Técnicas de Programación Avanzada (Java)

Curso 2021-2021

S_5 (Programación Interfaces) {

5.1 Concurrencia.

5.2 Inversión de Control. Definición y ejemplos.
Inyección de dependencias.

5.3 Expresiones avanzadas del lenguaje.

}

Dra. Nieves Cubo Mateo

29 Nov y 13 Dic 2021

Tema 1: Objetos y memoria.

1.1 Características básicas del lenguaje. Primer programa. Compilación y Ejecución. IDE.

1.2 Sentencias de control. Secuencia, selección e iteración.

1.3 Abstracción. Clases, objetos, métodos y atributos.

1.4 Sobrecarga de métodos y encapsulamiento.

Tema 2. Otros conceptos fundamentales de la Programación Orientada a Objetos

2.1 Herencia. Interfaces y clases abstractas. Agregación.

2.2 Polimorfismo.

2.3 Gestión de Excepciones.

2.4 Genericidad y plantillas.

2.5 Utilidades. Entrada y Salida.

2.6 Anotaciones.

Tema 3. Patrones de Diseño.

3.1 Concepto de Patrones de Diseño.

3.2 Patrones de creación.

3.3 Patrones estructurales.

3.4 Patrones de comportamiento.

Tema 4. Programación de Interfaces.

4.1 Interfaces Gráficas de Usuario.

4.2 Gestión de eventos.

Tema 5. Temas Avanzados.

5.1 Concurrencia.

5.2 Inversión de Control. Definición y ejemplos. Inyección de dependencias.

5.3 Expresiones avanzadas del lenguaje.

5.1 Concurrencia

Intro

https://docs.oracle.com/javase/tutorial/essential/concurrency/

Ejecuciones en paralelo (multithreading) para ahorrar recursos y memoria, aislar actividades en diferentes hilos de ejecución, etc. En java, siempre hay al menos 1 Thread (main)

Implementación similar a C++: Creando una clase que extienda a 'Thread' y sobrescribiendo el método 'run()'. El contenido de ese método será la tarea que se realice cuando una instancia de esa clase ejecute el método 'start()'.

Clase Thread – Métodos:

- run() actividad del thread (active desde start)
- start() activa run() y vuelve al llamante
- join() espera por la terminación (timeout opcional)
- interrupt() sale de un wait, sleep o join
- isInterrupted()
- yield()
- Deprecated: stop(), suspend(), resume()

Métodos estáticos

- sleep(milisegundos)
- currentTread()

Métodos de la clase Object que controlan la suspension de threads:

- wait(), wait(milisegundos)
- notify(), notifyAll()

5.1 Threads (Hilos)

Threads (hilos) – unidades básicas de código que pueden ser ejecutadas al mismo tiempo

Cómo implementar un hilo:

1. Extendiendo la clase *Thread* → https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html
2. Implementando la interfaz *Runnable* → https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html

En ambos casos, tendremos que instanciar después un Thread e invocar al método “start”.

```
Public class myThread extends Thread {

    @Override
    public void run() {
        //whatever
    }
}

myThread t = new myThread(143);
t.start();
```

```
public class A implements Runnable{

   @Override
   public void run(){
       //whatever
    }
}

Thread t= new Thread(new A());
t.start();
```

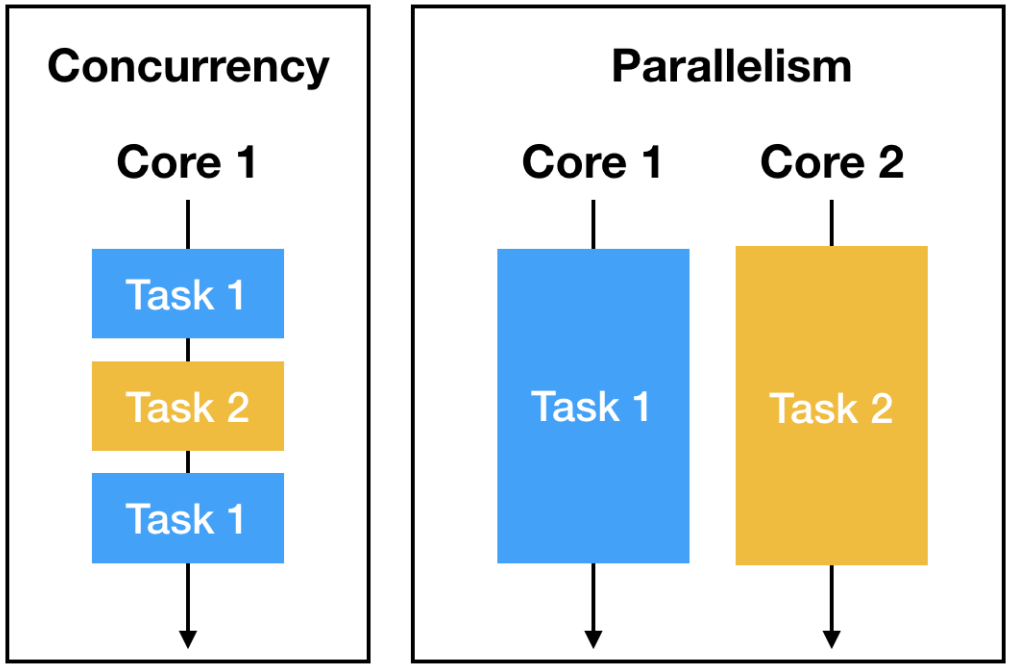
5.1 Concurrency vs Parallelism

- Más de un procesador corriendo simultáneamente y accediendo a los mismos recursos (“un ratito tu, un ratito yo,…”) Context switch

