Programación Orientada a Objetos

Expresiones Lambda

Ing. Aníbal Sardón Paniagua

C16290@utp.edu.pe

anibal.sardon@hotmail.com

Logros de Aprendizaje

Al finalizar la sesión el estudiante:

- Descubre las expresiones Lambda en Java.
- Aprende a utilizar las funciones Lambda sobre Colecciones de objetos.

Recordamos la sesión anterior

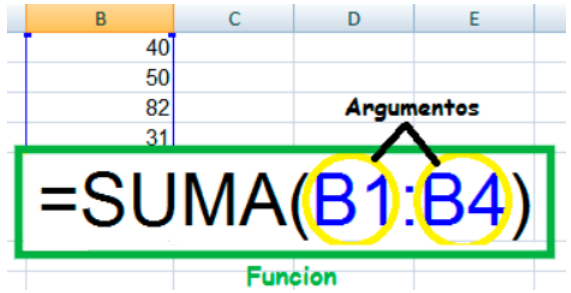
- ¿Qué son las clases genéricas?.
- ¿Cómo son las restricciones de tipos genéricos?

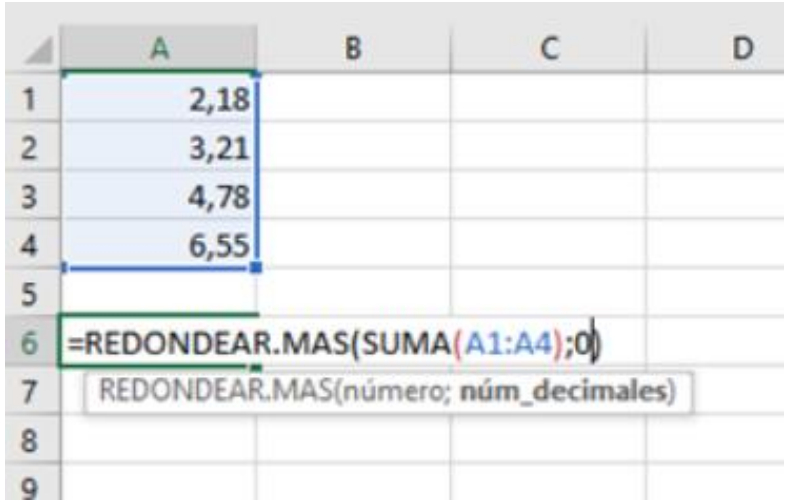
- ¿Dudas de la sesión anterior?

Saberes previos

¿Qué vemos en las imágenes?

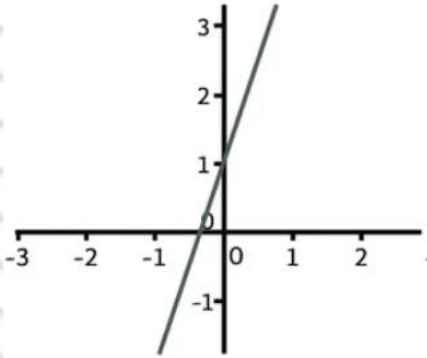
¿Cuál sería la interface?



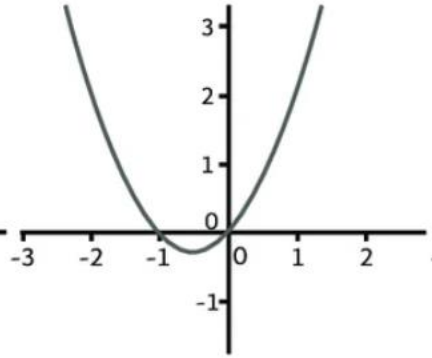


¿Cuál es la importancia del tema?

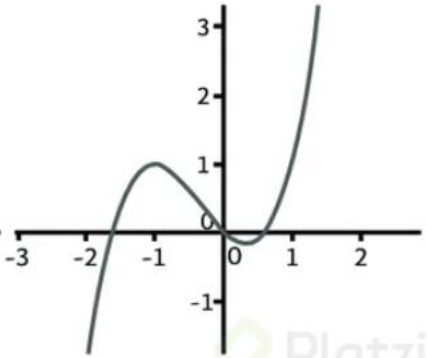
$f(x) = ax + b$ La recta



$f(x) = ax^2 + bx + c$ La parábola



$f(x) = ax^3 + bx^2 + cx + d$ La cúbica



Universidad Tecnológica del Perú

Temas a tratar

- Paradigma de Programación Funcional
- Expresiones Lambda
- Funciones Lambda sobre Colecciones

Programación Funcional

- Paradigma de programación donde la ejecución del programa no es más que la evaluación de funciones. *(cálculo Lambda – Alonso Church)*
- En programación funcional, un programa es un conjunto de funciones matemáticas que convierten unas entradas en unas salidas.
- Sin ningún estado *interno (inmutable – el dato interno en las variables no cambia)* y ningún efecto lateral *(el valor interno de la variable no cambia por otra sentencia)*

```
(cuadrado (+ 10 (cuadrado (+ 2 4)))) ; ⇒ 2116
```

```
(cuadrado (+ 10 (cuadrado (+ 2 4)))) ⇒
(cuadrado (+ 10 (cuadrado 6))) ⇒
(cuadrado (+ 10 36)) ⇒
(cuadrado 46) ⇒
2116
```

