Módulo 1: Programación en Java

Roger F. González Camacho

Agosto 2023

Gobierno de Canarias
Servicio Canario de Empleo
Centro GASTRONÓMICO del Atlántico
GOBIERNO DE ESPAÑA
MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL
MINISTERIO DE TRABAJO Y ECONOMÍA SOCIAL
SERVICIO PÚBLICO DE EMPLEO ESTATAL
SEPE

Arrays. Ejercicios

1. Crear una tabla de longitud 10 que se inicializará con números aleatorios comprendidos entre 1 y 100. Mostrar la suma de todos los números aleatorios que se guardan en la tabla.
2. Diseñar un programa que solicite al usuario que introduzca por teclado 5 números decimales. A continuación, mostrar los números en el mismo orden que se han introducido.
3. Escribir una aplicación que solicite al usuario cuántos números desea introducir. A continuación, introducir por teclado esa cantidad de números enteros, y por último, mostrar en el orden inverso al introducido.
4. **Diseñar la función: int maximo (int t [ ], que devuelva el máximo valor contenido en la tabla t.**

Arrays. Ejercicios

5. **Escribir la función int [ ] rellenaPares (int longitud, int fin), que crea y devuelve una tabla ordenada de la longitud especificada, que se encuentra rellena con números pares aleatorios comprendidos en el rango desde 2 hasta fin (inclusive).**
6. **Definir una función que tome como parámetros dos tablas, la primera con los 6 números de una apuesta de la primitiva, y la segunda (ordenada) con los 6 números de la combinación ganadora. La función devolverá el número de aciertos.**

Tratamiento de errores. Excepciones

A lo largo de nuestro aprendizaje de Java nos hemos topado en alguna ocasión con Errores, **pero éstos suelen ser los que nos ha indicado el compilador**. Un punto y coma por aquí, un nombre de variable incorrecto por allá, pueden hacer que nuestro compilador nos avise de estos descuidos. Cuando los vemos, se corrigen y obtenemos nuestra clase compilada correctamente.

Pero, **¿Sólo existen este tipo de Errores? ¿Podrían existir Errores no sintácticos en nuestros programas?** Está claro que sí, un programa perfectamente compilado en el que no existen Errores de sintaxis, puede generar otros tipos de Errores que quizá aparezcan en tiempo de ejecución. A estos Errores se les conoce como excepciones.

