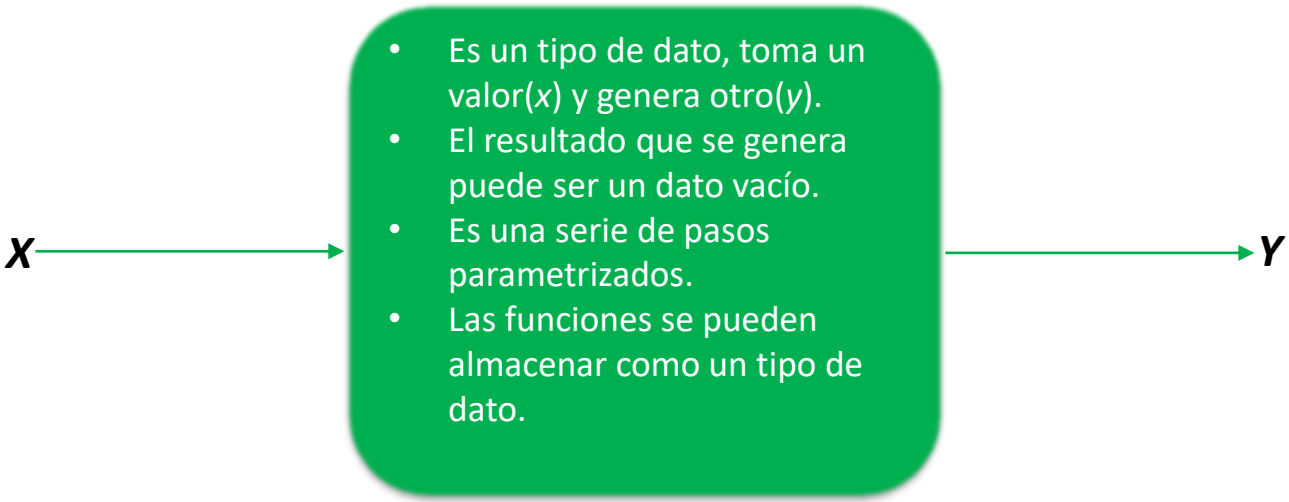
APUNTES DE PROGRAMACIÓN FUNCIONAL CON JAVA SE

¿QUÉ ES UNA FUNCIÓN?



FUNCIONES PURAS

```
suma( x: 3, y: 8)
```

- Las funciones puras generan el mismo resultado para el mismo parámetro.

```
public static int suma(int x, int y){
    return x + y;
}
```

