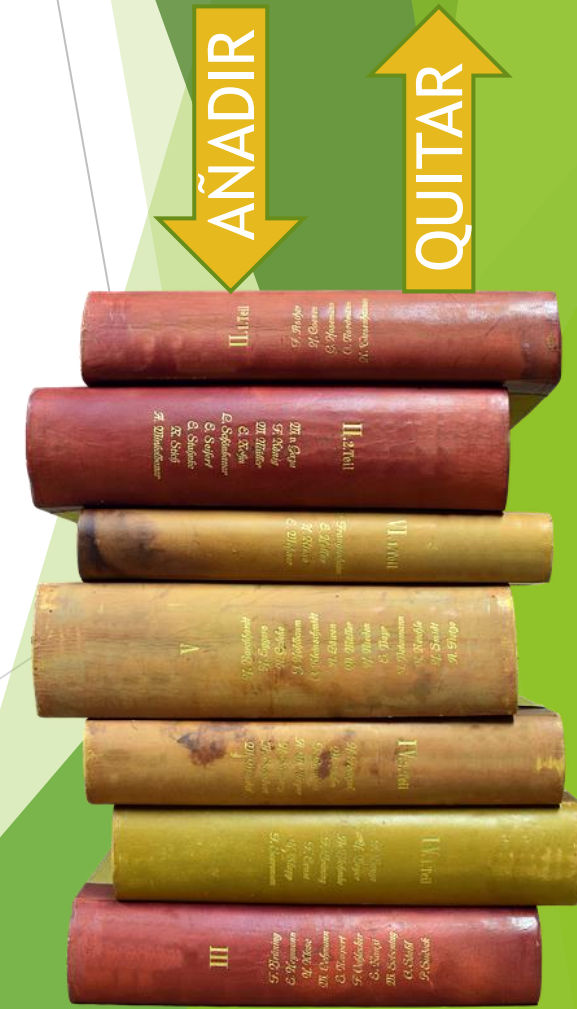
- ▶ **COLAS:** son **un tipo de listas** con las que se trabaja de la siguiente forma:
 - ▶ Los elementos se borran o extraen siempre del comienzo de la cola.
 - ▶ Los elementos se añaden siempre al final de la cola.
- ▶ Ejemplos:
 - ▶ La cola para comprar las entradas del cine.
 - ▶ La cola para pagar en el supermercado.
 - ▶ La cola de tareas pendientes de realizar.
 - ▶ La cola de mensajes a enviar en un servidor de e-mail.
- ▶ Son estructuras ligadas al procesamiento masivo de datos. Las peticiones van llegando, colocándose al final de la cola y van siendo atendidas por el programa por orden de llegada. Por eso, también se le llama estructuras FIFO (First In, First Out)



UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

- ▶ **PILAS:** son **un tipo de listas** con las que se trabaja de la siguiente forma:
 - ▶ Los elementos se añaden y se extraen siempre del mismo punto, llamado cima de la pila (top of stack).
- ▶ Son estructuras de tipo LIFO (Last In First Out).
- ▶ Ejemplos:
 - ▶ Una pila de libros o de platos para fregar.
 - ▶ Una pila de cartas en las que solo puedes coger la de arriba.
 - ▶ Una pila de acciones ejecutadas de un programa de modo que puedan ser “deshechas” mediante el botón “Undo” (deshacer).
 - ▶ Una pila de páginas web visitadas en un navegador para que podamos volver a la página anterior mediante el botón “Atrás”.



UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

- ▶ **CONJUNTOS:** son colecciones que cumple con dos criterios:
 - ▶ La **posición de los elementos NO es importante**. Es decir, NO hay un criterio de orden.
 - ▶ **NO se permite almacenar elementos duplicados.**
- ▶ Ejemplos:
 - ▶ Un conjunto de los miembros de una familia.
 - ▶ El conjunto de palabras de un idioma.
 - ▶ El conjunto de países del mundo.
 - ▶ El conjunto de habitantes de un pueblo.
- ▶ Los conjuntos son estructuras que están muy ligadas a las **operaciones de tipo “¿este elemento está contenido en el conjunto?”**. Es decir, ¿la palabra “dog” está en el idioma inglés? ¿El DNI 23423245E está contenido en el censo de los habitantes del pueblo?

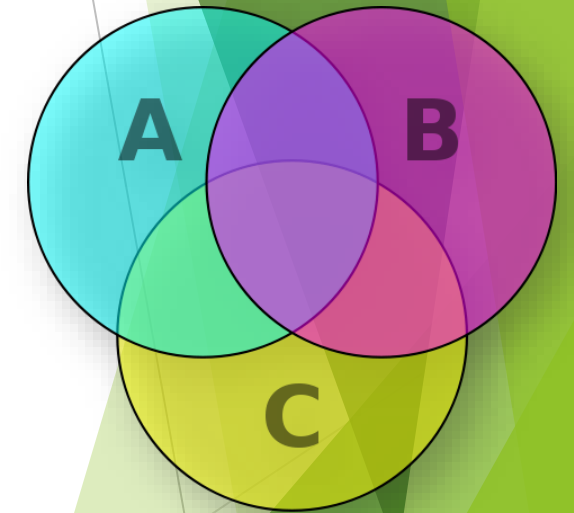


UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

- Además, si tenemos varios conjuntos que almacenan el mismo tipo de elementos, podremos realizar operaciones entre ellos, igual que se estudia en Matemáticas:

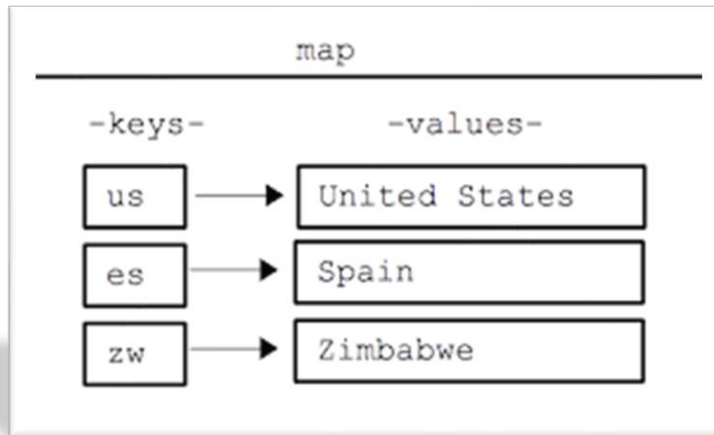
Union	Interseccion	Diferencia
		
Las personas que cuentan con un medio de transporte: Auto o Bicicleta	Las personas que cuentan Auto y Bicicleta	Las personas que cuentan Bicicleta pero no con Auto
Hugo, Paco, Luis	Paco	Hugo



UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

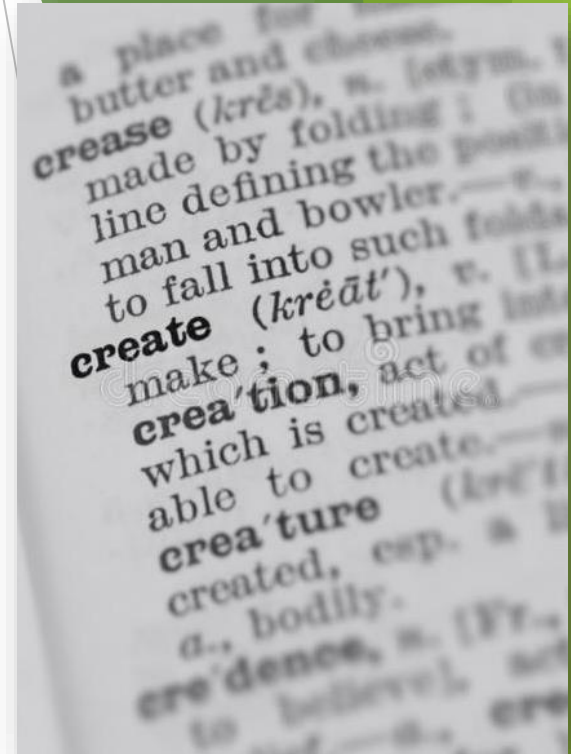
- **DICCIONARIOS:** son estructuras formadas por parejas clave-valor. Se toma un dato (clave) que identifica unívocamente a un elemento (valor) para así poder hacer búsquedas a mucha velocidad.
- Se comportan como un diccionario en el que la palabra a definir es la “clave de búsqueda” y la definición es el “valor buscado”.



Realizamos el
ejercicio 2 del
boletín

- Si hacemos una analogía con las tablas de una base de datos, la Primary Key sería la clave de búsqueda y los campos del resto del registro serían nuestro valor buscado.
- También se les llama tablas hash, arrays asociativos o mapas.

Son estructuras muy rápidas y potentes. Las estudiaremos en profundidad más adelante en este tema



UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

► El concepto de framework

- Un framework o marco de trabajo es un componente de software que ofrece al desarrollador un entorno o ambiente de trabajo para resolver un problema frecuente, reduciendo el tiempo/esfuerzo de desarrollo.
- Los frameworks se pueden clasificar según el problema que resuelven:
 - **Arquitectónicos:** permiten crear un “esqueleto” de la aplicación sobre el que empezar a trabajar, resolviendo de una forma estándar muchos de los problemas frecuentes que aparecen. Ejemplos: Spring, Angular, React...
 - **Autenticación:** permiten resolver de forma cómoda la autenticación de usuarios en distintas plataformas o aplicaciones.
 - **ORM (Object-Relational Mapping):** permiten automatizar el acceso a una base de datos relacional desde una aplicación orientada a objetos ofreciendo un conjunto de métodos y herramientas que le hacen la vida más fácil a los desarrolladores.
 - Y muchos más...