- Puede haber problemas si se quiere modificar la misma información a la vez

- Para evitar conflictos, deben coordinarse / sincronizarse las tareas (synchronized) → solo un hilo tiene acceso en cada momento, el resto está “bloqueado”

- A veces cuando queremos usar paralelismo, hacemos uso de concurrencia (falso paralelismo) – 1 CPU

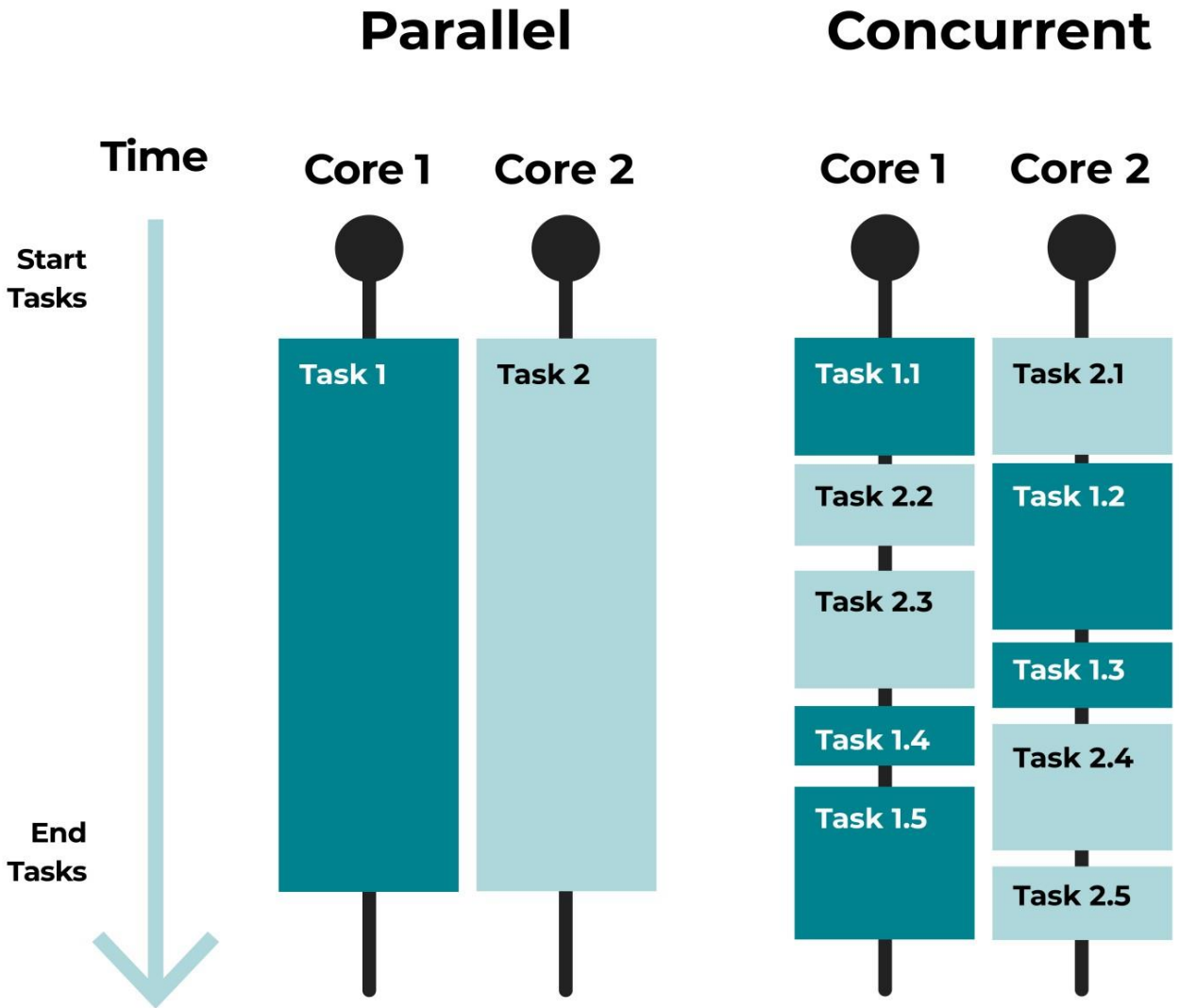


- Se ejecutan realmente diferentes tareas a la vez

- Requiere que el equipo disponga de diferentes CPUs (unidades de procesamiento)

5.1 Multithread en nuestros ordenadores

Cómo funciona con varios procesadores



- Tener multiples procesadores no implica que la ejecución de nuestro Código vaya a ser en paralelo

5.1 Concurrencia

Ejemplo de implementación

```
import java.lang.Math ;

class EjemploThread extends Thread {
   int numero;
   EjemploThread (int n) { numero = n; }

   @Override
   public void run() {
      try {
         while (true) {
               System.out.println (numero);
               sleep((long)(1000*Math.random()));
         }
      } catch (InterruptedException e) { return; } // acaba este thread
   }

   public static void main (String args[]) {
      for (int i=0; i<10; i++)
      new EjemploThread(i).start();
   }
}
```

Importante: se lanza con START para crear un nuevo hilo.