//Siempre será 8

{Las funciones puras no dependen ni del sistema ni del contexto.}

{Su resultado se puede predecir por lo tanto son deterministas. }

EFECTOS SECUNDARIOS

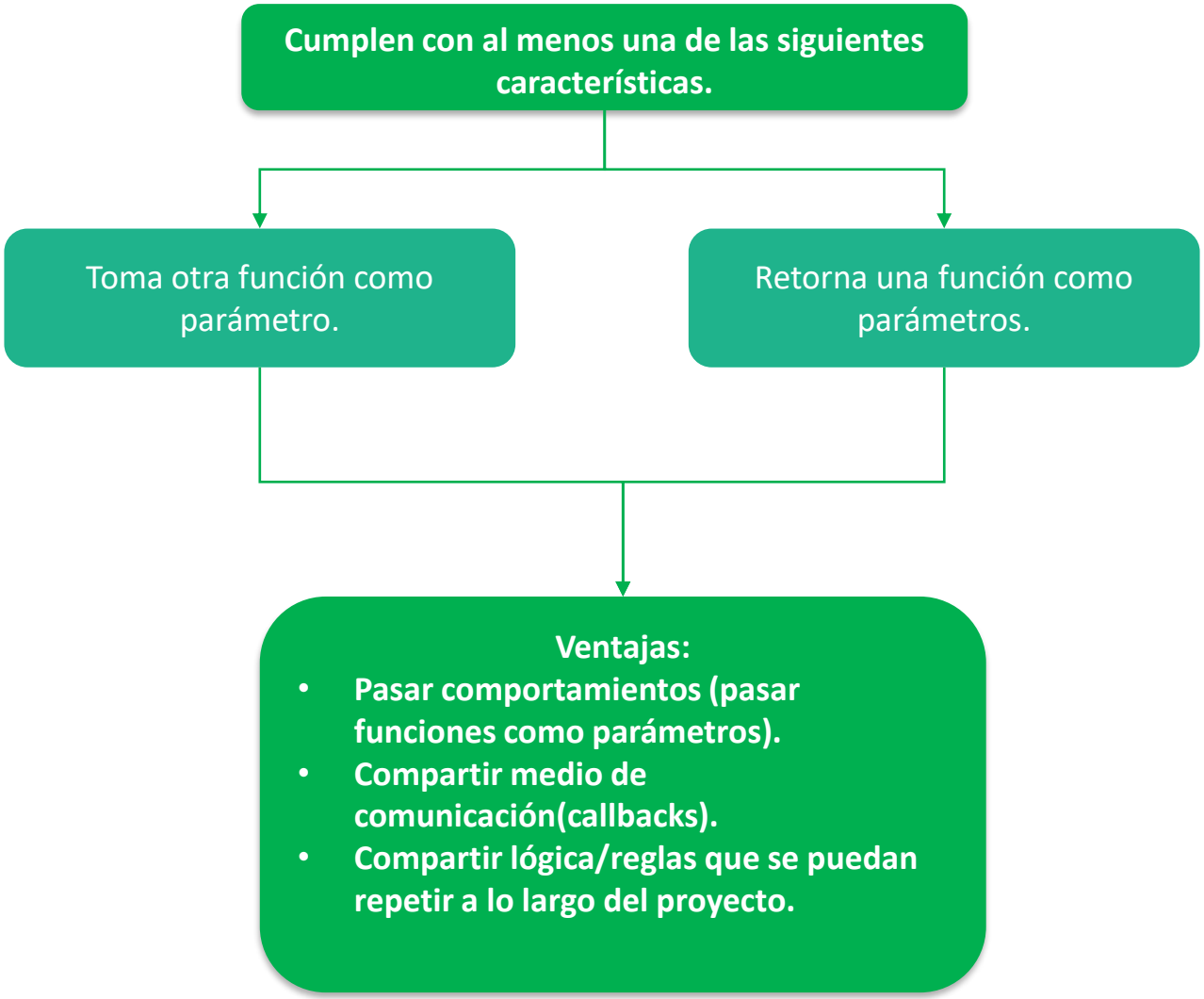
- Todo cambio observable desde fuera del sistema es un efecto secundario.
- Los efectos secundarios son inevitables.

- Los efectos secundarios se pueden reducir/controlar cuando suceden.

¿Por qué evitarlos?

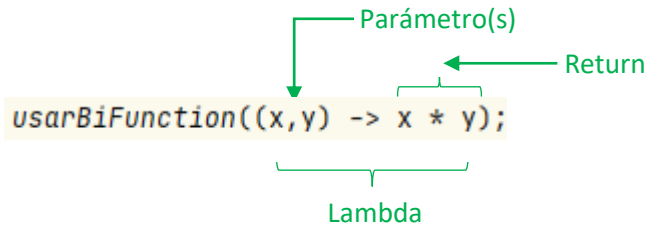
- **Nos ayuda a tener mejor estructura de nuestro código.**
- **Nos permite tener mas funciones puras.**
- **Tendremos mejor separadas las definiciones y responsabilidades de nuestro código.**
- **Aumenta la testeabilidad de nuestro sistema.**

FUNCIONES DE ORDEN MAYOR



FUNCIONES LAMBDA

Las funciones lambda son funciones anónimas, no tienen nombre.



Por que usarlas?

- **Es útil cuando requerimos de un comportamiento de un solo uso, es decir que difícilmente lo usaremos en otra parte del código.**
- **Es una regla que solo se requiere en un lugar.**
- **Es una función extremadamente simple.**

¡ Una lambda sigue siendo una función

INMUTABILIDAD DE LOS DATOS

Cuando hablamos de inmutabilidad nos referimos a los datos que no cambian y no se pueden modificar, por lo cual nos aseguramos que nuestros objetos/datos no cambien en lugares inesperados.

