GENERALITAT VALENCIANA
Conselleria d'Educació, Investigació, Cultura i Esport

IES LA ENCANTÁ

GOBIERNO DE ESPAÑA
MINISTERIO DE EDUCACIÓN, CULTURA Y DEPORTE

Unión Europea
Fondo Social Europeo
El FSE invierte en tu futuro

UNIDAD 5.

POO Avanzada y excepciones.

Índice

1. Ficha unidad didáctica.....1
2. Contenidos.....2
2.1. Excepciones.....2
2.1.1 Concepto de excepción.....2
2.1.2 Excepciones en Java (Jerarquía).....3
2.1.3 Manejo y captura.....7
2.1.4 Excepciones personalizadas.....9
2.1.5 Lanzamiento de excepciones.....10
2.1.6 Aserciones.....11
3. Actividades y ejercicios.....11

1. Ficha unidad didáctica.

OBJETIVOS DIDÁCTICOS	
OD1: Enunciar los tipos de herencia existentes. OD2: Exponer la características de las clases abstractas. OD3: Usar la herencia e interfaces de forma razonada dependiendo de los problemas a resolver. OD4: Elaborar jerarquía de clases que apliquen sobrecarga y sobreescritura métodos. OD5: Modificar y analizar código fuente que utilice herencia y jerarquía de clases. OD6: Reconocer las situaciones donde controlar excepciones. OD7: Elaborar software tolerante a fallos usando excepciones. OD8: Analizar código con uso de excepciones, o añadiendo estas para prever fallos. OD9: Desarrollar nuevas excepciones en función de las necesidades. EV1. Analizar el derecho a propiedad intelectual de programas, código y librerías, no haciendo un uso indebido o ilícito de estos. EV3. Caracterizar la importancia del trabajo en grupo en el ámbito empresarial. EV4. Valorar el trabajo por su calidad, no por la persona que lo realiza. RL2. Evaluar las consecuencias de una vida y trabajo sedentario para la salud. TIC2. Utilizar el uso de Internet como forma de actualización en nuevas tecnologías relacionadas con el ámbito laboral de la informática.	
RESULTADOS DE APRENDIZAJE	RA3, RA7
CONTENIDOS	
Relaciones entre clases. Composición. Herencia. Poliformismo. Herencia en otros lenguajes OO. UML y clases. Concepto de excepción. Excepciones en Java (Jerarquía) Manejo y captura. Excepciones personalizadas. Lanzamiento de excepciones. Aserciones.	
ORIENTACIONES METODOLÓGICAS	
Se hace ver al alumno la relación entre las actividades y su futura vida laboral. Relacionar los contenidos y actividades con otros módulos del ciclo. Las actividades guían a habituar a la documentación de los trabajos realizados y a la lectura y comprensión de documentación técnica.	
CRITERIO DE EVALUACIÓN	3d, 7a,7b, 7c, 7d, 7e, 7f, 7g, 7h

2. Contenidos.

2.1. Excepciones.

Los lenguajes compilados y/o con tipado de datos fuerte introducen reglas que evitan en gran parte un mal funcionamiento del software, pero muchas veces existen ocasiones en las que no es posible conocer antes de ejecutar el software ciertas situaciones que pueden dar error, por ejemplo ¿qué sucede si se intenta escribir un fichero en una ruta en la que no se tiene permisos?

2.1.1 Concepto de excepción.

Una excepción es una error en **tiempo de ejecución** que se produce de manera inesperada, las causas pueden ser ajenas al software o por falta de previsión de ciertas situaciones.

El ejemplo clásico es la división de un número entre 0, que provoca un fallo. Al crear el software o bien comprueba antes de hacer la división que el divisor no sea 0 o insertar código que trate el posible error en caso de que se produzca.

En Java cuando se produce un error en tiempo de ejecución, la JVM realiza un primer tratamiento devolviendo un objeto de tipo excepción, ya que como se ha tratado en Java todo son objetos.

```
public class Ejemplo1 {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int dividendo,divisor,resultado;
        Scanner input= new Scanner(System.in);
        dividendo=input.nextInt();
        divisor=input.nextInt();
        resultado=dividendo/divisor;
        System.out.println("El resultado es "+resultado);
    }
}
```

Al introducir los valores 5 y 0 se produce el siguiente error:

```
5
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
      at pedro.ieslaencanta.com.excepciones.Ejemplo1.main(Ejemplo1.java:23)
Command execution failed.
```

La línea 23 del código es resultado=dividendo/divisor;

