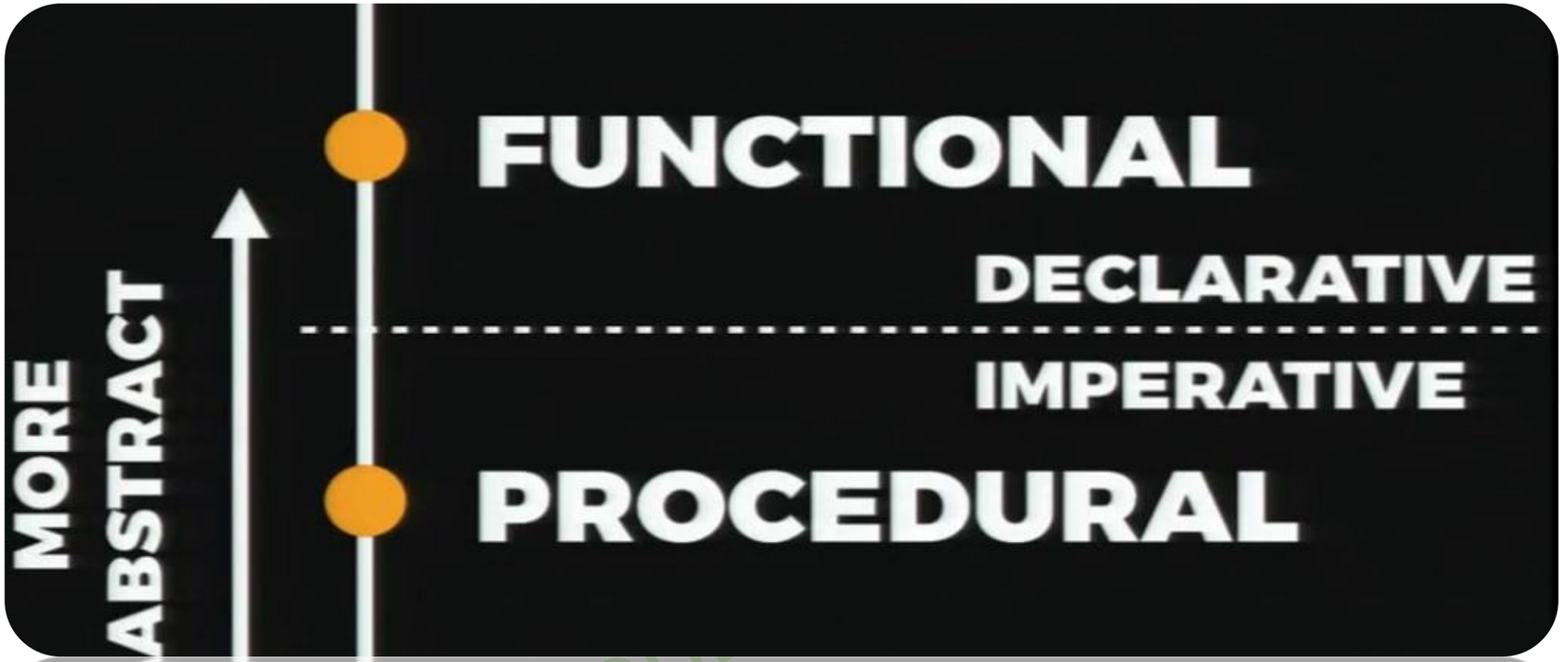


UD X – PROGRAMACIÓN DECLARATIVA EN JAVA

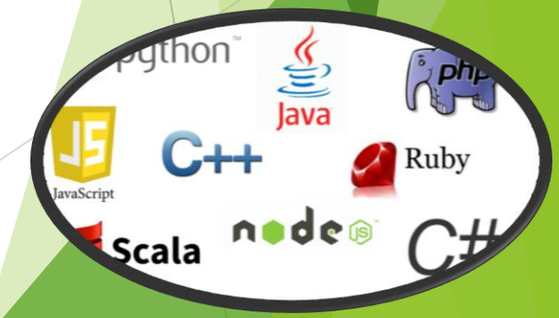
UD X - PROGRAMACIÓN DECLARATIVA EN JAVA
Índice

1 – Motivación

- ¿Te has encontrado con algún código por Internet "lleno de flechitas" y has pensado «eso se debe estudiar en segundo»?
- ¿Se te atragantan algunos bucles? ¿Quieres decirles adiós? ¿Te gustaría poder cambiarlos por algo como lo siguiente?

```
listaEmpleados.stream()
      .filter(empleado -> empleado.getSalario() > 2000)
      .map(empleado -> empleado.getNombre())
      .forEach(System.out::println);
```

- Si has respondido "Sí" a alguna de las preguntas anteriores, presta atención hasta el final del tema para aprender una **nueva forma de programar para tratar colecciones de datos** que incluyen todos los lenguajes de programación modernos.