Programación Funcional

Universidad Tecnológica del Perú

Ejemplo: Comparación entre paradigma imperativo vs funcional

```
List<Integer> numeros = List.of(18, 6, 4, 15, 55, 78, 12, 9, 8);
```

- **Imperativo:**

```
int contador = 0;
for(int numero : numeros) {
    if(numero > 10) {
        contador ++;
    }
}
System.out.println(contador);
```

- **Funcional:**

```
Long result = numeros.stream()
                     .filter(num -> num > 10)
                     .count();
System.out.println(result);
```

Programación Funcional

- Características de la programación **imperativa**
  - Variable = nombre de una zona de memoria
  - Asignación
  - Referencias
  - Pasos de ejecución
- Características de la programación **funcional**
  - Variable = nombre dado a un valor *(declaración)*
  - En lugar de pasos de ejecución se utiliza la composición de funciones
  - No existe asignación ni cambio de estado.
  - No existe mutación, se cumple la transferencia referencial:
    - Dentro de un mismo ámbito todas las ocurrencias de una variable y las llamadas a funciones devuelven el mismo valor

Expresiones Lambda

- Término adoptado de la programación funcional y corresponden con funciones de Java que normalmente son anónimas.
- Recibe cero o más argumentos y devuelven uno o ningún valor de retorno.
- Se declaran al mismo tiempo en donde se usa, y puede acceder a las variables locales del ámbito al que pertenece.
- Compuesta por dos elementos, separados por una **flecha** →
  - Parte **izquierda** de la flecha → Parámetros de entrada.
  - Parte **derecha** de la flecha → Expresión Lambda.

Ejemplo

Parámetros de entrada | Expresión lambda

(int x) ⟶ x + 1

n ⟶ n > 10

Expresiones Lambda

- Una función lambda es como una clase con un único método público.
- Devuelve lo que devuelve la operación.
- Comportamiento depende de donde se use *(filtrar, ordenar).*
- Permite ejecutar código *(ejemplo, hacer un System.out.println)*

```
() -> "Hello"

() -> System.out.println("Hello")

(String str) -> str.length()

(str) -> str.length()

str -> str.length()

(int i, int j) -> i + j

(i, j) -> i + j


() -> { System.out.println("Hello"); System.out.println("World"); }

(int i) -> { System.out.println("Hello"); return i; }
```

Expresiones Lambda en Colecciones

- La **API Streams** representan flujos de datos que estarán asociadas a una colección.
- Se utiliza para definir una operación que será aplicada a los elementos contenidos en la Colección.
- Permite procesar datos de modo declarativo.

Ejemplo:

```
Long result = numeros.stream()
                     .filter(num -> num > 10)
                     .count();
System.out.println(result);
```

Ejemplo:

```
Stream<Persona> stream = personas.stream();

List<String> nombres = stream

   // filtrado de los elementos que no tienen
nombre null
   .filter(p -> p.nombre!=null)

   // aplicar una conversión, de Persona a String
   // quedándonos con el nombre
   .map(p -> p.nombre)
   // a partir de aquí se tiene un Stream<String>

   // recolectar los elementos en una lista
   .collect(Collectors.toList());
```

Expresiones Lambda en Colecciones

- Ejemplo: Filtrado de elementos de una Lista de objetos Producto

```
class Producto {
  private int precio;
  public Producto(int p) { this.precio = p; }
  public int getPrecio() { return this.precio; }
}

List<Producto> productos = Arrays.asList(
                  new Producto(20),
                  new Producto(40),
                  new Producto (5));
```

Programación Imperativa

```
List<Producto> descuentos = new ArrayList<Producto>();
for (Producto p : productos)
    if (p.getPrecio()>10.0)
      descuentos.add(p);
```

Expresión Lambda

```
List<Producto> descuentos = productos.stream().
          filter(p -> p.getPrecio()>10.0).     // filtramos los > 10
          collect(Collectors.toList());        // los ponemos en una lista
```

Expresiones Lambda en Colecciones

- Ejemplo: Secuencia de datos con operaciones de agregación

```
List<Integer> numbers =
        Arrays.asList(1, 2, 3, 4, 5);
```

Programación Imperativa

```
int sum = 0;
for (int n : numbers) {
  if (n % 2 == 1) {
    int square = n * n;
    sum = sum + square;
  }
}

System.out.println(sum);
```

Expresión Lambda

```
int sum = numbers.stream()
          .filter(n -> n % 2 == 1)
          .map(n -> n * n)
          .reduce(0, Integer::sum);

System.out.println(sum);
```

