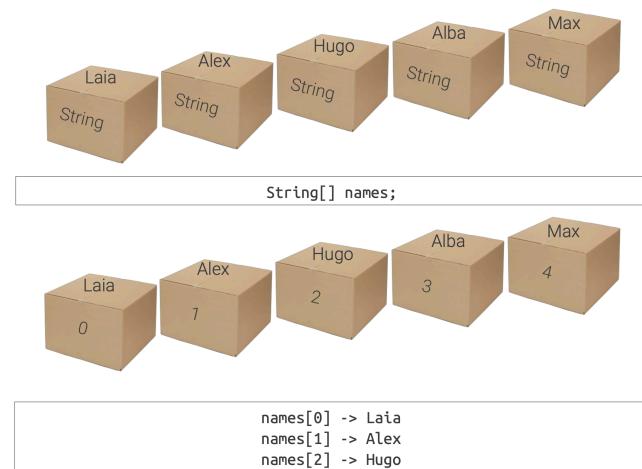


CONTROL DE EXCEPCIONES Y COLECCIONES DE DATOS I

ARRAYS AND COLLECTIONS

ARRAYS AND COLLECTIONS

Arrays



Sabemos que la mejor manera de almacenar múltiples datos del mismo tipo es usar arrays. Los arrays son un grupo de celdas que almacenan valores, y cada valor se almacena de forma separada, pero el array se trata como si fuera una sola variable.

Cada elemento en el array tiene un índice, un número entero que comienza en cero e indica la posición que ocupa el elemento en el array.

El índice permite acceder a cada elemento del array de forma individual.

Array Limitations

```
String[] names = new String[5];
```

Los arrays tienen algunas limitaciones. La primera es que tenemos que saber el número de elementos que tendrá en el momento de inicializarlo.

Y una vez que hemos inicializado un array con un número específico, no podemos añadir o eliminar elementos.

Esto significa que si intentamos acceder a un índice que no existe, porque supera el tamaño del array, se producirá un error en tiempo de ejecución y la aplicación se detendrá de forma inesperada.

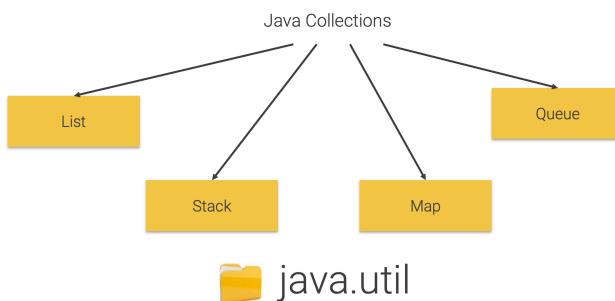


```
String[] names = new String[5];  
names[10] = "Matt";
```



```
String[] names = new String[5];
```

Java Collections



`add(E element)`

`iterator()`

`remove(E element)`

Otra limitación es que cuando comenzamos a rellenar con valores el array no podemos añadir elementos adicionales sino que tendremos que reemplazar alguno de los existentes y cuando eliminemos valores se quedarán los huecos en el array y tendremos que cambiar la posición de los elementos manualmente para eliminar estos huecos.

Estas limitaciones son el motivo por el cual Java introdujo lo que se conoce como colecciones. Las colecciones son simplemente un grupo de clases e interfaces que Java nos ofrece para trabajar con agrupaciones de elementos. Todas ellas se encuentran en el paquete `java.util`. Las colecciones nos permitirán trabajar con agrupaciones de elementos cuyo tamaño sea variable.

`add(E element)`

`iterator()`

Dispone de distintos elementos, como add, remove, iterator, etcétera, para añadir, eliminar, recorrer los elementos de la colección.

`String`

`Double`

`Car`

`boolean`

~~int~~

`Integer`

`StringBuilder`

`Book`

~~double~~

Es importante que entiendas que en una colección podremos almacenar objetos de cualquier tipo, o bien predefinidos por el lenguaje, o bien creados por nosotros, pero nunca tipos primitivos de datos.

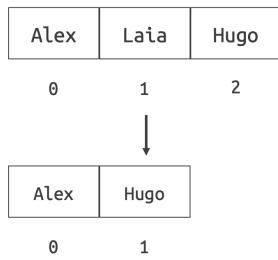
ArrayList: Example of a collection

Una de las colecciones más conocidas es `ArrayList`. La veremos en detalle más adelante. Ahora la usaremos como ejemplo de colección Java para esbozar la potencia de usar colecciones en nuestras aplicaciones.



Podremos inicializar una colección sin indicar el número de elementos que contendrá. Con la instrucción que ves en la captura hemos creado un `ArrayList` de strings, denominado `Names`, vacío, sin elementos. Para añadir un elemento a una colección, podremos usar el método `Add`, especificando como argumento el elemento a añadir.

```
ArrayList<String> names = new ArrayList<>();
names.add("Alex");
names.add("Laia");
names.add("Hugo");
names.remove("Laia");
```

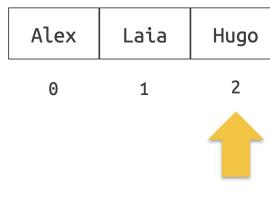


En este momento Names tiene tres elementos, y podríamos seguir añadiendo más usando el método Add. Usando el método Remove, podríamos eliminar de la colección el elemento especificado como argumento, además de reorganizar el resto de elementos para eliminar los huecos. Añadir y eliminar son las operaciones básicas.

Cualquier colección en Java tiene muchas operaciones más que iremos descubriendo.

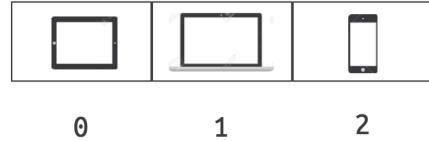
```
ArrayList<String> names = new ArrayList<>();
names.add("Alex");
names.add("Laia");
names.add("Hugo");

Iterator iterator = names.iterator();
while (iterator.hasNext()){
    System.out.print(iterator.next()+" ");
}
```



Toda colección dispone de un iterador, esto es un puntero, que con ayuda de un bucle nos permitirá recorrer la colección elemento a elemento. Y recuerda que una colección nos permitirá almacenar objetos de cualquier tipo, o bien predefinidos por el lenguaje, o bien creados por nosotros.

```
ArrayList<Producto> productos= new ArrayList<>();
productos.add(new Producto('Tablet',195.99));
productos.add(new Producto('PC',485));
productos.add(new Producto('Smartphone',95.99));
```



LISTS**LIST**

```

Module java.base
Package java.util
Interface List<E>

Type Parameters:
E - the type of elements in this list

All Superinterfaces:
Collection<E>, Iterable<E>

All Known Implementing Classes:
AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector

```

Una lista en Java permite trabajar con una colección ordenada de elementos, también se conoce como secuencia. Una lista en Java es una interfaz que se comporta de forma muy similar a un array. Mantiene el orden de inserción, permite obtener y modificar los valores de sus elementos a partir de su posición, puede almacenar valores nul y valores duplicados.

La diferencia con respecto a los arrays es que al ser una colección, su tamaño puede cambiar y por tanto, permite añadir y borrar elementos. Puedes consultar la documentación de Oracle sobre esta interfaz en [docs.oracle.com](https://docs.oracle.com/javase/8/docs/api/java/util/List.html).

Some of the List's methods

Comprobarás que dispone de los métodos básicos de cualquier colección Java, como son add, remove, para añadir o eliminar elementos. Y que a estos métodos, la interfaz list, añade otros que permiten manipular sus elementos por la posición que ocupan en la lista, como son contains, para consultar si el elemento especificado está en la lista o no, size, para saber cuántos elementos contiene, add, remove, para añadir o eliminar un elemento dado su índice, get y set, para obtener o actualizar el valor de un elemento dado su índice.

La clase más popular que implementa la interfaz List es ArrayList. Comencemos con ella.

ARRAYLIST

```

Module java.base
Package java.util
Class ArrayList<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.ArrayList<E>

Type Parameters:
E - the type of elements in this list

All Implemented Interfaces:
Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:
AttributeList, RoleList, RoleUnresolvedList

```

Un ArrayList es una clase que implementa la interfaz List. Es como un Array, pero con potentes métodos que permiten gestionar los elementos del array de forma simple.

Creating an ArrayList

```
ArrayList<String> names = new ArrayList<String>();
```

```
ArrayList<String> names = new ArrayList<String>();
```

```
ArrayList<String> names = new ArrayList<>(); //names is an empty ArrayList
```

Declaration and initialization

```
ArrayList<String> anotherArrayList; //anotherArrayList is null  
...  
anotherArrayList= new ArrayList<>(); // Now anotherArrayList is an empty ArrayList
```

Initialization

Declaration

```
ArrayList<String> anotherArrayList; //anotherArrayList is null  
...  
anotherArrayList= new ArrayList<>(); // Now anotherArrayList is an empty ArrayList
```

Is null

Para declarar e inicializar un ArrayList usamos la instrucción que ves en la captura. Cuando creamos un ArrayList hay que informar a Java el tipo de dato que va a almacenar. Esto se indica entre los símbolos mayor que y menor que. En el ejemplo que ves en la captura se ha creado un ArrayList de objetos string.

Observa que la palabra String aparece dos veces, en el lado izquierdo del igual y en el lado derecho del igual. ¿Piensas que la segunda vez que aparece es redundante? Pues así es. Desde Java versión 7 no es necesario indicar el tipo de dato a la derecha, tal como ves en la captura.

Esta secuencia declara un ArrayList de objetos String y lo inicializa en la misma línea. Este ArrayList no tiene elementos. Está vacío.

Podemos también declarar una variable de tipo arraylist, como ves en la captura, y luego inicializarla en otra línea de código.

Mientras esta variable no haya sido inicializada, su valor es null, como la de cualquier otro objeto que no haya sido inicializado.