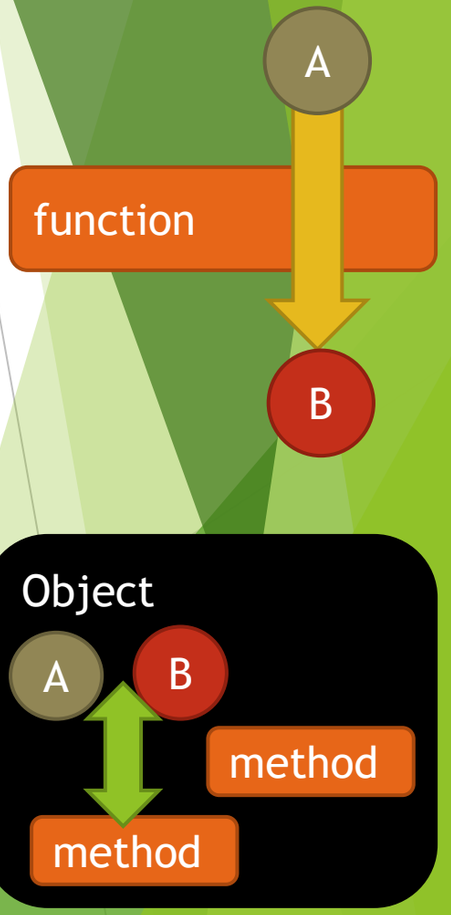


2 – Paradigmas de programación

- Un lenguaje de programación nos proporciona una **caja limitada de herramientas** que nos permiten resolver un problema.
- La misión del desarrollador de software consiste en saber **escoger la secuencia correcta de herramientas** que resuelve un problema para así poder codificarlo.
- El desarrollador/a, por tanto, debe **aplicar un "modelo de pensamiento"** que se ve determinado por la "caja de herramientas" que nos ofrece el lenguaje.
- A este **"modelo de pensamiento"** se le llama **paradigma de programación.** Es una forma concreta de resolver problemas con las herramientas que ofrece un lenguaje.
- Obviamente, el código resultante de un paradigma de programación dado quedará "marcado" con el estilo de pensamiento asociado.

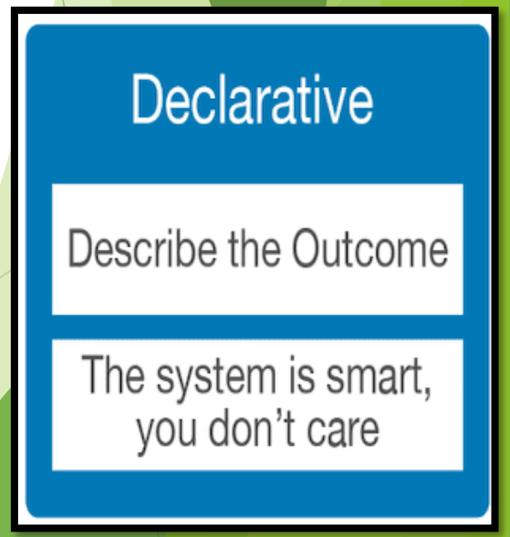
3 – Clasificación de los paradigmas de programación

- Según "**cómo se relacionan los datos y los procesos**" tenemos los siguientes paradigmas:
  - **PROCEDIMENTAL o ESTRUCTURADO**: los procesos se estructuran en funciones para poder ser reutilizados y los datos "pasan casualmente" por estas funciones para ser procesados pero no hay una relación fuerte entre los datos y los procesos.
    - Este modelo de pensamiento toma la **función como "ladrillo"** o pieza básica.
    - Lenguajes: C, Fortran, Cobol, Basic…
  - **ORIENTADO A OBJETOS**: los datos y los procesos se relacionan de forma estrecha porque encierran en una misma pieza de código que es el objeto. Ahora se usan los métodos para procesar los datos (propiedades) almacenados en el objeto.
    - Este modelo de pensamiento toma el **objeto como "ladrillo"** o pieza básica.
    - Lenguajes: Java, C++, Phyton, PHP, Kotlin…



