

Ingeniería de Sistemas e Informática

Effective Java

Mg. Fernando Díaz Sánchez

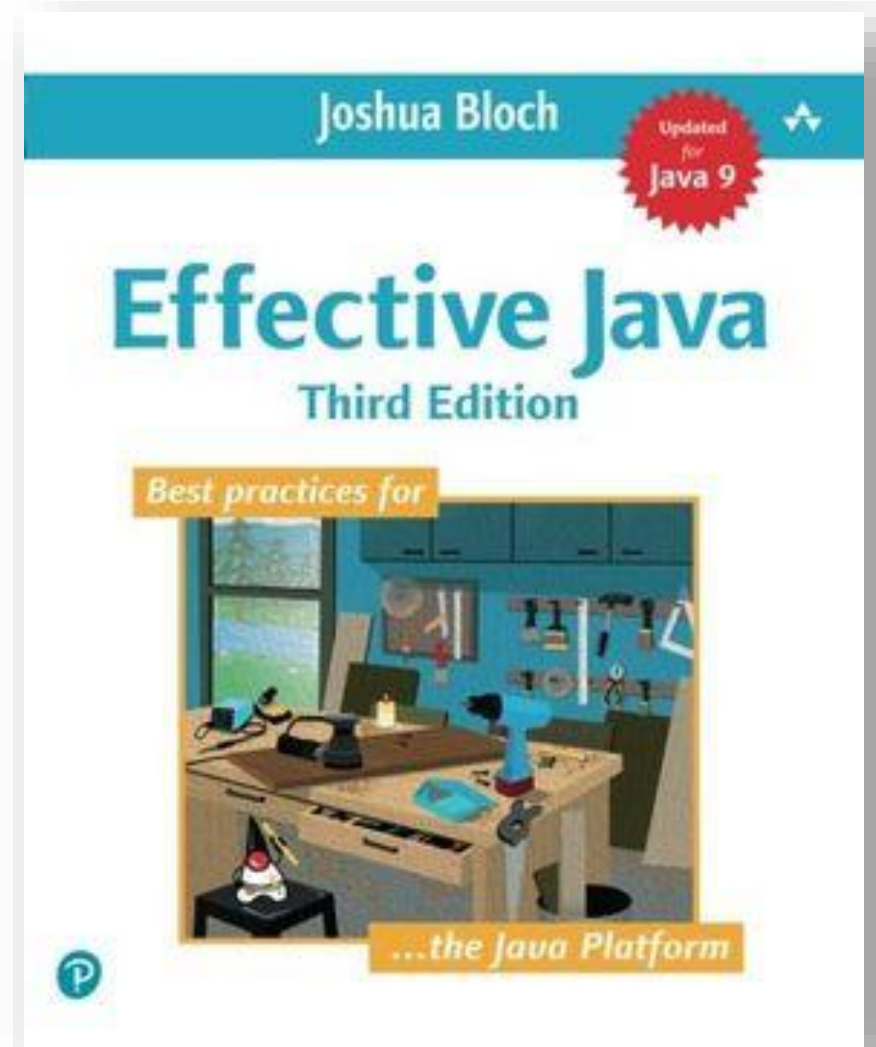
Antes de empezar

Código para la sesión de hoy

https://github.com/utpcix/effective_java.git

¿Por qué es importante?

- Sirven para brindar claridad y simplicidad al código. (Bloch, 2018)
- El arte de programar, al igual que en otras disciplinas consiste en aprender las reglas y comprender cuando se rompen y los efectos que conlleva
- Muestran un camino para que el programador mejore su nivel de manera notable



Creating and Destroying Objects

Common Methods

Classes and Interfaces

Enums

Lambdas and Streams

Methods

Exceptions

Usa Static Factory Methods

- Una clase debe brindar a sus clientes SFM en lugar de constructores o en conjunto con ellos. (Bloch, 2018)
- A diferencia de los constructores, los SFM tienen un nombre que al ser bien elegida resulta en un código más legible
- Si el programador lo decide, puede crear o no nuevos objetos o devolver subtipos de la clase

Creating and Destroying Objects



Usa Static Factory Methods

Nombre Estándar	Descripción	Ejemplo
from	Método conversor. Toma un parámetro y devuelve una instancia de la clase	<code>LocalDate.from(accesor_time)</code>
of	Método agregador. Toma uno o varios parámetros y retorna una instancia de la clase	<code>List.of("A", "B", "C")</code>
valueOf	Método alternativo a from y of	<code>Integer.valueOf("35")</code>
getInstance	Retorna una instancia que es descrita por sus parámetros (no garantiza nuevos objetos)	<code>Pool.getInstance(options)</code>
create	Igual que getInstance, pero garantiza nuevos objetos	<code>Array.newInstance(String.class,10)</code>
getType	Como getInstance pero de un tipo de dato especificado	<code>File.getFileStore(path)</code>