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> names = new ArrayList<>();
    }
}
```

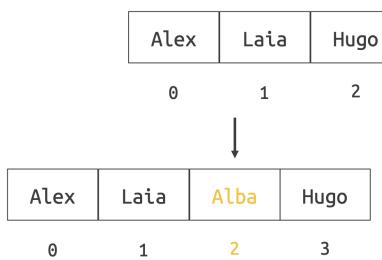
ArrayList pertenece al paquete java.util que no se importa implícitamente como el paquete Java.lang. Por lo tanto, tendremos que añadir la sentencia import para poder utilizar objetos de la clase ArrayList.

Adding elements to an ArrayList

```
ArrayList<String> names = new ArrayList<>();
names.add("Alex");
names.add("Laia");
names.add("Hugo");
```



```
ArrayList<String> names = new ArrayList<>();
names.add("Alex");
names.add("Laia");
names.add("Hugo");
names.add(2, "Alba");
```



Bien, hemos dicho que ArrayList es un Array dinámico, cuyo tamaño puede variar, así que vamos a empezar viendo cómo añadir elementos. Para añadir un elemento a un ArrayList usamos el método add, especificando como argumento el elemento añadir.

Debemos recordar que el tipo de dato del elemento debe coincidir con el tipo de dato que declaraste que iba a contener tu ArrayList. ¿Qué ha ocurrido? Hemos añadido tres elementos, tres objetos de tipo String a nuestro ArrayList, igual que en los Arrays el primer elemento de un ArrayList se almacena en la posición 0.

El método Add puede tener dos argumentos como ves en pantalla. El primer argumento indica en qué posición se quiere añadir el valor especificado como segundo argumento. Cuando un valor se añade a una posición que es ocupada por otro elemento, el elemento se desplaza a la derecha para dejar sitio al nuevo valor añadido.

Cuando se añade un elemento en una posición determinada, lo que hace Java es crear un nuevo array e inserta todos sus elementos en las posiciones siguientes a la posición que se ha especificado.

NullPointerException

```
ArrayList<String> names;
names.add("Alex");
```

NullPointerException

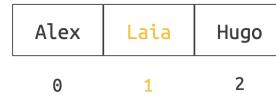
Antes de seguir, fíjate en una cosa importante. Si la variable ArrayList no ha sido inicializada, cuando llamemos a cualquiera de los métodos de la clase ArrayList, se producirá un error en tiempo de ejecución y el programa terminará.

Si no inicializamos la variable, su valor es null, e invocar a cualquiera de sus métodos produce una excepción del tipo `nullPointerException`.

Accessing elements of an ArrayList

```
ArrayList<String> names = new ArrayList<>();
names.add("Alex");
names.add("Laia");
names.add("Hugo");

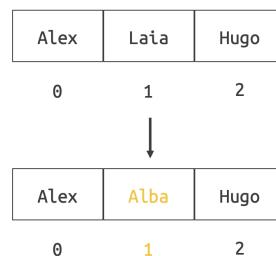
System.out.println(names.get(1)); //Prints Laia
```



Usamos el método `get` para acceder a un elemento en concreto, indicando su posición.

Modifying the elements of an ArrayList

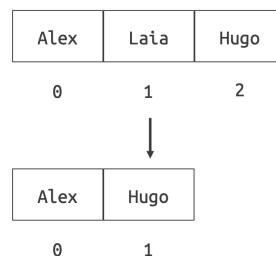
```
ArrayList<String> names = new ArrayList<>();
names.add("Alex");
names.add("Laia");
names.add("Hugo");
names.set(1,"Alba");
```



Podemos modificar el valor de un elemento de un `ArrayList` con el método `SET`. Este método tiene dos argumentos. El primer argumento indica la posición a modificar y el segundo argumento el nuevo valor.

Deleting the elements of an ArrayList

```
ArrayList<String> names = new ArrayList<>();
names.add("Alex");
names.add("Laia");
names.add("Hugo");
names.remove(1);
```



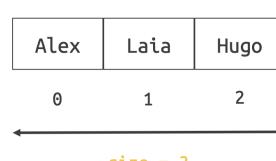
Para eliminar un elemento de un `ArrayList` usamos el método `REMOVE`. Este método tiene un argumento donde especificamos la posición a eliminar. Como parámetro de entrada del método Remove, también podemos especificar el valor a eliminar.

Loops & ArrayList

```
ArrayList<String> names = new ArrayList<>();
names.add("Alex");
names.add("Laia");
names.add("Hugo");

for (int i=0; i<names.size(); i++){
    System.out.print(names.get(i)+" ");
}

//Prints
Alex Laia Hugo
```



Usaremos bucles para recorrer los elementos de un `ArrayList`, de forma equivalente a como lo hacíamos con un `Array`. El método `Get` nos permitirá acceder a cada elemento según su posición. Y el método `Size`, que devuelve el número de elementos que tiene la colección, nos permitirá establecer la condición de salida del bucle.

Es el equivalente al length para Arrays.

Un ArrayList permite que lo recorramos de una forma más simple. Es el código que ves en el lado derecho de la pantalla.

```
ArrayList<String> names = ...
```

```
for (int i=0; i<arrays.size(); i++){
    System.out.print(names.get(i)+" ");
}
```

```
//Prints  
Alex Laia Hugo
```

```
ArrayList<String> names = ...
```

```
for (String element: names){
    System.out.print(element+" ");
}
```

```
//Prints  
Alex Laia Hugo
```

```
ArrayList<String> names = ...
```

```
for (int i=0; i<names.size(); i++){
    System.out.print(names.get(i)+" ");
}
```

```
//Prints  
Alex Laia Hugo
```

```
ArrayList<String> names = ...
```

```
for (String element: names){
    System.out.print(element+" ");
}
```

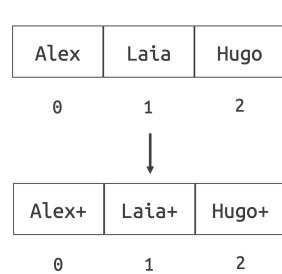
```
//Prints  
Alex Laia Hugo
```

Consiste en escribir el tipo de dato de los elementos que contiene el ArrayList, seguido del nombre de la variable, por ejemplo, element, que tomará el valor de cada uno de los elementos de la colección, seguido de dos puntos, y por último, el identificador de la variable de tipo ArrayList.

Advanced Loops & ArrayList

Pero hay otra manera de recorrer los elementos de una ArrayList, que solo deberíamos usar en las situaciones donde al mismo tiempo que estamos recorriendo la colección queremos modificarla para eliminar algún elemento o modificar el valor de algún elemento. Se trata de usar ListIterator.

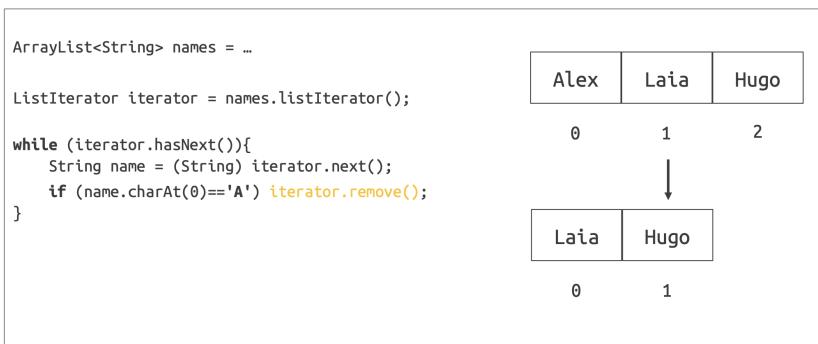
```
ArrayList<String> names = ...
ListIterator iterator = names.listIterator();
while (iterator.hasNext()){
    String name = (String) iterator.next();
    iterator.set(name+"+");
}
```



En el ejemplo que ves en pantalla, mientras se recorre la colección usando el ListIterator, se actualiza el valor de sus elementos. Para poder usar un iterador, primero tendremos que declararlo llamando al método listIterator de la colección.

Un iterador es como un puntero. Con el método Next iremos moviendo el puntero al siguiente elemento de la colección.

El método HasNext retornará True siempre que hayan elementos para recorrer. Por este motivo será la condición a utilizar en el bucle. Un ListIterator dispone del método set para actualizar con un nuevo valor el elemento actual. En el ejemplo que ves en pantalla, cada stream de la colección se actualiza introduciendo un símbolo más al final de la cadena de texto.



Un ListIterator también dispone del método remove para eliminar el elemento actual. En el ejemplo que ves en pantalla, si el elemento de la colección comienza por A, se eliminará de la colección. Es recomendable echar un vistazo a la documentación de Oracle sobre la clase ArrayList y su lista completa de métodos.

STACK

Module java.base
Package java.util
Interface List<E>

Type Parameters:
E - the type of elements in this list

All Superinterfaces:
Collection<E>, Iterable<E>

All Known Implementing Classes:
AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector

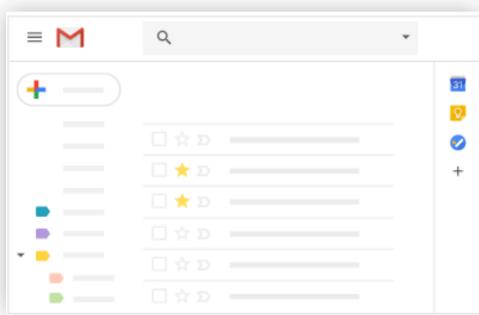
Otra clase popular que implementa la interfaz LIST es STACK.

Module java.base
Package java.util
Class Stack<E>

java.lang.Object
java.util.AbstractCollection<E>
java.util.AbstractList<E>
java.util.Vector<E>
java.util.Stack<E>

All Implemented Interfaces:
Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Una stack representa una pila de objetos, LIFO, LASTIN, FIRSTOUT, que significa que el último elemento introducido en la pila será el primer elemento en salir de ella. Como una pila de platos, el último plato que añades será el primer plato que coges. Igual que ArrayList, una stack internamente usa un array para almacenar los elementos.



Un ejemplo de cuando nos puede ser útil emplear una colección de tipo stack sería cuando desarrollamos algo parecido al sistema de correo electrónico. Cuando el servidor de correo electrónico recibe un nuevo email, debe añadir este email arriba en la pila de emails, de forma que el usuario pueda leer el email más reciente primero.

Creating a Stack

```
Stack<String> newsFeed = new Stack();
```

Para declarar e inicializar una stack usamos la instrucción que ves en pantalla. Igual que con ArrayList debes especificar el tipo de objeto que almacenarás en la colección, en este caso strings, y tendrás que añadir la sentencia import para poder utilizarla.