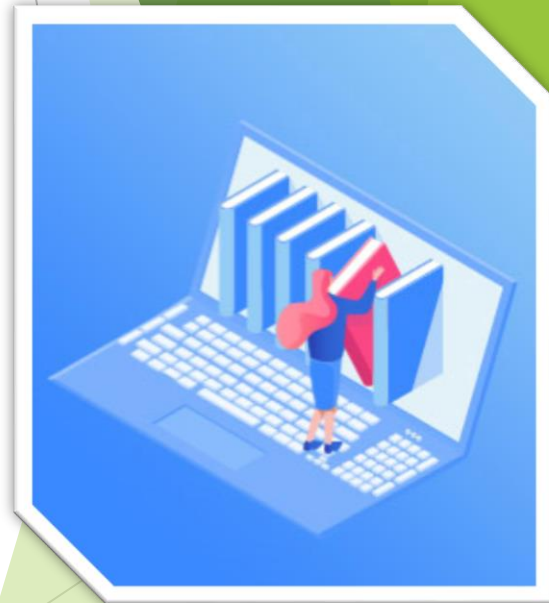
UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

► El Framework Collection de Java

- El JCF (Java Collections Framework) es el framework de Java que permite dar una solución cómoda, versátil y estándar al manejo de colecciones de objetos.
- Es importante recalcar que el JCF es un **estándar** y que, si usas este framework, tu código será rápidamente comprensible por la comunidad de desarrolladores Java.
- El JCF se compone de un **conjunto de clases e interfaces** que nos brinda una “forma de hacer las cosas”, nos da una filosofía de trabajo. Aprender esta forma de trabajar es uno de los objetivos de este tema.
- La web oficial es:

<https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html>

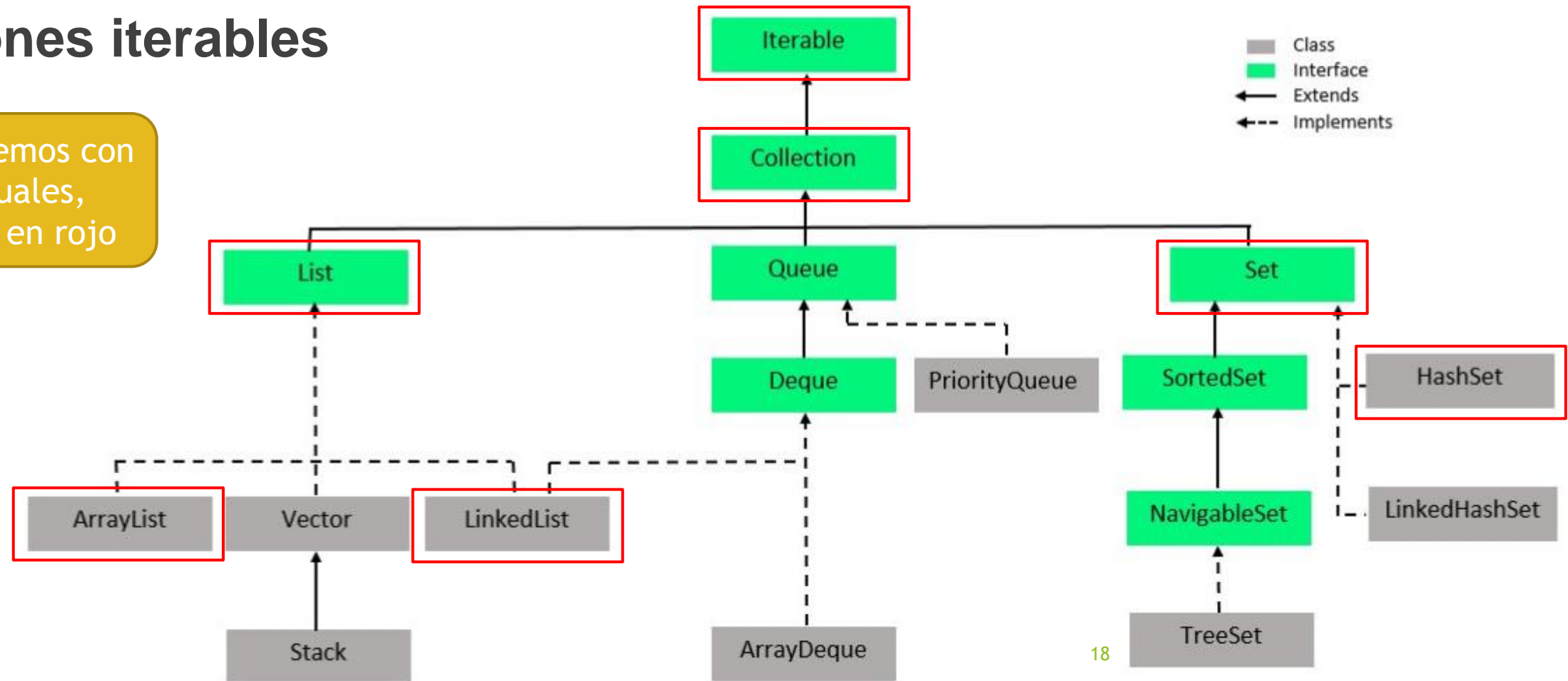


UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

- ▶ A vista de pájaro el JCF se compone de dos jerarquías de clases/interfaces independientes.
- ▶ **1) Colecciones iterables**

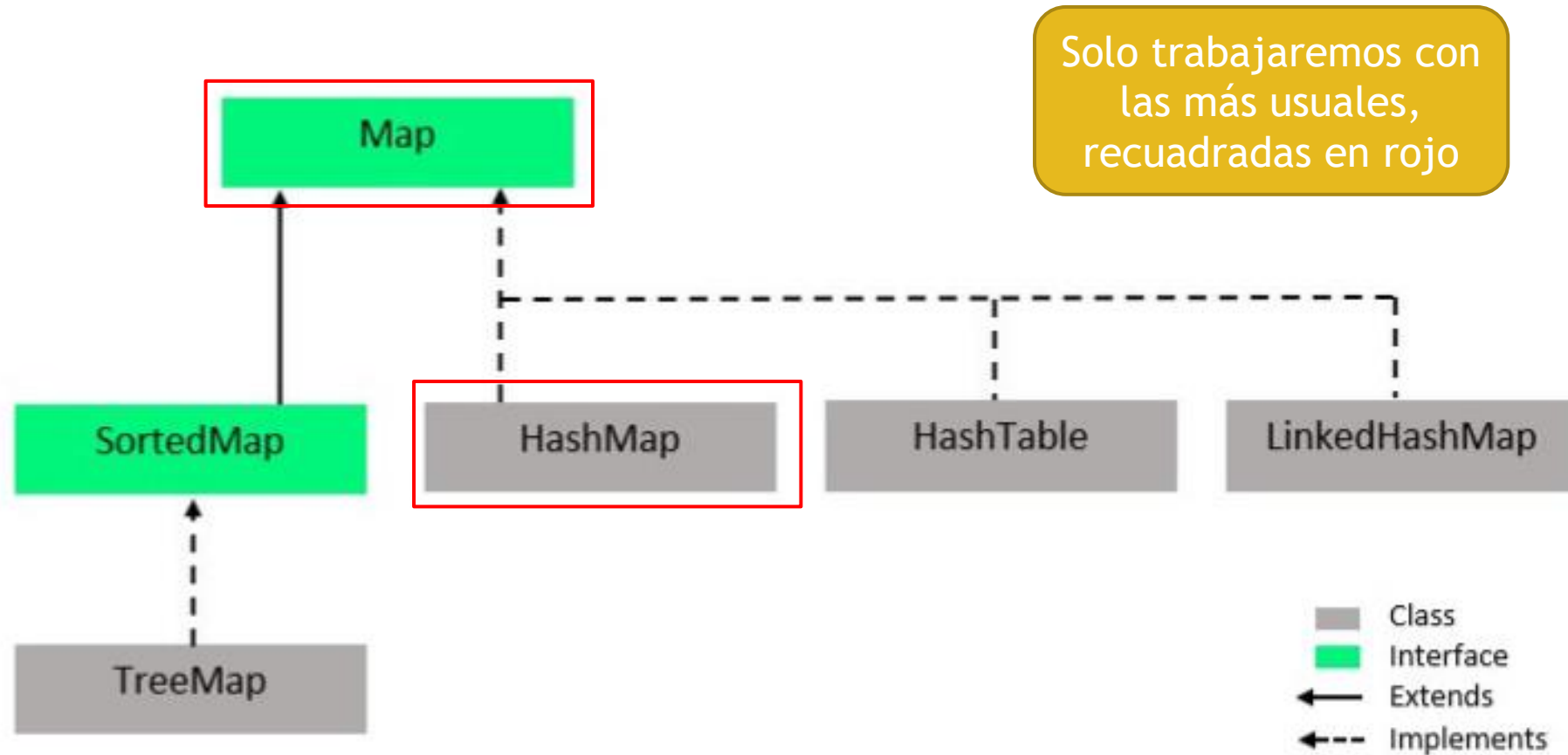
Solo trabajaremos con las más usuales, recuadradas en rojo



UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

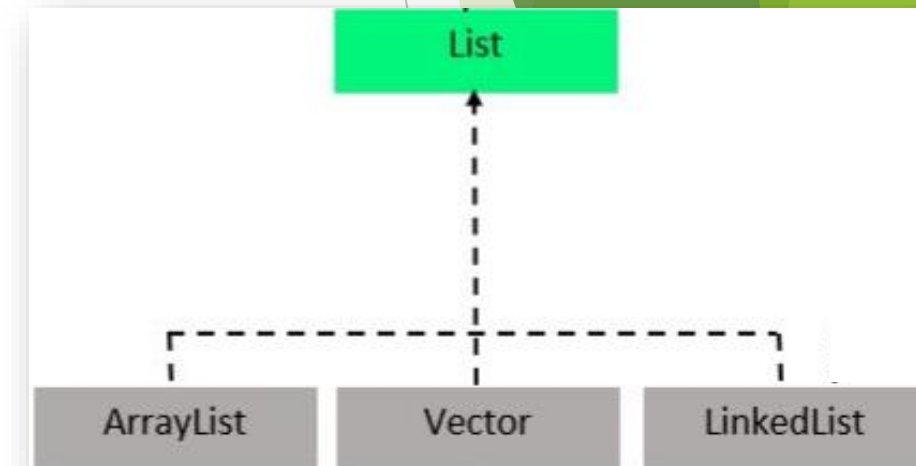
► 2) Colecciones tipo diccionario



UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

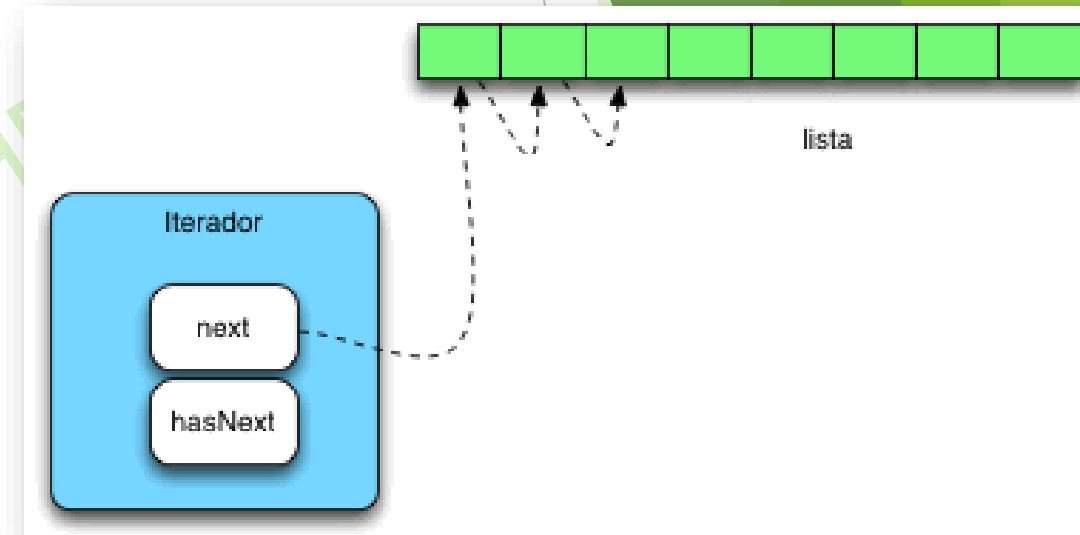
- **Para digerir los diagramas anteriores...**
- ... tenemos que recordar que las interfaces las podíamos entender como los “moldes” con los que podemos fabricar clases. Las interfaces dicen qué métodos deben tener las clases pero no especifican su código.
- Así, por ejemplo, la **interfaz List** especifica qué métodos debe tener una clase que quiera comportarse como una lista pero no da ningún código para ellos.
- Las **clases ArrayList, Vector y LinkedList** implementan la interfaz List y, por tanto, dan código a cada uno de los métodos de la interfaz. Cada clase lo realiza “a su manera”, haciendo que sea mejor o peor según para qué quiera utilizarse.



UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

- ▶ Para digerir los diagramas anteriores también tenemos que saber qué es un iterador
- ▶ Java entiende que una colección de objetos es una estructura que puede recorrerse (iterarse) pasando por todos y cada uno de los objetos que contiene.
- ▶ Para poder recorrer las colecciones se inventan una estructura llamada **iterador** que tiene dos métodos que nos permiten pasar por todos los elementos de una colección de forma sencilla:
 - ▶ **public boolean hasNext()** que nos dice si hay más elementos pendientes de recorrer.
 - ▶ **public <T> next()** que nos devuelve el siguiente elemento de la colección. En este caso <T> simboliza el tipo de datos que guarde la colección: String, Persona, Factura...



De este modo, TODAS las colecciones podrán recorrerse usando un iterador

UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

► Ejemplo:

```
public static void main(String[] args) {  
    List beatles = Arrays.asList("John", "Paul", "Ringo", "George");  
    Iterator iter = beatles.iterator();  
  
    while(iter.hasNext()) {  
        System.out.println(iter.next());  
    }  
}
```



- Fíjate que creamos una lista, después le pedimos que nos de un objeto capaz de recorrerla o iterar sobre ella y por último hacemos un bucle que usa el iterador y que nos permite recorrer la colección.
- Lo interesante es que **haríamos exactamente lo mismo** si tuviéramos que recorrer un conjunto o una pila o una cola...
- Como hemos empezado a “coquetear” con las listas, vamos a verlas.

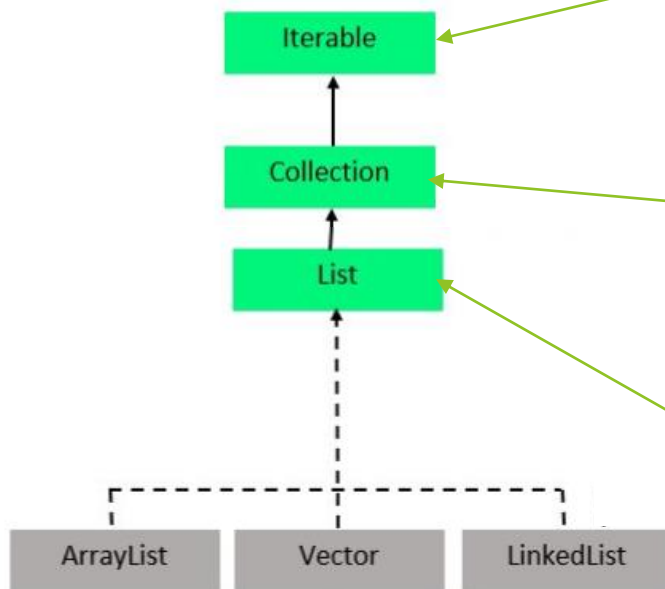
Puedes probar el código anterior descargando [U4.P1.Iteradores.zip](#)

UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

► 2.1 – Listas

- Vamos a intentar digerir el diagrama de interfaces y clases que nos permite crear y usar listas.



La interfaz **Iterable** solo define 1 método `public Iterator iterator()` que nos permitirá pedirle a una colección que nos devuelva un objeto **Iterator** capaz de recorrerla

La interfaz **Collection** extiende de **Iterable** de modo que también contendrá el método `iterator()`. Además define un conjunto de métodos comunes a todas las colecciones: añadir un elemento, borrar un elemento, consultar si está vacía...

La interfaz **List** extiende de **Collection** de modo que también contendrá todos los métodos anteriores y además otros nuevos, propios de las listas: obtiene el elemento de la posición X, borra el elemento de la posición Y...

Shopping list

Milk 
Apples 
Eggs 
Toilet rolls 
Bananas 
Bread 

UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

- ▶ Las **clases** ArrayList, Vector y LinkedList **implementan la interfaz List**, dando código a todos los métodos que esta interfaz contiene. De hecho, no añaden ningún método más.
- ▶ Vamos a partir de la implementación de **ArrayList** y vamos echar un ojo a los métodos más utilizados.

Para ello, lo mejor es verlo con un ejemplo de uso:

CREAMOS UNA LISTA Y OPERAMOS CON ELLA

La lista conserva el orden de inserción de los elementos:

[Juan, Ana, Pedro]

Se admiten duplicados:

[Juan, Ana, Pedro, Ana]

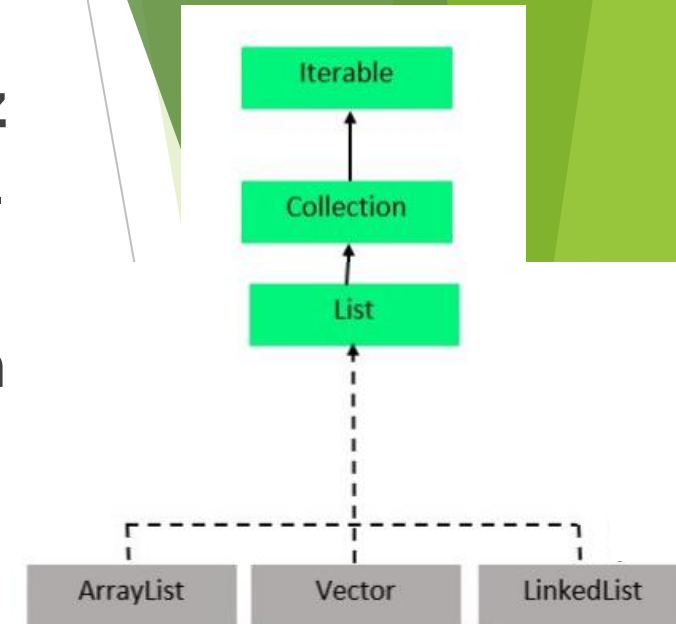
Insertamos 'Luisa' en la posición 1 y se desplazan los elementos de su derecha:

[Juan, Luisa, Ana, Pedro, Ana]

Actualizamos el elemento de la posición 2 con un nuevo valor

[Juan, Luisa, Irene, Pedro, Ana]

Descargamos y estudiamos el código de U4.P2.Listas.zip



UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

- Del código de U4.P2.Listas hay dos líneas que hay que explicar:

1) Fíjate que hemos escrito lo siguiente:

```
// CREACIÓN DE UNA LISTA  
List lista = new ArrayList();
```

En vez de
escribir esto:

```
// CREACIÓN DE UNA LISTA  
ArrayList lista = new ArrayList();
```

Estamos aprovechando el polimorfismo de las interfaces **usando una referencia a interfaz en vez de usar una referencia a la propia clase**. Esto se considera **una buena práctica** y nos aporta la siguiente ventaja:

- Si mañana quiero cambiar mi código y dejar de usar ArrayList para usar otra implementación de una lista solo tendría que cambiar la línea anterior y no impactaría en ninguna otra parte del código. Por ejemplo, podríamos simplemente modificar lo siguiente y todo seguiría funcionando bien:

```
// CREACIÓN DE UNA LISTA  
List lista = new LinkedList();
```



UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

2) Fíjate que hemos escrito lo siguiente:

```
Iterator iter = lista.iterator();  
while (iter.hasNext()) {  
    String elem = (String) iter.next();  
    System.out.println("Elemento: "+elem);  
}
```

Realizamos los ejercicios del 3 al 6 del boletín de problemas

El método **next** tiene la siguiente firma: **public Object next()** así que si hemos guardado String en nuestra colección tendremos que **usar el operador de conversión de tipos explícita** para evitar que el compilador se “queje” de nuestro código.

Con lo que sabemos hasta ahora, las colecciones solo saben guardar objetos de tipo **Object**, aunque tú guardes un String, una Persona... ellas piensan que le has metido un Object. Por eso tenemos que hacer la conversión.

Más adelante en este tema estudiaremos los tipos “**genéricos**” que nos solucionará este problema.



UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

► ArrayList VS LinkedList

- **ArrayList** es una clase que implementa la interfaz List y que utiliza internamente un array para almacenar los objetos.
- **LinkedList** es una clase que implementa la interfaz List y que utiliza internamente una “cadena de nodos doblemente enlazados” para almacenar los objetos. Cada nodo es capaz de guardar el objeto y las referencias a su nodo anterior y a su siguiente de la lista.



¿Cuál es mejor? ¿Cuándo usar cada una de ellas?

La respuesta es "DEPENDEN"...



UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

Operación	ArrayList	LinkedList
Acceso posicional a un elemento mediante el método <i>get(pos)</i>	Inmediato	Lento. Para llegar al elemento N hay que pasar por los N-1 anteriores
Insertar un elemento mediante <i>add(pos, elem)</i>	Lento. Requiere desplazar elementos para insertar el nuevo elemento	Rápido. Sólo hay que crear el nuevo nodo y cambiar las referencias con los nodos anterior y posterior
Eliminar un elemento mediante <i>remove(pos)</i>	Lento. Requiere desplazar elementos para “tapar” el hueco del elemento borrado	Rápido. Sólo hay que borrar el nodo y actualizar las referencias de los nodos anterior y posterior

- Si utilizamos **referencias a la interfaz List** y, en algún momento, se nos hace necesario pasar de un ArrayList a una LinkedList o viceversa, tendremos que **modificar solo una línea de código**.