Static Factory Methods – Caso de Uso

Se requiere una clase sencilla para guardar las canciones y sus artistas teniendo en cuenta las siguientes consideraciones

- La clase no debe permitir la creación de instancias con el operador `new`
- La clase puede crear una lista de instancias a partir de un archivo csv
- La clase puede crear una instancia a partir de un objeto del mismo tipo
- El constructor debe ser reemplazado con el método `valueOf`

Song
-artist: String -title: String
<u>+from(file: String): List<Song></u> <u>+of(song: Song): Song</u> <u>+valueOf(artist: String, title: String): Song</u>

Static Factory Methods – Caso de Uso

Creating and Destroying Objects

```
11 public final class Song {
12     private final String artist;
13     private final String title;
14
15     private Song(String artist, String title) {
16         this.artist = artist;
17         this.title = title;
18     }
19
20     @ public static Song of(Song song){
21         return new Song(song.artist, song.title);
22     }
23
24     @ public static Song valueOf(String artist, String title){
25         return new Song(artist, title);
26     }
```

```
28 @ public static List<Song> from(String path)
29     throws IOException {
30     List<Song> lista = new ArrayList<>();
31     List<String> lineas = TextUTP.readlines(path,
32         TextUTP.OS.WINDOWS);
33     for (String linea : lineas) {
34         String[] data = linea.split(regex: ";");
35         String artist = data[0];
36         String title = data[1];
37         lista.add( new Song(artist, title) );
38     }
39     return lista;
40 }
```


Sobrecargar el método equals y hashCode

- Toda clase concreta debe tener un contrato general explícito mediante el uso del método equals y hashCode (Bloch, 2018)
- Se debe tener mucho cuidado con este método si se utilizarán junto con estructuras como HashMap o HashSet, quienes dependen directamente de él
- Debemos asegurarnos que al sobrecargar el método equals, cumplamos la **relación de equivalencia**