Ventajas:

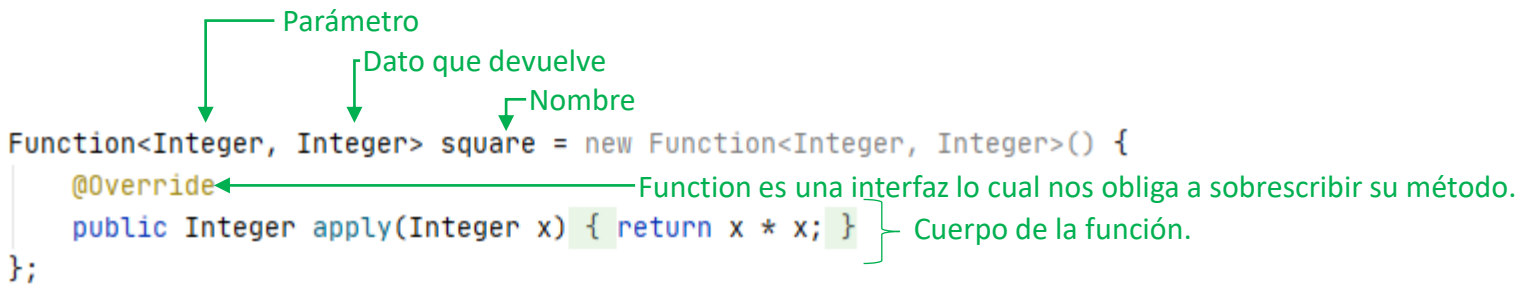
- Una vez creadas las instancias de los objetos no se puede alterar.
- Facilita crear funciones puras disminuyendo los efectos secundarios.
- Facilita usar threads/concurrencia.

Desventajas:

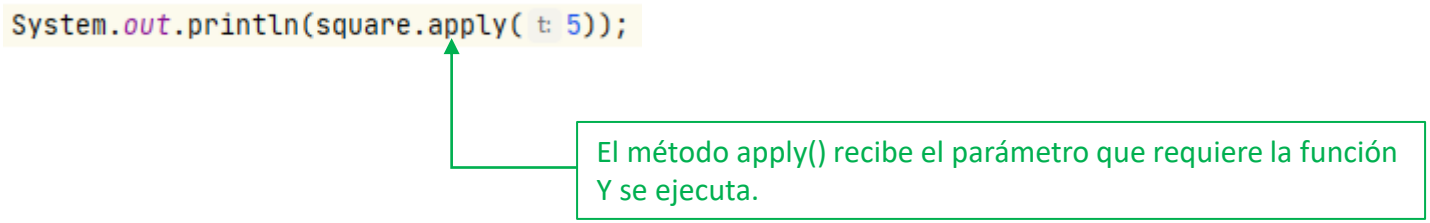
- Por cada modificación se necesita una nueva instancia del dato.
- Requiere especial atención al diseño.
- Existen objetos mutables(pueden provenir de librerías de terceros o datos externos)fuera de nuestro alcance.

java.util.function: Function

Crear una función:



Invocar una función:

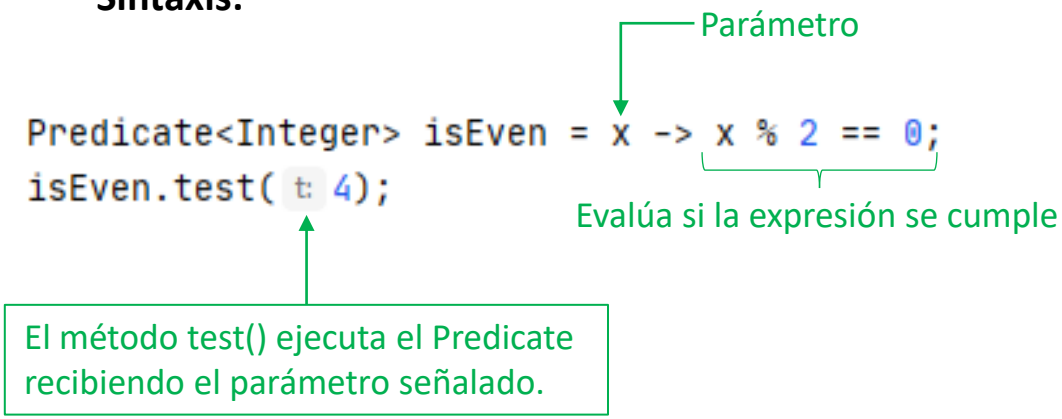


Las funciones no reciben ni devuelven como parámetro datos primitivos, sino objetos.

java.util.function: Predicate

La interfaz Predicate nos devuelve un objeto del tipo Boolean, esto es útil especialmente en testing.

Sintaxis:



java.util.function: Consumer y Supplier

Consumer es una interfaz del tipo genérico, que nos permite generar funciones simples, toma o consume un dato pero no retorna ningún valor.

Dato que se va a consumir

Nombre

Parámetro

```
Consumer<CLIArguments> consumerHelper = cliArguments1 -> {
    if (cliArguments1.isHelp()){
        System.out.println("Manual solicitado");
    }
```

Forma en que se van a consumir los datos

```
consumerHelper.accept(cliArguments);
```

Para invocar a nuestro Consumer usamos el método accept().

Consumer es especialmente útil para consumir listados de datos donde podemos operar o modificar cada dato de la lista.

Un Supplier es una interfaz que nos permite crear funciones que no requieran saber la procedencia de los datos, no recibe argumentos pero devuelve un tipo de dato.

Dato que va a generar

Nombre

Indica que no recibirá ningún dato

```
Supplier<CLIArguments> generator = () -> new CLIArguments();
return generator.get();
```

Retornamos nuestro Supplier con el método get().

Supplier es especialmente útil para generar archivos bajo demanda.

Nota: Los ejemplos de Supplier y Consumer generan y consumen datos del tipo CLIArguments, enseguida se muestra la definición del objeto.

```
public class CLIArguments {
    private boolean isHelp;

    public boolean isHelp() {
        return isHelp;
    }
}
```

java.util.function: Operators y BiFunction

Los Operators son funciones simplificadas que nos permiten ver fácilmente que hacen o sobre que trabajan.

UnaryOperator es un tipo de Operator que trabaja sobre un solo tipo de dato y nos devuelve el mismo tipo de dato.

Dato sobre el que trabaja

Nombre

Parámetro

```
UnaryOperator<String> quote = text -> "\"" + text + "\"";
```

Return

```
System.out.println(quote.apply( t: "Hola Estudiante de Platzi!"));
```

Muchas de las veces nuestras funciones requieren de dos parámetros para lo cual UnaryOperator no nos es útil, en cambio usamos BinaryOperator, nos sirve si ambos parámetros y el resultado son del mismo tipo.

Parámetros

Return

```
BinaryOperator<Integer> multiply = (x, y) -> x * y;
System.out.println(multiply.apply( t: 2, u: 3));
```

Ambas interfaces extienden de Function por lo tanto usan el método apply() para invocar nuestras funciones

Otro caso muy común es que los parámetros que requieren nuestras funciones no necesariamente son del mismo tipo, en estas situaciones BiFunction nos es de gran ayuda.

Datos que recibirá

Dato que devolverá

```
BiFunction<String, Integer, String> leftPad = (s, i) -> String.format( s: "%" + i + "s", s);

System.out.println(
        leftPad.apply( t: "Hello",  u: 10) // "     Hello"
);
```

Parámetros

Return

BiFunction también usa el método apply() para invocar nuestra función.

SAM y FunctionalInterface

SAM (*Single Abstract Method)* indica que tenemos una interfaz con un solo método sin definir, con esto SAM nos permite crear nuestras propias interfaces. En Java 8 se agrego la etiqueta *@FunctionalInterface* para indicar que nuestra interfaz se usara como función.