5.1 Thread Safety

Cómo acceder a la información de forma segura

- Si varios hilos tienen acceso a la misma variable o sección de código, deben establecerse métodos de acceso sincronizados

- Si dos o más hilos compiten por la misma información, y el orden puede afectar al resultado, se genera una condición de carrera (*race condition*). Las partes del Código que generan estas situaciones se denominan secciones críticas (*critical section*)

- Synchronized: puede usarse para métodos, bloques de Código (estáticos o no)

```
synchronized(objectidentifier) {
   // Access shared variables and other shared resources
}
```

Ejemplo de sincronización

```
public class TestThread {

public static void main(String args[]) {
  PrintDemo PD = new PrintDemo();

  ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD);
  ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD);

  T1.start();
  T2.start();

// wait for threads to end
  try {
     T1.join();
     T2.join();
  } catch ( Exception e) {
     System.out.println("Interrupted");
  }
 }
}
```

https://www.tutorialspoint.com/java/java_thread_synchronization.htm

```
class ThreadDemo extends Thread {
 private Thread t;
 private String threadName;
 PrintDemo  PD;

 ThreadDemo( String name,  PrintDemo pd) {
  threadName = name;
  PD = pd;
}

 public void run() {
  synchronized(PD) { //Si elimino esto..
     PD.printCount();
  }
  System.out.println("Thread " +  threadName + " exiting.");
}

 public void start () {
   System.out.println("Starting " +  threadName );
   if (t == null) {
      t = new Thread (this, threadName);
      t.start ();
   }
 }
}
```

```
class PrintDemo {
 public void printCount() {
   try {
      for(int i = 5; i > 0; i--) {
         System.out.println("Counter   ---   "  + i );
      }
   } catch (Exception e) {
      System.out.println("Thread  interrupted.");
   }
  }
}
```

Sin sincronizar vs sincronizado

```
//synchronized(PD) {
```

```
Starting Thread - 1
Starting Thread - 2
Counter   ---   5
Counter   ---   4
Counter   ---   3
Counter   ---   5
Counter   ---   2
Counter   ---   1
Counter   ---   4
Thread Thread - 1  exiting.
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 2  exiting.
```

```
synchronized(PD) {
```

```
Starting Thread - 1
Starting Thread - 2
Counter   ---   5
Counter   ---   4
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 1  exiting.
Counter   ---   5
Counter   ---   4
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 2  exiting.
```

Otras formas de trabajar con multihilos (Java)

- Framework *executor*
- *Phaser class*
- *Fork & Join*
- *Stream frameworks*

https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

https://github.com/shubhamvrkr/java-concurrency-examples

5.2 Inversión de Control. Definición y ejemplos. Inyección de dependencias.

Inversión del control (IoC)

Recordad: *Estructuras de control (Tema 1)*

- Estilo de programación en el cual un framework o librería controla el flujo de un programa.

- Muy útil cuando se usan frameworks de desarrollo. Es el framework el que toma el control, el que define el flujo de actuación o el ciclo de vida de una petición. Es decir, es el framework quien ejecuta el código de usuario.

- La **Inversión de control** puede implementarse mediante:
  - *Inyección de dependencias* (se puede crear un contenedor que gestione las instancias u objetos)
  - *Eventos* (como en las aplicaciones con GUI)

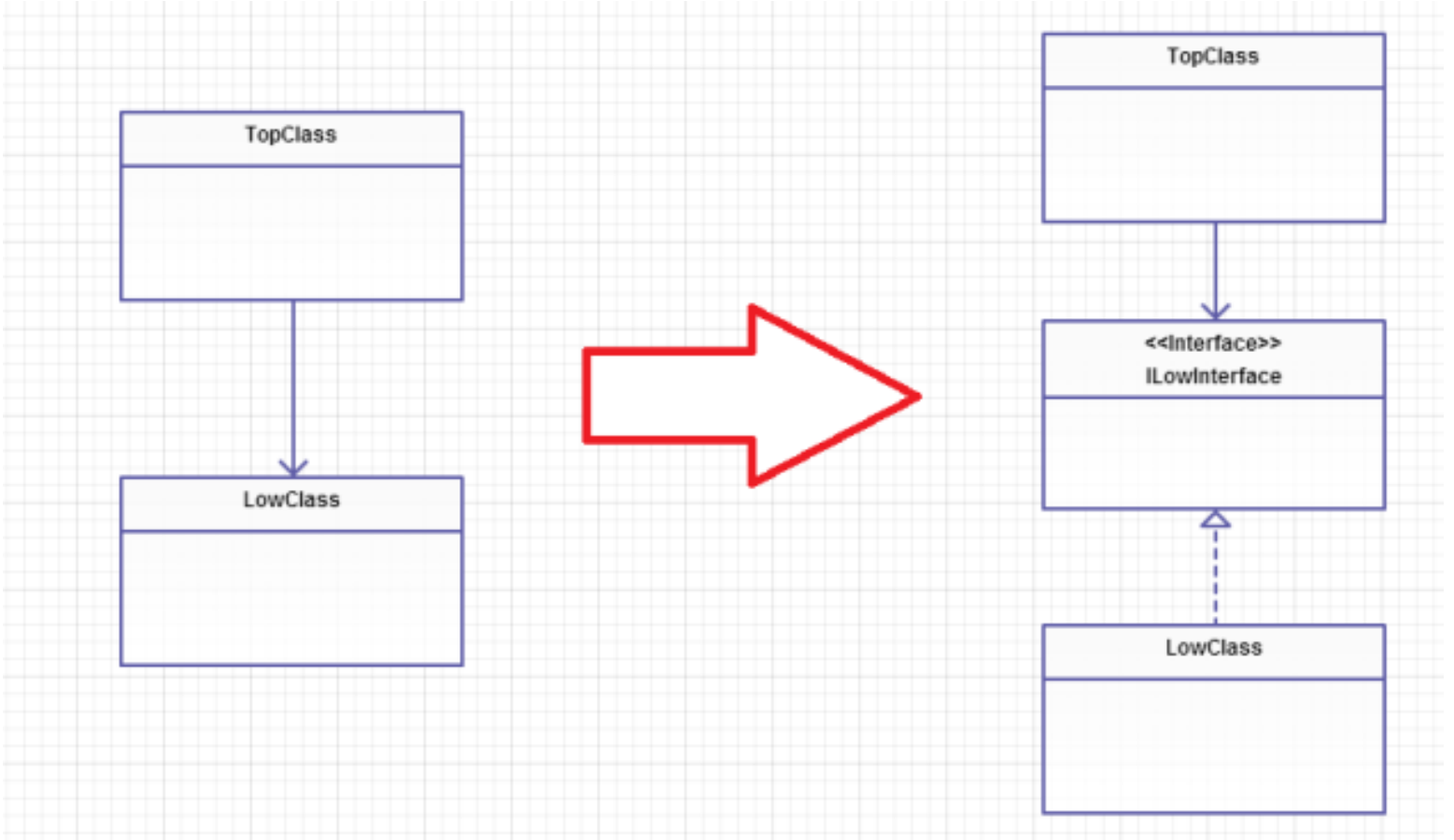
5.2 Inversión de Control. Definición y ejemplos. Inyección de dependencias.