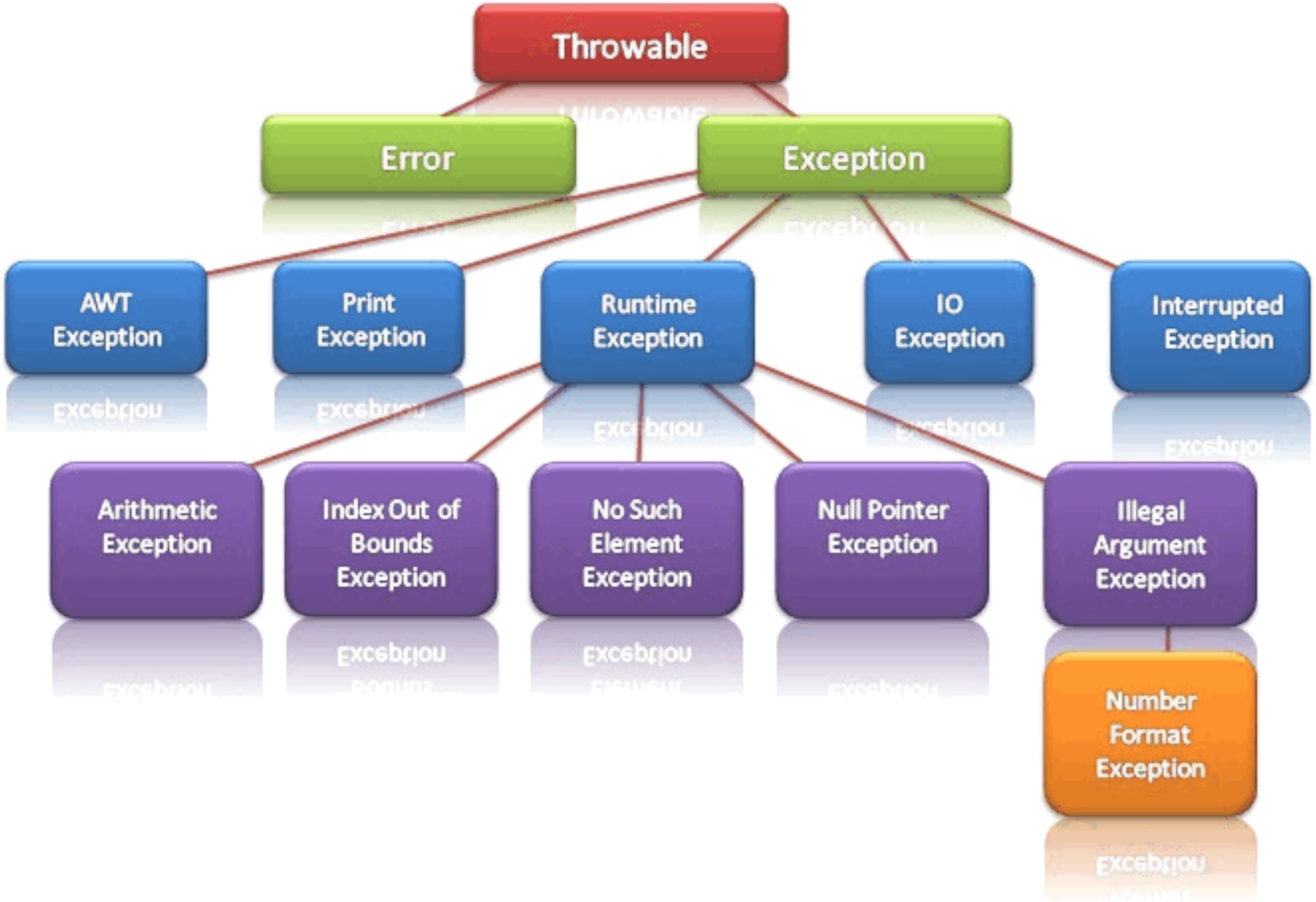
Tratamiento de errores. Excepciones

Aprenderemos a gestionar de manera adecuada estas excepciones y tendremos la oportunidad de utilizar el potente sistema de manejo de Errores que Java incorpora. La potencia de este sistema de manejo de Errores radica en:

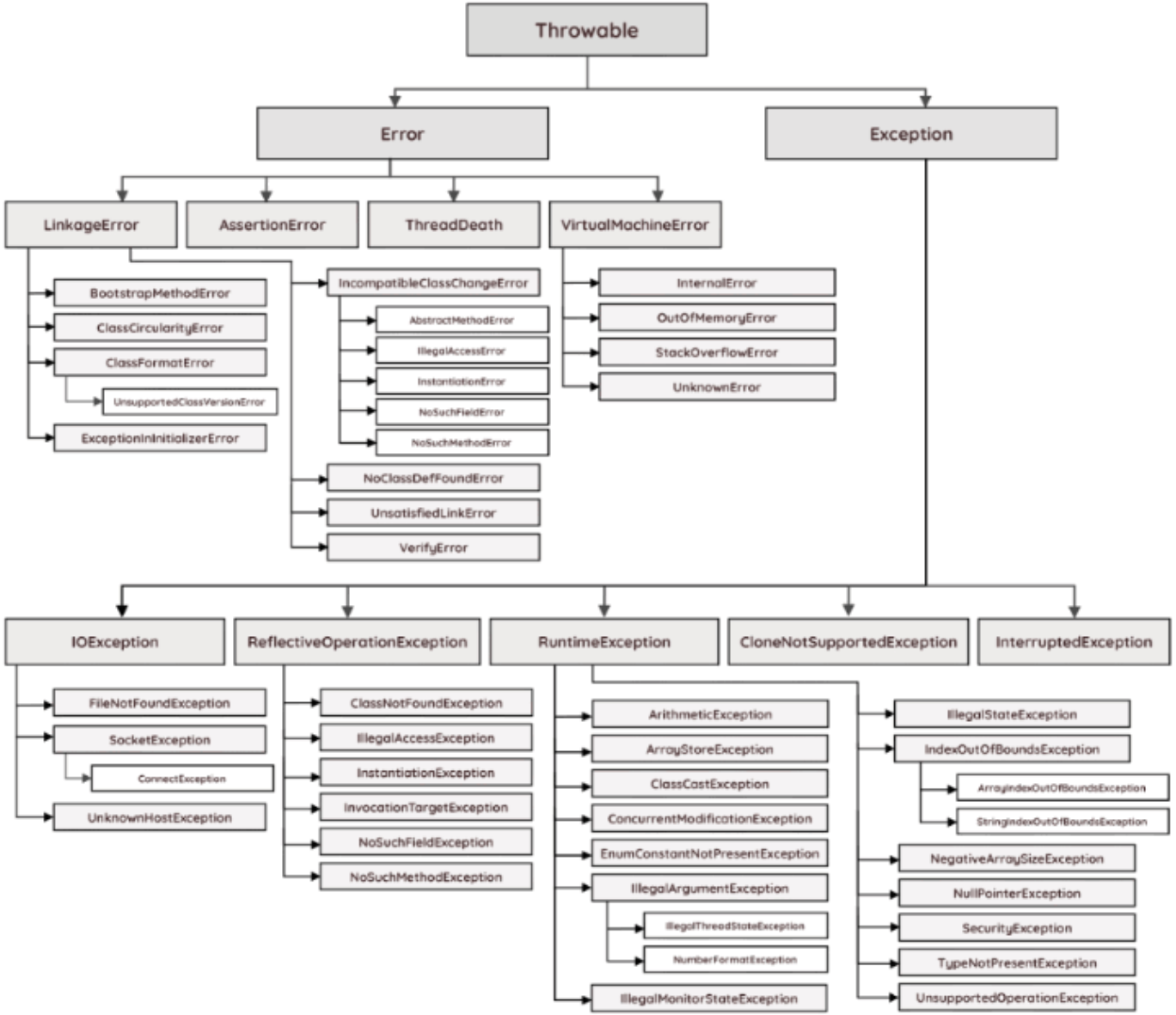
- **Que el código que se encarga de manejar los Errores**, es perfectamente identificable en los programas. Este código puede estar separado del código que maneja la aplicación.