Expresiones Lambda en Colecciones

Operaciones Intermedias

- Son ejecutadas en forma “lazy”, es decir no se ejecutan hasta que se alcanza una operación terminal.
  - **filter**: devuelve un flujo de elementos que satisfacen el predicado pasado como parámetro a la operación. Es decir, filtra los elementos de la colección mediante un predicado.
  - **map**: devuelve un flujo de elementos después de que hayan sido procesados por la función pasada como parámetro.
  - **distinct**: es un caso especial de la operación de filtro. **Distinct** devuelve un flujo de elementos de modo que cada elemento es único en el flujo. *(Elimina duplicados)*

Function	Preserves count	Preserves type	Preserves order
map	✅	❌	✅
filter	❌	✅	✅
distinct	❌	✅	✅
sorted	✅	✅	❌
peek	✅	✅	✅

Expresiones Lambda en Colecciones

Operaciones Terminales

- Esta operación iniciará la ejecución de todas las operaciones “Lazy” anteriores presentes en la secuencia. Las operaciones de terminal devuelven tipos concretos.
  - **reduce:** puede llamar a la operación concreta como, Integer :: sum. *(Acumula elementos)*
  - **forEach:** no devuelve un tipo concreto, pero puede agregar un efecto secundario como imprimir cada elemento.
  - **collect:** es un tipo especial de **reduce** que toma todos los elementos de la secuencia y puede producir un **Set , Map o List.**

Function	Output	When to use
reduce	concrete type	to cumulate elements
collect	list, map or set	to group elements
forEach	side effect	to perform a side effect on elements

Expresiones Lambda en Colecciones

Universidad Tecnológica del Perú

Ejemplos:

- Obtenga los apellidos únicos en mayúsculas de los primeros 15 autores de libros que tengan 50 años o más.

```
library.stream()
        .map(book -> book.getAuthor())
        .filter(author -> author.getAge() >= 50)
        .map(Author::getSurname)
        .map(String::toUpperCase)
        .distinct()
        .limit(15)
        .collect(toList()));
```

Expresiones Lambda en Colecciones

Ejemplos:

- Imprima la suma de las edades de todas las autoras menores de 25 años.

```
library.stream()
        .map(Book::getAuthor)
        .filter(author -> author.getGender() == Gender.FEMALE)
        .map(Author::getAge)
        .filter(age -> age < 25)
        .reduce(0, Integer::sum)
```

Expresiones Lambda en Colecciones

Ejemplos:

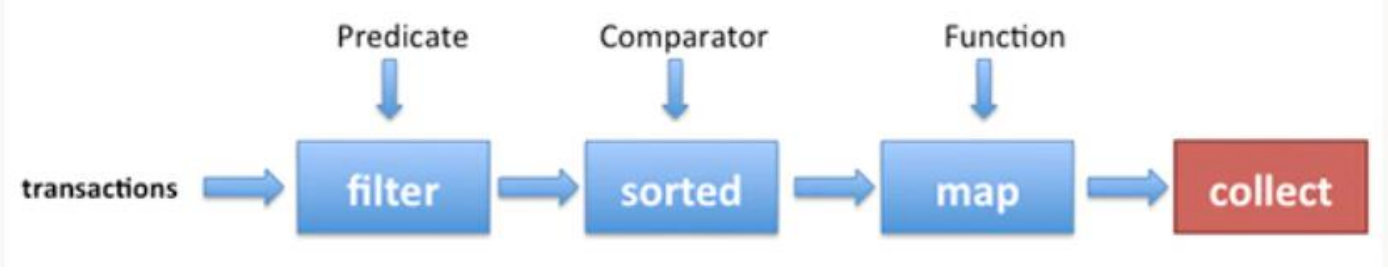
- Encontrar todas las transacciones del tipo grocery y obtener un listado de identificaciones de transacciones ordenadas de mayor a menor por valor de transacción.

```
List<Transaction> groceryTransactions = new Arraylist<>();

for(Transaction t: transactions){
    if(t.getType() == Transaction.GROCERY){
        groceryTransactions.add(t);
    }
}
Collections.sort(groceryTransactions, new Comparator(){
    public int compare(Transaction t1, Transaction t2){
        return t2.getValue().compareTo(t1.getValue());
}
});
```

```
List<Integer> transactionIds = new ArrayList<>();
for(Transaction t: groceryTransactions){
    transactionsIds.add(t.getId());  }
```

```
List<Integer> transactionsIds =
transactions.stream()
.filter(t -> t.getType() == Transaction.GROCERY)
.sorted(comparing(Transaction::getValue).reversed())
.map(Transaction::getId)
.collect(toList());
```



Preguntas

Manos a la Obra, a programar ….

Conclusiones

- ¿Qué aprendiste en esta sesión?
- Te invitamos a compartir tus conclusiones en clase.

Resumen

1. En programación funcional un programa es un conjunto de funciones matemáticas que convierten unas entradas en unas salidas.
2. Una expresión Lambda es un término adoptado de la programación funcional y corresponden con funciones de Java que normalmente son anónimas
3. La API Streams representan flujos de datos que estarán asociadas a una colección.
4. Se utiliza para definir una operación que será aplicada a los elementos contenidos en la Colección


UTP
Universidad
Tecnológica
del Perú