```
import java.util.Stack;

public class Main {

    public static void main(String[] args) {
        Stack<String> newsFeed = new Stack();
    }
}
```

Some of the List's methods

La clase Stack, como implementa la interfaz List, dispone de sus métodos de forma equivalente a lo que hemos visto con la clase ArrayList. Y añade los que ves en pantalla para permitir gestionar los elementos como una pila LIFO.

Adding elements to a Stack

```
Stack<String> newsFeed = new Stack();
newsFeed.push("Morning news");
newsFeed.push("Afternoon news");
newsFeed.push("Evening news");
```

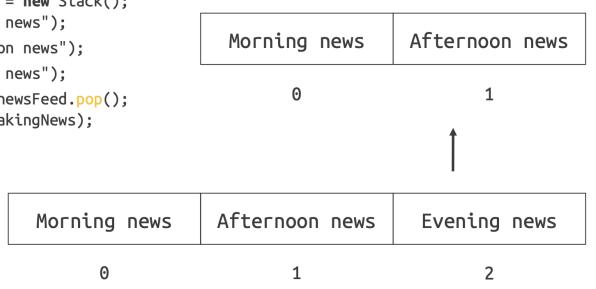
Morning news	Afternoon news	Evening news
0	1	2

Para añadir un elemento a una Stack LIFO usamos el método Push, especificando como argumento el elemento a añadir.

Deleting elements from a Stack

```
Stack<String> newsFeed = new Stack();
newsFeed.push("Morning news");
newsFeed.push("Afternoon news");
newsFeed.push("Evening news");
String breakingNews = newsFeed.pop();
System.out.println(breakingNews);

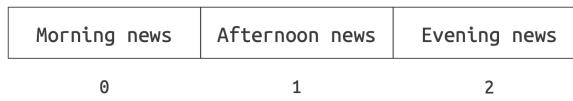
//Prints
Evening news
```



Accessing elements from a stack

```
Stack<String> newsFeed = new Stack();
newsFeed.push("Morning news");
newsFeed.push("Afternoon news");
newsFeed.push("Evening news");
String breakingNews = newsFeed.peek();
System.out.println(breakingNews);

//Prints
Evening news
```



Y para eliminar un elemento de una stack LIFO usamos el método POP.

Y usaremos el método PICK si queremos acceder al último elemento de la stack sin eliminarlo. Es recomendable echar un vistazo a la documentación de Oracle sobre la clase Stack y su lista completa de métodos.

QUEUES

QUEUE



FIFO
First In - First Out

Otro tipo de colección usado en Java son las queues o colas, como sugiere su nombre representa una línea de elementos situados uno detrás de otro.

A diferencia de la Stack o pila es FIFO, First In, First Out, donde el primer elemento que se añade a la cola es el primer elemento al que se accede o se elimina.

Module: java.base
Package: java.util
Interface Queue<E>

Type Parameters:
E - the type of elements held in this queue

All Superinterfaces:
Collection<E>, Iterable<E>

All Known Subinterfaces:
BlockingQueue<E>, BlockingQueue<E>, Deque<E>, TransferQueue<E>

All Known Implementing Classes:
AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, SynchronousQueue

Una queue en Java es una interfaz. Puedes consultar la documentación de Oracle sobre esta interfaz y sus métodos.

Some of the Queues's methods

Comprobarás que dispone de los métodos básicos de cualquier colección Java, como son Add y Remove, y que a estos métodos la interfaz Queue añade otros que permiten operaciones de inserción y extracción.

```
Module java.base
Package java.util
Interface Queue<E>

Type Parameters:
E - the type of elements held in this queue

All Superinterfaces:
Collection<E>, Iterable<E>

All Known Subinterfaces:
BlockingDeque<E>, BlockingQueue<E>, Deque<E>, TransferQueue<E>

All Known Implementing Classes:
AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, LinkedListTransferQueue, PriorityBlockingQueue, PriorityQueue, SynchronousQueue
```

La clase más popular que implementa la interfaz Queue es LinkedList. Veámosla.

LINKEDLIST

```
Module java.base
Package java.util
Class LinkedList<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.AbstractSequentialList<E>
                java.util.LinkedList<E>

Type Parameters:
E - the type of elements held in this collection

All Implemented Interfaces:
Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, List<E>, Queue<E>
```

Una LinkedList es una clase que en su jerarquía extiende lista pero adicionalmente implementa la interfaz Queue. Entonces tenemos los métodos que tiene el ArrayList y además los que tienen las colas.

Creating a LinkedList

```
import java.util.LinkedList;

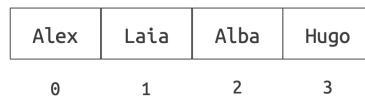
public class Main {

    public static void main(String[] args) {
        LinkedList<String> names = new LinkedList<>();
    }
}
```

Importamos, declaramos e inicializamos un LinkedList en nuestro proyecto de la misma forma que lo hacemos con un ArrayList.

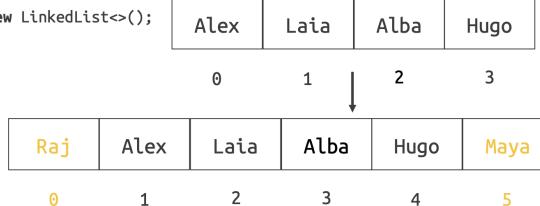
Adding elements to a LinkedList

```
LinkedList<String> names = new LinkedList<>();
names.add("Alex");
names.add("Laia");
names.add("Hugo");
names.add(2, "Alba");
```



```
LinkedList<String> names = new LinkedList<>();
names.add("Alex");
names.add("Laia");
names.add("Hugo");
names.add(2, "Alba");

names.addFirst("Raj");
names.addLast("Maya");
```



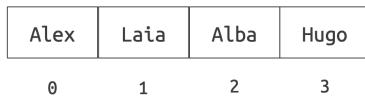
Y añadimos elementos de la misma forma utilizando el método Add.

O con los métodos adicionales de la interfaz Queue, AddFirst, AddLast, para añadir al principio o al final de la lista.

Retrieving elements from a LinkedList

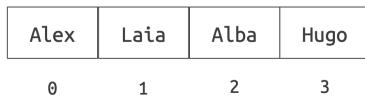
```
System.out.println("The LinkedList elements are:");
for (Iterator i = names.iterator(); i.hasNext();) {
    System.out.println(i.next());
}

//The LinkedList elements are
//Alex
//Laia
//Alba
//Hugo
```



Podemos recorrer los elementos de una Linked List utilizando un iterador.

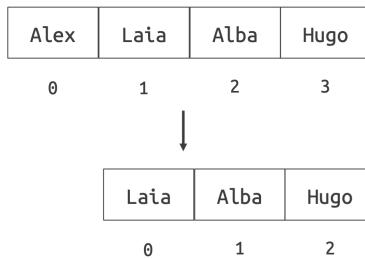
```
System.out.println(names.indexOf("Laia")); //1
System.out.println(names.indexOf("Lucas")); //-1
System.out.println(names.get(2)); //Alba
```



Con el método de la interfaz ListIndexOf podemos consultar si un elemento está en la lista, nos devolverá su índice o menos uno en el caso de que no esté en la lista. Con el método Get y utilizando el índice como parámetro obtendremos el elemento.

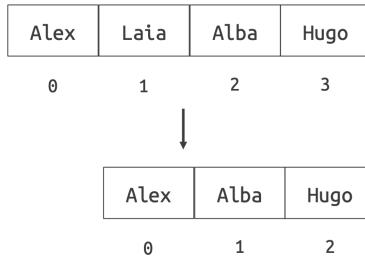
Deleting elements from a LinkedList

```
String name = names.remove(); //Alex
```



Para eliminar elementos de una Linked List podemos usar el método Remove. Si no indicamos parámetros de entrada en el método, se borrará el primer elemento de la lista.

```
String name = names.remove(1); //Laia
```



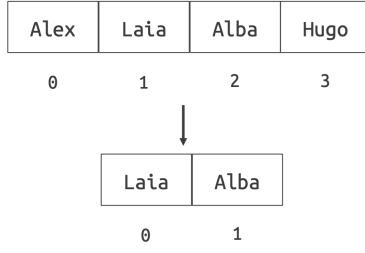
Si como parámetro especificamos un entero, se eliminará el elemento que ocupe esta posición.

```
boolean deleted = names.remove("Raj"); //false
```



Y si especificamos un objeto, se eliminará la primera ocurrencia del elemento especificado si lo encuentra.

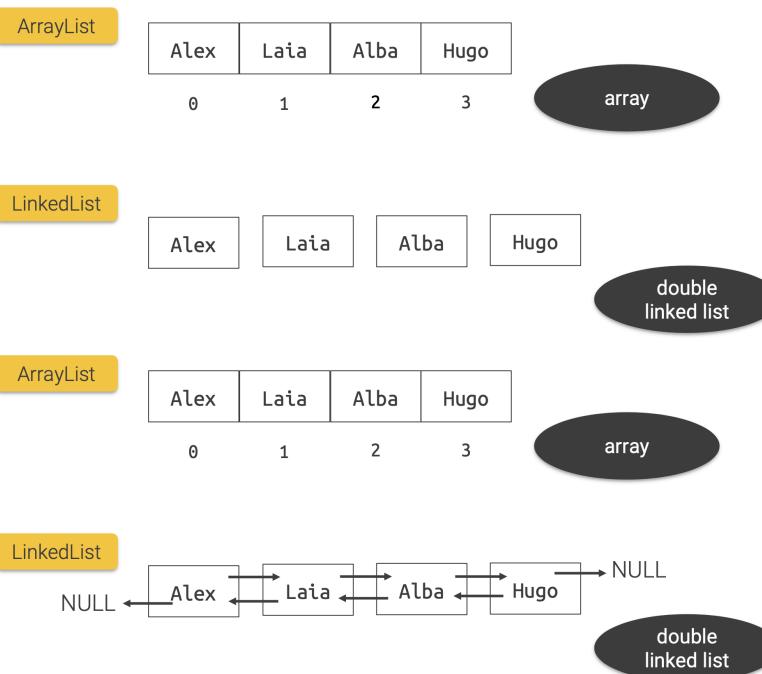
```
String name1 = names.removeFirst(); //Alex
String name2 = names.removeLast(); //Hugo
```



Y adivina. Igual que LinkedList, por implementar la Interface Queue, tenía los métodos addFirst y addLast, también dispone de removeFirst y removeLast para eliminar el primer y último elemento de la lista. Es recomendable echar un vistazo a la documentación de Oracle sobre la clase LinkedList y su lista completa de métodos.

LINKEDLIST VS ARRAYLIST

Entonces, si un LinkedList es un ArrayList con más métodos ¿por qué no utilizar siempre LinkedList? Esta decisión dependerá del caso concreto en que nos encontremos, porque en unas situaciones un ArrayList será mejor y en otras ocasiones lo será un LinkedList. Y esto es debido a que ArrayList y LinkedList están implementadas de manera distinta.



Como sabemos, el ArrayList está implementado utilizando un Array. En cambio, una LinkedList se implementa utilizando una estructura de datos llamada Lista Dblemente Enlazada.

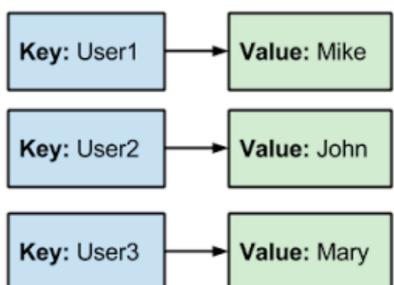
¿Qué significa esto? Pues que cada nodo de la lista contiene además del dato dos referencias, una al nodo anterior y otra al nodo siguiente.

Como consecuencia de estas implementaciones distintas, en una linked list podemos añadir elementos a la cabeza mucho más rápido, y también insertar o eliminar elementos en cualquier posición utilizando un iterador.

Sin embargo, una linked list ocupa más memoria que una arraylist debido a los punteros al anterior y posterior elemento. Pero la principal desventaja de una linked list frente a arraylist es que las operaciones de GET tardan bastante más. Por lo tanto, la recomendación es usar LinkedList en colecciones donde sean frecuentes las operaciones de añadir o eliminar elementos, ya que será mucho más rápido que si usas ArrayList. En el caso de que necesites una colección read-only, esto es, que apenas se modifique su número de elementos, mejor usar ArrayList.

MAPS

MAP



Los mapas son una colección que permite agrupar elementos a los que se accede mediante una clave, la cual debe ser única para los diferentes elementos de la colección. Un map en Java es una interfaz. Puedes consultar la documentación de Oracle sobre esta interfaz y sus métodos.