ArrayList es la clase más utilizada del API de Java



Usaremos **LinkedList** si la lista se va a tratar con muchas inserciones/borrados y pocos accesos por posición.

UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

► Ordenando listas

- En los ejercicios hemos ordenado nuestras listas de elementos String utilizando el método estático **Collections.sort (lista)**
- Esto es posible porque la **clase String** “sabe compararse” y puede decir si un elemento es menor/mayor o igual a otro.
- Para que una clase cualquiera (Persona, Artículo, String...) “sepa compararse”, Java solo le pide una cosa: «**que implemente la interfaz Comparable**».
- Esta interfaz solo tiene el método `public int compareTo (Object obj)` y debe comportarse así:
 - Si **this < obj** entonces este método debe devolver un entero < 0
 - Si **this > obj** entonces este método debe devolver un entero > 0
 - Si **this = obj** entonces este método debe devolver un 0



COMPARABLE

Descargamos y estudiamos
el código de
U4.P3.ProbandoComparable

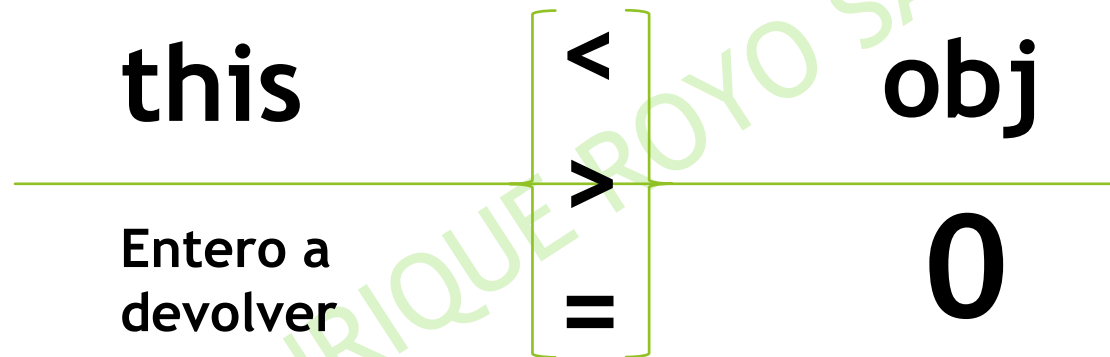
UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

► Trucos cuando usamos Comparable

- Como puede ser un poco lioso esto de devolver un entero $< \text{ó} > \text{ó} = 0$ tenemos dos recursos mentales para solucionarlo:

- Pensar en el siguiente **paralelismo**:



- Usar **una resta** para el cálculo (no siempre se puede). Ejemplo:

```
public int compareTo(Object obj) {  
    Empleado emp = (Empleado) obj;  
    int resultado = (int) (this.salarario - emp.getSalario());  
    return resultado;  
}
```

Realizamos los ejercicios del 7 al 10 del boletín de problemas

En Java se llama “orden natural” al criterio de orden que establece el método *compareTo*

UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

► Conclusiones sobre el trabajo con listas

- Las usaremos siempre que necesitemos conservar un orden en los elementos o bien si éstos puedan repetirse.
 - Para buscar un elemento en una lista se hace secuencialmente. Si queremos una colección “especialista” en búsquedas rápidas, las listas no son nuestro mejor candidato.
 - ArrayList es la implementación más utilizada.
 - Usaremos siempre referencias a la interfaz List (buenas prácticas).
-
- La clase **Vector** también implementa la interfaz List. Es como un ArrayList pero que no permite el acceso simultáneo de dos hilos de programación. No la estudiamos.
 - **Las colas (Queue) y las pilas (Stack) tampoco las estudiamos**

CONCLUSION

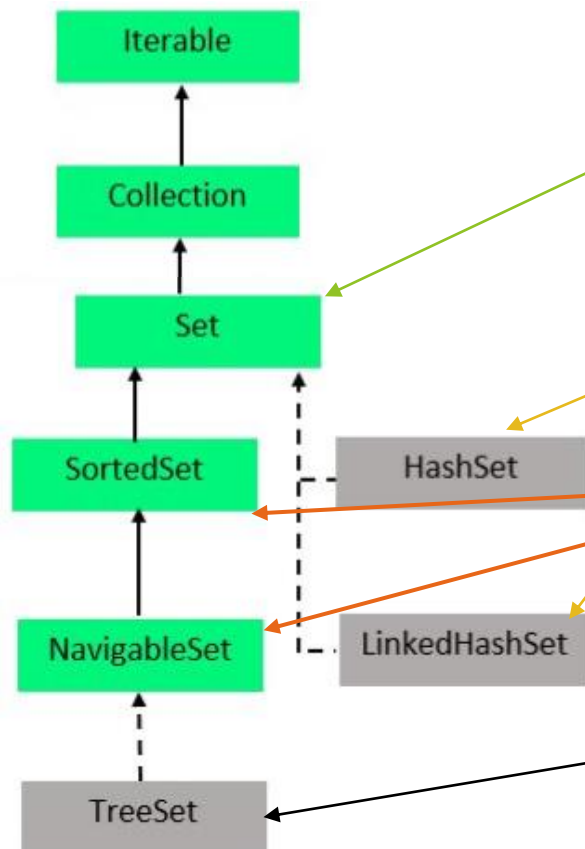


UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

► 2.2 – Conjuntos. Clases Wrapper

- Vamos a intentar digerir el diagrama de interfaces y clases que nos permite crear y usar conjuntos.



La interfaz **Set** extiende de **Collection** así que hereda todos los métodos de esta interfaz. De hecho, **Set** no añade ningún otro método pero sí especifica cómo debe comportarse un conjunto

HashSet y **LinkedHashSet** son las dos clases que implementan la interfaz **Set**. Solo estudiaremos **HashSet**

SortedSet y **NavigableSet** son dos interfaces que permiten establecer un orden en un conjunto

TreeSet es una implementación de las interfaces anteriores que usa un “árbol binario” para almacenar los datos de forma ordenada. No la estudiaremos.

32



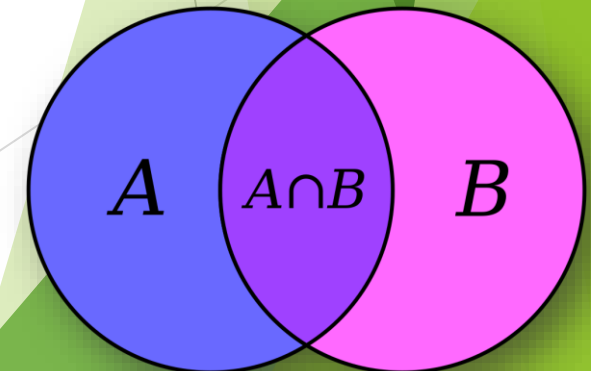
UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

- ▶ Te recuerdo las características de los conjuntos:
 - ▶ No se permiten duplicados.
 - ▶ Los elementos almacenados no guardan ningún orden.
 - ▶ Son expertos en buscar rápidamente si un elemento está contenido o no en el conjunto.
 - ▶ Además permiten operaciones entre conjuntos: unión, intersección, diferencia...
- ▶ Estudiamos cómo se usa la **clase HashSet** que **implementa la interfaz Set**. Como siempre, mejor con un ejemplo:

Descargamos y estudiamos el código de U4.P4.Conjuntos.zip

Realizamos los ejercicios 11 y 12 del boletín de problemas



UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

- ▶ **Clases Wrapper (envoltorio)**
- ▶ Hasta ahora siempre hemos guardado objetos en nuestras listas o conjuntos. ¿Pero qué pasa si quiero **guardar un tipo primitivo** como un **int** o un **double**? Pues **NO se puede**, las colecciones solo saben guardar objetos. 😱
- ▶ Pero ¿y si hacemos un “truco”? ¿Y si creamos un objeto que “envuelva” a cada uno de los tipos primitivos? Entonces sí podríamos guardarlos en una colección.

¡Qué buena idea!

La cosa es que esto ya se les había ocurrido a los “padres de Java” y ellos han creado una jerarquía de clases “envoltorio”.

Veámosla:



PRIMITIVOS

boolean

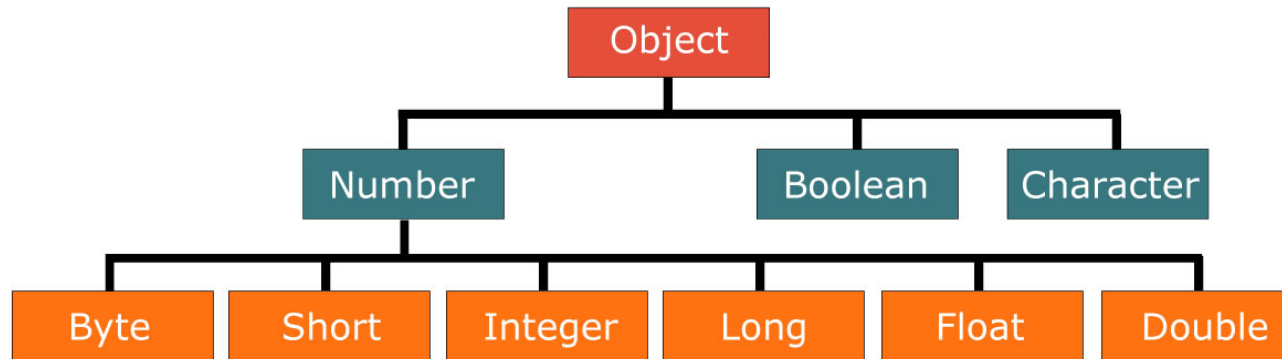
int

double

UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

- La jerarquía de clases envoltorio es la siguiente:



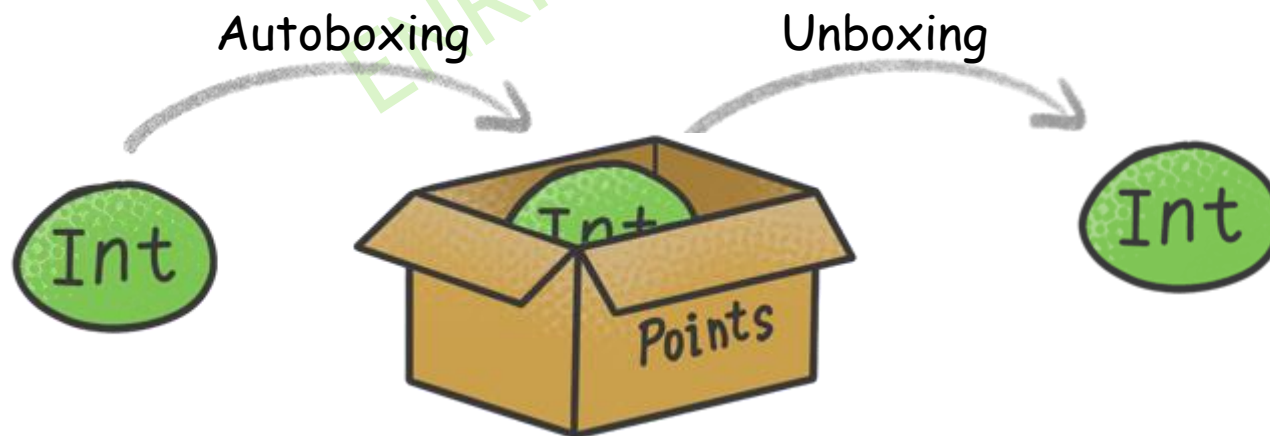
- **Number** es una clase abstracta que tiene los siguientes métodos: *byteValue()*, *shortValue()*, *intValue()*, *longValue()*, *floatValue()*, *doubleValue()*. Cada uno retorna el tipo primitivo correspondiente que mejor represente al valor que guarda internamente la clase.
- A su vez, cada subclase contiene un conjunto de métodos y constantes que son muy útiles para manejar este tipo de objetos (comparar, devolver el mayor de dos números, analizar una cadena de texto para “extraer” un valor numérico de ella, convertir un número en cadena...)

Descargamos y estudiamos el código de U4.P5.Wrappers.zip

UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

- ▶ Las clases Wrappers existen desde la versión 1.0 de Java. A partir de la versión 1.5 se introdujo el **Autoboxing/Unboxing**, que solo es un mecanismo que realiza automáticamente las conversiones de tipo entre valores primitivos y los wrappers, consiguiendo así una sintaxis más limpia.
- ▶ Se usa la metáfora de “meter” el tipo primitivo en una caja (boxing) cuando lo encerramos en la clase wrapper y “sacarlo” de la caja (unboxing) cuando obtenemos de nuevo el tipo primitivo.



UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

- Es tan sencillo como lo siguiente:

```
int miEntero = 10;
// Esto sería AUTOBOXING
Integer nuevoObjeto3 = miEntero;
Integer nuevoObjeto4 = 1000;

// Esto sería UNBOXING
int result1 = nuevoObjeto3;
// Y esto también
int result2 = nuevoObjeto3 + nuevoObjeto4;
```

- Básicamente Java hace automáticamente la conversión que corresponda dependiendo del tipo de dato que se espere en cada situación. ¡FÁCIL!

Descomentamos la segunda mitad del código de U4.P5.Wrappers.zip

Realizamos los ejercicios 13 y 14 del boletín de problemas

UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

- **Funcionamiento interno de un HashSet**
- Decíamos que los conjuntos son expertos en búsquedas. Mientras que en una lista hacemos una búsqueda secuencial de un elemento, con los conjuntos se consigue un acceso "casi" inmediato al elemento buscado.
- Vamos a ver el siguiente código que **compara el rendimiento de las listas y los conjuntos** cuando tenemos que realizar muchas búsquedas:

Descargamos y estudiamos el
código de
[U4.P6.RendimientoConjuntos.zip](#)

- Esta velocidad se consigue gracias a la combinación de dos elementos que vamos a estudiar:
 1. Las funciones de hashing
 2. Las tablas hash o diccionarios o arrays asociativos



UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

1. Las funciones de hashing

- ▶ Son funciones matemáticas que toman una secuencia de bytes (de un objeto, de un archivo, de una contraseña...) y lo convierten en un código numérico. Ese código es como un “resumen” o “identificador único” de esa secuencia de bytes.
- ▶ El método **public int hashCode()** de la clase **Object** nos devuelve el código hash de un objeto:

```
public static void main(String[] args) {  
    Object obj = new Object();  
    System.out.println("Hash code = "+obj.hashCode());  
}
```

Hash code = 914504136



UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

- ▶ El método **hashCode()** puede ser sobrescrito por las subclases de Object pero te exige cumplir un “contrato”:
 - ▶ **Consistencia:** si llamamos calculamos el hashCode de un mismo objeto en distintos instantes temporales, el código retornado debe ser el mismo.
 - ▶ **Igualdad:** si dos objetos se consideran iguales según el método **equals** entonces el código hash de ambos objetos también debe ser el mismo. Esto implica que los métodos **equals** y **hashCode** van “emparejados”.
- ▶ Entonces podríamos crear un **hashCode()** que siempre devuelve el valor 1 y cumpliría ambas reglas... pero **¿nos serviría?**
- ▶ No, si queremos obtener el beneficio de las búsquedas rápidas en los conjuntos. Para ello debemos cumplir además con la siguiente cláusula:
 - ▶ **Diferencia:** si dos objetos son distintos según el método **equals** entonces los códigos hash de ambos objetos deben ser distintos.

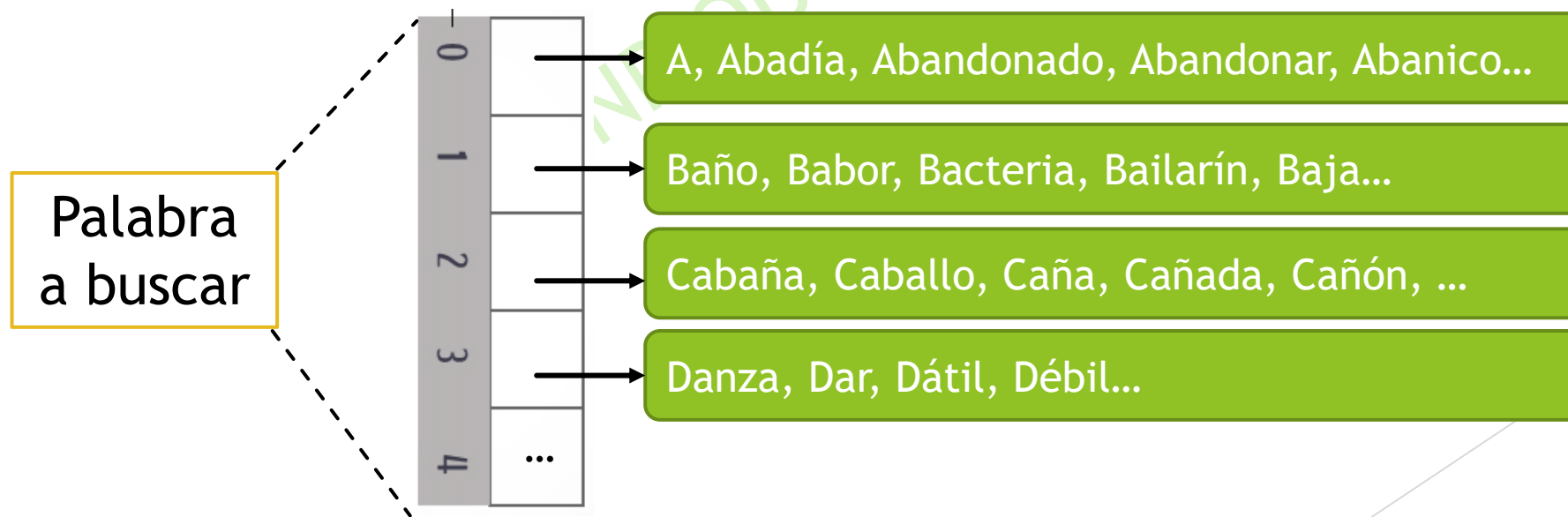


UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

2. Las tablas hash

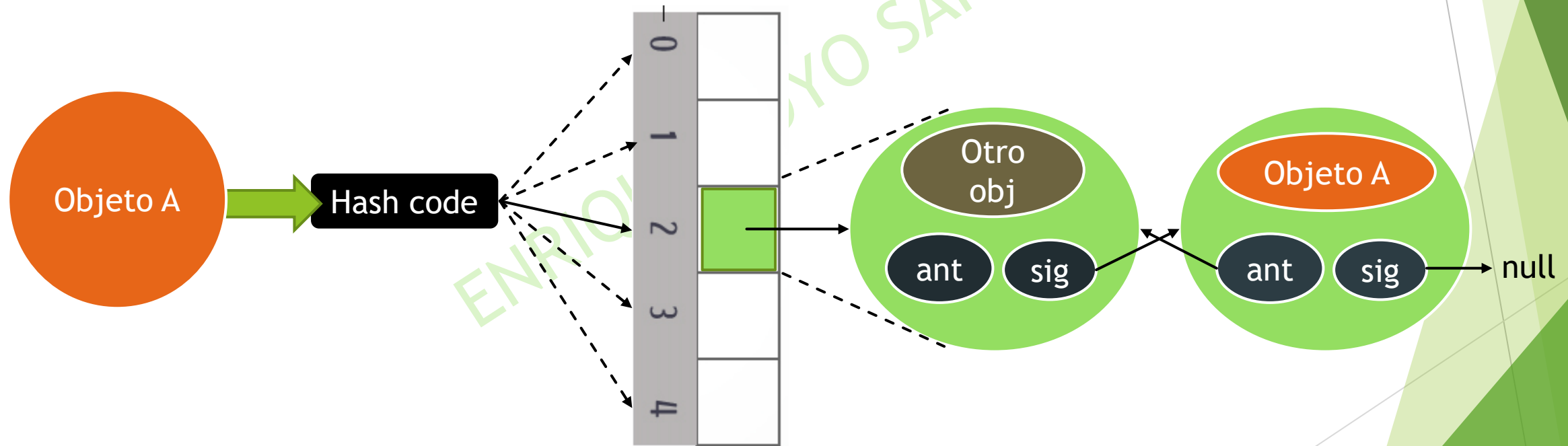
- ▶ La idea de funcionamiento de las tablas hash **se basa en cómo buscaríamos información en un diccionario.**
- ▶ En castellano, todas las palabras empiezan por una letra de la A-Z (27 letras). Si asignásemos a cada letra una posición de un **array** y dentro de cada celda del array guardamos una **lista** con las palabras que comienzan por esa letra **podríamos acelerar las búsquedas.**



UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

- En Java se hace algo parecido. Se saca el código hash del objeto y tras un “ajuste” matemático se calcula la posición del array en la que se guardará el objeto. Si hay varios objetos en dicha posición se almacenan en una lista enlazada (LinkedList) por orden de llegada.