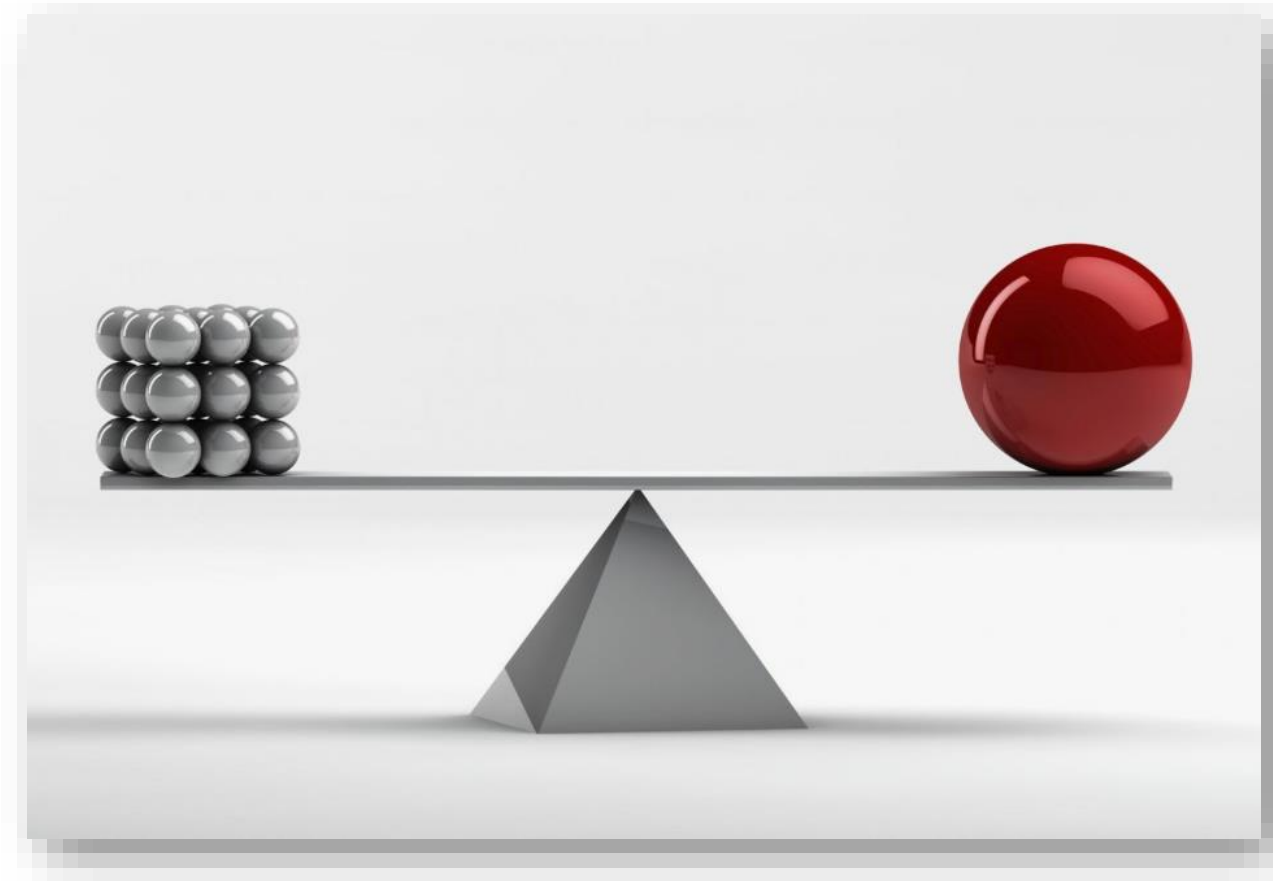


Imagen extraída de:

https://images.inc.com/uploaded_files/image/1920x1080/getty_163740246_77306.jpg

Sobrecargar el método equals y hashCode

Relación de equivalencia

Propiedad	Descripción
Reflexivo	Para cualquier referencia no-null del valor <code>x</code> , la operación <code>x.equals(x)</code> debe retornar <code>true</code>
Simétrico	Para cualquier referencia no-null del valor <code>x</code> , <code>y</code> , la operación <code>x.equals(y)</code> debe retornar <code>true</code> sí y solo si <code>y.equals(x)</code> retorna <code>true</code>
Transitivo	Para cualquier referencia no-null del valor <code>x</code> , <code>y</code> , <code>z</code> , si <code>x.equals(y)</code> retorna <code>true</code> y <code>y.equals(z)</code> retorna <code>true</code> , entonces <code>x.equals(z)</code> debe retornar <code>true</code>
Consistente	Para cualquier referencia no-null del valor <code>x</code> , <code>y</code> , el resultado de múltiples invocaciones de <code>x.equals(y)</code> debe retornar el mismo resultado

Nota: `x.equals(null)` siempre debe retornar `false`

Sobrecargar el método equals y hashCode

```
31      @Override
32      public boolean equals(Object o) {
33          if (this == o) return true;
34          if (!(o instanceof Software software))
35              return false;
36          return getMajor_version()
37              == software.getMajor_version()
38              && Objects.equals(getName(),
39                              software.getName());
40      }
```

Software
-name: String
-int: major
-int: minor
+equals(): boolean
+hashCode(): int

```
38      @Override
39      public int hashCode() {
40          // Joshua Bloch Method
41          //      int result = Objects.hashCode(getName());
42          //      result = 31 * result + Objects.hashCode(getMajor_version());
43          //      return result;
44
45          // Primes Method
46          int P = 16908799;
47          int hash = 0;
48          String key = getName() + getMajor_version();
49          for (int i = 0; i < key.length(); i++) {
50              hash = (127 * hash + key.charAt(i)) % P;
51          }
52          return hash;
53
54          // IntelliJ IDEA method
55          //      return Objects.hash(getName(), getMajor_version());
56      }
```

Sobrecargar el método equals y hashCode

```
// 1. Reflexive: x.equals(x) == true
System.out.println("1. Reflexive: x.equals(x) == true");
System.out.println("s1.equals(s1) = " + s1.equals(s1));

// 2. Symmetric: x.equals(y) == y.equals(x)
System.out.println("2. Symmetric: x.equals(y) == y.equals(x)");
System.out.println("(s1.equals(s2) == s2.equals(s1)) = " + (s1.equals(s2) == s2.equals(s1)));

// 3. Transitive: if x.equals(y) == true and y.equals(z) == true then x.equals(z) == true
System.out.println("3. Transitive: if x.equals(y) == true and y.equals(z) == true then x.equals(z) == true");
boolean transitive = (s1.equals(s2) == s2.equals(s3)) ? s1.equals(s3) : false;
System.out.println("transitive = " + transitive);

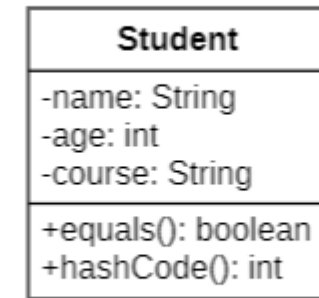
// 4. Consistent: x.equals(y) == true; x.equals(y) == true; x.equals(y) == true; ...
System.out.println("4. Consistent: x.equals(y) == true; x.equals(y) == true; x.equals(y) == true; ...");
boolean consistent = (s1.equals(s2) && s1.equals(s2) && s1.equals(s2) && s1.equals(s2));
System.out.println("consistent = " + consistent);

// 5. if 2 objects equals then hashCode too
System.out.println("5. if 2 objects equals then hashCode too");
System.out.println("(s1.hashCode() == s2.hashCode()) = " + (s1.hashCode() == s2.hashCode()));
```