```

compact1, compact2, compact3
java.util
Interface Map<K,V>

Type Parameters:
K - the type of keys maintained by this map
V - the type of mapped values

All Known Subinterfaces:
Bindings, ConcurrentMap<K,V>, ConcurrentNavigableMap<K,V>, LogicalMessageContext, MessageContext,
NavigableMap<K,V>, SOAPMessageContext, SortedMap<K,V>

All Known Implementing Classes:
AbstractMap, Attributes,AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap,
Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints,
SimpleBindings, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

```

Some of the Map's methods

El método get recupera un valor asociado a una clave dada. El método put permite añadir una pareja de clave-valor. Asocia la clave especificada en el key con el valor especificado en el value. El método remove elimina la clave-valor referenciada por la clave proporcionada.

El método keys devuelve un objeto en numeración con las claves que contiene la colección map. El método values devuelve un objeto collection con los valores que contiene la colección map. El método entrySet obtiene una colección set de los elementos del map. La clase más popular que implementa la interface queda es HighStable. Echemos un vistazo.

HASHTABLE

```
Module java.base
Package java.util

Class Hashtable<K,V>

java.lang.Object
    java.util.Dictionary<K,V>
        java.util.Hashtable<K,V>

Type Parameters:
K - the type of keys maintained by this map
V - the type of mapped values

All Implemented Interfaces:
Serializable, Cloneable, Map<K,V>

Direct Known Subclasses:
Properties, UIDefaults

public class Hashtable<K,V>
extends Dictionary<K,V>
implements Map<K,V>, Cloneable, Serializable
```

Una Hashtable es una clase que implementa la interface map y que nos permite almacenar elementos con pares clave-valor.

Creating a HashTable

```
import java.util.Hashtable;

public class Main {
    public static void main(String[] args) {
        Hashtable<String, String> hashtable = new Hashtable<>();
    }
}
```

Como vemos en la captura, para declarar una Hashtable debemos especificar el tipo tanto de la clave como del objeto que queremos almacenar.

```
Hashtable<String, Integer> numbers
    = new Hashtable<String, Integer>();

numbers.put("one", 1);
numbers.put("two", 2);
numbers.put("three", 3);
```

En el ejemplo que ves en la captura se crea la hash table numbers, que usa los nombres de los números como claves.

```
Hashtable<String, String> example1 = new Hashtable<>();

Hashtable<Integer, String> example2 = new Hashtable<>();

Hashtable<String, Person> example3 = new Hashtable<>();

Hashtable<int, Double> example4 = new Hashtable<>();
```

Podemos utilizar tipos de datos diferentes tanto en la clave como en el valor. Siempre deberán ser objetos. Nunca tipos primitivos.

Keys on HashTable

Hashtable **keys** must be of a **type** that implements **comparable** interface
So they can be compared to secure that **keys are unique**

```
Hashtable<String, String> hashtable = new Hashtable<>();
Module: java.base
Package: java.lang
Class: String
java.lang.Object
java.lang.String
All Implemented Interfaces:
Serializable, CharSequence, Comparable<String>
```

En el tipo de dato de la key existe una restricción adicional. El tipo de dato debe permitir ser comparado con otro. Por este motivo, el tipo de dato es necesario que implemente la interface comparable. De este modo, las claves se pueden comparar para garantizar que son únicas.

En el ejemplo que ves en la captura, el tipo de dato key es String. La clase String implementa la interface comparable.

Adding elements to a HashTable

```
Hashtable<String, String> hashtable = new Hashtable<>();

hashtable.put("Key1", "Chaitanya");
hashtable.put("Key2", "Ajeet");
hashtable.put("Key3", "Peter");
hashtable.put("Key4", "Ricky");
hashtable.put("Key5", "Mona");
```

Addinn kay-value pairs

Para añadir elementos a una hash table, utilizamos el método put. El primer parámetro es la clave, el segundo es el valor.

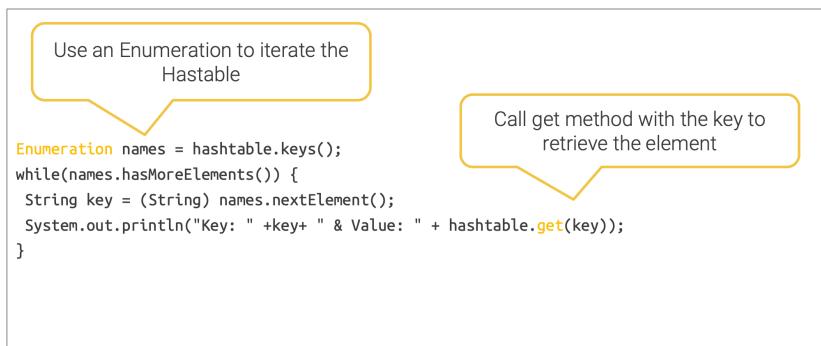
```
Hashtable<String, String> hashtable = new Hashtable<>();

hashtable.put("Key1", "Chaitanya");
hashtable.put("Key2", "Ajeet");
hashtable.put("Key3", "Peter");
hashtable.put("Key4", "Ricky");
hashtable.put(null, "Mona");
```

Si intentamos añadir un elemento con clave o valor null, dará error. Una hash table no permite almacenar ni claves ni valores null.

Iterating a HashTable

Para poder recorrer la colección, el método keys nos proporcionará un enumerador con el que poder iterar la colección.



Una hash table es más rápida que otras colecciones si podemos prever la cantidad de elementos que almacenará, y si además no eliminamos ni añadimos demasiados elementos. Por lo tanto, utiliza una Hashtable para almacenar grandes cantidades de información que sufrirán pocos cambios y deban ser accedidos mediante una clave única.

Consulta la documentación de Oracle sobre la clase Hashtable y su lista completa de métodos.

WORKING WITH COLLECTIONS

COLLECTIONS CLASS

java.util

Class Collections

java.lang.Object
java.util.Collections

```

public class Collections
extends Object

```

This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends.

The methods of this class all throw a `NullPointerException` if the collections or class objects provided to them are null.

The documentation for the polymorphic algorithms contained in this class generally includes a brief description of the *implementation*. Such descriptions should be regarded as *implementation notes*, rather than parts of the *specification*. Implementors should feel free to substitute other algorithms, so long as the specification itself is adhered to. (For example, the algorithm used by `sort` does not have to be a mergesort, but it does have to be *stable*.)

The "destructive" algorithms contained in this class, that is, the algorithms that modify the collection on which they operate, are specified to throw `UnsupportedOperationException` if the collection does not support the appropriate mutation primitive(s), such as the `set` method. These algorithms may, but are not required to, throw this exception if an invocation would have no effect on the collection. For example, invoking the `sort` method on an unmodifiable list that is already sorted may or may not throw `UnsupportedOperationException`.

This class is a member of the Java Collections Framework.

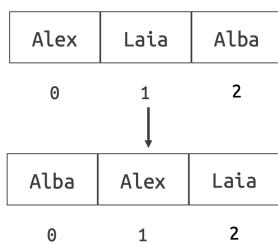
El framework de colecciones de Java proporciona la clase Collections. Es una clase famosa porque contiene métodos, todos ellos estáticos, que nos permiten trabajar con colecciones. Por ejemplo, para añadir una colección dentro de otra en una determinada posición o para obtener el valor mínimo o máximo de una determinada colección, invertir los elementos de una colección, etc.

Sort method

```
ArrayList<String> names = new ArrayList<String>();
names.add("Alex");
names.add("Laia");
names.add("Alba");

Collections.sort(names);

