Funciones de alto orden

Matemática estructural y lógica

ISIS-1104

Esta clase solo compila en Java 8 o superior

Supongan que tenemos el siguiente método

```
public static List<Integer> sum2(List<Integer> lista) {
    ArrayList<Integer> nLista = new ArrayList<Integer>();
    for (Integer i: lista) {
        nLista.add(i + 2);
    }
    return nLista;
}
```

Y este otro método

```
public static List<Integer> mul2(List<Integer> lista) {
    ArrayList<Integer> nLista = new ArrayList<Integer>();
    for (Integer i: lista) {
        nLista.add(i * 2);
    }
    return nLista;
}
```

Ingenieria de software 101

- Estos métodos son iguales salvo la función que estoy **aplicando** sobre los elementos de la lista.
- ¿Es posible volver ésta función **argumento** de mi método?
- Justamente este es el problema que resuelven las funciones de **alto orden**.

Funciones como variables

En Java 8, `Function<U, V>` es una función que recibe una variable de tipo `U` y retorna una variable de tipo `V`

```
Function<Integer, Integer> doblar = i -> 2 * i;
Function<Integer, Boolean> esPositivo = b -> b > 0;
Function<String, Boolean> esVacio = s -> s.length() == 0;
```

Funciones de alto orden: `map`

Tengamos en cuenta el siguiente método

```
public static List<Integer> map(List<Integer> lista,
    Function<Integer, Integer> op) {
    ArrayList<Integer> nLista = new ArrayList<Integer>();
    for (Integer i: lista) {
        nLista.add( op.apply(i) );
    }
    return nLista;
}
```

Funciones de alto orden: `map`

Ahora podemos rescribir `sum2` y `mul2`:

```
public static List<Integer> sum2(List<Integer> lista) {
    return map(lista, x -> 2 + x);
}

public static List<Integer> mul2(List<Integer> lista) {
    return map(lista, x -> 2 * x);
}
```

Funciones de alto orden: `map`

Podemos mejorar `map` aun mas

```
public static List<Integer> map(List<Integer> lista,
    Function<Integer, Integer> fun) {
    ArrayList<Integer> nLista = new ArrayList<Integer>();
    for (Integer i: lista) {
        nLista.add( fun.apply(i) );
    }
    return nLista;
}
```

Funciones de alto orden: `map`

Usando *generics* podemos hacer que funcione para cualquier tipo

```
public static <T> List<T> map(List<T> lista,
    Function<T, T> fun) {
    ArrayList<T> nLista = new ArrayList<T>();
    for (T i: lista) {
        nLista.add( fun.apply(i) );
    }
    return nLista;
}
```

Funciones de alto orden: `map`

Ahora las siguientes funciones pueden implementarse usando `map`

```
public static List<Boolean> negar(List<Boolean> lista) {
    return map(lista, x -> !(x));
}

public static List<String> admirar(List<String> lista) {
    return map(lista, x -> "¡" + x + "!");
}
```

Funciones de alto orden: `filter`

Supongan que tenemos el siguiente método

```
public static List<Integer> pos(List<Integer> lista) {
    ArrayList<Integer> nLista = new ArrayList<Integer>();
    for (Integer i: lista) {
        if (i >= 0) {
            nLista.add(i);
        }
    }
    return nLista;
}
```

Funciones de alto orden: `filter`

Y también tenemos este otro método

```
public static List<Integer> neg(List<Integer> lista) {
    ArrayList<Integer> nLista = new ArrayList<Integer>();
    for (Integer i: lista) {
        if (i < 0) {
            nLista.add(i);
        }
    }
    return nLista;
}
```

Funciones de alto orden: `filter`

Necesitamos una función similar a `map`, pero que sea capaz de filtrar en vez de aplicar

```
public static <T> List<T> filter(List<T> lista,
    Function<T, Boolean> pred) {
    ArrayList<T> nLista = new ArrayList<T>();
    for (T i: lista) {
        if( pred.apply(i) ) {
            nLista.add(i);
        }
    }
    return nLista;
}
```

Funciones de alto orden: `filter`

Ahora podemos rescribir `pos` y `neg`:

```
public static List<Integer> pos(List<Integer> lista) {
    return filter(lista, x -> x >= 0);
}

public static List<Integer> neg(List<Integer> lista) {
    return filter(lista, x -> x < 0);
}
```

Funciones de alto orden: `fold`

Supongan que tenemos el siguiente método

```
public static Integer superSuma(List<Integer> lista) {
    Integer acum = 0;
    for (Integer i: lista) {
        acum = acum + i;
    }
    return acum;
}
```

Funciones de alto orden: `fold`

Y también tenemos este otro método

```
public static Boolean superO(List<Boolean> lista) {
    Boolean acum = false;
    for (Boolean b: lista) {
        acum = acum || b;
    }
    return acum;
}
```

Funciones de alto orden: `fold`

Para reescribir ambos métodos introducimos una función capaz de `reducir` una lista a un único valor

```
public static <T> T fold(List<T> list,
  BiFunction<T, T, T> op,
  T init) {
  T acum = init;
  for (T elem: list) {
      acum = op.apply(acum, elem);
  }
  return acc;
}
```

Funciones de alto orden: `fold`

Ahora podemos rescribir `superSuma` y `superO`:

```
public static Integer superSuma(List<Integer> lista) {
    return fold(lista, (x, y) -> x + y, 0);
}

public static List<Integer> superO(List<Integer> lista) {
    return filter(lista, (x, y) -> x || y, false);
}
```

En resumen

Si tengo una `lista`:

- `map` me permite **aplicar** una **función** a todos sus elementos.
- `filter` me permite **filtrar** sus elementos usando un **predicado**.
- `fold` me permite **acumular** sus elementos usando una **operación**.