Equals y hashCode – Caso de Uso

Se requiere llevar el control de los talleres que llevan los estudiantes de la UTP y para motivarlos se les pide que registren su canción favorita para que puedan entrar a un sorteo y ganar una cena con el artista

- La clase solo debe guardar datos del nombre, la edad y el curso en el que el estudiante está inscrito
- Si el nombre y la edad de 2 objetos son iguales, se considerarán como objetos iguales, al margen del curso que llevan
- La relación entre el estudiante y su canción deberán registrarse en un HashMap



Equals y hashCode – Caso de Uso

Common Methods

```
32      @Override
33      public boolean equals(Object o) {
34          if (this == o) return true;
35          if (!(o instanceof Student student)) return false;
36          return getAge() == student.getAge()
37              && Objects.equals(getName(),
38                  student.getName());
39      }
40
41      @Override
42      public int hashCode() {
43          return Objects.hash(getName(), getAge());
44      }
```

Student
-name: String
-age: int
-course: String
+equals(): boolean
+hashCode(): int

```
Student juan = Student.valueOf( name: "Juan", age: 20, course: "Java");
Student carla = Student.valueOf( name: "Carla", age: 20, course: "Java");
Student rosa = Student.valueOf( name: "Rosa", age: 20, course: "Java");
Student random = Student.valueOf( name: "Juan", age: 20, course: "Python");
```

```
// Testing equals
// ¿los objetos juan y carla son iguales?
```

```
String ruta = "src/main/resources/songs.csv";
List<Song> lista = Song.from(ruta);
```

```
HashMap<Student, Song> studentSong = new HashMap<>();
studentSong.put(juan, lista.get(0));
studentSong.put(carla, lista.get(10));
studentSong.put(rosa, lista.get(20));
```

```
// ¿Qué canción le gusta a Carla?
// ¿Qué pasa si elimino los métodos equals y hashCode
// y pregunto por el estudiante random?
```

```
Song songForStudent = studentSong.get(random);
System.out.println("songForStudent = " + songForStudent);
```

Maximiza la inmutabilidad

- Una clase inmutable es una clase cuyas instancias no pueden ser modificadas (Bloch, 2018)
- Muchas librerías en Java contienen clases inmutables, una de ellas es la clase String
- Las clases inmutables más son sencillas de diseñar, implementar y usar que las clases mutables



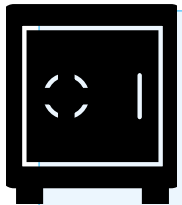
Maximiza la inmutabilidad



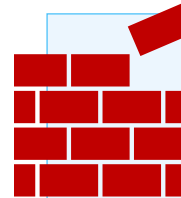
No brindes métodos que modifiquen el estado de un objeto (mutators)



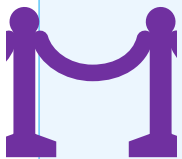
Asegúrate que la clase no puede ser extendida (class final)



Todos los atributos deben ser final



Todos los atributos deben ser privados



Asegura el acceso exclusivo a componentes mutables (p.e. arrays)



Utiliza copias defensivas en constructores y getters

Maximiza la inmutabilidad

```
11 // 2. Class can't be extended
12 public final class Project {
13     // 1. Make all fields final and private
14     private final String name;
15     private final String manager;
16     private final LocalDate init_date;
17     private final List<String> members;
18
19     // 3. Use defensive copies
20     @ public Project(String name, String manager,
21                     LocalDate init_date,
22                     List<String> members) {
23         this.name = new String(name);
24         this.manager = new String(manager);
25         this.init_date = LocalDate.of(init_date.getYear(),
26                                     init_date.getMonth(),
27                                     init_date.getDayOfMonth());
28         this.members = members.stream()
29                             // From Apache Commons Lang
30                             .map(SerializationUtils::clone)
31                             .collect(Collectors.toList());
32     }
```