for (String name:names){
    System.out.println(name);
}
```



Como muestra de la potencia de esta clase, vamos a ver uno de sus métodos, el método Sort, que ordena de menor a mayor la colección introducida como parámetro de entrada en el método sobreescribiéndola. Es recomendable echar un vistazo a la documentación de Oracle sobre la clase Collections y su lista completa de métodos.

COLLECTIONS DEMO

Demo 1

Veamos ahora una demo de uso de las colecciones LinkedList, Map y Stack. En esta demo veremos cómo utilizar algunas colecciones de Java. Comencemos viendo cómo funciona la pila. En esta clase de demostración, creamos un objeto Stack. Lo llenamos de información con el método `fillData`, fijaros que algunos elementos están repetidos y luego realizamos varias operaciones. El método `peek` utilizado aquí nos ha devuelto el último elemento que se ha añadido a la pila a continuación mostramos todo el contenido de la pila veremos que está en el mismo orden en el que hemos añadido los elementos.

Luego la primera llamada al método `pop` nos devuelve el mismo elemento que la anterior llamada a `peek`, pero a siguientes llamadas a este método nos van devolviendo los elementos siguientes. Fijaros, esto es debido a que `pop` retorna el primer elemento de la lista pero además lo elimina. Como podemos comprobar cuando volvemos a mostrar todo el contenido de la pila aquí debajo.

Demo 2

En esta otra clase de demostración tenemos una Linked List.

En el método `FillData` vemos que le estamos añadiendo los mismos elementos y con el mismo orden que anteriormente hemos añadido a la pila. Después de llenar los datos, printamos el primer elemento, el último elemento y a continuación la colección entera, como vemos aquí. Podemos recuperar por delante y por detrás con `PollFirst` y `PollLast`. Aquí lo tenemos. Hemos obtenido los dos primeros y los dos últimos elementos respectivamente y los cuatro han sido eliminados de la colección, como podemos ver cuando volvemos a imprimir la lista aquí debajo.

Acto seguido, añadimos dos nuevos elementos a esta lista doblemente enlazada, uno por delante y otro por detrás. Por último, mostramos la lista por consola de tres formas distintas. Como lo hemos hecho siempre, utilizando un `for`, y luego empezando por delante y empezando por detrás, utilizando, en este caso, un iterador `ListIterator`. Aquí vemos el resultado, como lo hemos hecho siempre, imprimiendo desde delante e imprimiendo desde atrás.

Demo 3

De la interfaz Map hemos visto la clase Hashtable, pero antes de desarrollar un ejemplo con esta clase os voy a mostrar otra colección la clase Hashmap que también implementa esta interface. En esta clase declaramos una colección map de tipo Hashmap y la instanciamos como tipo map, aquí lo vemos. Esto nos permitirá después cambiar el tipo de colección map sin hacer modificaciones en el

código. Una vez instanciado, el objeto map llamamos al método fillData que llenará de datos la colección. Fijaros que entre ellos tenemos valores null.

A continuación declaramos un iterador. Recordad que un iterador es un puntero que nos permitirá ir recorriendo uno a uno todos los elementos de una colección. Hemos visto inicializar un iterador para HighStable con el método keys, pero recordad que estamos utilizando esta HighStable como un objeto map, así que utilizaremos el método EntrySet, que es común para todas las colecciones map. Con el iterador vamos tratando elemento a elemento como un objeto clave valor entry y lo mostraremos por pantalla.

Acto seguido realizamos una inserción y una eliminación y volveremos a mostrar el contenido. Aquí finalmente recuperamos una de las entradas mediante su clave. Vemos que lo que nos devuelve el iterador no tiene por qué coincidir con el orden de inserción que hemos realizado antes. Si ahora cambiamos el tipo de HashMap por HashTable, vemos que no tenemos ningún error de compilación. Pero ejecutemos. Ahora sí obtenemos un error, nullPointerException en tiempo de ejecución, puesto que HashTable no admite valores ni claves null. Si comentamos este valor y volvemos a ejecutar, ya no tenemos ningún problema.

ENUMS

ENUMS

Enums es un tipo de dato especial de Java y otros lenguajes que nos permitirá definir colecciones de constantes. En Java podemos utilizar Enums para definir colecciones de constantes como pueden ser las estaciones del año, los días de la semana, etc. Un Enum en Java es un tipo de clase especial y a diferencia de las otras colecciones de Java, Enum se encuentra en el paquete java.lang.

Creating an enum

Creamos un Enum de la siguiente forma. Como los elementos que contiene son constantes, usamos mayúsculas en su definición.

Fields in enums

En los Enum podemos definir atributos. En tal caso, deberemos especificar también un constructor. Como que los Enums tienen valores constantes, este constructor se crea implícitamente como privado.

Method in enums

Podemos declarar métodos dentro de los enums, por ejemplo, para devolver el valor de un elemento.

Enums usage

Y aquí tenemos un ejemplo de cómo podríamos usar el enum Syson. En este caso, en una sentencia switch case, como si fuera un tipo primitivo de datos, evaluamos un objeto Syson como expresión en el switch y podemos utilizar los valores constantes del enum para establecer los bloques case.

Enums inside classes

Y también podemos usar los enums que hayamos definido como tipo de dato de cualquier atributo de una clase. Puedes encontrar información detallada de la clase enum en la documentación oficial de oracle.

CONTROL DE EXCEPCIONES Y COLECCIONES DE DATOS II

EXCEPTIONS: INTRODUCTION

EXCEPTIONS: INTRODUCTION

Sabemos que en tiempo de ejecución pueden producirse situaciones que alteren el deseado transcurso de un programa. Por ejemplo, si un recurso deja de estar disponible y no podemos acceder a él.

Something is going wrong

throw



Do something to keep the application working.

catch

Los lenguajes de programación deben ofrecer al programador mecanismos para gestionar estas situaciones con el fin de establecer alternativas en lugar de provocar una salida brusca del programa. La gestión de excepciones es el conjunto de mecanismos que un lenguaje de programación proporciona para detectar y gestionar los errores que se pueden producir en tiempo de ejecución.

Según el modelo de gestión de excepciones en Java, al producirse un error, la máquina virtual lanza un aviso, lo que conocemos como **throw**, que el programador debería poder capturar utilizando la cláusula **catch** para resolver la situación problemática y mantener la aplicación en funcionamiento.

Error VS Exception

Veamos ahora qué dos situaciones no deseadas pueden producirse en tiempo de ejecución. Java distingue entre error y excepción.

Error

Irrecoverable situation

Related to environment

There is no programmable fix

Exception

Recoverable situation

Related to application

May be anticipated and should be handled

`OutOfMemoryError`

`class Error`

`ArrayIndexOutOfBoundsException`

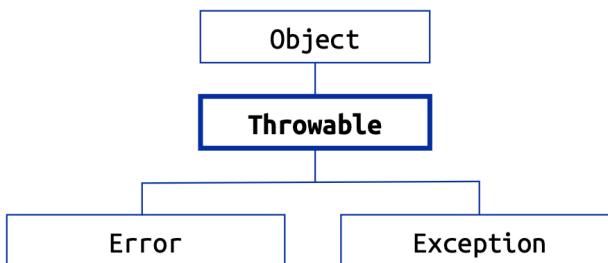
`class Exception`

Los errores son situaciones irrecuperables que no tienen solución y que no dependen del programador. No se deberían de producir nunca, pero cuando se producen provocan la interrupción brusca del programa. Por ejemplo, la máquina virtual se queda sin recursos para continuar la ejecución del programa.

Las excepciones, valga la redundancia, son situaciones excepcionales que los programas se pueden encontrar en tiempo de ejecución, incluyendo los errores de programación. El programador puede prever cada tipo de excepción y escribir el código para su gestión.

Por ejemplo, se producirá una excepción si intentamos acceder a una posición inexistente dentro de un array. Java engloba los posibles errores y excepciones en las clases Error y Exception según corresponda.

Class Throwable



Como podemos ver en la imagen, las clases Error y Exception se encuentran al mismo nivel dentro de la jerarquía de clases de Java, justo por debajo de la superclase Throwable. Esto significa que ambas heredan sus características y métodos.

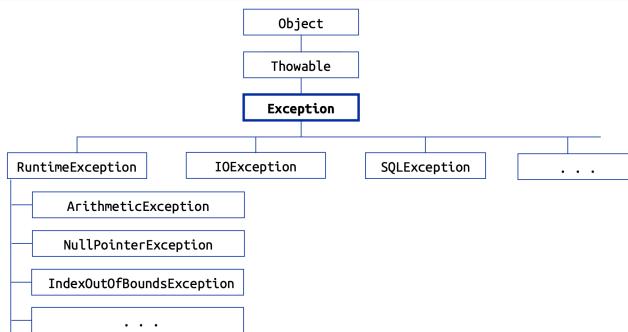
Constructors	
Modifier	Constructor and Description
	<code>Throwable()</code> Constructs a new throwable with <code>null</code> as its detail message.
	<code>Throwable(String message)</code> Constructs a new throwable with the specified detail message.
	<code>Throwable(String message, Throwable cause)</code> Constructs a new throwable with the specified detail message and cause.
protected	<code>Throwable(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace)</code> Constructs a new throwable with the specified detail message, cause, <code>suppression</code> enabled or disabled, and writable stack trace enabled or disabled.
	<code>Throwable(Throwable cause)</code> Constructs a new throwable with the specified cause and a detail message of <code>(cause==null ? null : cause.toString())</code> (which typically contains the class and detail message of <code>cause</code>).

Como podemos ver en la documentación de la clase Throwable, disponible en Docs.Oracle.com, hay dos constructores que incorporan la posibilidad de crear un objeto throwable, indicando otro objeto throwable como causante del nuevo objeto. Esto permite encadenar los errores y las excepciones.

Methods	
Modifier and Type	Method and Description
void	<code>addSuppressed(Throwable exception)</code> Appends the specified exception to the exceptions that were suppressed in order to deliver this exception.
Throwable	<code>fillInStackTrace()</code> Fills in the execution stack trace.
Throwable	<code>getCause()</code> Returns the cause of this throwable or <code>null</code> if the cause is nonexistent or unknown.
String	<code>getLocalizedMessage()</code> Creates a localized description of this throwable.
String	<code>getMessage()</code> Returns the detail message string of this throwable.
StackTraceElement[]	<code>getStackTrace()</code> Provides programmatic access to the stack trace information printed by <code>printStackTrace()</code> .
Throwable[]	<code>getSuppressed()</code> Returns an array containing all of the exceptions that were suppressed, typically by the <code>try-with-resources</code> statement, in order to deliver this exception.
Throwable	<code>initCause(Throwable cause)</code> Initializes the <code>cause</code> of this throwable to the specified value.
void	<code>printStackTrace()</code> Prints this throwable and its backtrace to the standard error stream.
void	<code>printStackTrace(PrintStream s)</code> Prints this throwable and its backtrace to the specified print stream.
void	<code>printStackTrace(PrintWriter s)</code> Prints this throwable and its backtrace to the specified print writer.
void	<code>setStackTrace(StackTraceElement[] stackTrace)</code> Sets the stack trace elements that will be returned by <code>getStackTrace()</code> and printed by <code>printStackTrace()</code> and related methods.
String	<code>toString()</code> Returns a short description of this throwable.

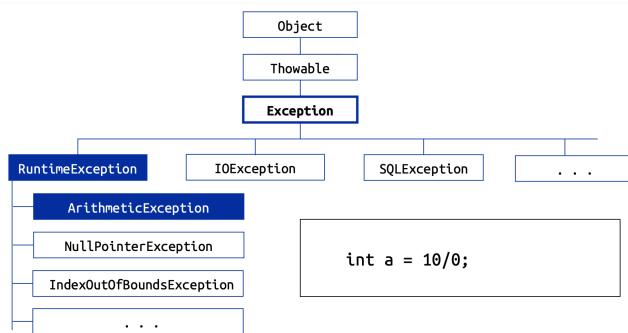
La clase `throwable` dispone de métodos para conocer el contexto en el que se ha producido la situación problemática y, por tanto, poder actuar en consecuencia. En este caso, nos será de especial utilidad los métodos `GetCalls`, que nos devuelve un objeto `throwable` que contiene la causa del error. `GetMessage`, que devuelve un mensaje descriptivo del error producido. `PrintStackTrace`, que muestra por el canal de errores el contexto donde se ha producido el error, incluyendo la cascada de llamadas desde el método `main` que han llevado al punto en el que se ha levantado el error, o `toString`, que devuelve una breve descripción del objeto en formato de texto.

EXCEPTION: TYPES

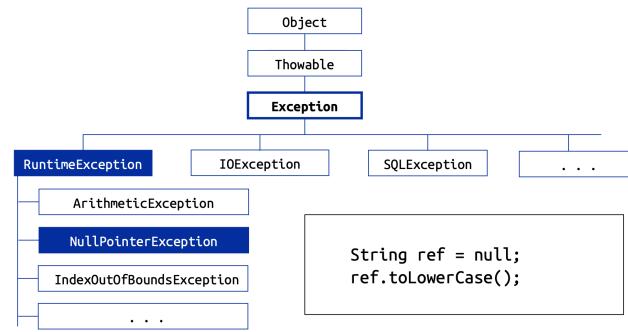


Al producirse una excepción en un programa, se crea un objeto de la subclase de `Exception` a la que pertenece la excepción. En pantalla podemos ver cómo se organiza la jerarquía de la clase `Exception` en Java. Las excepciones que se producen por algún problema dentro de la máquina virtual Java se engloban en la subclase `RuntimeException`. Según el problema ocurrido, el objeto que representará la excepción producida será una subclase de `RuntimeException`.

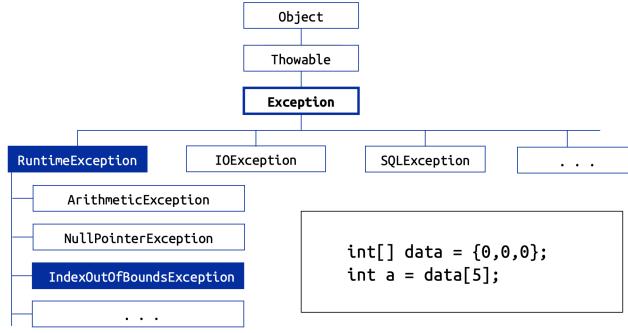
A continuación veremos algunas de ellas.



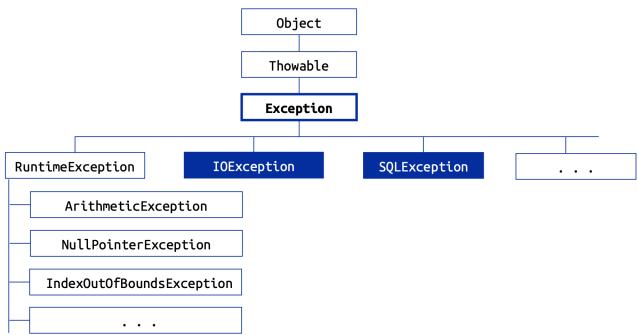
Obtendremos una `ArithmeticException` si el problema se ha producido durante una operación aritmética, como vemos en el código de ejemplo en pantalla, si intentamos dividir por cero.



Intentar utilizar una referencia a un objeto que no ha sido creado, es decir, cuyo valor es `null`, provocará una `null pointer exception`.



Si intentamos acceder a un índice inexistente, bien sea dentro de un string, un vector o como en el código del ejemplo, un array, hará que se produzca una `index out of bounds exception`.



Otras subclases que heredan directamente de Exception son por ejemplo IOException o SQLException.

Implicit VS Explicit

Una clasificación no oficial pero que nos puede ayudar en nuestra tarea como programadores es diferenciar entre excepciones implícitas o explícitas.

Implicit	Explicit
RunTimeExceptions	Not RunTimeExceptions
May/Should be controlled	Must be controlled
Can't be worked around in run time	May be worked around in run time

Las excepciones implícitas son aquellas que pertenecen a una subclase de RuntimeException. Habitualmente representan errores de programación. La misma máquina virtual se encargará de gestionarlas si no lo hace el propio programador.

Algo recomendable para mejorar la percepción de los usuarios sobre la aplicación. Pero debemos ser conscientes que no podremos resolver estas excepciones en tiempo de ejecución.

Implicit Exception