Inyección de dependencias (DI)

La inyección de dependencias **es un patrón de diseño** que permite construir software con poco acoplamiento.

El patrón funciona con un objeto que se encarga de construir las dependencias que una clase necesita y se las suministra (“inyecta”).

La clase no crea directamente los objetos que necesita, sino que los recibe de otra clase.



Inyección de dependencias

Es un patrón de diseño orientado a objetos, en el que se suministran objetos a una clase en lugar de ser la propia clase la que cree dichos objetos.

```
public class Vehiculo {

   private Motor motor = new Motor();

public Double enAceleracionDePedal(int presionDePedal) {
      motor.setPresionDePedal(presionDePedal);
      int torque = motor.getTorque();
      Double velocidad = ...
return velocidad;
   }

}
```

```
public class Vehiculo {

   private Motor motor = null;

   public void setMotor(Motor motor){
            this.motor = motor;
   }

public Double enAceleracionDePedal(int presionDePedal) {
      Double velocidad = null;
      if (null != motor){
         motor.setPresionDePedal(presionDePedal);
         int torque = motor.getTorque();
         velocidad = ... //realiza el cálculo
      }
      return velocidad;
   }
}
```

```
public class VehiculoFactory {

   public Vehiculo construyeVehiculo() {
      Vehiculo vehiculo = new Vehiculo();
      Motor motor = new Motor();
      vehiculo.setMotor(motor);
      return vehiculo;
   }

}
```

Estructuras de datos

- **Array (built-in array):** tamano no se puede modificar. Hace falta crear otro array.
  - String [] cats = new String(5);
    cats[0]="Kitty";

- **ArrayList:** Se pueden anadir y eliminar elementos (set, get, remove, size, sort,)
  - import java.util.ArrayList;
    import java.util.Collections; //Para usar sort
    ArrayList<String> cats = new ArrayList<String>();
    cats.add("Kitty");