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
  <version>3.14.0</version>
</dependency>
```



Inmutabilidad – Caso de Uso

Se requiere poder generar listas de canciones, que tengan las siguientes características

- Los objetos creados no pueden ser modificados mediante Setters
- La lista de canciones no puede contener elementos repetidos
- La clase solo permite la creación de objetos mediante el método from
- La clase implementa copias defensivas
- La clase no puede ser extendida por otras subclases



Inmutabilidad – Caso de Uso

```
10 public final class Playlist {
11     private final String name;
12     private final Set<Song> songSet;
13
14     @ private Playlist(String name, Set<Song> songSet) {
15         this.name = new String(name);
16         this.songSet = songSet.stream()
17             // Song Class must implement Serializable
18             .map(SerializationUtils::clone)
19             .collect(Collectors.toSet());
20     }
21
22     @ public static Playlist from(String name,
23                                   List<Song> songList){
24         Set<Song> collect = songList.stream()
25             .collect(Collectors.toSet());
26         return new Playlist(name, collect);
27     }
28
29     @ public String getName() {
30         return new String( name );
31     }
32
33     public Set<Song> getSongSet() {
34         return songSet.stream()
35             .map(SerializationUtils::clone)
36             .collect(Collectors.toSet());
37     }
```

```
String ruta = "src/main/resources/songs.csv";
List<Song> lista = Song.from(ruta);
List<Song> miSeleccion = lista.stream()
    .filter(s -> s.getArtist().contains("Metallica"))
    .collect(Collectors.toUnmodifiableList());

Playlist mySpotify = Playlist.from( name: "Mi Super " +
    "Lista de Canciones",
    miSeleccion);

Set<Song> mis_canciones = mySpotify.getSongSet();

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

Inmutabilidad – Caso de Uso

Classes and Interfaces

Se puede apreciar que las direcciones de memoria asignadas a los objetos de tipo Song se encuentran en ubicaciones diferentes

Esto garantiza que si el programador utiliza el getter, o el método from se crearán copias defensivas para garantizar la inmutabilidad

```
✓  miSeleccion = {ImmutableCollections$List12@1201} size = 2
  >  0 = {Song@1345} "Song{artist='Metallica', title='No Leaf Clover (Live)'}"
  >  1 = {Song@1346} "Song{artist='Metallica', title='I Disappear'}"
✓  mySpotify = {PlayList@1202} "PlayList{name='Mi Super Lista de Canciones',
  >  name = "Mi Super Lista de Canciones"
✓  songSet = {HashSet@1350} size = 2
  >  0 = {Song@1352} "Song{artist='Metallica', title='I Disappear'}"
  >  1 = {Song@1353} "Song{artist='Metallica', title='No Leaf Clover (Live)'}"
✓  mis_canciones = {HashSet@1203} size = 2
  >  0 = {Song@1357} "Song{artist='Metallica', title='I Disappear'}"
  >  1 = {Song@1358} "Song{artist='Metallica', title='No Leaf Clover (Live)'}"
```

Usa Enums

- Un Enum es un tipo enumerado cuyos valores consisten de un conjunto fijo de constantes (Bloch, 2018)
- En Java, los enums son clases que exportan una instancia por cada enumeración
- Permiten un mejor control cuando se requiere utilizar un conjunto finito de valores comunes a varios objetos