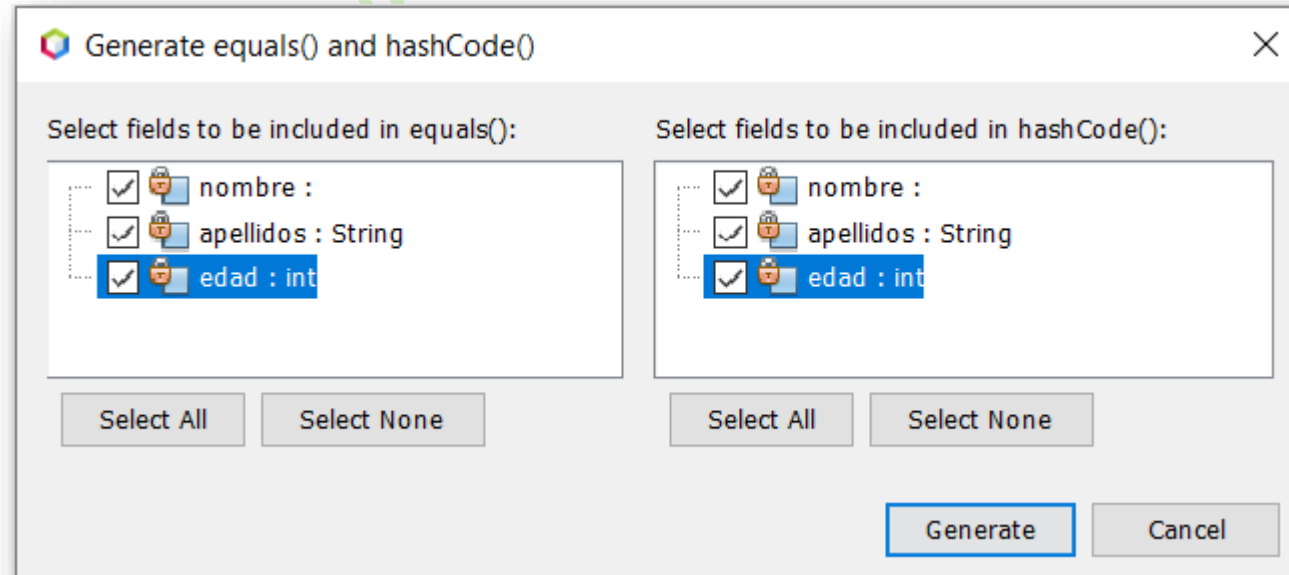
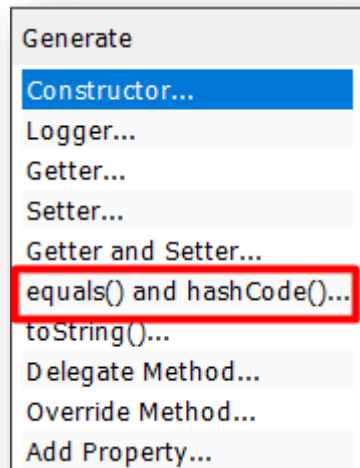


- De este modo, se aceleran considerablemente las búsquedas.

UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

- Decíamos antes que **equals** y **hashCode** debían ir emparejados. Es decir, debemos **utilizar las mismas propiedades** para establecer el criterio de igualdad y para generar el código hash.
- Si no lo hiciéramos así entonces se podrían “perderse” objetos en la estructura, ocupando espacio pero sin poderse “rescatar”.
- Al final el IDE nos lo pone fácil y casi que nos obliga a emparejarlos:



UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

- Pero no todas las propiedades de las clases van a ser buenas candidatas para generar esta pareja de métodos...
- ...Imagina que en una clase Vehículo seleccionamos propiedades que cambian su valor en el tiempo para generar el **hashCode** y el **equals**:

```
public class Vehiculo {  
    private String matricula, marca, modelo, nombrePropietario;  
    private boolean arrancado;  
    private double numLitrosDeposito;  
}
```

¿Qué pasaría si escogemos arrancado y numLitrosDeposito?

- En cuanto estas propiedades cambien su valor el código hash resultante sería distinto y también cambiaría el criterio de igualdad. Dos objetos que antes se consideraban iguales, ahora no... las búsquedas en la tabla hash fallarían ¡UN DESASTRE!



UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

- ▶ Tenemos que procurar que las propiedades implicadas en los métodos **equals** y **hashCode** NO CAMBIEN en el tiempo o que cambien lo menos posible.
- ▶ Ejemplos de “propiedades candidatas” y su clasificación:

Buenas candidatas	Malas candidatas
DNI, Nombre, Apellidos, Fecha de nacimiento, matrícula de un coche, número de factura, número de serie, código de barras, nombre de usuario...	edad, estaCasada, tieneTrabajo, númeroHijos, número de litros en el depósito, número de kilómetros recorridos, total de la factura, precio del producto...

- ▶ Si deseamos forzar que una propiedad de un objeto no cambie se le puede poner el modificador **final** para que su valor no pueda ser modificado (o mutado). Se dice entonces que estas propiedades son **INMUTABLES**.

Realizamos los ejercicios 15 y 16 del boletín de problemas

Soy INMUTABLE

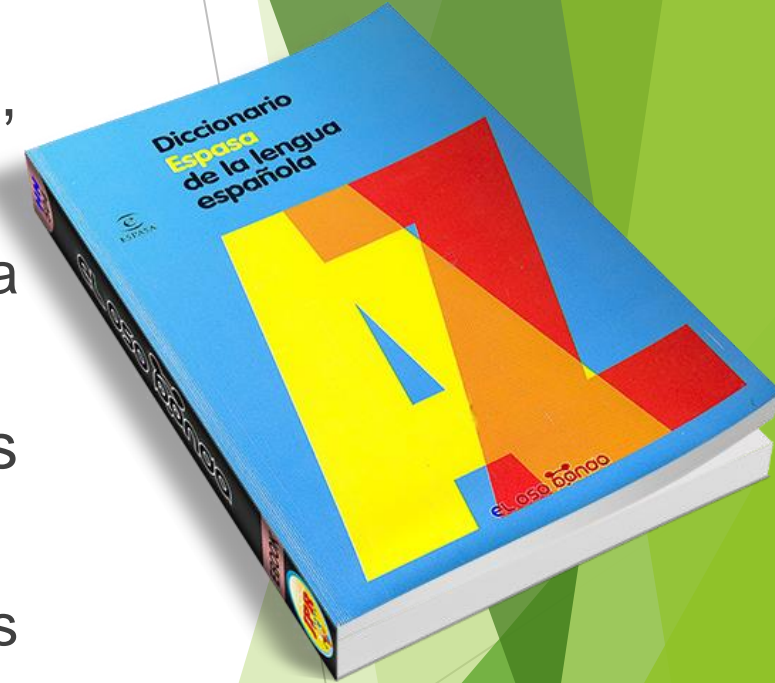


UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

► 2.3 – Diccionarios

- Esta estructura recibe muchos nombres distintos: diccionario, mapa, array asociativo, tabla hash, tabla de dispersión...
 - **To map** = asignar, determinar, corresponder. En la jerga informática española se dice "mapear"...
- Un diccionario es una estructura de búsqueda que almacena dos tipos de elementos:
 - **Claves:** son utilizadas como identificadores para localizar los valores. En inglés se usan los términos: *key* (llave o clave) o *entry* (entrada).
 - **Valores:** son los elementos que se almacenan para posteriormente ser buscados.
- Con lo que ya sabemos, vamos a intentar digerir esta estructura mediante el ejemplo del código *U4.P7.Diccionarios*

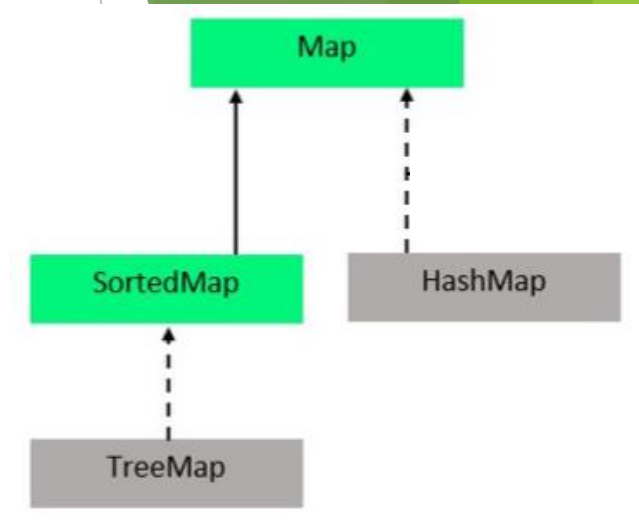


Descargamos y
estudiamos el
código de
U4.P7.Diccionarios

UD 4 - TIPOS AVANZADOS DE DATOS

2 - Colecciones de datos

- ▶ Después de ver el código anterior podemos observar algunas cosas:
 - ▶ Básicamente, tenemos una estructura que funciona como un HashSet pero separando los conceptos de clave y valor.
 - ▶ La interfaz Map no descende de Collection, así que no podremos pedirle un iterador directamente.
 - ▶ Sin embargo, tanto las claves como los valores se almacenan internamente como una colección. Podemos obtenerlas (métodos keySet() y values()) y sacar un iterador de estas colecciones para recorrer la estructura.
 - ▶ El tipo de dato de las claves y de los valores pueden ser diferentes. Por ejemplo: **Clave** = matrícula (String) y **Valor** = clase Vehículo. **Clave** = DNI (String) y **Valor** = clase Empleado.



Realizamos los ejercicios 17 y 18 del boletín de problemas

UD 4 - TIPOS AVANZADOS DE DATOS

3 - Genéricos

► 3 – Genéricos

- Desde Java 1.5 disponemos de una notación, ampliamente difundida, que permite “concretar” o restringir los tipos de datos contenidos en clases o métodos en tiempo de compilación....

¿ein? ¿y eso qué es? Vamos a verlo...