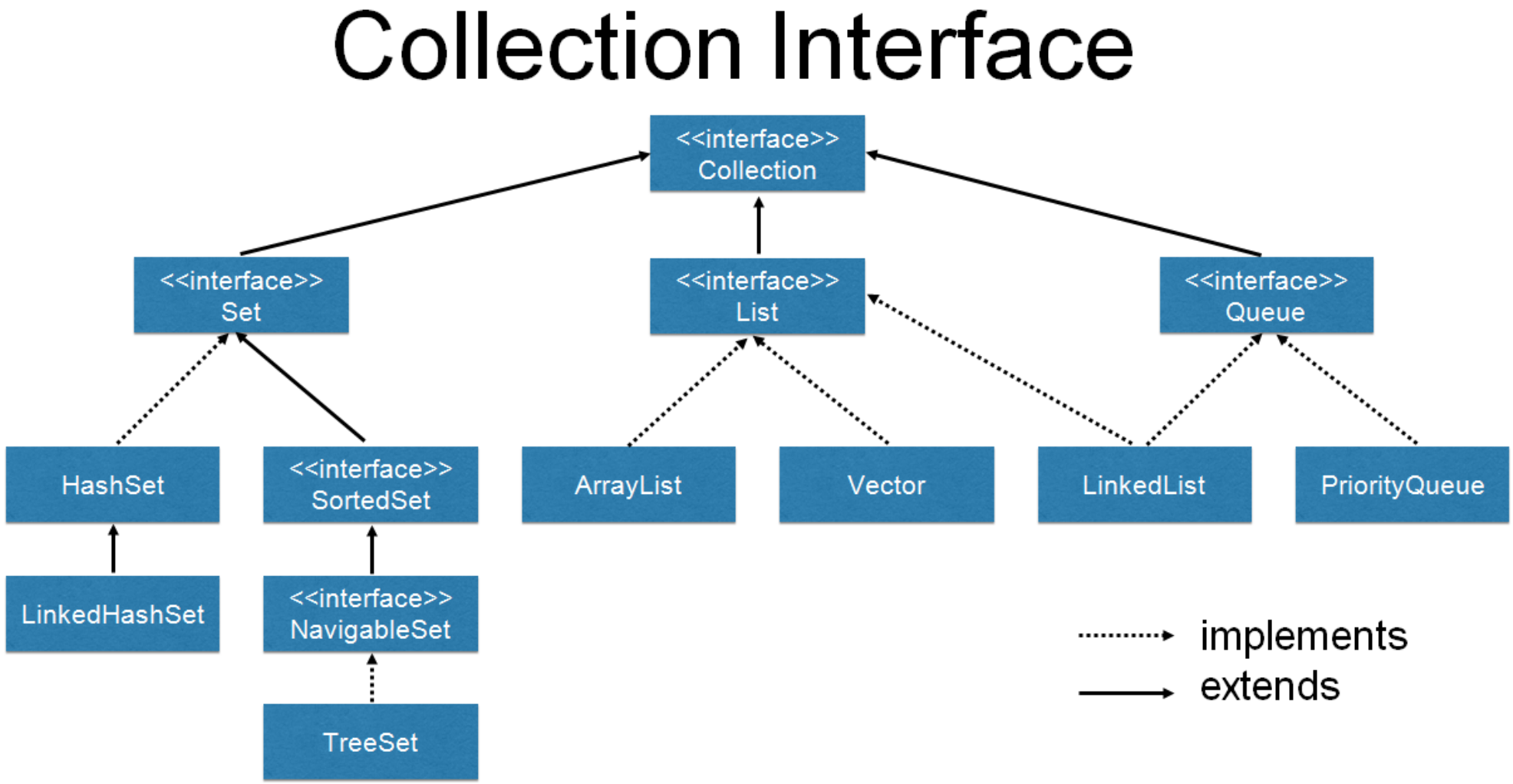
Objects, Wrappers

- **LinkedList**: similar a ArrayList (ambos implementan la interfaz List), pero son elementos enlazados (nodos) en orden (addFirst, addLast, removeFirst, removeLast, getFirst, getLast, pop, peek, ..), **no se puede acceder de forma random**.
  - import java.util.LinkedList;
    LinkedList<String> cats = new LinkedList<String>();
    cats.add("Kitty");

- **HashMap:** como los diccionarios <key/value>
- **HashSet:** como los pools, saco con items únicos

5.3 Expresiones avanzadas del lenguaje.

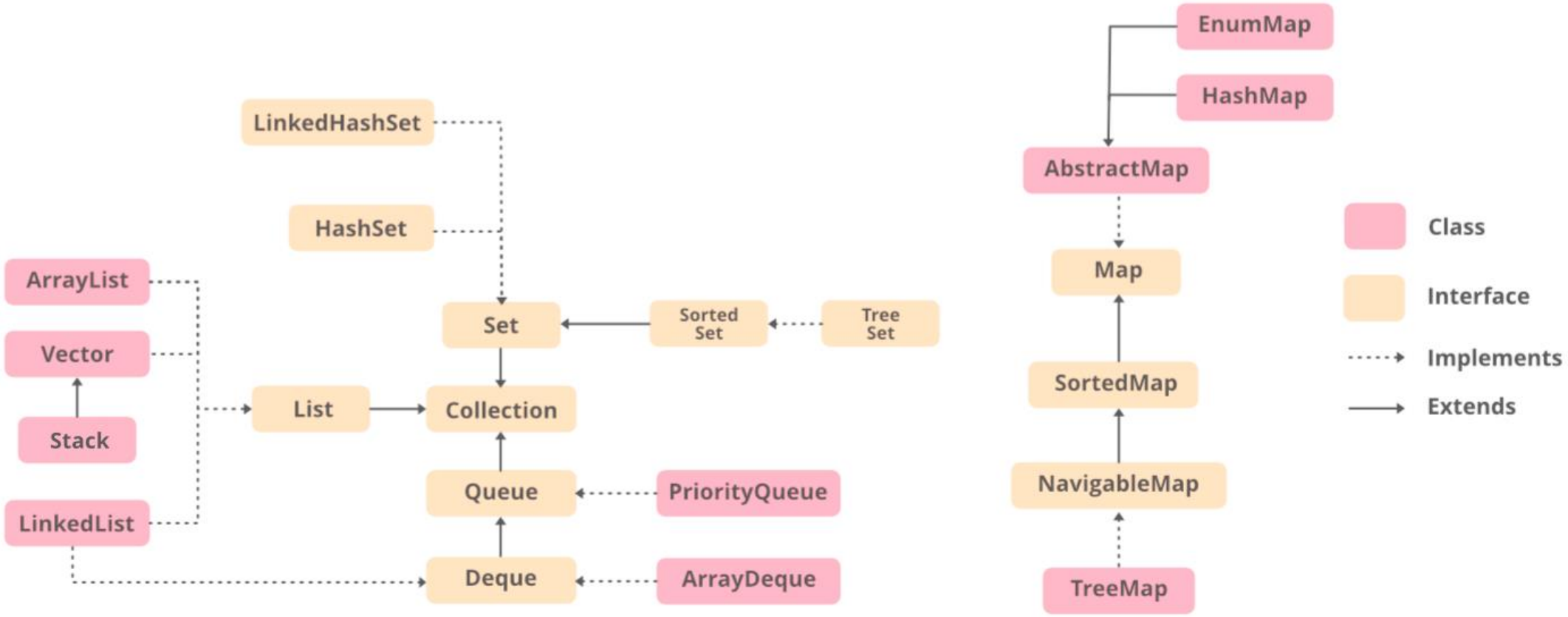
Collections (java.util.Collection)



https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html

5.3 Expresiones avanzadas del lenguaje.

Collections (java.util.Collection)



https://www.geeksforgeeks.org/collections-in-java-2/

5.3 Expresiones avanzadas del lenguaje.

Clases anónimas


```
new Book("Design Patterns") {
    @Override
    public String description() {
       return "Famous GoF book.";
    }
}
```

Las clases anónimas son expresiones, por lo que deben ser parte de una declaración.

Se emplean como solución rápida para implementar una clase que **se va utilizar una vez** y de forma inmediata.