Creando una interfaz:

```
public class AgeUtils {
    @FunctionalInterface
    interface  TriFunction <T, U, V, R>{
        R apply(T t, U u, V v );
    }
}
```

Esta sintaxis nos indica que los datos que recibirá y devolverá la función son genéricos, es decir, se desconoce de que tipo serán hasta que el usuario de la interfaz defina los tipos.

Método de nuestra interfaz(sin definir)

Datos que recibe

Dato que devuelve

Usando nuestra interfaz:

```
public static void main(String[] args) {
    Function<Integer, String> addCeros = x -> x < 10 ? "0" + x : String.valueOf(x);

    TriFunction<Integer, Integer, Integer, LocalDate> parseDate =
            (day, month, year) -> LocalDate.parse(year + "-" + addCeros.apply(month) + "-" +
                    addCeros.apply(day));

    TriFunction<Integer, Integer, Integer, Integer> calculateAge =
            (day, month, year) -> Period.between(
                    parseDate.apply(day, month, year), LocalDate.now()
                    ).getYears();
}
```

Indicamos los datos que recibirá y devolverá

Definimos nuestra función

```
System.out.println(calculateAge.apply( t: 10, u: 10, v: 1992));
```

Invocamos a nuestra función con el método apply() que creamos en nuestra SAM.

Operador de Referencia

Java nos permite integrar programación funcional a nuestros proyectos donde ya están definidas nuestras clases, métodos, objetos, etc. Con esto podemos hacer referencia a esos métodos desde nuestras funciones a partir de un operador.

```
public static void main(String[] args) {

    List<String> profesores = getList( ...elements: "Nicolas", "Juan", "Zulema");

    //Imprimiendo la lista sin usar operador de referencia
    Consumer<String> printer = text -> System.out.println(text);
    profesores.forEach(printer);

    //imprimiendo la lsita usando operador de referencia
    profesores.forEach(System.out::println);
}

static <T> List<T> getList(T... elements){
    return Arrays.asList(elements);
}
```

Indica que recibirá una cantidad indefinida de elementos.

Operador de referencia

Para poder hacer referencia a otro método es necesario que nuestra función reciba el mismo parámetro(tipo de dato) y genere el mismo resultado que el método al cual hacemos referencia.

Inferencia de tipos

En programación funcional con Java no es necesario indicar explícitamente los tipos de datos que recibirán nuestras funciones, esto sucede porque en tiempo de compilación java corrobora, basado en la definición de nuestra función que los datos que se pasan a través de esta sean del tipo que corresponden.

```
List<String> alumnos = NombresUtils.getList( ...elements: "Hugo", "Paco", "Luis");
alumnos.forEach((String name) -> System.out.println(name));
```

Si estuviéramos obligados a indicar el tipo de dato seria así.

```
alumnos.forEach(name -> System.out.println(name));
```

Java infiere que el dato de entrada que requiere forEach() es un String, no es necesario indicarlo de forma explicita.

```
alumnos.forEach(System.out::println);
```

Incluso puedes utilizar directamente un operador de referencia.

Comprendiendo la sintaxis de las funciones lambda

```
List<String> cursos = NombresUtils.getList( ...elements: "Java", "Functional");

cursos.forEach(curso -> System.out.println(curso));
```

Sabemos que es una lambda por que no esta
Almacenada en ningún lugar, si quisiéramos utilizarla en otra parte de nuestro código no podríamos.

```
usarZero(() -> 2);
```

Este es un ejemplo de una lambda que no recibe argumentos.

```
static void usarPredicado(Predicate<String> predicado){}
usarPredicado(text -> text.isEmpty());
```

En este ejemplo pasamos una lambda dentro de un predicado.
Observa que no tenemos que definir el tipo de dato que recibe o retorna.

```
static void usarBiFunction(BiFunction<Integer, Integer, Integer> operacion ){}
```

En este ejemplo usamos una lambda dentro de una BiFunction,
esta es la sintaxis de una lambda cuando se requieren mas de un parámetro.

```
usarBiFunction((x, y) -> {
    System.out.println("X: " + x + ", Y: " + y);
    return x - y;
});
```

Hay ocasiones en las cuales nuestras lambdas ocupan mas de una línea,
nota que la sintaxis es de la sig. Forma () -> {}.
En una lambda sencilla no es necesario
indicar explícitamente el return, sin embargo en nuestro ejemplo
el cuerpo de nuestra lambda al ser mas grande java necesita tener de forma
explicita en donde tendrá el retorno.

```
    @FunctionalInterface
    interface OperarNada{ void nada();}
}
```

```
usarNada(() -> {
    System.out.println("Hola");
});
```

Existen funciones en las cuales no se recibe ningún dato,
pero tampoco retornan algún valor en este caso usamos () -> {}
como el ejemplo anterior.