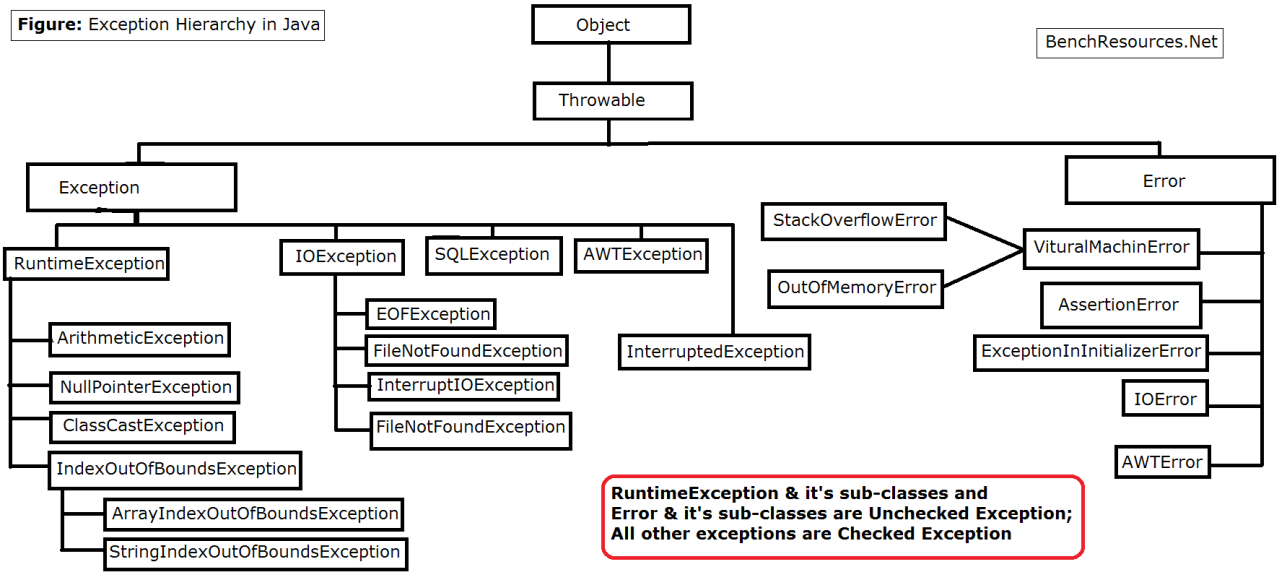
¿En el código anterior existe alguna otra situación en la que se producirá un error en tiempo de ejecución?

Cuando se detecta un fallo y se crea un objeto de tipo excepción, a este proceso se le denomina **lanzar** una excepción, y en caso de no **tratarla** provoca la salida del programa de forma abrupta.

2.1.2 Excepciones en Java (Jerarquía).

Es posible que en una misma instrucción se produzcan diferentes tipos de excepción, en el ejemplo anterior se pueden dar fallos y lanzarse excepciones si se divide entre 0 y tambiñén si lo que se introduce por teclado es diferente a un entero.

Existe una jerarquía de excepciones dependiendo de la causa que desencadene dicha excepción, esta jerarquía por supuesto utililiza herencia. El diagrama de clases de los errores y excepciones en Java es el siguiente:



¿De qué objeto hereda todo el sistema de excepciones? ¿Qué métodos se sabe que tendrá de forma obligatoria cualquier excepción?¿Se cumple con lo tratado al inicio del tema?

La primera clase para los errores es Thowable, la API de dicha clase se encuentra en: https://docs.oracle.com/javase/9/docs/api/java/lang/Throwable.html, esta clase contiene la pila completa (llamada a métodos de diferentes clases desde el principal hasta la llamada que produce el error) entre otra información, un mensaje de error descriptivo de esta heredan 2 nuevas clases:

Error: Se producen cuando es un error grave y difícilmente es posible su tratamiento para que el software no falle, por ejemplo fallo de la máquina virtual, desbordamiento de la memoria o problemas con hilos. **No se suele gestionar las situaciones que hacen lanzar este tipo de error, ya que no es responsabilidad del software y poco se puee hacer.**

Por ejemplo el siguiente código hace salta un error por desbordamiento del heap (memoría dinámica que se usa en tiempo de ejecución y que és limitada).

```
  public static void main(String[] args) {
        String str="desbordar";
        while(true){
            str+=str+"al montón";
        }
    }
```

Lanzándose el siguiente error:

```
Exception in thread "main" java.lang.OutOfMemoryError: Overflow: String length
out of range
      at
java.base/java.lang.StringConcatHelper.checkOverflow(StringConcatHelper.java:57
)
      at java.base/java.lang.StringConcatHelper.mix(StringConcatHelper.java:138)
      at
pedro.ieslaencanta.com.excepciones.DesbordamientoHeap.main(DesbordamientoHeap.j
ava:19)
```

Exception: Errores producidos en tiempo de ejecución por el software, por ejemplo intentar acceder a una posición de un vector que no existe. **Estas si han de ser tratadas y previstas, al menos en los puntos críticos.**

Java define a partir de esta clase una gran cantidad de subclases para el tratamiento de

una gran cantidad de situaciones, en la siguiente imagen se pueden observar todas las subclases de Exception.