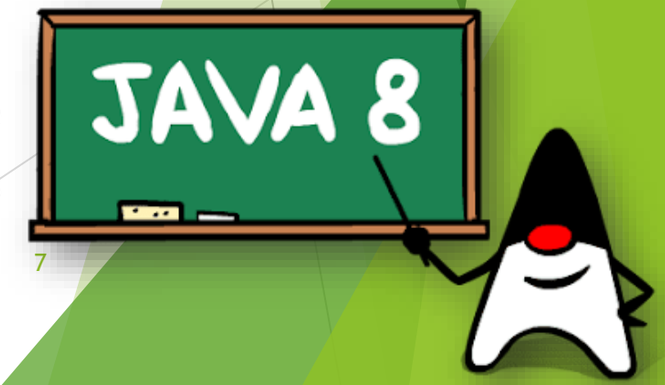
UD X – PROGRAMACIÓN DECLARATIVA EN JAVA
3 – Clasificación de los paradigmas de programación

- Según "**cómo se describe la solución del problema**" tenemos los siguientes paradigmas:
  - **IMPERATIVO**: la solución del problema se describe como una secuencia de órdenes (asignaciones, bucles, condicionales…) que debe realizar el ordenador secuencialmente.
    - Lenguajes procedimentales e imperativos: C, Fortran, Cobol, Basic…
    - Lenguajes orientados a objetos e imperativos: Java, C++, Phyton, PHP, Kotlin…
  - **DECLARATIVO**: la solución al problema se describe mediante una declaración del resultado que esperamos que el ordenador fabrique para nosotros, pero no le damos detalles de cómo tiene que hacerlo.
    - Este enfoque es el que sigue SQL cuando hacemos una "SELECT * FROM …", permitiendo indicar qué resultado queremos sin entrar en el detalle de qué pasos tendrá que dar el ordenador para conseguirlo.
    - Lenguajes declarativos: Lisp, Haskell, ProLog…



4 – Lenguajes multiparadigma

- Con el paso del tiempo se ha comprobado que ciertos problemas se resuelven de una forma más sencilla usando un paradigma concreto.
- Así pues **los lenguajes modernos** están incorporando las cajas de herramientas de otros lenguajes y **se están volviendo multiparadigma**.
- El ejemplo más claro es **JavaScript**. Este lenguaje es orientado a objetos, procedimental, imperativo y declarativo a la vez… vamos, que es todo y nada al mismo tiempo.
- A partir de la versión Java 8 (liberada en 2014), **Java se hace multiparadigma** incorporando el paradigma declarativo que vamos a estudiar en esta unidad.



5 – Expresiones Lambda y referencias a métodos

- Para poder trabajar con la programación declarativa de Java necesitamos hacer una introducción a estas dos técnicas que nos **permiten escribir nuestros métodos/funciones de una forma cómoda y reducida**.
- En el Anexo de esta unidad tenemos más información sobre "programación funcional", que es el término que se usa para englobar este conjunto de técnicas.

EXPRESIONES LAMBDA

- Es una nueva notación nos permite crear **expresiones cuyo tipo es una función/método.** Definiremos por tanto métodos sin nombre (anónimos) que podremos usar como una expresión en nuestro código (en la parte derecha de una asignación, como parámetro de entrada de un método…)

UD X – PROGRAMACIÓN DECLARATIVA EN JAVA
5 – Expresiones Lambda y referencias a métodos

- La sintaxis de esta notación es la siguiente:

Parámetros de entrada	-> Operador flecha	Código del método

- Veamos algunos ejemplos:
  - () -> System.out.println("Hola mundo Lambda");
  - () -> **{** System.out.println("Este bloque de código tiene");
    System.out.println("Más de una instrucción"); **}**
  - (nombre) -> System.out.println("Hola "+nombre);
  - nombre -> System.out.println("Hola "+nombre); // Sin () si hay 1 solo param.
  - (nom, ape) -> System.out.println("Hola "+nom+" "+ape);
  - (nom, ape1, ape2) -> System.out.println("Hola "+nom+" "+ape1+" "+ape2);

LAMBDA

UD X – PROGRAMACIÓN DECLARATIVA EN JAVA
5 – Expresiones Lambda y referencias a métodos