Se emplean mucho en listeners, eventos…

Implementación de interfaces (sin parámetros) – definición de métodos:

```
new Runnable() {
    @Override
    public void run() {
        ...
    }
}
```

https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html

5.3 Expresiones avanzadas del lenguaje.

Clases anónimas – Ejemplo Listeners/Eventos

```
public class Button1ActionListener implements ActionListener {
    private JTextField jText;

    public Button1ActionListener(JTextField jText) {
        this.jText = jText;
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        jText.setText("button 1 clicked");
    }
}
```

Con clases anónimas:

```
// Previously defined: jButton1; jText;

jButton1.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        jText.setText("button 1 clicked");
    }
});
```

Con exp. Lambda:

```
jButton1.addActionListener(e -> jText.setText("button 1 clicked"));
```

5.3 Expresiones avanzadas del lenguaje.

Lambda expressions

Una expresión lambda es una bloque corto de código que toma parámetros y devuelve un valor. Las expresiones lambda son similares a los métodos, pero no necesitan un nombre y se pueden implementar directamente en el cuerpo de un método.

Son interfaces funcionales, solo tienen un método abstracto.

```
((parameter) -> {expression});
```

```
ArrayList<Integer> numbers = new
ArrayList<Integer>();
numbers.add(5);
numbers.add(9);
numbers.add(8);
numbers.add(1);

for (int n:numbers)
System.out.println(n);
```

Sencillo para simplificar bucles con arraylist, usando el método forEach y pasando una función como argumento

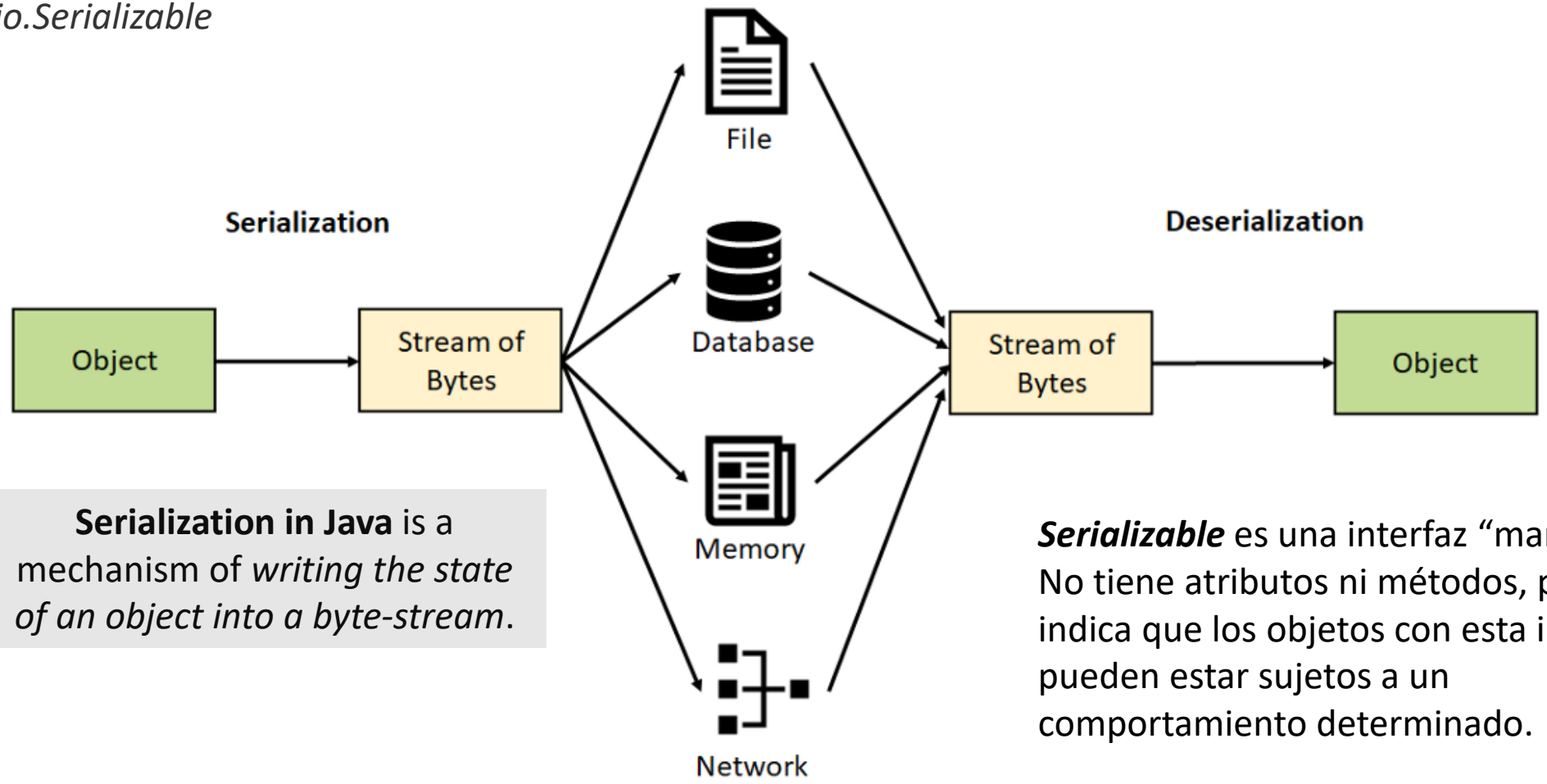
```
numbers.forEach( (n) -> { System.out.println(n); } );
```

https://www.w3schools.com/java/java_lambda.asp

5.3 Expresiones avanzadas del lenguaje.

Clases serializables – guardar y extraer objetos

java.io.Serializable



Serialization in Java is a mechanism of *writing the state of an object into a byte-stream*.

Serializable es una interfaz "marcador". No tiene atributos ni métodos, pero indica que los objetos con esta interfaz pueden estar sujetos a un comportamiento determinado.

https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html

5.3 Expresiones avanzadas del lenguaje.

Clases serializables – guardar y extraer objetos

```
import java.io.Serializable;
public class Student implements Serializable{
 int id;
 String name;
 public Student(int id, String name) {
  this.id = id;
  this.name = name;
 }
}
```

Para serializar un objeto, llamamos al método **writeObject ()** de la clase *ObjectOutputStream*, y para la deserialización llamamos al método **readObject ()** de la clase *ObjectInputStream*.