Cada subclase añade información particular que permite tratar de forma más detallada, en la siguiente imagen se observan todas las clases que heredan de Exception:

Direct Known Subclasses:

AbsentInformationException, AclNotFoundException, ActivationException, AgentInitializationException, AgentLoadException, AlreadyBoundException, ApplicationException, AttachNotSupportedException, AWTException, BackingStoreException, BadAttributeValueExpException, BadBinaryOpValueExpException, BadLocationException, BadStringOperationException, BrokenBarrierException, CardException, CertificateException, ClassNotLoadedException, CloneNotSupportedException, DataFormatException, DatatypeConfigurationException, DestroyFailedException, ExecutionControl.ExecutionControlException, ExecutionException, ExpandVetoException, FontFormatException, GeneralSecurityException, GSSException, IllegalClassFormatException, IllegalConnectorArgumentsException, IncompatibleThreadStateException, InterruptedException, IntrospectionException, InvalidApplicationException, InvalidMidiDataException, InvalidPreferencesFormatException, InvalidTargetObjectTypeException, InvalidTypeException, InvocationException, IOException, JAXBException, JMException, JShellException, KeySelectorException, LambdaConversionException, LastOwnerException, LineUnavailableException, MarshalException, MidiUnavailableException, MimeTypeParseException, MimeTypeParseException, NamingException, NoninvertibleTransformException, NonInvertibleTransformException, NotBoundException, NotOwnerException, ParseException, ParserConfigurationException, PrinterException, PrintException, PrivilegedActionException, PropertyVetoException, ReflectiveOperationException, RefreshFailedException, RemarshalException, RuntimeException, SAXException, ScriptException, ServerNotActiveException, SOAPException, SQLException, StringConcatException, TimeoutException, TooManyListenersException, TransformerException, TransformException, UnavailableServiceException, UnmodifiableClassException, UnsupportedAudioFileException, UnsupportedCallbackException, UnsupportedFlavorException, UnsupportedLookAndFeelException, URIReferenceException, URISyntaxException, UserException, VMStartException, XAException, XMLParseException, XMLSignatureException, XMLStreamException, XPathException

---

Los más destacados son RunTimeException y IOException.

- **RunTimeException**. Lanza excepciones relacionadas con la programación, como división entre 0 o intento de acceso a indices de vector que no existen. Las subclases de RuntimeException son:

AnnotationTypeMismatchException, ArithmeticException, ArrayStoreException, BufferOverflowException, BufferUnderflowException, CannotRedoException, CannotUndoException, CatalogException, ClassCastException, ClassNotPreparedException, CMMException, CompletionException, ConcurrentModificationException, DataBindingException, DateTimeException, DOMException, DuplicateRequestException, EmptyStackException, EnumConstantNotPresentException, EventException, FileSystemAlreadyExistsException, FileSystemNotFoundException, FindException, IllegalArgumentException, IllegalCallerException, IllegalMonitorStateException, IllegalPathStateException, IllegalStateException, IllformedLocaleException, ImagingOpException, InaccessibleObjectException, IncompleteAnnotationException, InconsistentDebugInfoException, IndexOutOfBoundsException, InternalException, InvalidCodeIndexException, InvalidLineNumberException, InvalidModuleDescriptorException, InvalidModuleException, InvalidRequestStateException, InvalidStackFrameException, JarSignerException, JMRuntimeException, JSException, LayerInstantiationException, LSException, MalformedParameterizedTypeException, MalformedParametersException, MediaException, MirroredTypesException, MissingResourceException, NashornException, NativeMethodException, NegativeArraySizeException, NoSuchDynamicMethodException, NoSuchElementException, NoSuchMechanismException, NullPointerException, ObjectCollectedException, ProfileDataException, ProviderException, ProviderNotFoundException, RangeException, RasterFormatException, RejectedExecutionException, ResolutionException, ResourceRequestDeniedException, SecurityException, SPIResolutionException, SystemException, TypeConstraintException, TypeNotPresentException, UncheckedIOException, UndeclaredThrowableException, UnknownEntityException, UnknownTreeException, UnmodifiableModuleException, UnmodifiableSetException, UnsupportedOperationException, VMDisconnectedException, VMMismatchException, VMOutOfMemoryException, WebServiceException, WrongMethodTypeException, XPathException

¿Qué excepción salta cuándo una clase no implementa el método clone?

- **IOException**. Muchos de los errores de los programas provienen por fallos de entrada salida, el caso más común son los ficheros, pero pueden ser otros como todos los derivados con conexiones de red, como HttpTimeoutException, que se lanza cuando una conexión HTTP tarde más de lo esperado.