- **3.1. Necesidad. Uso de colecciones con genéricos**
- Para comprender los genéricos tenemos que entender primero **qué problemas se presentan** al no usar esta notación.
- Para ello vamos a estudiar el siguiente trozo de código que lo tenemos disponible en **U4.P8.NecesidadGenericos**

<GENERIC>



UD 4 - TIPOS AVANZADOS DE DATOS

3 - Genéricos

Código disponible en
U4.P8.NecesidadGenericos

```
public class NecesidadGenericos {  
    public static void main(String[] args) {  
        List listaBlanca = new ArrayList();  
        listaBlanca.add("954784512");  
        listaBlanca.add("654987123");  
        listaBlanca.add("658789451");  
        listaBlanca.add(954865784); // El IDE no nos avisa  
  
        Iterator iter = listaBlanca.iterator();  
        while(iter.hasNext()) {  
            String elem = (String) iter.next();  
            System.out.println("Teléfono permitido: "+elem);  
        }  
    }  
}
```

Queremos guardar los teléfonos como String, sin embargo nos equivocamos y metemos un Integer pero nadie nos avisa de nuestro error

Al hacer la conversión de Object a String todo va bien hasta que llegamos al Integer y se provoca una excepción

Teléfono permitido: 954784512
Teléfono permitido: 654987123
Teléfono permitido: 658789451

Exception in thread "main" java.lang.ClassCastException: class java.lang.Integer cannot be cast to class java.lang.String
at es.tuespiral.u4.p7.necesidadgenericos.NecesidadGenericos.main(NecesidadGenericos.java:14)

UD 4 - TIPOS AVANZADOS DE DATOS

3 - Genéricos

- Si rescribimos el código anterior con genéricos podemos decirle al compilador que **queremos usar String tanto en la lista como en el iterador**:

```
public class NecesidadGenericos {  
    public static void main(String[] args) {  
        List<String> listaBlanca = new ArrayList<String>();  
        listaBlanca.add("954784512");  
        listaBlanca.add("654987123");  
        listaBlanca.add("658789451");  
        listaBlanca.add(954865784); // El IDE ahora sí nos avisa  
  
        Iterator<String> iter = listaBlanca.iterator();  
        while(iter.hasNext()) {  
            String elem = iter.next(); // No hace falta el (String)  
            System.out.println("Teléfono permitido: "+elem);  
        }  
    }  
}
```

Con la notación de genéricos encerramos entre <> el tipo de datos que queremos usar

Ahora el IDE sí nos avisa de nuestro error

Además, ya no hace falta hacer la conversión explícita de tipos porque el compilador ya sabe con qué tipo de dato estamos trabajando

UD 4 - TIPOS AVANZADOS DE DATOS

3 - Genéricos

- Podemos usar una notación abreviada cuando creamos la colección de la siguiente forma:

```
List<String> listaBlanca = new ArrayList<>();
```

- El tipo que hay en la parte izquierda se aplica también en la parte derecha.
- A los <> que quedan en la parte derecha se les llama “operador diamante”.
- Veamos algunos ejemplos más:

```
Set<Cliente> conjunto = new HashSet<>();
```

```
Map<String, Cliente> diccionario = new HashMap<>();
```

- Como las colecciones almacenan *Object* y este tipo es demasiado general, **los genéricos nos permiten “concretar” o restringir el tipo de datos que queremos guardar**. De este modo, el compilador hace comprobaciones del tipo de datos que se guarda en cada momento y nos permite crear un código más seguro y fiable. **Así es como se usan la colecciones hoy en día.**



Realizamos los ejercicios 19 y 20 del boletín de problemas



UD 4 - TIPOS AVANZADOS DE DATOS

3 - Genéricos

BYE BYE ITERATORS - SYNTACTIC SUGAR

- Otra ventaja de usar genéricos al declarar nuestras colecciones es que podemos **prescindir de los iteradores** obteniendo una sintaxis más "endulzada". Observa el siguiente ejemplo:

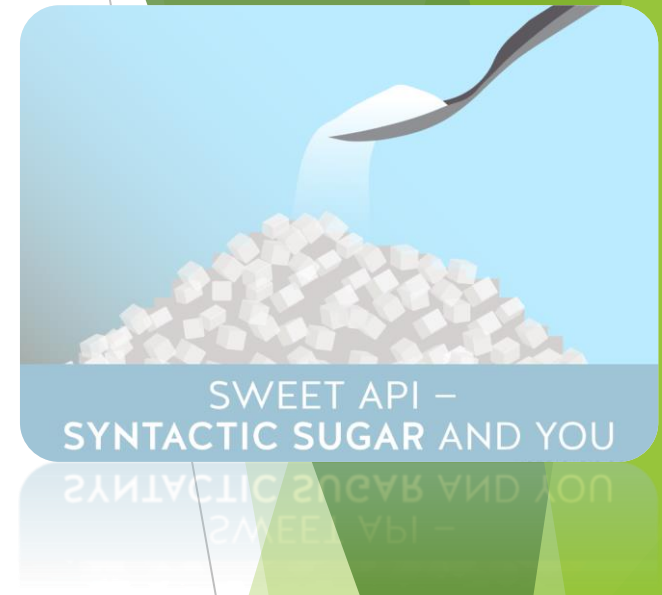
```
Set<User> usuarios = new HashSet<>();
```

```
for (User user : usuarios) {  
    System.out.println(user);  
}
```

=

```
Iterator<User> iter = usuarios.iterator();  
while (iter.hasNext()) {  
    System.out.println(iter.next());  
}
```

- Si usamos un bucle **for** con la sintaxis tipo "for-each" conseguimos un código más limpio y legible. El compilador hará el trabajo por nosotros y lo "reescribirá" usando un iterador.



UD 4 - TIPOS AVANZADOS DE DATOS

3 - Genéricos

► 3.2. Otros usos de los genéricos

- El uso más frecuente de los tipos genéricos es el que hemos visto en las colecciones. Pero vamos a ver dos ejemplos de uso más:
- **Creación de clases que puedan contener un tipo genérico**
- Java nos proporciona las colecciones que dan solución a las necesidades más habituales de una aplicación, pero también nos deja crear nuestras propias colecciones y estructuras de datos.
- Un ejemplo muy sencillo consiste en modelar una clase contenedora que permite guardar un objeto de un tipo de dato cualquiera, como si fuera una **caja genérica**. Eso sí, una vez que concretemos qué tipo de objeto queremos guardar en la caja, no permitiremos guardar otra cosa. Así que si decidimos guardar ropa en nuestra caja, no permitiremos guardar comida.
- Veamos el código:



UD 4 - TIPOS AVANZADOS DE DATOS

3 - Genéricos

- Cuando definimos la clase, ahora también decimos que vamos a trabajar con un tipo genérico **<T>** en el interior de la clase.

```
public class CajaGenerica <T> {  
    private T contenido;  
  
    public void guarda(T objeto) {  
        contenido = objeto;  
    }  
  
    public T saca() {  
        T auxiliar = contenido;  
        contenido = null;  
        return auxiliar;  
    }  
  
    public boolean isVacia() {  
        return contenido == null;  
    }  
}
```

```
public class Comida {  
}
```

```
public class Ropa {  
}
```

Código disponible en
U4.P9.CajaGenerica

```
public class PruebaCajaGenerica {  
    public static void main(String[] args) {  
        CajaGenerica<Ropa> cajaRopa = new CajaGenerica<>();  
        CajaGenerica<Comida> cajaComida = new CajaGenerica<>();  
  
        Ropa pantalon = new Ropa();  
        Comida pan = new Comida();  
  
        cajaRopa.guarda(pantalon);  
        pantalon = cajaRopa.saca();  
        cajaComida.guarda(pan);  
  
        cajaRopa.guarda(pan);  
    }  
}
```

Si intentamos guardar
otra cosa, el compilador
nos da un error

UD 4 - TIPOS AVANZADOS DE DATOS

3 - Genéricos

- ▶ Con esta técnica conseguimos que la clase **CajaGenerica** reciba como parámetro el tipo <T> del objeto a almacenar.
- ▶ El ejemplo que hemos creado es muy sencillo y es el que debéis aprender. Así que se os podría pedir que crearais la clase Bicicleta (puede llevar Persona), o la clase KinderSorpresa (puede contener Jugete) o la clase Camion (puede llevar Carga)...
- ▶ Usando genéricos podemos implementar cualquier estructura de datos:
 - ▶ Pilas
 - ▶ Colas (simples, de doble cabeza, de prioridades...)
 - ▶ Listas (simples, doblemente enlazadas...)
 - ▶ Árboles (binarios, ternarios, N-arios...)
 - ▶ Bosques (colección de árboles)
 - ▶ Grafos (direccionales, no direccionales)
 - ▶ Redes...

La creación de estructuras de datos es un tarea compleja y se sale del ámbito de este curso. No obstante, se deja en el repositorio una implementación básica de una ListaEnlazada a modo de ejemplo (U4.P10.ListaEnlazada)




Realizamos los ejercicios 21 y 22 del boletín de problemas

UD 4 - TIPOS AVANZADOS DE DATOS

3 - Genéricos

- **Creación de métodos genéricos en clases de utilidades**
- El otro ejemplo de uso que vamos a estudiar es la creación de métodos que admitan tipos genéricos y que sirvan como utilidades estáticas.
- Podríamos crear un método estático que reciba como parámetro una colección de cualquier tipo de dato y la imprima en la pantalla o bien nos diga que la colección está vacía si fuera el caso. Observa el siguiente código:

```
public static void main(String[] args) {  
    List<String> lista = new ArrayList<>();  
    lista.add("Hola");  
    lista.add("Caracola");  
    lista.add("Adios");  
    lista.add("Caracol");  
    UtilidadGenerica.imprimeColeccion(lista);  
  
    System.out.println("");  
    Set<Integer> conjunto = new HashSet<>();  
    UtilidadGenerica.imprimeColeccion(conjunto);  
}
```



```
Elemento 1 = Hola  
Elemento 2 = Caracola  
Elemento 3 = Adios  
Elemento 4 = Caracol  
  
Coleccion vacía
```

UD 4 - TIPOS AVANZADOS DE DATOS

3 - Genéricos

- El código de esta utilidad es:

```
public class UtilidadGenerica {  
    public static <T> void imprimeColeccion (Collection<T> coleccion) {  
        if (coleccion.isEmpty())  
            System.out.println("Coleccion vacía");  
        else {  
            Iterator iter = coleccion.iterator();  
            int i = 1;  
            while(iter.hasNext()) {  
                System.out.println("Elemento "+i+" = "+iter.next());  
                i++;  
            }  
        }  
    }  
}
```

Poniendo <T> justo antes del tipo devuelto por el método indicamos que dicho método va a usar un tipo genérico T

Fíjate que la clase no es genérica pero el método sí

Código disponible en
U4.P11.MetodoGenerico

- De este modo **conseguimos independizar el algoritmo** de recorrido e impresión en pantalla de una colección de datos sin importar el tipo de colección que se trate o el tipo de dato que contenga.