Chaining

Seguramente has visto algún fragmento de código similar a la siguiente sintaxis:
Clase clase = new Clase();
clase.metodo1().metodo2().metodo3();
A esto se le conoce como chaining donde encadenamos el resultado de una ejecución con respecto a otra ejecución.

```
StringBuilder stringBuilder = new StringBuilder();
stringBuilder.append("Hola")
        .append("alumno")
        .append("de")
        .append("Platzi");
```

Entendiendo la composición de funciones

En clases anteriores revisamos qué condiciones se cumplen para tener una *high order function,* en donde estas requerían recibir como parámetro o devolver como resultado una función o ambas, tener este tipo de funciones nos permite generar composición de funciones, es decir, encadenar o asociar nuestras lambdas y funciones.

```
Function<Integer,Integer> multiplyBy3 = x -> {
    System.out.println("Multiplicando por x: " + x + "Pror 3");
    return x * 3;
};

Function<Integer, Integer> add1MultypplyBy3 =
        multiplyBy3.compose(y -> {
            System.out.println("Le agregue 1 a: " + y);
            return y + 1;
});
```

El método compose() nos permite componer funciones asociando una función a una lambda, consecuencia de esto la lambda se ejecuta primero y después la función.

```
System.out.println(
        add1MultypplyBy3.apply( t: 5)
);
```

Observa el orden en que se ejecutan las funciones

```
> Task :modules:MathOperations.main()
Le agregue 1 a: 5
Multiplicando por x: 6 Por 3
18
```

1
```
Function<Integer,Integer> multiplyBy3 = x -> {
    System.out.println("Multiplicando por x: " + x + " Por 3");
    return x * 3;
};
```

2
```
Function<Integer, Integer> add1MultypplyBy3 =
        multiplyBy3.compose(y -> {
            System.out.println("Le agregue 1 a: " + y);
            return y + 1;
});
```

3
```
Function<Integer, Integer> andSquare =
        add1MultypplyBy3.andThen(x -> {
            System.out.println("Estoy eleveando " + x + " al cuadrado");
            return x * x;
        });
```

El método andThen() nos permite agregar pasos después de la ejecución de una función, en este caso el orden es el inverso de compose(), primero se ejecuta la función y después la lambda.

```
System.out.println(
        andSquare.apply( t: 3)
);
```

```
> Task :modules:MathOperations.main()
Le agregue 1 a: 3
Multiplicando por x: 4 Por 3
Estoy eleveando 12 al cuadrado
144
```

2 → compose()

1

3 → andThen()

La clase Optional

Un problema bastante común en Java son los ***null***, este tipo de datos indican que es un valor desconocido y pude lanzar una excepción pues el objeto o variable puede no estar definido o inicializado, o talvez un dato inexistente en una base de datos. Podemos programar de forma manual una serie de validaciones cuando iteramos sobre una colección, base de datos etc. para evitar este tipo de datos o podemos usar la alternativa que Java nos ofrece, la clase Optional.

```
List<String> names = getNames();
if (names != null){
    //operar con nombres
}
```

En este ejemplo no se hace uso de Optional y se tiene que hacer validaciones a cada elemento de la lista, este procedimiento puede ser engorroso pues tendríamos que agregar una serie de validaciones para cada caso, por ejemplo la lista que se muestra esta conformada por Strings y este puede estar vacío pero no significa que el elemento sea nulo.

```
Optional<String> optional = Optional.of("Java 8");
```

Una forma de usar Optional es operar sobre un dato que si tenemos en estos casos usamos el método of().

```
optional = Optional.ofNullable(uknownResult());
```

Es posible que no estemos seguros del valor que pondremos en el optional en estos casos podemos usar ofNullable().

```
return Optional.empty();
```

A veces simplemente queremos evitar devolver un null pero no tenemos un valor para retornar.

```
if (optional.isPresent()) {
    dato = optional.get();
}
```

Incluso podemos revisar la presencia de un dato con isPresent() y de ser así obtener ese dato con get().