Direct Known Subclasses:

AttachOperationFailedException, ChangedCharSetException, CharacterCodingException, CharConversionException, ClosedChannelException, ClosedConnectionException, EOFException, FileLockInterruptionException, FileNotFoundException, FilerException, FileSystemException, HttpRetryException, HttpTimeoutException, IIOException, InterruptedByTimeoutException, InterruptedIOException, InvalidPropertiesFormatException, JMXProviderException, JMXServerErrorException, LoadException, MalformedURLException, ObjectStreamException, ProtocolException, RemoteException, SaslException, SocketException, SSLException, SyncFailedException, TransportTimeoutException, UnknownHostException, UnknownServiceException, UnsupportedDataTypeException, UnsupportedEncodingException, UserPrincipalNotFoundException, UTFDataFormatException, WebSocketHandshakeException, ZipException

Las excepciones de tipo RuntimeException **no existe obligación de ser capturadas**, en cambio el resto son de tipo **Checked**, lo que **obliga a que sea capturad**a , en caso de no serlo se obtiene un error de compilación, por ejemplo, en el ejericio de generar figuras geométricas al guardar la información en un archivo:

FileWriter fichero = null;

```
            fichero = new FileWriter(ruta);
            fichero.write(this.aSVG());
            fichero.close();
```

Al compilar se obtiene el siguiente error (además el IDE también marca el error):

```
*.java:89: error: unreported exception IOException; must be caught or declared
to be thrown
            fichero = new FileWriter(ruta);
*.java:90: error: unreported exception IOException; must be caught or declared
to be thrown
            fichero.write(this.aSVG());
*.java:91: error: unreported exception IOException; must be caught or declared
to be thrown
            fichero.close();
3 errors
```

Indicando que o bien se ha de capturar o bien ha de poder lanzarse, los detalles de como caputar o indicar que ese método puede lanzar excepciones se tratan en el siguiente punto.

De las excepciones de I/O, ¿existen más relacionadas con la red o con el sistema de ficheros? ¿Cuál piensas que es la razón?

2.1.3 Manejo y captura.

Ya se conoce el proceso que se desencadena al producirse un error, en caso de ser un error no controlable se lanza un objeto de la clase Error, en caso de ser otro tipo de error se lanza un objeto de tipo Exception o derivado, en este caso si es posible su tratamiento.

En primer lugar se ha de identificar los lugares en que se pueden lanzar excepciones, por ejemplo, en los lugares en que se realiza una división.

En este punto o bien se captura y maneja la excepción o bien se indica que es posible que el método produzca una excepción siendo responsabilidad de quien realiza la llamada al método.

Caso 1:Capturar y manejar la excepción. Existen una estructura para la captura y manejo de excepciones:

```
try{
      //código que puede hacer que se lance una exepción
}catch (Exception e1){
      //manejo de la excepción e1, que será alguno de los tipos que heredan de
      exception
}catch(Exception e2){
      //manejo de la excepción e2, diferente a e1, que será alguno de los tipos
      que heredan de exception
```

```
}catch(Exception….
      //se manejan todos los tipos que se deseee
finally{
      //código que se ejecuta se produzca o no la excepción
}
```

Al manejar diferentes tipos de excepciones, el anidamiento a de ir de la más concreta a la menos concreta, ya que se captura la primera que coincide con la jerarquía de clases.

En el ejemplo anterior, para capturar 2 tipos de excepciones:

```
    public void guardarDibujo(String ruta) {
        FileWriter fichero = null;
        try {

            fichero = new FileWriter(ruta);
            fichero.write(this.aSVG());


        } catch (IOException ex) {
            System.err.println("Excepción por IO");
        } catch (Exception ex2) {
            System.err.println("Excepción general, la más alta");

        } finally {
            try {
                fichero.close();
                System.out.println("Este código se ejeucta siempre, salte o no
                salte");
            } catch (IOException ex) {
                Logger.getLogger(Dibujo.class.getName()).log(Level.SEVERE,
null, ex);
            }
        }

    }
```

 En el ejemplo anterior ¿qué se ejecuta primero en caso de lanzarse una excepción ex o ex2?. ¿Qué codigo se ejecuta siempre, se lance o no se lance la excepción? ¿Cuál es la razón para que se cierre el fichero en finaly? ¿Qué consecuencias puede tener no hacer lo finally?

Los métodos más destacados de una excepción son: String getMessage que da una descripción del error y printlStackTrace que muestra la pila de llamadas a objetos y métodos que ha desencadenado la expceción.

🎯 **Nunca se ha de dejar sin código un catch ya que no se tendría constancia del fallo del programa, además es más que recomendable, casi obligatorio poseer algún tipo de sistema para realizar un log en fichero u otro medio que registre lo sucedido.**

En el siguiente enlace se listan algunas buenas prácticas con resptecto al tratamiento de excepciones.

https://www.clubdetecnologia.net/blog/2017/java-buenas-practicas-para-el-manejo-de-excepciones/

2. Propagación de la excepción. En caso no tratarse el error, la excepción se propaga al método siguiente hasta que o bien encuentra un try… o se llega al método static main. En ocasiones al usar una librería, clase o método el compilador indica que es posible que se produzca una excepción de algún tipo y que esta debe tratarse **obligatoriamente**, teniendo 2 alternativas:

tratarla con try catch.

Indicar que en el método a su vez puede lanzar excepciones del tipo que se obliga a lanzar, en el ejemplo del fichero es obligatorio tratar excepciones de entrada/salida, sino se trata en el método es necesario indicar en la defínción del mismo que quien lo use habrá de tratarlo:

```
public void guardarDibujo(String ruta) throws IOException {
        FileWriter fichero = null;
        fichero = new FileWriter(ruta);
        fichero.write(this.aSVG());
        System.out.println("Este código se ejeucta siempre, salte o no salte");
    }
```

Ahora quien llame al método guardarDibujo(String ruta) o bien sigue propagando la excepción.