```

Public class Main {
    public static void main (String[]){
        String filename=null;
        if (filename.contains("aaaa")){
            //TODO code application
        }else{
            //TODO code application
        }
    }
}
  
```

Example

En el siguiente código, podemos ver un ejemplo de excepción implícita. Vemos que en la tercera línea se declara la variable StringFileName y a la vez se inicializa con un valor nulo. En la línea inmediatamente siguiente, intentamos utilizar el método contains de la variable stringFileName.

Implicit
RunTimeExceptions
May/Should be controlled
Can't be worked around in run time

Esto producirá una excepción en tiempo de ejecución de tipo runtimeException. Concretamente obtendremos una nullPointerException, que se produce siempre que intentamos utilizar métodos de un objeto null.

NullPointerException
unchecked

Las excepciones implícitas también se conocen como uncheck, puesto que el compilador de Java no las detecta. Un ejemplo de excepción implícita o uncheck es NullPointerException.

Explicit Exception

```
Public class Main {
    public static void main (string[]){
        //TODO code application
        String filename="employees.dat";
        FileReader fr = new FileReader(filename);
    }
}
```

Example

unreported exception FileNotFoundException must be caught or declared to be thrown

Las excepciones explícitas son todas aquellas que no pertenecen a ninguna subclase de RuntimeException. Son excepciones que el programador está obligado a tener en cuenta, mediante el uso de cláusulas try-catch o declarando que el método pudo lanzar una excepción con la palabra reservada throws.

Ahora vemos un ejemplo de una excepción explícita. Después de escribir este código en nuestro IDE, el compilador nos alertará que algo no está bien, típicamente subrayando la línea donde está el problema.

Explicit

Not RunTimeExceptions

Must be controlled

May be worked around in run time

FileNotFoundException

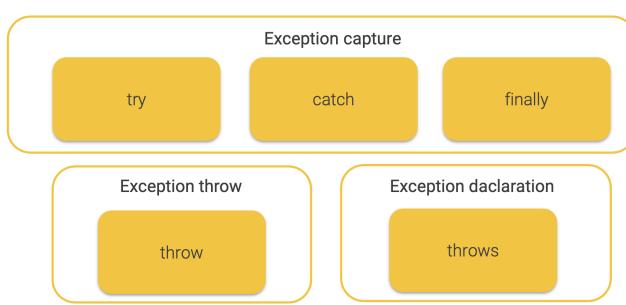
checked

Explorando la descripción textual obtendremos más información. En este caso, nos indica que la línea en cuestión puede lanzar una excepción del tipo FileNotFoundException. No se puede garantizar que el fichero Employees.dat vaya a estar siempre disponible. Es algo que no depende del lenguaje, sino de factores externos. Debemos, por lo tanto, controlar la posible excepción

Las excepciones explícitas también se conocen como check, puesto que el compilador de Java detectará que es posible que se produzcan e instará al programador a enmendar el código para su control. Un ejemplo de excepción explícita o check es FileNotFoundException.

EXCEPTION HANDLING: CAPTURE**EXCEPTION HANDLING: INTRODUCTION**

Exception handling keywords



El manejo de excepciones en Java se gestiona a través de cinco palabras clave. Veamos en primer lugar en qué ámbito se utilizan cada una de ellas. Try, catch y finally son utilizadas en la captura de las excepciones y su posterior tratamiento.

Throw se utiliza para el lanzamiento de las excepciones. Y throws se utiliza para su declaración.

EXCEPTION CAPTURE

Try, catch & finally clauses

```
try{
    //code that may cause an Exception
}catch (nombreClasseExcepcion1 e1){
    //actions to do when de Exception1 happens
}catch (nombreClasseExcepcion2 e2){
    //actions to do when de Exception2 happens
} finally{
    //code to be executed always
}
```

Las palabras clave Try, Catch y Finally forman un subsistema interrelacionado en el que el uso de uno implica el uso de otro. Como vemos en pantalla, se emplean para enclausurar bloques de código. Dentro de un bloque Try, escribiremos código susceptible de causar una excepción.

Un bloque Try debe ir siempre seguido como mínimo de un bloque catch. En los bloques catch, escribiremos las acciones que queremos hacer cuando se produce la excepción. Un bloque catch va precedido de una declaración que define la excepción a la que el bloque da respuesta.

Podemos encadenar varios bloques catch para dar respuesta a distintos tipos de excepciones producidas dentro del bloque try. Al finalizar el bloque try o un bloque catch si se ha producido una excepción se ejecutará el código del bloque Finally y se continuará la ejecución del programa. Los bloques finally son de utilidad para cerrar recursos tales como conexiones o archivos ante una finalización prematura del programa. El bloque finally es opcional, pero cuando está su contenido se ejecutará siempre, incluso si los bloques try o catch que le preceden devuelven el método con un return.

Demo ejemplo 1

Veamos un ejemplo práctico de la captura de excepciones utilizando las cláusulas try, catch y final. El siguiente código debe cargar el contenido de un fichero txt en una variable inputStream, utilizando para ello un objeto FileInputStream. Vemos que nuestro IDE, en este caso IntelliJ, realza el objeto FileInputStream indicando que puede lanzar una excepción de tipo FileNotFoundException, lo cual no se está capturando.

```
public static void main(String[] args) {
    InputStream in = null;
    try {
        System.out.println("About to open a file");
        in = new FileInputStream("missingfile.txt");
        System.out.println("File open");
        int s = in.read();
    } catch(FileNotFoundException e){
        System.out.println(e.getClass().getName()+" - "+ e.getMessage());
    } catch(IOException e){
        System.out.println(e.getClass().getName()+" - "+ e.getMessage());
    } finally{
        System.out.println("Application end");
        try{
            if(in != null){
                in.close();
            }
        } catch(IOException e){
            System.out.println("Failed to close file");
        }
    }
}
```

Para hacerlo utilizaremos la cláusula try. Una cláusula try siempre debe ir seguida de una cláusula catch o finally, capturaremos la excepción con una cláusula catch. En nuestro caso, capturaremos todo tipo de excepciones. Ahora el código ya no tiene errores de compilación, pero dejarlo así es una mala práctica. En nuestro caso, informaremos por consola que algo no ha ido bien.

También mostraremos la descripción del error.