Enums

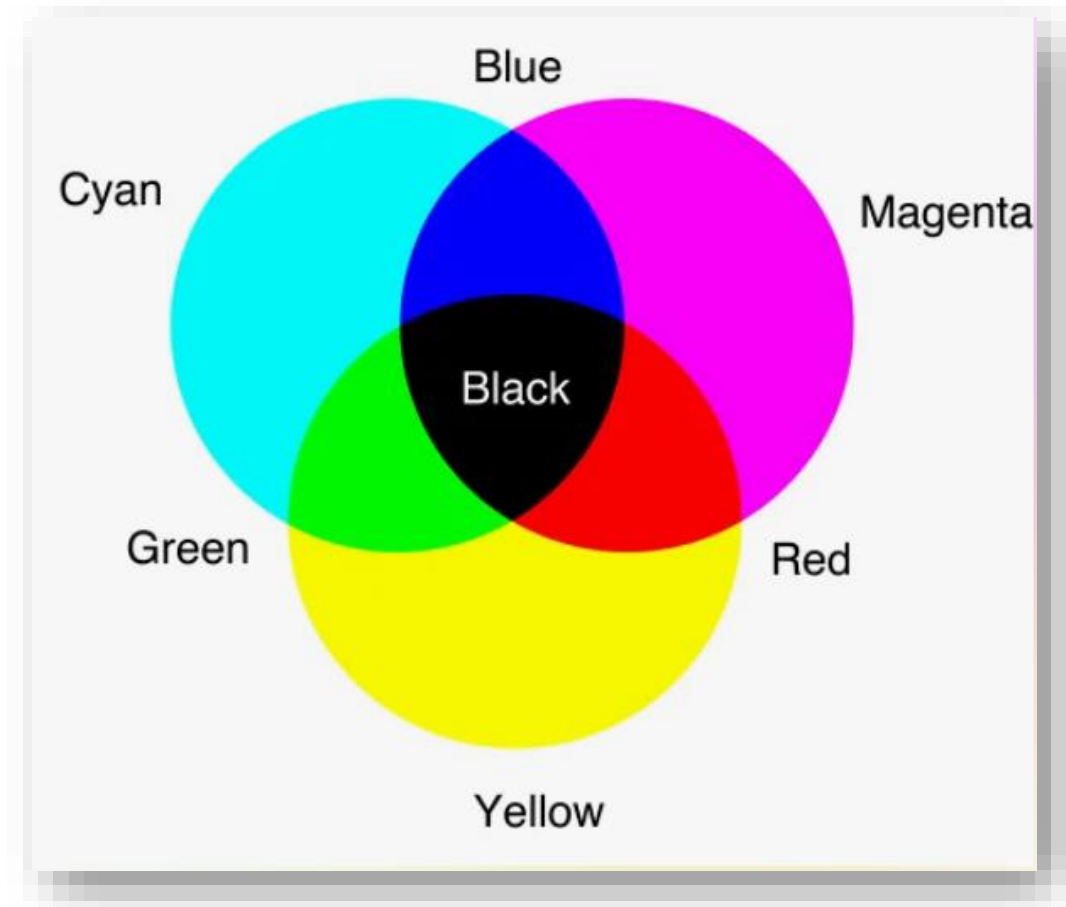


Imagen extraída de:

<https://i.pinimg.com/originals/57/7e/fb/577efbfb7c7b7078ebf4709e6c518727.png>

Usa Enums

Enums

```
3 public enum Priority {  
4     LOW,  
5     MEDIUM,  
6     HIGH  
7 }
```

Se recomienda utilizar Enums en lugar de cadenas de texto

```
Priority pri = Priority.LOW;  
  
String val = switch (pri){  
    case LOW -> "Low Priority";  
    case MEDIUM -> "Medium Priority";  
    case HIGH -> "High Priority";  
    default -> "Unkown";  
};
```

Usa Method References

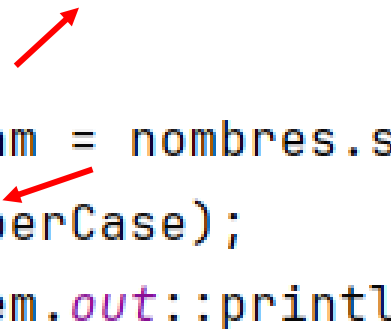
- Aunque las lambdas brindan una gran flexibilidad, Java provee una manera más sencilla a través de los Method References (Bloch, 2018)
- Permiten escribir código más legible y sencillo
- Su uso sin embargo no es conveniente cuando el nombre de las clases es muy extenso

Lambdas and Streams



Use Method References

```
List<String> nombres = new ArrayList<>(Arrays  
    .asList("Juan", "Ana", "Rosa"));  
nombres.forEach(System.out::println);  
  
Stream<String> stringStream = nombres.stream()  
    .map(String::toUpperCase);  
stringStream.forEach(System.out::println);
```



Usa Functional Interfaces

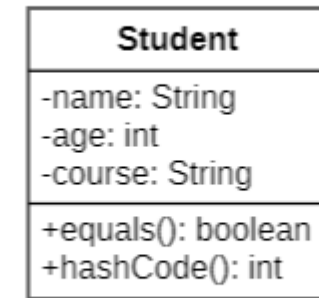
Functional Interface	Arguments	Return	Signature	Example
UnaryOperator<T>	T t1	T	T apply(T t1)	String::toLowerCase
BinaryOperator<T>	T t1, T t2	T	T apply(T t1, T t2)	BigInteger::add
Predicate<T>	T t1	boolean	boolean test(T t1)	Collection::isEmpty
Function<T,R>	T t1	R	R apply(T t1)	Arrays::asList
Supplier<T>		T	T get()	Instant::now
Consumer<T>	T t1	void	void accept(T t1)	System.out::println

En Java, hay muchas Functional Interfaces, sin embargo, se pueden resumir en estas 6

Functional Interfaces & Method References – Caso de Uso

Se requiere poder analizar los datos de los estudiantes mediante una clase utilitaria de tal forma que se permita conocer la siguiente información