2.1.4 Excepciones personalizadas.

Las excepciones no son más que objetos de clases que heredan de Exception, y por tanto es posible heredar de las mismas para personalizar o crear nuevas excepciones.

En el caso de querer personalizar la excepción aritmética para concretar en la división de 0.

```
/**
 *
 * @author Pedro
 */
public class DivisionporCeroException extends ArithmeticException {
    public DivisionporCeroException(String msg){
        super(msg);
    }
    public DivisionporCeroException(){
        super("Fallo al dividir por 0");
    }

}
```

El uso de estas excepciones personalizadas se realiza capturando la excepción base y lanzando la nueva (creándola), como se ve en el punto siguiente.

2.1.5 Lanzamiento de excepciones.

Hasta ahora se han capturado y tratado las excepciones, pero también es posible lanzarlas para avisar de un uso incorrecto de por ejemplo, las librerías creadas.

En la práctica de Arkanoid a la hora de acceder a la matriz de bloques o ladrillos en muchas ocasiones durante el desarrollo se ha dado el caso o bien de acceder a una posición que está fuera de la matriz, ya sea por filas y/o columnas o accediendo a una posicion correcta esta es nula.

Es interesante capturar las excepciones anteriores:

- NullPointerException
- ArrayIndexOutOfBoundsException

Y a partir de estas crear las excepciones personalizadas con la información, por ejemplo indicar que se ha sobrepasado el tamaño o que el elemento es nulo, o incluso ir más allá,

y no esperar a capturar la excepción.

🎯 **Es necesario lanzar una excepción en caso de que los parámetros o el uso de clases y/o métodos no sea el correcto.**

Para lanzar una excepción, ya sea de las predefinidas o de las personalizadas se utiliza la palabra reservada:

throws

Por ejemplo, en la división por 0 se captura la excepción Aritmética y a continuación con la información se lanza la nueva excepción creada anteriormente:

```
   int dividendo,divisor,resultado;
        Scanner input= new Scanner(System.in);
        dividendo=input.nextInt();
        divisor=input.nextInt();
        try{
        resultado=dividendo/divisor;
        }catch ( ArithmeticException e){
            throw  new DivisionporCeroException("División por 0 personalizado
"+e.getMessage());
        }
        System.out.println("El resultado es "+resultado);
```

Al dividir por 0 se obtiene la siguiente salida:

```
Exception in thread "main"
pedro.ieslaencanta.com.excepciones.DivisionporCeroException: División por 0
personalizado / by zero
      at pedro.ieslaencanta.com.excepciones.Ejemplo1.main(Ejemplo1.java:26)
```

En las buenas prácticas de excepciones se recomienda añadir la máxima información posible, se añade la excepción inicial a la personalizada

Otra opción es no esperar a que se lance la excepción, sino detectar el problema y hacer saltar la personalizada.

```
      int dividendo, divisor, resultado;
      Scanner input = new Scanner(System.in);
      dividendo = input.nextInt();
      divisor = input.nextInt();
      if (divisor == 0) {
        throw new DivisionporCeroException("División por 0 personalizado ");
      }
```

```
        resultado = dividendo / divisor;
        System.out.println("El resultado es " + resultado);
```

Al ejecutar el código anterior:

```
Exception in thread "main"
pedro.ieslaencanta.com.excepciones.DivisionporCeroException: División por 0
personalizado
        at pedro.ieslaencanta.com.excepciones.Ejemplo1.main(Ejemplo1.java:24)
```

2.1.6 Aserciones.

Las aserciones se definene como un predicado (expresión que se evalua a cierto o falso) y se utiliza para garantizar que se cumple el predicado en un punto del programa.

Java implementa desde la versión 1.4 las aserciones, pero **están desaconsejado su uso** y solo sirve para la fase de desarrollo, no utilizandonse en producción, un ejemplo de assert:

```
public void ejemplo(Object parametro) {
    assert parametro!=null;
    if (null==parametro) {
        System.err.println("Parametro null");
        return;
    }
    ….
}
```

Al ejecutar el código de forma normal el assert no se evalua, en cambio al ejecutarlo en modo activación de aserciones si se evalua:

```
java -ea MiAplicacion.jar
```

Saltando una excepción en caso de ser el parámetro nulo.

Relacionado con las aserciones y para evitar usos inadecuados o que provoquen fallos en el mismo se define la programación por contrato, que establece una serie de condiciones y obligaciones en el software de forma similar a una relación de negocios, por ejemplo, que no se intente acceder a un elemento inexistente, las partes que intervienen confian entre ellas.

En la programación por contrato se tienen 3 tipos de aserciones:

- Precondiciones: Condiciones que se han de cumplir para realizar la llamada. Es responsabilidad del elemento que hace uso del método cumplir con las

condiciones.