- Si ahora queremos que nuestro método devuelva algo podemos usar la instrucción **return** como siempre:
  - num -> return num + 1;
- Pero también podemos indicarlo **de forma "implícita"** según sea el tipo de la expresión de la última línea del bloque de código:
  - num -> num + 1;

  Esta expresión es equivalente a la anterior. Como la última línea contiene una expresión que devuelve un valor de tipo numérico, el compilador entiende que se ha calculado ese valor para devolverlo y **no hace falta poner el return**.
- Veamos algunos ejemplos más:
  - (n1, n2) -> n1 + n2;
  - () -> Math.random() * 1000;
  - num -> num % 2 == 0; // True si es par y false si es impar

return opcional

UD X – PROGRAMACIÓN DECLARATIVA EN JAVA
5 – Expresiones Lambda y referencias a métodos

- Para terminar, algunos ejemplos más:
  - rango -> Math.random() * rango;
  - (n1, n2) -> n1 > n2;
  - (persona) -> persona.getEdad() >= 18;
  - (empleado) -> empleado.getSalario > 2000;
  - (emp1, emp2) -> { if (emp1.getSalario() > emp2.getSalario()) return 1;
    else if (emp1.getSalario() < emp2.getSalario()) return -1;
    else return 0; }

- Como vemos esta notación nos permite crear de forma cómoda y concisa métodos anónimos. En el siguiente apartado le daremos sentido a todo esto y veremos cómo se insertan estas expresiones lambda en nuestro código. Pero antes tenemos que ver las "*referencias a métodos*".

UD X – PROGRAMACIÓN DECLARATIVA EN JAVA
5 – Expresiones Lambda y referencias a métodos

REFERENCIAS A MÉTODOS

- Observa este código:

  () -> Math.random();

  Estamos definiendo una expresión lambda que simplemente llama al método estático *random* de la clase **Math**.

- Las referencias a métodos nos permiten escribir este tipo de expresiones lambda con otra sintaxis más concisa. El siguiente código es equivalente al anterior:

  Math::random;

- Otros ejemplos:
  - System.out::println;
  - Integer::parseInt;
  - String::compareTo;

¿Ein? Esto es muy RARO…
¿Para qué me sirve todo esto?
En el siguiente apartado se desvelará el "secreto"…

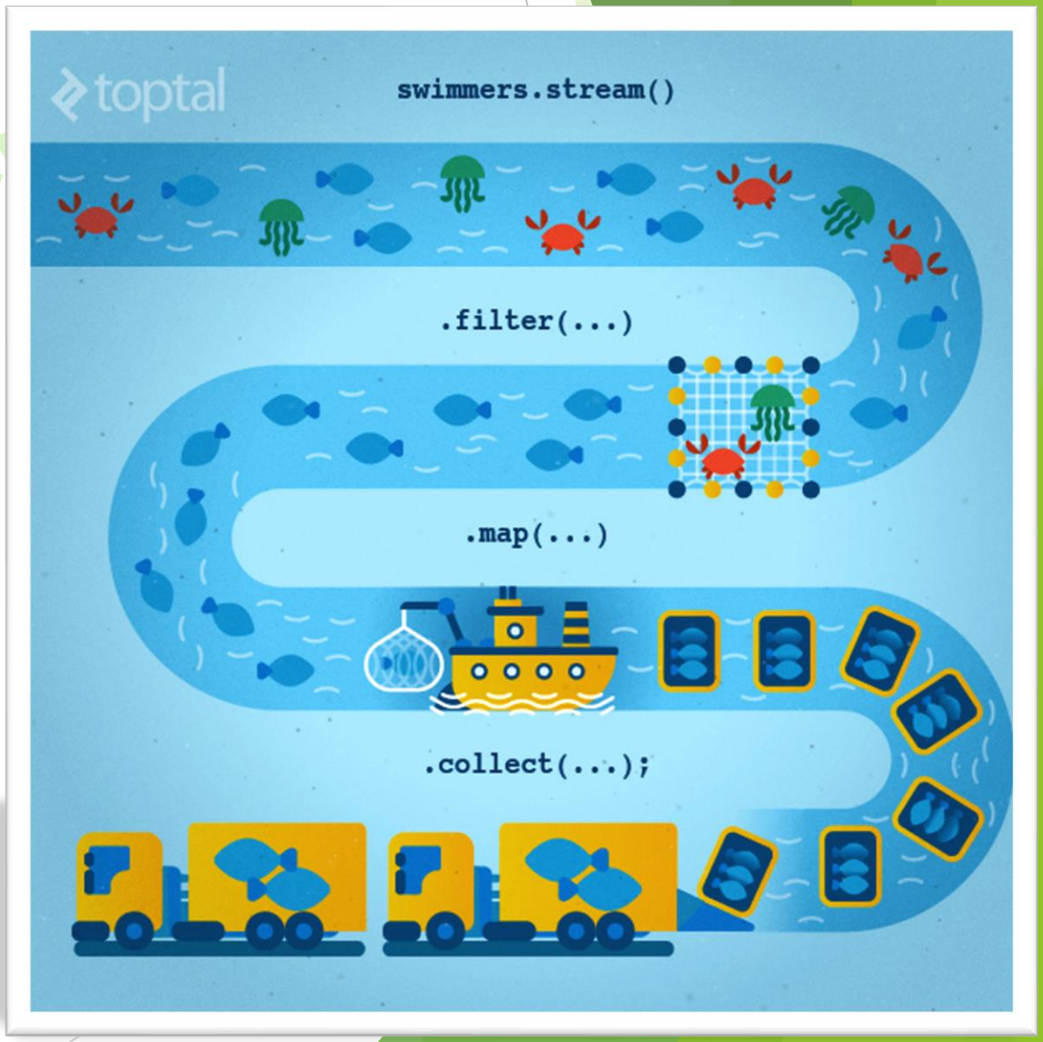
UD X – PROGRAMACIÓN DECLARATIVA EN JAVA
6 – Programación declarativa con el API Stream de Java