- Lista de estudiantes que llevan el curso de Java
- Lista de estudiantes que llevan el curso de Python
- Lista de estudiantes que cubren una lista de requerimientos personalizada mediante el uso de streams



Functional Interfaces & Method References – Caso de Uso

```
9 public class StudentUtils {
10
11 @   public static boolean isTakingJava(Student student){
12     return student.getCourse().equalsIgnoreCase( anotherString: "java");
13 }
14
15 @   public static boolean isTakingPython(Student student){
16     return student.getCourse().equalsIgnoreCase( anotherString: "python");
17 }
18
19 @   public static boolean testRequirements(List<Predicate<Student>>
20                                           requirements,
21                                           Student student){
22     for (Predicate<Student> requirement : requirements) {
23         if ( requirement.test(student) == false ) return false;
24     }
25     return true;
26 }
27
28 }
```

Las Functional Interfaces permiten flexibilizar las posibilidades de las aplicaciones mediante el paso de funcionalidad mediante parámetros

Functional Interfaces & Method References – Caso de Uso

```
14 Student juan = Student.valueOf( name: "Juan", age: 20, course: "Java");
15 Student carla = Student.valueOf( name: "Carla", age: 17, course: "Java");
16 Student rosa = Student.valueOf( name: "Rosa", age: 20, course: "Python");
17
18 List<Student> lista = List.of(juan, carla, rosa);
19
20 // Listar estudiantes que solo llevan Java (usando Method References)
21 List<Student> onlyJava = lista.stream().filter(StudentUtils::isTakingJava)
22     .collect(Collectors.toUnmodifiableList());
23 System.out.println("onlyJava.size() = " + onlyJava.size());
24
25 // Diseñar reglas especiales para filtrado de datos
26 Predicate<Student> rule1 = StudentUtils::isTakingJava;
27 Predicate<Student> rule2 = (s -> s.getAge() >= 15);
28
29 List<Predicate<Student>> req_list = List.of(rule1, rule2);
30 // ¿Juan cumple los requerimientos?
31 boolean juan_req_ok = StudentUtils.testRequirements(req_list, juan);
32 System.out.println("juan_req_ok = " + juan_req_ok);
33
34 // ¿Que estudiantes cubren los requerimientos?
35 System.out.println("Cubren requerimientos");
36 for (Student student : lista) {
37     boolean cover = StudentUtils.testRequirements(req_list, student);
38     if (cover) System.out.println(student);
39 }
```

Las Functional Interfaces permiten flexibilizar las posibilidades de las aplicaciones mediante el paso de funcionalidad mediante parámetros

Ejemplo de mapeo de funciones

```
// Standard Functional Interfaces
Function<String, String> function1 = String::toUpperCase;
Function<String, String> function2 = String::toLowerCase;
Function<String, String> function3 = String::trim;
Function<String, String> function4 = FreeBSD::install;
List<String> software = new ArrayList<>(Arrays.asList("vim", "apache24", "wget"));

HashMap<String, Function<String, String>> functionList = new HashMap<>();
functionList.put("upper", function1);
functionList.put("lower", function2);
functionList.put("trim", function3);
functionList.put("install", function4);

String userCmd = "install"; // User election

// Get the function reference (method reference) and exec for every software
Function<String, String> stringFunction = functionList.get(userCmd);
Stream<String> stringStreamAdv = software.stream().map(stringFunction);
stringStreamAdv.forEach(System.out::println);
```

Las Functional Interfaces permiten pasar funciones a otros métodos de manera sencilla

Siempre revisa los parámetros

- Por lo general, los métodos y constructores tienen restricciones sobre los valores que pueden venir en sus parámetros (Bloch, 2018)
- Hay un principio general de programación que consiste en intentar detectar errores antes de que ocurran
- Si un parámetro tiene datos inválidos, debemos generar una excepción