- Postcondiciones: Define que va a obtener el cliente, el método ha de garantizar de que se devuelva lo pactado.
- Invariantes: Estado de un objeto que se ha de cumplir cuando se crea el objeto, así como antes y despueés de ejecutar **cualquier** método.

De forma nativa no se tiene aserciones utilizables y por tanto tampoco precondiciones, postcondiciones ni invariantes, pero existen librerías externas que permiten establecer aserciones, las más destacadas son:

- **Paquete del framework Spring**, para la definicion de aserciones: https://docs.spring.io/spring-framework/docs/5.0.0.M4_to_5.0.0.M5/Spring%20Framework%205.0.0.M5/org/springframework/util/Assert.html. Un ejemplo de uso:

```
public class Car {
    private String state = "stop";

    public void drive(int speed) {
        Assert.isTrue(speed > 0, "speed must be positive");
        this.state = "drive";
        // ...
    }
}
```

Posee aserciones de diferente tipo como : isTrue, state, notNull…

Artículo sobre Spring Asserts https://www.baeldung.com/spring-assert

Guava, de Google, conjunto de librerías que posee una gran cantidad de complementos como cálculos matemáticos, cache, trabajo con cadenas o colecciones entre otras. Se encuentra en Github en la dirección: https://github.com/google/guava

Posee las siguientes librerías:

Guava: Google Core Libraries for Java HEAD-jre-SNAPSHOT API

Packages

Package	Description
com.google.common.annotations	Common annotation types.
com.google.common.base	Basic utility libraries and interfaces.
com.google.common.cache	This package contains caching utilities.
com.google.common.collect	This package contains generic collection interfaces and implementations, and other utilities for working with collections.
com.google.common.escape	Interfaces, utilities, and simple implementations of escapers and encoders.
com.google.common.eventbus	The EventBus allows publish-subscribe-style communication between components without requiring the components to explicitly register with one another (and thus be aware of each other).
com.google.common.graph	An API for representing graph (node and edge) data.
com.google.common.hash	Hash functions and related structures.
com.google.common.html	Escapers for HTML.
com.google.common.io	This package contains utility methods and classes for working with Java I/O; for example input streams, output streams, readers, writers, and files.
com.google.common.math	Arithmetic functions operating on primitive values and `BigInteger` instances.
com.google.common.net	This package contains utility methods and classes for working with net addresses (numeric IP and domain names).
com.google.common.primitives	Static utilities for working with the eight primitive types and `void`, and value types for treating them as unsigned.
com.google.common.reflect	This package contains utilities to work with Java reflection.
com.google.common.util.concurrent	Concurrency utilities.
com.google.common.xml	Escapers for XML.

Para saber más: Página con las diferentes librerías de Guava https://www.tutorialspoint.com/guava/guava_preconditions_class.htm

Las precondiciones se definen en la librería com.google.commons.base.Preconditions. La API de la clase Precondicitions se encuentra en :https://guava.dev/releases/snapshot-jre/api/docs/com/google/common/base/Preconditions.html.

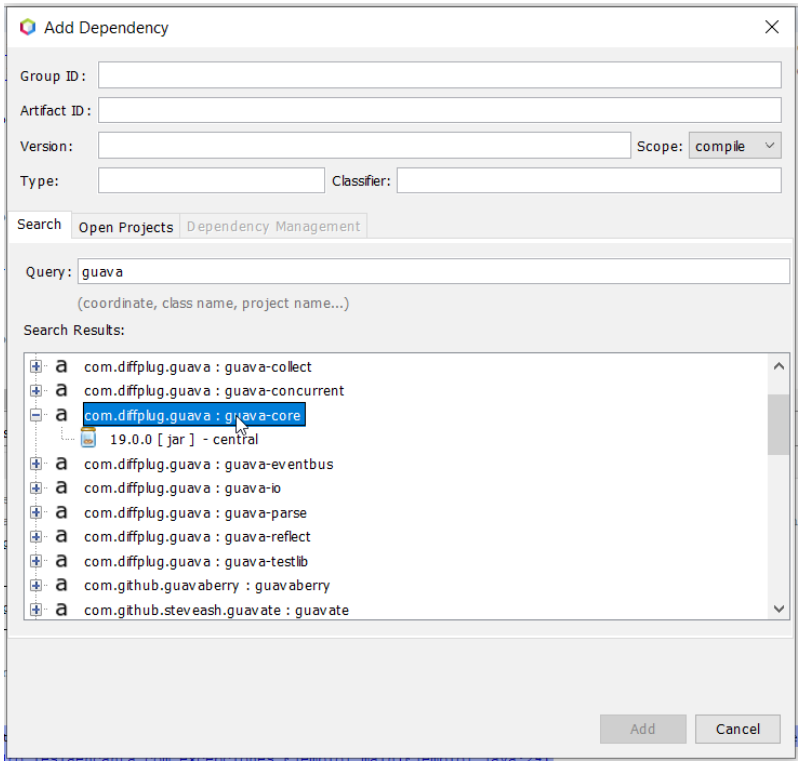
Algunos de los métodos que posee:

Method Summary

All Methods | Static Methods | Concrete Methods

Modifier and Type	Method
static void	checkArgument(boolean expression)
static void	checkArgument(boolean expression, Object errorMessage)
static void	checkArgument(boolean b, String errorMessageTemplate, char p1)
static void	checkArgument(boolean b, String errorMessageTemplate, char p1, char p2)
static void	checkArgument(boolean b, String errorMessageTemplate, char p1, int p2)
static void	checkArgument(boolean b, String errorMessageTemplate, char p1, long p2)
static void	checkArgument(boolean b, String errorMessageTemplate, char p1, Object p2)
static void	checkArgument(boolean b, String errorMessageTemplate, int p1)
static void	checkArgument(boolean b, String errorMessageTemplate, int p1, char p2)
static void	checkArgument(boolean b, String errorMessageTemplate, int p1, int p2)
static void	checkArgument(boolean b, String errorMessageTemplate, int p1, long p2)
static void	checkArgument(boolean b, String errorMessageTemplate, int p1, Object p2)
static void	checkArgument(boolean b, String errorMessageTemplate, long p1)
static void	checkArgument(boolean b, String errorMessageTemplate, long p1, char p2)
static void	checkArgument(boolean b, String errorMessageTemplate, long p1, int p2)
static void	checkArgument(boolean b, String errorMessageTemplate, long p1, long p2)
static void	checkArgument(boolean b, String errorMessageTemplate, long p1, Object p2)
static void	checkArgument(boolean expression, String errorMessageTemplate, @Nullable Object... errorMessageArgs)
static void	checkArgument(boolean b, String errorMessageTemplate, Object p1)
static void	checkArgument(boolean b, String errorMessageTemplate, Object p1, char p2)
static void	checkArgument(boolean b, String errorMessageTemplate, Object p1, int p2)
static void	checkArgument(boolean b, String errorMessageTemplate, Object p1, long p2)
static void	checkArgument(boolean b, String errorMessageTemplate, Object p1, Object p2)
static void	checkArgument(boolean b, String errorMessageTemplate, Object p1, Object p2, Object p3)
static void	checkArgument(boolean b, String errorMessageTemplate, Object p1, Object p2, Object p3, Object p4)
static int	checkElementIndex(int index, int size)
static int	checkElementIndex(int index, int size, String desc)
static <T> T	checkNotNull(T reference)

En el siguiente ejemplo se establece una serie de precondiciones sobre la clase cuenta bancaria, en primer lugar se ha de importar la librería al proyecto, la librería se encuentra en el repositorio de Maven:



La clase movimiento:

```
public class Movimiento {
    private int cantidad;
    private String concepto;
    private Date fecha;
    public Movimiento(String concepto, int cantidad) {
        this.cantidad = cantidad;
        this.concepto = concepto;
        //fecha actual
        this.fecha = new Date();
    }
    public int getCantidad() {
        return cantidad;
    }
    public Date getFecha() {
        return fecha;
    }
    public String getConcepto() {
        return concepto;
    }
}
```

La clase cuenta:

```
enum ESTADO_CUENTA {
    BLOQUEADA,
    ACTIVA,
    DEBAJA
}

public class Cuenta {

    String usuario;
    private ESTADO_CUENTA estado;
    int saldo;
    Date fecha_creacion;
    Movimiento movimientos[];
    int contador = 0;
    public Cuenta(String usuario, int saldo_inicial) {
        this.usuario = usuario;
        this.saldo = saldo_inicial;
        this.fecha_creacion = new Date();
        this.movimientos = new Movimiento[50];
        contador = 0;
        this.estado=ESTADO_CUENTA.ACTIVA;
    }
    public Movimiento ingresar(int cantidad, String concepto) {
        this.saldo += cantidad;
        this.movimientos[contador] = new Movimiento(concepto, cantidad);
        contador++;
        return this.movimientos[contador - 1];
    }
    public Movimiento sacardinero(int cantidad, String concepto) {

        this.saldo -= cantidad;
        this.movimientos[contador] = new Movimiento(concepto, cantidad);
        contador++;
        return this.movimientos[contador - 1];
    }
    public void bloquear() {
        this.estado=ESTADO_CUENTA.BLOQUEADA;
    }
    public void activar(){
        this.estado=ESTADO_CUENTA.BLOQUEADA;
    }
```

```
    public void dar_de_baja(){
        this.estado=ESTADO_CUENTA.DEBAJA;
    }
    public ESTADO_CUENTA getEstado() {
        return estado;
    }
}
```

En la clase cuenta existen muchas situaciones en las que se han de cumplir ciertas condiciones antes de ejecutar el código propio del método, por ejemplo, no puede activarse una cuenta que se ha dado de baja, o un ingreso no puede ser negativo.