6 – Programación declarativa con el API Stream de Java

- Java 8 define un concepto nuevo llamado **Stream** que representa un **flujo de datos** que puede ser tratado por nuestro programa.
- Observa la imagen y después lee lo siguiente:

  «*Por el flujo de un río va una colección de seres acuáticos (objetos). A continuación, filtramos el flujo para dejar pasar solo a los peces. Luego los pescamos y los transformamos en un producto enlatado. Por último, guardamos en un camión de trasporte la nueva colección de productos enlatados*»

- Este es el tipo de tratamiento que el API Stream de Java permite hacer con las colecciones de objetos (arrays, listas, conjuntos…)



UD X – PROGRAMACIÓN DECLARATIVA EN JAVA
6 – Programación declarativa con el API Stream de Java

- Las clases que componen el API Stream se alojan en el paquete **java.util.stream**, sin embargo, la filosofía de los Streams ha "impregnado" otros paquetes del sistema pudiendo "pedirle un flujo de datos" a objetos que antes no ofrecían esa posibilidad. Observa:

```
// Tenemos una lista de objetos Empleado
List<Empleado> lista = Empleado.listAll();

// Le pedimos un stream de Empleado a la lista
Stream<Empleado> empleados = lista.stream();
```

- El API Stream, combinado con la notación lambda, nos permite filtrar, ordenar, transformar, reducir y recolectar los elementos de un flujo de datos de forma cómoda. Veámoslo con un primer ejemplo:

Abrimos el proyecto 'Stream' y el archivo
…a.primerospasos.DemoStream.java

UD X – PROGRAMACIÓN DECLARATIVA EN JAVA
6 – Programación declarativa con el API Stream de Java

Características

- **Los Streams son "de un solo recorrido" e "inmutables"**, es decir que no pueden ser modificados. Lo entendemos en el siguiente código:

Abrimos el proyecto 'Stream' y el archivo
…b.caracteristicas.Caracteristicas.java

Flujo habitual de trabajo con un Stream

- Normalmente, lo que se hace es tomar un objeto como "fuente", pedirle un Stream, realizar una secuencia de **operaciones intermedias** y finalizar con una **operación terminal** que permite obtener el resultado deseado.