Como podrás notar Optional nos ofrece muchas formas de controlar los *null,*
si quieres conocer todos los métodos de esta clase te recomiendo que le des un vistazo a la API de java 8 y revises la documentación de Optional, aquí te dejo el link:
https://docs.oracle.com/javase/8/docs/api/

Streams

Java en su versión 8 incluyo la clase Stream.
Stream es una especie de colección, pero se diferencia de estas ya que Stream es auto iterable, es decir con Stream podemos tener datos ya definidos o podemos proveer una manera de ir generando los datos que pasaran mediante este flujo.
Hay una aspecto de los Streams que debemos tener siempre en mente:
"**Los Streams solo pueden ser consumidos una vez**"

```
List<String> courseList = NombresUtils.getList(
        ...elements: "Java",
        "FrontEnd",
        "FullStack"
);

for (String course: courseList) {
    String newCourseName = course.toLowerCase().replace( charSequence: "!",  charSequence1: "!!");
    System.out.println("Platzi: " + newCourseName);
}
```

Puedes observar en el ejemplo que en este tipo de colecciones tenemos que definir que operaciones se van aplicar sobre cada dato al iterar, esto indica que la colección no es auto iterable.

```java
Stream<String> coursesStream = Stream.of(
        "Java",
        "FrontEnd",
        "FullStack");
```

La forma mas sencilla para generar un Stream es usando el método of() en ejemplo generamos un Stream a partir de datos que ya conocemos.

```java
Stream<Integer> courseLenghtStream = coursesStream.map(course -> course.length());
```

Podemos agregar operaciones dentro del Stream.
Aquí, de forma implícita hacemos uso de su principal característica, ser auto iterable.

```java
Stream<String> justJavaCourses = coursesStream.filter(course -> course.contains("java"));
```

Este es otro ejemplo donde hacemos uso de su capacidad auto iterable.

Stream listeners

Habíamos revisado en el tema anterior que un aspecto importante a considerar de los Streams es que solo pueden consumirse una vez, esto nos obliga a almacenar un nuevo tipo de variable por cada operación, lo cual en apariencia parece poco útil, sin embargo nosotros podemos explotar el verdadero potencial de los Streams haciendo uso de un viejo amigo: chaining.

```
Stream<String> emphasisCourses = coursesStream.map(course -> course + "!");
Stream<String> justJavaCourses = emphasisCourses.filter(course -> course.contains("java"));
justJavaCourses.forEach(System.out::println);
```

En este ejemplo se genera una nueva variable por cada operación para poder hacer uso del Stream.

Ahora veamos como usar chaining en nuestros Streams:

```
Stream<String> coursesStream2 = courseList.stream();
```

Con java 8 podemos convertir nuestras colecciones en Streams de forma directa, en este ejemplo usamos la List de la clase anterior.

```
    coursesStream2.map(course -> course + "!!")
            .filter(course -> course.contains("Java"))
            .forEach(System.out::println);

}
```

En este ejemplo observamos lo fácil que es implementar el chaining, es posible que al ser pocos métodos que implementamos en nuestro Stream notemos poca diferencia respecto al ejemplo anterior, !pero imagina que tengamos que usar 5 o 10 operaciones a nuestro Stream, lo engorroso que seria crear una variable por cada método! Eso le restaría legibilidad a nuestro código.