Se puede realizar con if antes de ejecutar el código o utilizar precondiciones, en el segundo caso usando Guava:

```
enum ESTADO_CUENTA {
    BLOQUEADA,
    ACTIVA,
    DEBAJA
}

public class Cuenta {

    String usuario;
    private ESTADO_CUENTA estado;
    int saldo;
    Date fecha_creacion;
    Movimiento movimientos[];
    int contador = 0;

    public Cuenta(String usuario, int saldo_inicial) {
        Preconditions.checkArgument(saldo_inicial > 0,
                "Illegal Argument passed: Negative value %s.", saldo_inicial);
        Preconditions.checkNotNull(usuario, usuario,
                "Please check the Object supplied, its null!");
        Preconditions.checkState(this.estado != ESTADO_CUENTA.ACTIVA,
                "Illegal state");
        this.usuario = usuario;
        this.saldo = saldo_inicial;
        this.fecha_creacion = new Date();
        this.movimientos = new Movimiento[50];
        contador = 0;
```

```
        this.estado = ESTADO_CUENTA.ACTIVA;
    }

    public Movimiento ingresar(int cantidad, String concepto) {
        Preconditions.checkArgument(cantidad > 0,
                "Illegal Argument passed: Negative value %s.", cantidad);
        Preconditions.checkState(this.estado != ESTADO_CUENTA.ACTIVA,
                "Illegal state");
        this.saldo += cantidad;
        this.movimientos[contador] = new Movimiento(concepto, cantidad);
        contador++;
        return this.movimientos[contador - 1];
    }

    public Movimiento sacardinero(int cantidad, String concepto) {
        Preconditions.checkArgument(cantidad < 0,
                "Illegal Argument passed: Positive value %s.", cantidad);
        Preconditions.checkState(this.estado != ESTADO_CUENTA.ACTIVA,
                "Illegal state");
        this.saldo -= cantidad;
        this.movimientos[contador] = new Movimiento(concepto, cantidad);
        contador++;
        return this.movimientos[contador - 1];
    }

    public void bloquear() {
        Preconditions.checkState(this.estado == ESTADO_CUENTA.DEBAJA,
                "Illegal state");
        this.estado = ESTADO_CUENTA.BLOQUEADA;
    }

    public void activar() {
        Preconditions.checkState(this.estado == ESTADO_CUENTA.DEBAJA,
                "Illegal state");
        this.estado = ESTADO_CUENTA.BLOQUEADA;
    }

    public void dar_de_baja() {
        this.estado = ESTADO_CUENTA.DEBAJA;
    }
```

```
    public ESTADO_CUENTA getEstado() {
        return estado;
    }
}
```

Al ejecutar el siguiente código, se observa como se lanza una excepción al no cumplir la precondición:

```
public static void main(String[] args) {
        // TODO code application logic here
        Cuenta c= new Cuenta(null,10);
    }
```

Salida:

```
Exception in thread "main" java.lang.NullPointerException: null [Please check
the Object supplied, its null!]
      at
com.google.common.base.Preconditions.checkNotNull(Preconditions.java:253)
      at pedro.ieslaencanta.com.precondicines.Cuenta.<init>(Cuenta.java:32)
      at pedro.ieslaencanta.com.precondicines.Principal.main(Principal.java:18)
```

La programación orientada a aspectos va un paso más alla, extrayendo la lógica accesora de las acciones o métodos en otros puntos del programa, siendo posible su reutilización. En otros lenguajes como Javascript con node.js existen conceptos similares

Para saber más POA https://www.youtube.com/watch?v=AjXPs9nVHow

3. Actividades y ejercicios.

1. ¿Qué es una excepción?
2. ¿Cuándo se produce?
3. ¿De quién hereda Thorwable? ¿Quíen hereda de Thorowable?
4. Los errores de tipo Error, ¿cuándo se producen? ¿cómo tratarlos?
5. Indicar toda la jerarquía de la excepcióon ArryIndexOutOfBound, ¿cuándo se produce esa excepción?
6. ¿En qué casos se lanza la expceción NullPointerException? Explicar algún ejemplo.
7. ¿Qué excepción se lanza en caso de no tener conexión a Internet?

8. Explicar cada una las partes para capturar una excepción:

```
try{
}
catch (Exception e1){ ...}
catch (Exception e2){….}
finally{…}
```

9. ¿Qué relación tiene que existir entre e1 y e2 en el ejemplo anterior?

10. En el ejercicio 8, ¿qué codigo se ejecuta siempre? ¿qué codigo sería conveniente colocar en ese bloque?

11. Indicar la razón para que siempre se tenga que poner código de tratamiento en los catch.

12. Crea una excepción en el caso de seleccioanr una celda en el ajedrez que no tenga figura.

13. Explicar cómo lanzara la excepción anterior.

14. ¿Qué son las aserciones?

15. Expliar en uso de aserciones en Java. ¿a partir de que versión existe? ¿cómo se activan?. Indicar la razón para que no se usa.

16. Crear una clase coche con métodos para acelera, frena, girar volante, comprobar cinturon de seguridad, parar, arrancar, abrir pueta, cerrar puerta.

17. Insertar precondiciones con Ghuava para intentar evitar un mal funcionamiento, por ejemplo no es posible abrir una puerta si tiene una velocidad mayor que 0.

18. Explicar la POA y relacionarla con las aserciones.