Abrimos el proyecto 'Stream' y el archivo
…c.flujotrabajo.FlujoTrabajoHabitual.java

UD X – PROGRAMACIÓN DECLARATIVA EN JAVA
6 – Programación declarativa con el API Stream de Java

Operaciones terminales

- Todos los ejemplos que hemos visto han acabado en la misma operación terminal **foreach(System.out::println)**, que está bien para propósitos didácticos pero no para el día a día de un proyecto. Veamos otras operaciones terminales:

Abrimos el proyecto 'Stream' y el archivo
…d.operacionesterminales.OperacionesTerminales.java

- Hemos visto las operaciones terminales más habituales pero hay más que no estudiaremos: allMatch, anyMatch, noneMatch, count…

Los Streams nos ofrecen muchas opciones

- Existen Streams dedicados a procesar datos de algún tipo primitivo, Streams "paralelos" que aprovechan los distintos núcleos de la CPU, operaciones que permiten hacer lo mismo que con un GROUP BY en SQL…

UD X – PROGRAMACIÓN DECLARATIVA EN JAVA
6 – Programación declarativa con el API Stream de Java

Rendimiento de los Streams

- Los Streams son una nueva estructura de datos que nos permite hacer recorridos por una colección de datos a una "velocidad intermedia":

  Array > *ArrayList* > *LinkedList* > ***Streams*** > *Ficheros* > *Base de datos*

- Dependiendo del tipo Stream que usemos tendremos un mejor o peor rendimiento:
  - *Streams solo para datos primitivos*: son los + rápidos (no los estudiamos)
  - *Streams generales*: son los que hemos estudiado.
  - *Streams paralelos*: son como los anteriores pero distribuyen la carga de trabajo entre los distintos núcleos del procesador. Mejoran el rendimiento cuando hacemos operaciones "pesadas" con cada elemento.
- Mira este estudio comparativo entre ArrayList, Stream y ParallelStream

Abrimos el proyecto 'Stream' y el archivo …e.rendimiento.RendimientoStreams.java

Anexo – Programación funcional con Java

- En este anexo vamos a profundizar más en las expresiones lambda. Veremos cómo se implementan en Java y qué hace el compilador cuando se encuentra con una expresión de este tipo.
- **La programación funcional** es una técnica utilizada en programación declarativa que se basa en que **las funciones se convierten en "ciudadanos de primera clase"**, es decir, las funciones ahora pueden asignarse a variables y ser pasadas/retornadas como parámetros a otros métodos o funciones.
- Para integrar estos nuevos conceptos, los creadores de Java incorporan dos nuevos elementos al lenguaje:
  - La notación Lambda
  - Las interfaces funcionales

UD X – PROGRAMACIÓN DECLARATIVA EN JAVA
Anexo – Programación funcional con Java

- Con los Streams hemos pasado expresiones lambdas como parámetros a un método. Ahora nos queda ver eso de "**asignar funciones a variables".**
- Para conseguirlo de forma cómoda se usa la **notación lambda**. Observa la parte derecha de la asignación:

```
Function<Integer, Integer> funcionIncrementa = num -> num + 1;
```

tipo | variable | expresión lambda

- Como vimos anteriormente, esta notación debe entenderse como:

```
param-entrada -> expresión-a-devolver
```

Abrimos el proyecto 'ProgramacionFuncional' y el archivo …a.base.ProgramacionFuncional.java

UD X – PROGRAMACIÓN DECLARATIVA EN JAVA
Anexo – Programación funcional con Java

- El tipo de dato **Function<T, R>** se trata una **interfaz funcional** que representa a cualquier función que tenga un parámetro de entrada de tipo T y que devuelva un parámetro de salida de tipo R.
- Esta interfaz está definida en el paquete **java.util.function** y, si observamos su documentación, vemos que define un método abstracto (sin código) llamado **apply(T t)** que devuelve un dato de tipo R:

R	**apply**(T t) Applies this function to the given argument.

Para Java, una **Interfaz Funcional** es una interfaz que presenta UN SOLO método abstracto (sin código)

- La "magia" se produce cuando hacemos la asignación:

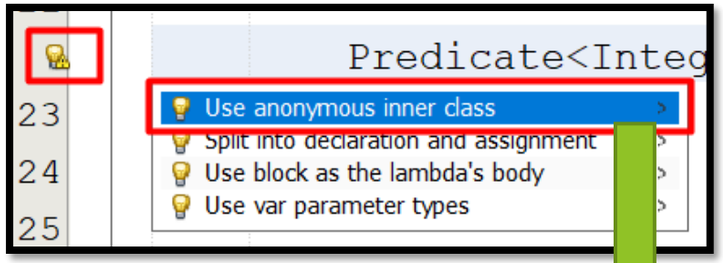
```
Function<Integer, Integer> funcionIncrementa = num -> num + 1;
```

porque la expresión lambda se "inyecta" como código en el método **apply**. Es como si se escribiese lo siguiente:

```
@Override
public Integer apply(Integer num) {
    return num + 1;
}
```