```

public static void main(String[] args) {
    InputStream in = null;
    try {
        System.out.println("About to open a file");
        in = new FileInputStream("missingfile.txt");
        System.out.println("File open");
        int s = in.read();
    } catch(FileNotFoundException e){
        System.out.println(e.getClass().getName()+" - "+ e.getMessage());
    } catch(IOException e){
        System.out.println(e.getClass().getName()+" - "+ e.getMessage());
    } finally{
        System.out.println("Application end");
        try{
            if(in != null){
                in.close();
            }
        } catch(IOException e){
            System.out.println("Failed to close file");
        }
    }
}

```



Demo ejemplo 2

```

public static void main(String[] args) {
    InputStream in = null;
    try {
        System.out.println("About to open a file");
        in = new FileInputStream("missingfile.txt");
        System.out.println("File open");
        int s = in.read();
    } catch(FileNotFoundException e){
        System.out.println(e.getClass().getName()+" - "+ e.getMessage());
    } catch(IOException e){
        System.out.println(e.getClass().getName()+" - "+ e.getMessage());
    } finally{
        System.out.println("Application end");
        try{
            if(in != null){
                in.close();
            }
        } catch(IOException e){
            System.out.println("Failed to close file");
        }
    }
}

```

java.io.IOException

Finalmente, de forma opcional, podemos incluir también una cláusula Finally, que utilizaremos para indicar que la aplicación ha terminado, independientemente de si se han producido errores o no. A continuación, iniciaremos la aplicación. En la consola, vemos los mensajes que ha dejado su ejecución, que nos da a conocer que se ha producido un error debido a que no se ha encontrado el fichero missingfile.txt.

Veamos ahora una versión extendida del código de la demo para adentrarnos en las características de la captura de excepciones en Java. En este caso, en lugar de tener un único bloque catch para capturar todo tipo de excepciones, tenemos dos bloques catch más específicos que capturan solo excepciones del tipo FileNotFoundException o IOException.

Los necesitamos porque debemos capturar obligatoriamente las excepciones de tipo check que pueden producir el constructor FileInputStream, que puede arrojar una FileNotFoundException, o el método read de la clase InputStream, que puede arrojar una IOException.

La ordenación de los bloques Catch no es casual, puesto que estamos capturando dos excepciones que forman parte de la misma jerarquía. FileNotFoundException es una subclase de IOException. En estos casos, siempre debemos capturar primero la excepción más específica. Centrémonos ahora en el bloque Finally.

Sabemos que de ser especificado este bloque, se ejecutará siempre al final, y que lo debemos utilizar para liberar recursos abiertos dentro del bloque Try. En este caso, cerraremos el objeto InputStream, que podría haber sido abierto previamente. El método Close de la clase InputStream también puede arrojar una IOException, por lo que debemos capturar la posible excepción mediante otra cláusula try-catch.

Vemos así que también podemos utilizar este tipo de cláusulas de forma anidada.

Multi-catch

```

try {
    System.out.println("About to open a file");
    in = new FileInputStream("missingfile.txt");
    System.out.println("File open");
    int x = in.read();
    int y = 20 / x
} catch(ArithmeticException e){
    System.out.println(e.getClass().getName()+" - "+ e.getMessage());
} catch(FileNotFoundException e){
    System.out.println(e.getClass().getName()+" - "+ e.getMessage());
} catch(IOException e){
    System.out.println(e.getClass().getName()+" - "+ e.getMessage());
}

```

Fíjémonos ahora en esta nueva versión de código con una instrucción adicional que puede arrojar otro tipo de excepción. En este caso, dependiendo del valor de X, podemos obtener una arithmetic exception. La posible nueva excepción se controla en su correspondiente bloque Catch.

```

try {
    System.out.println("About to open a file");
    in = new FileInputStream("missingfile.txt");
    System.out.println("File open");
    int x = in.read();
    int y = 20 / x
} catch(FileNotFoundException e){
    System.out.println(e.getClass().getName()+" - "+ e.getMessage());
} catch(IOException e){
    System.out.println(e.getClass().getName()+" - "+ e.getMessage());
} catch(ArithmaticException e){
    System.out.println(e.getClass().getName()+" - "+ e.getMessage());
}

```

De este modo, tenemos tres bloques Catch para capturar tres tipos distintos de excepciones. Esto nos permitiría poder gestionar cada una de ellas de una forma distinta. Pero en el ejemplo, vemos que estamos realizando la misma acción para cada una de ellas.

```

try {
    System.out.println("About to open a file");
    in = new FileInputStream("missingfile.txt");
    System.out.println("File open");
    int x = in.read();
    int y = 20 / x
} catch(FileNotFoundException | ArithmaticException e){
    System.out.println(e.getClass().getName()+" - "+ e.getMessage());
} catch(IOException e){
    System.out.println(e.getClass().getName()+" - "+ e.getMessage());
}

```

La característica Multicatch de Java nos ayuda a optimizar el código permitiendo la captura de más de una excepción en un mismo bloque catch. Como vemos en la imagen, ahora las excepciones FileNotFoundException y ArithmaticException son capturadas por el mismo bloque catch.

```

try {
    System.out.println("About to open a file");
    in = new FileInputStream("missingfile.txt");
    System.out.println("File open");
    int x = in.read();
    int y = 20 / x
} catch(FileNotFoundException | IOException | ArithmaticException e){ ✗
    System.out.println(e.getClass().getName()+" - "+ e.getMessage());
}

```

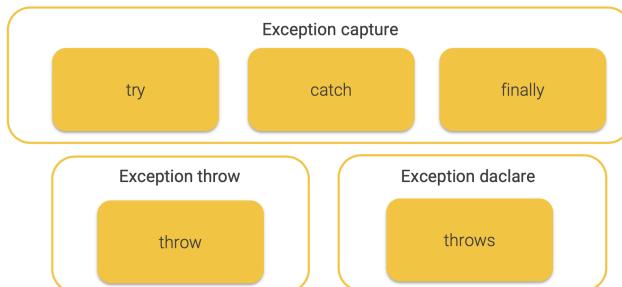
Types in multi-cast must be disjoint: FileNotFoundException is a subclass of IOException

En la declaración del bloque Multicatch debemos separar las excepciones que queremos capturar mediante el símbolo or. Debemos tener en cuenta que no podemos capturar en un bloque Multi catch excepciones que formen parte de la misma jerarquía. De este modo, este bloque Multicatch sería incorrecto, porque FileNotFoundException es una subclase de IOException.

EXCEPTION HANDLING: DECLARE AND THROW

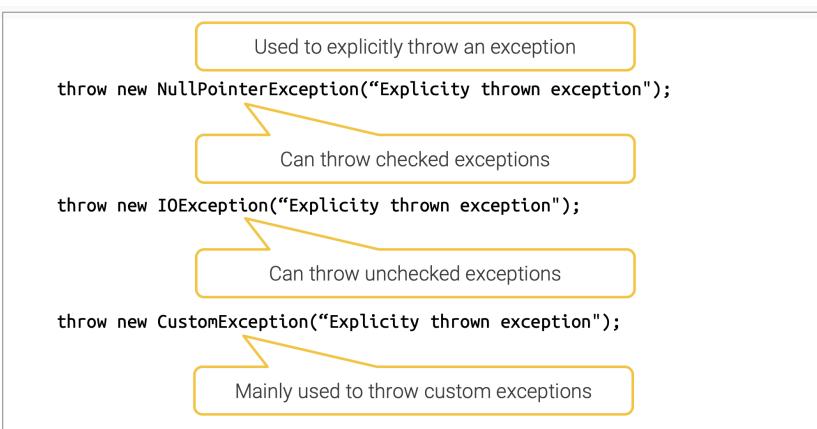
EXCEPTION HANDLING: INTRODUCTION

Exception handling keywords



Conocemos el uso de las cláusulas de captura de excepciones try catch y finally. Veamos ahora cómo se utiliza throw y Throws para declararlas.

EXCEPTION THROW



La palabra clave Throw en Java se emplea para lanzar una excepción desde un método o cualquier bloque de código.

Podemos lanzar de este modo excepciones de tipo check o uncheck indistintamente, pero throw se utiliza principalmente para lanzar excepciones personalizadas.

Throw y rethrow

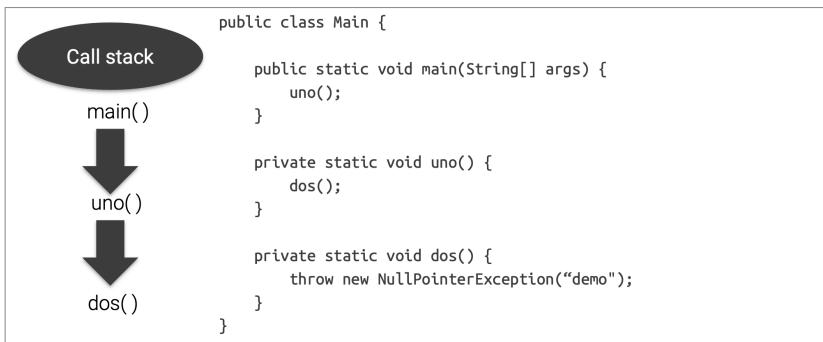
```