UD 4 - TIPOS AVANZADOS DE DATOS

3 - Genéricos

- ▶ La clase **java.util.Collections** sigue esta técnica de crear métodos estáticos genéricos para trabajar con colecciones. Esta clase implementa un conjunto de utilidades genéricas y estáticas como:
 - ▶ Realizar la búsqueda binaria en una lista ordenada.
 - ▶ Invertir el orden de una lista.
 - ▶ Copiar una lista en otra.
 - ▶ Ordenar o desordenar una lista.
 - ▶ Rellenar una lista de objetos...
- ▶ La documentación oficial de la clase es:
<https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html>
 - ▶ Si abrimos el enlace y observamos un poco, nos deben llamar la atención ciertas notaciones... Vamos a verlas, sobre todo para que podáis consultar la documentación oficial sin “asustaros”.

COLLECTIONS

UD 4 - TIPOS AVANZADOS DE DATOS

3 - Genéricos

- Se utilizan expresiones para limitar los tipos genéricos que podemos usar. Veamos las más comunes:

- **<? super T>** indicaría al compilador que solo se aceptan tipos T y sus superclases. Por ejemplo:

```
List <? super Felino> superFelinos = new ArrayList<>();
```

```
superFelinos.add( new Felino() );
```

```
superFelinos.add( new Mamifero() );
```

```
superFelinos.add( new Object() );
```

```
superFelinos.add( new Tigre() ); ERROR
```

<? super Felino> limita los tipos posibles a objetos de tipo Felino y sus superclases

- **<? extends T>** indicaría al compilador que solo se aceptan tipos T y sus subclases. Por ejemplo:

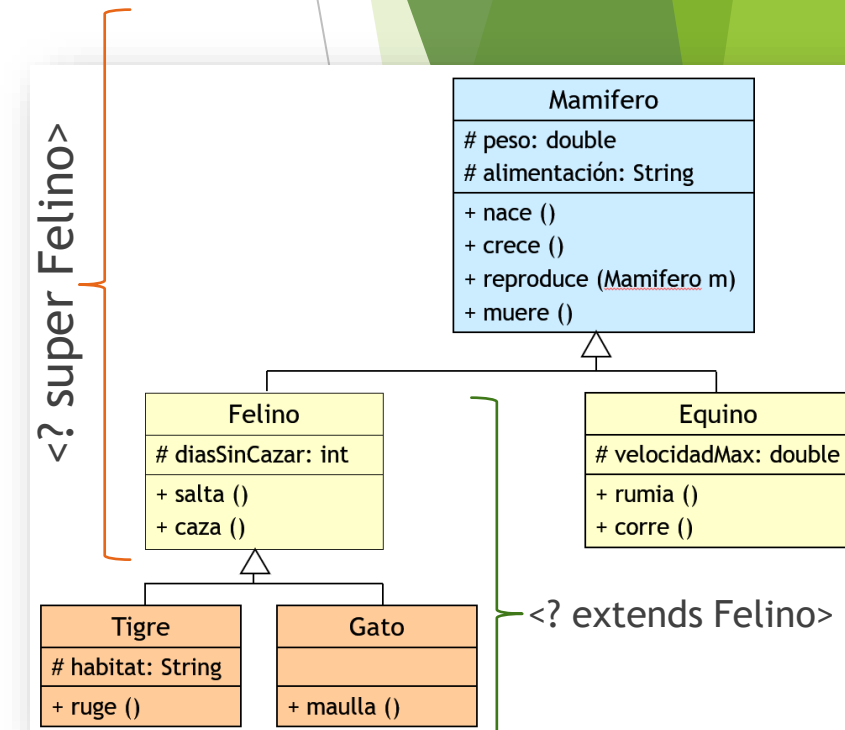
```
List <? extends Felino> felinosZoo = new ArrayList<>();
```

```
felinosZoo.add( new Felino() );
```

```
felinosZoo.add( new Tigre() );
```

```
felinosZoo.add( new Equino() ); ERROR
```

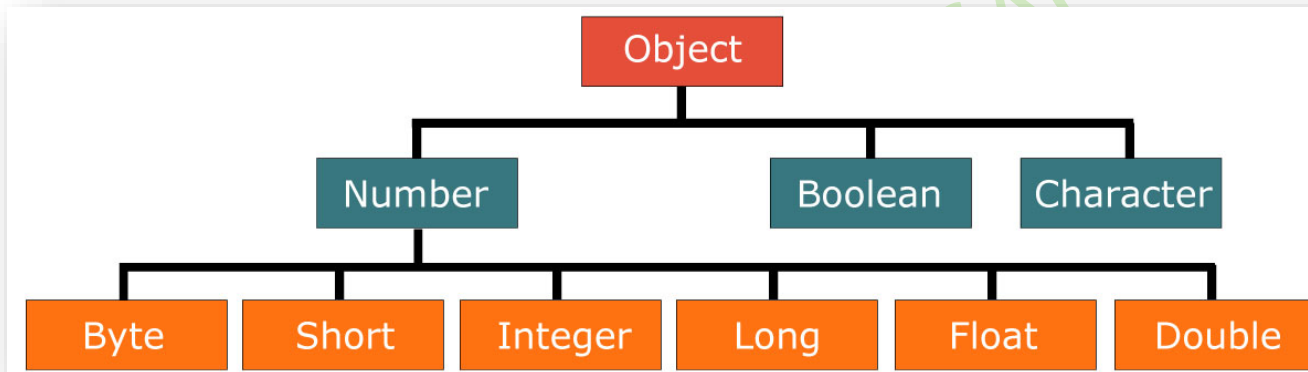
<? extends Felino> limita los tipos posibles a objetos de tipo Felino y sus subclases



UD 4 - TIPOS AVANZADOS DE DATOS

3 - Genéricos

- ▶ Si recordamos la jerarquía de clases Wrapper y para consolidar las dos expresiones que acabamos de estudiar, pregúntate:
 - ▶ ¿cómo crearías un conjunto que solo pudiera admitir tipos Long y sus superclases.
 - ▶ ¿Y cómo crearíamos una lista que solo pudiera contener cualquier tipo de número?



- ▶ **Además de <T> se usan otras letras.** Realmente podemos usar las que queramos, sin embargo, las que se usan habitualmente son:
 - ▶ <E> para tipos genéricos que sirvan como elementos de una colección.
 - ▶ <N> para tipos genéricos que sean un Number
 - ▶ <K, V> para las claves y valores de un Map respectivamente.

UD 4 - TIPOS AVANZADOS DE DATOS

3 - Genéricos

- ▶ **Conclusiones del uso de genéricos para este módulo profesional**
- ▶ Dado que guardar un Object en una colección es demasiado general y posibilita que se comentan errores, debemos usar los **genéricos para restringir el tipo de datos que vamos a almacenar**, obteniendo un código más fiable y seguro. Este es el caso más frecuente de utilización de los genéricos y tenemos que saber manejarlos.

CONCLUSION

- ▶ Por otro lado, tenemos que saber crear una **clase genérica sencilla** que almacene un objeto a modo del ejemplo de la CajaGenerica.
- ▶ La creación de estructuras de datos más complejas y la utilización de métodos genéricos **se sale del ámbito de este módulo profesional**. Sólo quiero que sepáis que existen estas técnicas.



UD 4 - TIPOS AVANZADOS DE DATOS

Anexo I. Fechas y horas

► Anexo I. Fechas y horas

- Desde la versión 1.0, Java disponía de las clases para representar “el tiempo”. Eran las clases: *java.util.Calendar* y *java.util.Date*.
- Sin embargo, estas clases presentaban varios problemas:
 - Eran engorrosas de utilizar porque el diseño de las clases no era bueno.
 - No contemplaban las zonas horarias y los desarrolladores tenían que añadir código extra para manejar las distintas franjas horarias del planeta Tierra.
 - No eran seguras en entornos multihilo (no eran thread-safe)
- Hasta la versión 8 de Java no se han modificado estas clases, así que todavía hay mucho código escrito con las versiones antiguas.
- Nosotros vamos a pasearnos por la nueva API de fechas y horas que está en el paquete *java.time*



UD 4 - TIPOS AVANZADOS DE DATOS

Anexo I. Fechas y horas

- ▶ El paquete *java.time* incluye muchas clases pero las básicas son:
 - ▶ **LocalDate**: representa solo fechas (sin la hora) y nos facilita su manejo para declararlas, sumar y restar fechas y compararlas.
 - ▶ **LocalTime**: representa solo horas, sin ninguna fecha asociada, pudiendo así compararlas, sumar o restar tiempo a las mismas...
 - ▶ **LocalDateTime**: como puedes suponer, es una combinación de las dos anteriores, que permite hacer lo mismo con fechas y horas simultáneamente.
 - ▶ **Instant**: se usa para almacenar un instante determinado en el tiempo o “timestamp” (se podría traducir como “registro o marca temporal”) en la hora UTC. Esta clase permite almacenar una fecha/hora con precisión de nanosegundos. Es muy útil para manejar instantes temporales de manera neutra permitiendo así el “intercambio de instantes” entre distintos sistemas.
- ▶ Cabe destacar que los constructores de estas clases son **private** y que se usan métodos estáticos a modo de “fábrica” de objetos.
- ▶ Para saber más: <https://www.campusmvp.es/recursos/post/como-manejar-correctamente-fechas-en-java-el-paquete-java-time.aspx>⁶³



Código de ejemplo en
U4.P12.FechaHora

UD 4 - TIPOS AVANZADOS DE DATOS

Cierre de la unidad

- ▶ En un aplicación **necesitamos procesar colecciones de datos con potencia y flexibilidad** y vemos que los arrays se nos “quedan cortos”.
- ▶ El framework **Collection es un estándar mundialmente utilizado** que nos brinda una forma de manipular y procesar colecciones de objetos.
- ▶ Es importante tomar conciencia de cómo este framework utiliza la herencia, las interfaces y el polimorfismo para resolver el problema, **generando un código flexible y preparado para adaptarse al cambio**.
- ▶ Además, el uso del framework con la **notación de genéricos** nos permite restringir el tipo de un elementos que almacena una colección, obteniendo un código más seguro, fiable y legible.
- ▶ Para “digerir” este framework hemos tenido que tocar además otros **contenidos secundarios**, aunque no menos importantes: casting de referencias de objetos, clases wrappers, interfaz Comparable, funciones de hashing... Todo seguirá usándose en las próximas unidades.

Confía en la
**CALIDAD DE TUS
CONOCIMIENTOS,**
no en la cantidad



UD 4 - TIPOS AVANZADOS DE DATOS