UD X – PROGRAMACIÓN DECLARATIVA EN JAVA
Anexo – Programación funcional con Java

Observa que el IDE nos ofrece la **posibilidad de cambiar a otra notación equivalente** (clases anónimas) al poner el cursor sobre la expresión lambda.




```
Function<Integer, Integer> funcionIncrementa = new Function<Integer, Integer>() {
    @Override
    public Integer apply(Integer num) {
        return num + 1;
    }
};
```

- Como vemos para llamar a la función que hemos inyectado con la expresión lambda escribimos algo como: **funcionIncrementa.apply(4)**

UD X – PROGRAMACIÓN DECLARATIVA EN JAVA
Anexo – Programación funcional con Java

- Vamos a seguir profundizando un poco más en esto de la programación funcional viendo dos conceptos más:
  - Las funciones se pueden **encadenar** permitiendo crear funciones nuevas como combinaciones de otras existentes.

Abrimos el proyecto 'ProgramacionFuncional' y el archivo …b.encadenamiento.Encadenamiento.java

  - Según los parámetros de entrada y de salida que presenta una función podemos hablar de **distintos tipos de funciones**. Veamos los tipos de funciones más usuales definidos en el paquete java.util.function:

Abrimos el proyecto 'ProgramacionFuncional' y el archivo …c.tipos.TiposDeFunciones.java

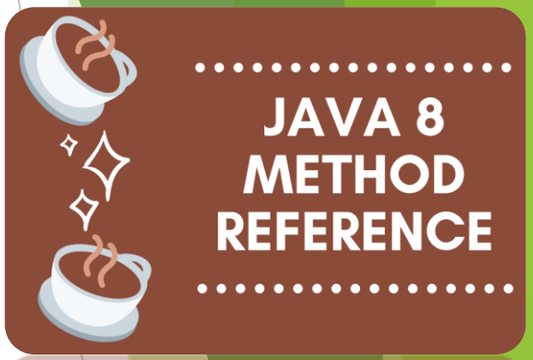
UD X – PROGRAMACIÓN DECLARATIVA EN JAVA
Anexo – Programación funcional con Java

- También podemos asignar a una variable un método ya definido en el sistema, usando la notación de **referencia a métodos**. Observa:

```
Consumer<Object> imprime = System.out::println;
```

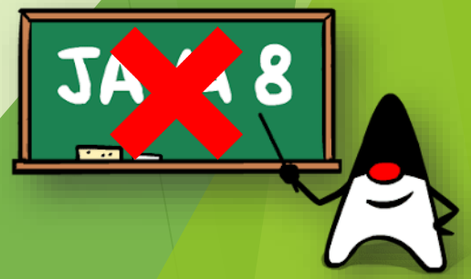
- En este caso estamos refiriéndonos a un método estático, pero esta notación también se puede usar con cualquier método de un objeto, incluso con constructores.

Abrimos el proyecto 'ProgramacionFuncional' y el archivo …d.referenciasmetodos.ReferenciasAMetodos.java



- Aunque las expresiones Lambda se suelen combinar con los Streams, también podrían usarse en el "Java clásico" anterior a la Java 8. Para ello, analiza el siguiente código:

Abrimos el proyecto 'ProgramacionFuncional' y el archivo …e.aplicacion.FuncionesJavaClasico.java



Cierre de la unidad

- Acabamos de hacer una introducción a la programación declarativa en Java. Lo bueno es que **todos estos conceptos se repiten en otros lenguajes** (JavaScript, Phyton, Kotlin…) con variaciones en la sintaxis y/o semántica.
- Aunque la velocidad de ejecución de un código declarativo pueda ser más lenta que la de su código imperativo equivalente, la **solución declarativa es más sencilla, se desarrollará en menos tiempo y será más fácil** de mantener porque es menos propensa a errores. La tendencia actual es que los lenguajes y los frameworks se vayan volviendo cada vez **más declarativos y menos imperativos**.
- Para saber más:
  - 'Java 8' de MitoCode (en español)
    - https://www.youtube.com/watch?v=n2plQQwJes0&list=PLvimn1Ins-419yVe5iPfiXrg4mZJl5kLS&index=12
  - Vídeos de AmigosCode (en inglés)
    - https://www.youtube.com/watch?v=Q93JsQ8vcwY&t=114s
    - https://www.youtube.com/watch?v=f5j1TaJlc0w

Martin Fowler