String y Wrappers: implementan *Serializable*. Los datos primitivos pueden ser escritos directamente por el *ObjectOutputStream.*

Algunas clases como **JPanel** lo implementan también.

https://www.javatpoint.com/serialization-in-java

Si no desea serializar ningún miembro de datos de una clase, puede marcarlo como transitorio (**transient**). Tras serializar y deserializar, se obtendrá un valor por defecto de dicha variable.

Si una clase implementa el marcador, todas las clases hijas lo tendrán también (**herencia**).

Si una clase contiene a otra por **agregación**, está deberá ser también Serializable, o el proceso no podrá llevarse a cabo.

En general, todos los datos de la clase deben ser Serializables para que pueda ejecutarse sin excepciones.

5.3 Expresiones avanzadas del lenguaje.

Serializable - Guardar objetos [ObjectOutputStream.writeObject(Obj)]

```
import java.io.*;
class Persist{
 public static void main(String args[]){
  try{
  //Creating the object
  Student s1 =new Student(211,"ravi");
  //Creating stream and writing the object
  FileOutputStream fout=new FileOutputStream("f.txt");
  ObjectOutputStream out=new ObjectOutputStream(fout);
  out.writeObject(s1);
  out.flush();
  //closing the stream
  out.close();
  System.out.println("success");
  }catch(Exception e){System.out.println(e);}
 }
}
```

5.3 Expresiones avanzadas del lenguaje.

Serializable - Extraer objetos [ObjectInputStream.readObject()]

```
import java.io.*;
class Depersist{
 public static void main(String args[]){
  try{
  //Creating stream to read the object
  ObjectInputStream in=new ObjectInputStream(new FileInputStream("f.txt"));
  Student s=(Student)in.readObject();
  //printing the data of the serialized object
  System.out.println(s.id+" "+s.name);
  //closing the stream
  in.close();
  }catch(Exception e){System.out.println(e);}
 }
}
```

CAST explícito tras es readObject (**recomendado**).

Para serializar un objeto, llamamos al método **writeObject ()** de la clase *ObjectOutputStream*,
y para la deserialización llamamos al método **readObject ()** de la clase *ObjectInputStream*.

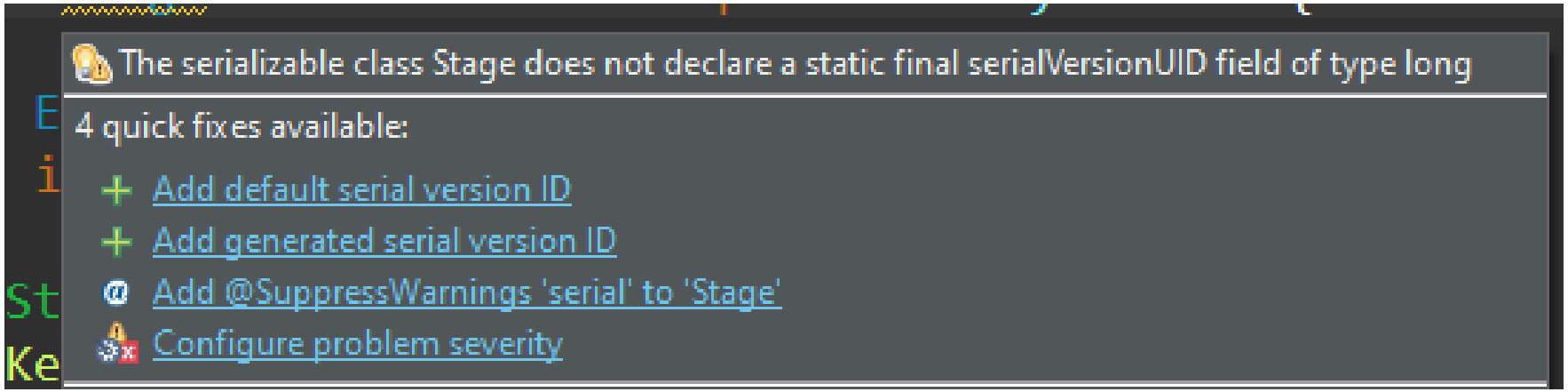
Ambos métodos pueden ser sobreescritos si se desea cambiar el comportamiento de los mismos, indicando cómo deben guardarse los datos de una clase.

Java Custom Serialization

https://howtodoinjava.com/java/serialization/custom-serialization-readobject-writeobject/