static void demoThrow()
{
    try
    {
        throw new NullPointerException("Explicitly thrown exception");
    }
    catch(NullPointerException e)
    {
        System.out.println("Caught inside demoThrow().");
        throw e; // rethrowing the exception
    }
}
  
```

En este ejemplo, vemos una sentencia Throw dentro de un bloque Try, y que es capturada en un bloque Catch dentro del mismo método. En determinados casos, también puede resultar útil relanzar esta misma excepción después de realizar alguna acción dentro del bloque Catch.

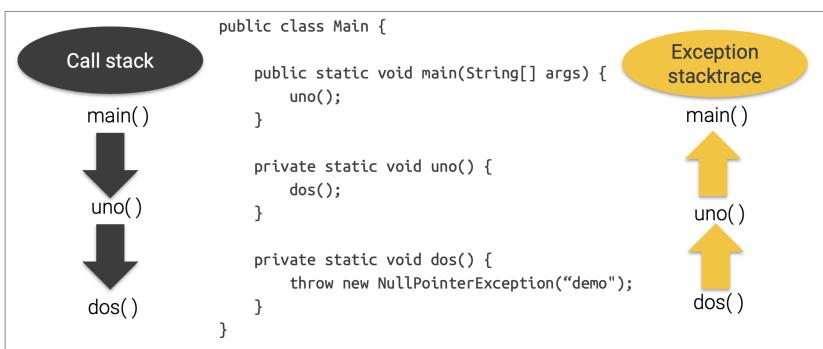
En tal caso, la excepción deberá ser capturada en un método anterior en la pila de ejecución del programa.

Exception propagation



Veamos cómo funciona la propagación de las excepciones en la pila de ejecución del programa. En el código del ejemplo, vemos una aplicación que en su método Main realiza una llamada a otro método llamado 1. El método 1 realiza una llamada al método 2. Esta secuencia de métodos se conoce como pila de llamadas.

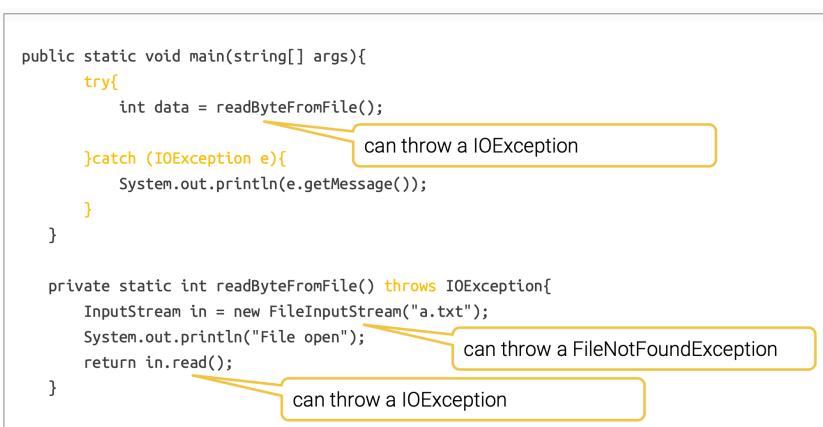
Si en tiempo de ejecución se lanza o se produce una excepción y ésta no es capturada dentro del mismo método, la máquina virtual aborta la ejecución del método donde se ha producido la excepción y propaga la excepción al método inmediatamente superior.



Si en el método superior tampoco se captura la excepción, se aborta la ejecución de éste y se propaga la excepción a su superior, y así sucesivamente hasta que la excepción es gestionada. Si la excepción no es gestionada y se llega al main, y aquí tampoco se gestiona, se interrumpe el programa.

Podemos encontrar información relativa al recorrido de la excepción desde que se lanza hasta que es capturada en la pila de llamadas de la excepción, más conocida por su término en inglés Stack Trace.

EXCEPTION DECLARE



Cuando un método puede propagar una excepción hacia un método superior, siempre es conveniente especificarlo en su declaración. En el ejemplo, el método ReadByteFromFile puede producir dos tipos de excepción, una File Not Found Exception y una EO Exception. Debemos añadir, por lo tanto, la sentencia Throws en la declaración.

De este modo, quien utilice el método sabrá que éste puede arrojar una excepción y podrá tomar las medidas para su control.

Demo ejemplo 1

Veamos un ejemplo práctico del uso de throw y throws para la declaración y lanzamiento de excepciones. Para ello, observemos el siguiente programa, el objetivo del cual es inicializar la variable data con un valor entero. Para ello, utilizamos el método getValue. Este método podemos suponer que contiene múltiples líneas y por el motivo que sea, deseamos lanzar una excepción. Por ejemplo, una IOException. Para lanzar una excepción de cualquier tipo, debemos empezar la sentencia con la palabra throw, seguido del operador new, y el nombre de la excepción que queremos lanzar.

Pasando como parámetro un texto, un mensaje del error producido. Ahora nuestro IDE ha detectado una excepción check que no está siendo controlada. Para hacerlo, podemos utilizar los bloques try-catch, pero en nuestro caso, escogeremos propagar la excepción al método superior. Podemos hacer esto utilizando la sentencia throws, seguido del nombre de la excepción que queremos propagar. Ahora vemos que el aviso que no estamos controlando una excepción check se ha desplazado al método main.

Concretamente, en la sentencia donde hacemos uso del método getValue, que ahora puede arrojar una IOException. En este caso, añadiremos las cláusulas tryCatch para controlarlo. Eliminamos este código, que es inalcanzable, puesto que por arriba estamos lanzando siempre una excepción, y obtenemos un programa libre de errores. Ahora podemos ejecutarlo.

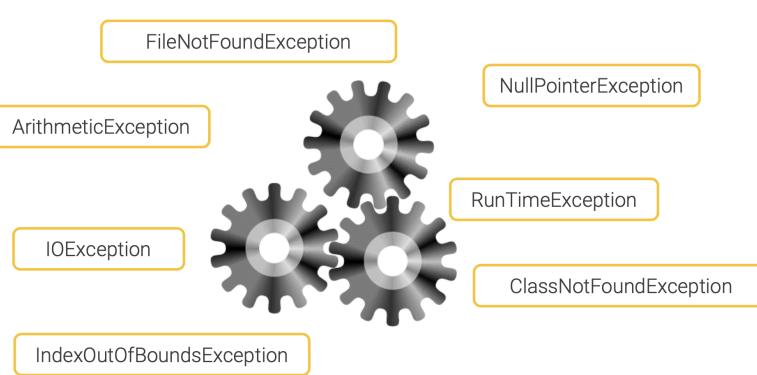
Demo ejemplo 2

Hagamos el ejemplo más real utilizando un objeto scanner para que podamos introducir valores por consola. Supongamos que solo queremos aceptar números enteros. En tal caso, retornaremos el valor. Y de lo contrario, lanzaremos la excepción. Ejecutemos el programa. Introduzcamos un valor numérico, por ejemplo el 1, y vemos que el programa finaliza sin problemas.

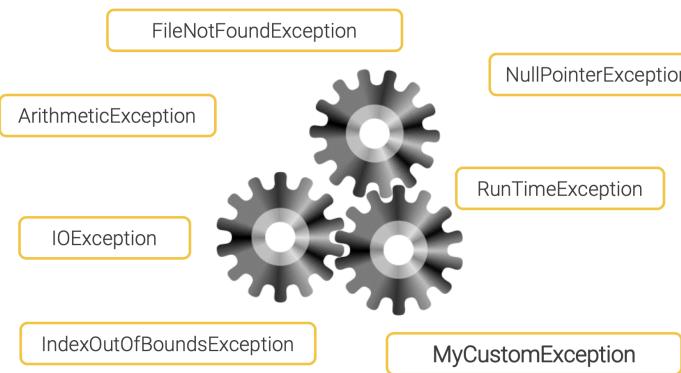
Introduzcamos ahora otro valor que no sea numérico. Vemos que se ha lanzado la excepción. Hemos visto que la propagación de excepciones es útil como medio para enviar un aviso a otra parte del programa, indicándole que algo está sucediendo y no es posible continuar con la ejecución normal del método. Utilizamos throw para lanzar una excepción de cualquier tipo en cualquier momento. Y debemos utilizar throws en la declaración de un método que propaga algún tipo de excepción.

CUSTOM EXCEPTIONS

CUSTOM EXCEPTIONS: INTRODUCTION



Sabemos que durante la ejecución de un programa pueden producirse excepciones por motivos diversos, mediante el uso de las técnicas de control de excepciones podemos gestionar estas situaciones y mantener nuestra aplicación en funcionamiento. Algunas de estas excepciones son lanzadas por la máquina virtual de Java y otras por el propio programador.

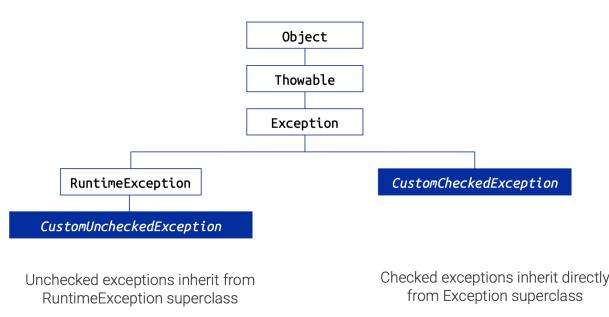


A veces el programador no encuentra entre las excepciones existentes ninguna adecuada a las características de la situación que quiere notificar.

En este caso, resulta práctico definir una excepción personalizada que sirva exactamente para el propósito específico que necesita el programador.

CUSTOM EXCEPTIONS: CLASSES

Custom exception classes hierarchy



Las excepciones personalizadas que queramos crear tendrán su propia clase y ocuparán su propio lugar dentro de la jerarquía de clases de Java.

Si queremos una excepción de tipo Unchecked, deberemos extender la clase `RuntimeException`. Si lo que queremos es crear una excepción de tipo Check, extenderemos directamente la clase `Exception`.

Custom exception class

```

Inherits from Exception superclass
public class MyCustomException extends Exception {
    public MyCustomException(String message) {
        super(message);
    }
}
We should call de superclass constructor
  
```

En el código de la pantalla podemos ver un ejemplo de excepción personalizada. En la declaración de la clase indicamos que queremos extender una excepción. En el constructor de la nueva clase realizaremos una llamada al constructor de la clase padre.

Demo ejemplo

Veamos un uso práctico de las excepciones personalizadas en Java. Para ello, fijémonos en la siguiente clase `Account`, que representa una cuenta bancaria. Tiene un único atributo `Balance`, para reflejar el saldo de la cuenta. Dispone de dos constructores y tres métodos.

Uno para obtener el saldo actual y otros dos para realizar ingresos y reintegros. Por otro lado, tenemos la clase `Automated Teller Machine` que representa un cajero automático. Como único atributo, tiene un objeto `Account` y luego tiene el método `main` y dos métodos más para simular ingresos y reintegros. Ejecutemos el programa. Vemos que tras crear una cuenta a cero y realizar un ingreso de 100, podemos también realizar un reembolso de 120, resultando en un saldo negativo de 20. Podemos considerar este caso como un problema, puesto que esta cuenta no dispone del 120 y no debería haberse permitido un reembolso superior a su saldo.

Vayamos entonces al método withdraw de la clase account a controlar esta casuística. Controlaremos que si el saldo actual es inferior al reembolso deseado abortaremos la ejecución del método lanzando una excepción. En caso contrario, permitiremos que se haga la operación. debemos especificar que este método propaga una excepción genérica y nos veremos obligados a controlar esta excepción cuando hagamos uso del método. Hemos utilizado la clase genérica Exception para lanzar la excepción. Esto prácticamente no nos da información de lo que ha ocurrido y dificulta, por ejemplo, comunicar al usuario qué ha sucedido y cómo puede proceder. Para mejorar la comunicación con el usuario, debemos también mejorar la comunicación entre clases. Para ello, haremos uso de una excepción personalizada.

Crearemos la clase LowBalanceException que extenderá de la clase exception. En el constructor llamaremos al constructor de la super clase con la keyword super. Ya tenemos la excepción personalizada creada. Vayamos a hacer uso de ella, en lugar de lanzar una excepción genérica, lanzaremos una LowBalanceException, vemos que nos marca que debemos especificar siempre un mensaje. Esto es porque solo hemos creado un constructor que obligatoriamente debe recibir un string. Crearemos pues el mensaje. Lanzaremos hacia método superior una LowBalanceException y vemos en el método main que no nos da ningún mensaje de error porque ya estamos capturando la excepción más genérica posible. Pero podemos añadir otro bloque catch para capturar excepciones específicas como la Low Balance Exception.y lo que haremos será mostrar el mensaje que nos da la descripción. Vemos que ahora el mensaje es más descriptivo cuando se produce la situación específica que queríamos controlar. Si se produciera algún otro tipo de excepción, ésta sería capturada en el segundo bloque Catch.