Stream de tipo específico y Paralelismo

Hasta ahora dentro de los Stream solo hemos trabajado con objetos pero esta clase también nos permite trabajar sobre datos primitivos muchos de estos Stream específicos ya vienen definidos en Java.

```
IntStream infiniteStream = IntStream.iterate( i: 0, x -> x + 1);
infiniteStream.limit(1000)
        .parallel()
        .filter(x -> x % 2 == 0)
        .forEach(System.out::println);
```

Valor inicial

Parámetro

En este ejemplo trabajamos sobre un Stream del tipo int sobre el cual vamos a estar iterando.

El método iterate() seguirá iterando infinitamente haste que limitemos el método.

Stream tiene soporte para concurrencia, esto nos es muy útil cuando tenemos una gran cantidad de datos a operar y es necesario usar hilos, para ello hacemos uso del método pararllel(). Es importante mencionar que este método no es idóneo si lo que nos interesa es el orden de los datos.

Operaciones intermedias

Cuando trabajamos con Streams operamos con métodos que pueden devolver otro dato del tipo Stream, cuando esto ocurre sabemos que es un método que realiza una operación intermedia, esto es así porque un Stream esta diseñado para ser eficiente para operar con datos pero no para ser usado por el usuario final, es decir no puedes usar Streams en una API o si tus datos van a ser consumidos por un servicio programado en otro lenguaje estos no pueden consumir Streams.

Algunos ejemplos de operaciones intermedias:

```
Stream<Integer> evenNumbersStream = Stream.iterate( t: 0, i -> i + 1);
evenNumbersStream.filter(i -> i % 2 == 0); //Solo los numeros pares.
```

Este ejemplo toma un Predicate y filtra los elementos para obtener solo los pares para generar un nuevo Stream.

```
Stream<String> heroesNamesStream = Stream.of("Peter", "Logan", "Luisa", "Clark", "Gwen", "Logan", "Peter");
Stream<String> uniqueHeroesNamesStream = heroesNamesStream.distinct();// "Peter", "Logan", "Luisa", "Clark", "Gwen"
```

En este ejemplo generamos un nuevo Stream de elementos únicos.

Existen otras operaciones que no necesariamente operan generando un nuevo Stream sin que alteran el orden del mismo como sorted(), hay otras operaciones mas especificas que convierten el Stream de un tipo a otro como mapTo().

Operaciones finales

Vimos en la clase pasada que las operaciones intermedias generan otro Stream, para darle utilidad a los datos que operamos en un Stream es necesario contar con operaciones finales estas operaciones nos generan un valor final después de iterar los elementos del Stream.

Algunos ejemplos de operaciones finales:

```
Stream<Integer> agesStream = Stream.of(19, 21, 35, 45, 12);
boolean allLegalDrinkingAge = agesStream.allMatch(age -> age > 18);
```

Este ejemplo revisamos si todos los elementos cumplen con cierto Predicate.

```
Stream<Integer> yearsStream = Stream.of(1990, 1991, 1994, 2000, 2010, 2019, 2020);
long yearsCount = yearsStream.count(); //7, solo nos dice cuantos datos tuvo el stream.
```

Podemos generar un long que nos indique cuantos elementos tiene el Stream.

```
Stream<List<String>> coursesStream = getCourses();
Optional<List<String>> coursesOptional = coursesStream.findAny();
```

Podemos devolver cualquier elemento del Stream, incluso un Optional.empty() Si no hay elementos en el Stream.

Collectors

Ahora sabemos que existen operaciones intermedias y finales, sabemos que las intermedias nos devuelven un nuevo Stream y las finales nos devuelven un resultado final, por ejemplo podemos obtener colecciones, arrays, primitivos o incluso JSON si nuestros datos serán consumidos por otros servicios, de esta manera nuestros datos serán mas comprensibles para otros sistemas, los Collectors nos ayudan a recopilar el contenido del Stream en una sola estructura de datos.

```
IntStream infiniteStream = IntStream.iterate( 0, x -> x + 1);
List<Integer>numbersList = infiniteStream.limit(1000)
        .filter(x -> x % 2 == 0)
        .boxed()
        .collect(Collectors.toList());
```

Convierte nuestro Stream en un Stream de datos específicos.

Podemos observar en este Stream una serie de operaciones intermedias y al final genera una nueva estructura de datos, en este caso una lista.

λ

Espero te sea útil.