5.3 Expresiones avanzadas del lenguaje.

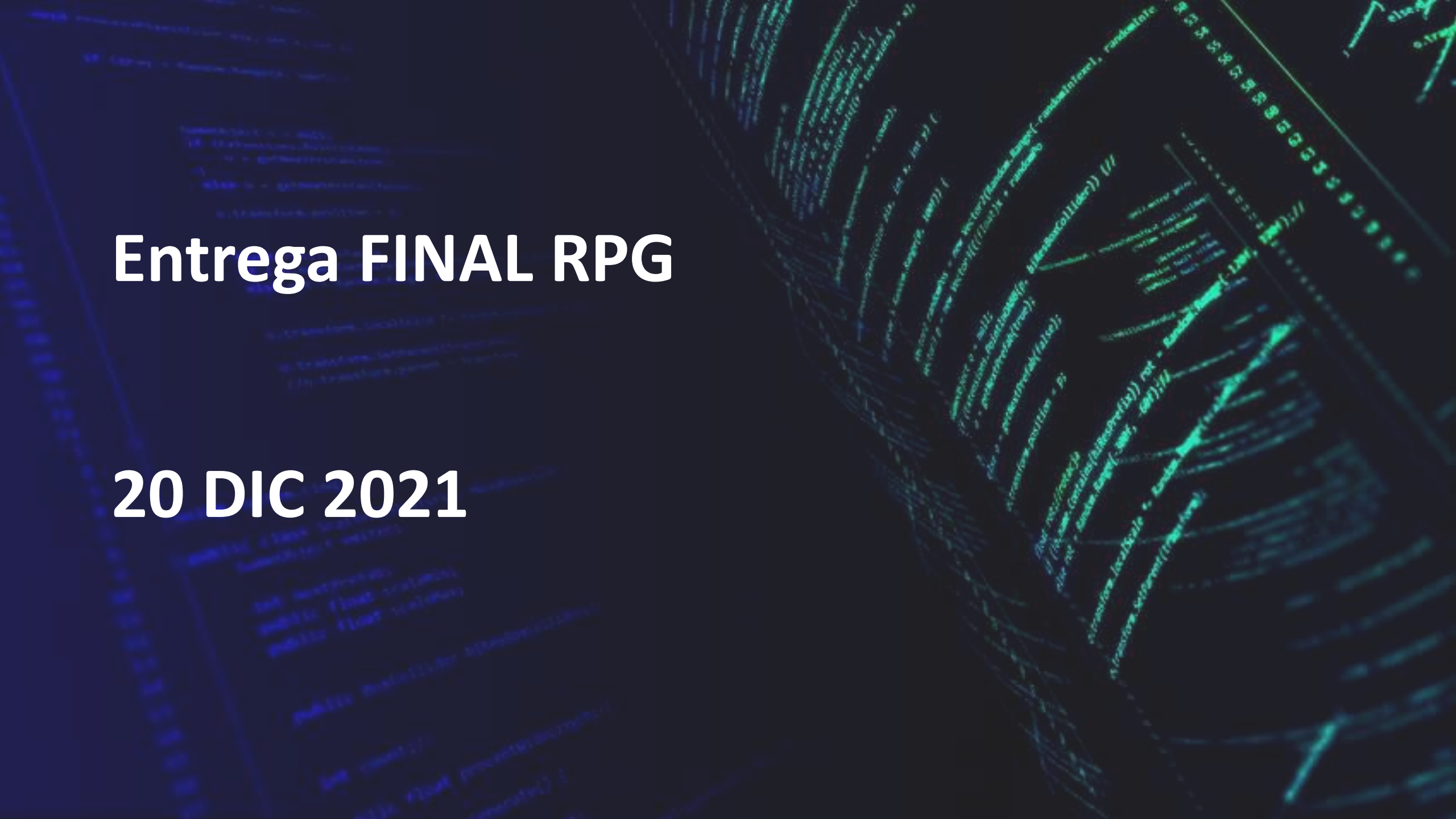
Clases serializables – guardar y extraer objetos



El proceso de serialización en tiempo de ejecución asocia una identificación con cada clase serializable que se conoce como ***SerialVersionUID***. Se utiliza para verificar el remitente y el receptor del objeto serializado, que deben tener el mismo SerialVersionUID; de lo contrario, se lanzará *InvalidClassException* cuando deserialice el objeto.

También podemos declarar nuestro propio SerialVersionUID en la clase Serializable. Para hacerlo, se debe crear un campo SerialVersionUID y asignarle un valor. Debe ser **de tipo largo con estático y final**. Se sugiere declarar explícitamente el campo serialVersionUID en la clase y tenerlo también privado. Por ejemplo:

```
private static final long serialVersionUID=1L;
```



Entrega FINAL RPG

20 DIC 2021

¿Qué debo entregar el día 20/12/21? 1 archivo ZIP con:

- Código Fuente (exportar como siempre, ***File > Export > General > Archive File.***
- Ejecutable (***File> Export> Runnable JAR file***) [*Package required libraries into generated JAR*]
- Documentación JavaDocs [*Create JavaDocs for members with visibility: Private*]
- Diagrama/s UML – Imagina que tuvieras que explicarlo a la clase. Debe ser entendible y sencillo. Se puede incluir además otros diagramas más complejos, relaciones entre paquetes, etc.
- Documento README.txt con los siguientes datos:
  - Autor
  - Fecha
  - Título
  - Abstract(<300 palabras)
  - Fuentes empleadas (si has cogido código de otros lados, citas y links)
  - Otros: comentarios que deba tener en cuenta la persona que ejecute el código, necesidades de librerías, flags de compilación, etc

Si el ‘.jar’ te da problemas…

- *“Al lanzarlo con doble click no se abre… “*
  - Windows: Abrir desde el CMD: java –jar file.jar

- *“No se me cargan las imágenes. “*
  - Revisa que estén en una carpeta tipo *source*
  - Introduce las imágenes de esta manera, como resources de la clase:

```
private BufferedImage image = null;

 public Entity() {
   try {
        image = ImageIO.read(getClass().getResource(dummy) );
   } catch (IOException e) {
 }
```

Bibliografía

- Deitel, H. M. & Deitel, P. J.(2008). *Java: como programar*. Pearson education. Séptima edición.
- Sumérgete en los patrones de diseño. V2021-1.7. Alexander Shvets. https://refactoring.guru/es/design-patterns/book

  Versión online: https://refactoring.guru/es/design-patterns/catalog
- The Java tutorials. https://docs.oracle.com/javase/tutorial/
- Páginas de apoyo para la sintaxis, bibliotecas, etc.:
  - https://www.sololearn.com
  - https://www.w3schools.com/java/default.asp
  - https://docstore.mik.ua/orelly/java-ent/jnut/index.htm

Técnicas de Programación Avanzada (Java)

Curso 2021-2021

```
exit(); //Gracias!
```

Dra. Nieves Cubo Mateo