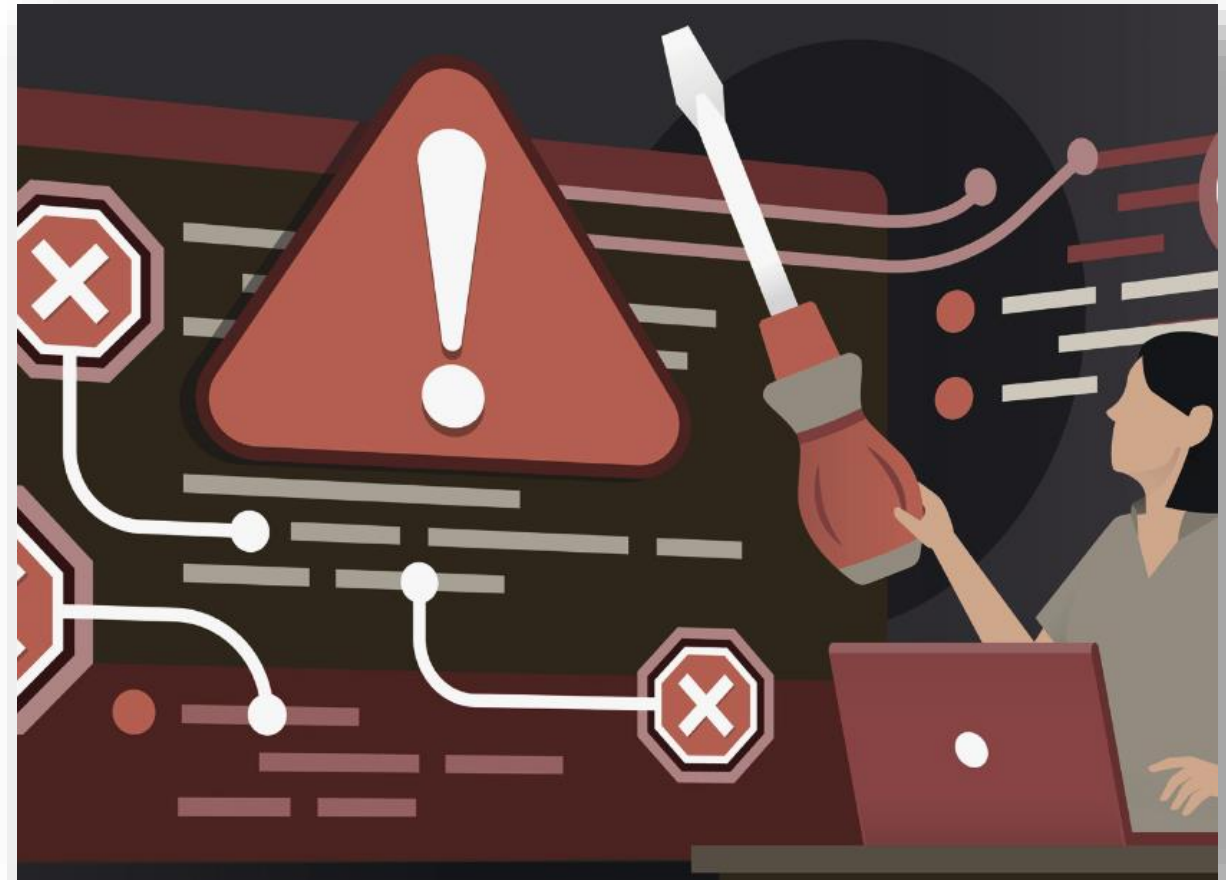
Siempre revisa los parámetros

```
7 public class Course {
8     private final String name;
9     private final String faculty;
10    private final int semester;
11
12    private Course(String name, String faculty, int semester) {
13        this.name = name;
14        this.faculty = faculty;
15        this.semester = semester;
16    }
17
18    @ public static Course of(String name, String faculty, int semester) {
19        // Checking arguments
20        checkArgument(expression: semester > 0 && semester <= 10,
21            String.format("invalid semester: %d (must be 1-10)", semester));
22        checkArgument(expression: name != null, errorMessage: "name cannot be null");
23        checkArgument(expression: faculty != null, errorMessage: "faculty cannot be null");
24        return new Course(name, faculty, semester);
25    }
```

```
<dependency>
  <groupId>com.google.guava</groupId>
  <artifactId>guava</artifactId>
  <version>32.1.3-jre</version>
</dependency>
```

Recomendaciones sobre excepciones

- Evita abusar de las excepciones
- Usa en gran medida las excepciones estándares de Java
- Incluye información detallada cuando ocurre una excepción
- Registra las excepciones en un log
- **Nunca ignores las excepciones**



Recomendaciones sobre excepciones

```
9  ▶ public static void main(String[] args) {  
10  
11      Path web = null;  
12      String ruta = "https://www.miweb.com";  
13      //String ruta = "C:/mi_ruta";  
14      try {  
15          web = Path.of(ruta); // wrong ruta  
16      } catch (InvalidPathException e) {  
17          // Ignore exception <----- DON'T DO THIS !!!  
18      }  
19  
20      System.out.println("web = " + web);  
21      System.out.println(web.equals("www.miweb.com")); // BOOM !  
22  
23  }
```

Siempre se debe evitar anular o ignorar las excepciones, porque permiten que un sistema entre en un estado inconsistente y un eventual colapso



Effective Java



GRACIAS

fidiaz@utp.edu.pe

Bibliografía



- Joshua Bloch (2018). Effective Java, Third Edition. Addison-Wesley.