- **Que Java tiene una gran cantidad de Errores** estándar asociados a multitud de fallos comunes, como **por ejemplo divisiones por cero, fallos de entrada de datos, etc. Al tener tantas excepciones localizadas, podemos gestionar de manera específica cada uno de los Errores que se produzcan.**

Tratamiento de errores. Excepciones



Tratamiento de errores. Excepciones



Tratamiento de errores. Excepciones

Una excepción es un evento, que ocurre durante la ejecución de un programa y rompe el flujo normal de instrucciones de dicho programa.

Gracias a las excepciones, el posible gestionar los posibles errores que se pueden producir en un programa, controlando el comportamiento de este en dichas situaciones.

Cuando se produce un error dentro de un método o programa, se crea un objeto Exception, que contiene información sobre el error, y se propaga dicho objeto al sistema, rompiendo el flujo natural de ejecución. Este objeto Exception, puede ser capturado por nuestro programa o en el caso de no ser capturado, provocar que el programa termine.

Clasificación de Excepciones

- **Excepciones controladas ("Checked Exception")** Son errores previstos y que están controlados por el código, de modo que el sistema puede recuperarse ante dichas situaciones.

  Todas estas excepciones deberán ser capturadas o propagadas, es decir, cuando en un punto de un programa se pueda generar una excepción de este tipo, dicha sección de código deberá estar delimitada por un bloque try-catch o deberá pertenecer a un método que indique explícitamente que puede generar excepciones.

- **Errores.** Situaciones de error excepcionales debido a elementos externos al programa y que no pueden ser previstas. Este tipo de excepción no tiene sentido que sea controlada, ya que se dan cuando se producen situaciones graves frente a las cuales el programa no puede actuar.

  Por ejemplo: Errores de la máquina virtual de Java y errores de falta de memoria.

Clasificación de Excepciones

- **Excepciones en tiempo de ejecución (“Runtime Exception”).** Son situaciones de error excepcionales e internas al sistema. Generalmente son provocadas por errores en la programación. Por lo tanto este tipo de excepciones deberían evitarse por medio de una correcta programación. No es necesario declarar o capturar estas excepciones, ya que es mucho más costoso su gestión a través de excepciones que evitar dichas situaciones por código.

Tratamiento de errores. Excepciones

En Java, **las excepciones están representadas por clases**. El **paquete java.lang.Exception y sus subpaquetes contienen todos los tipos de excepciones**. **Todas las excepciones derivarán de la clase Throwable**, existiendo clases más específicas. **Por debajo de la clase Throwable existen las clases Error y Exception**. Errores una clase que se encargará de los Errores que se produzcan en la máquina virtual, no en nuestros programas. Y la clase Exception será la que a nosotros nos interese conocer, pues gestiona los Errores provocados en los programas.

Capturar una excepción.

Para poder capturar excepciones, emplearemos la estructura de captura de excepciones **try-catch-finally**.

Básicamente, para capturar una excepción lo que haremos será declarar bloques de código donde es posible que ocurra una excepción. Esto lo haremos mediante un bloque try (intentar). Si ocurre una excepción dentro de estos bloques, se lanza una excepción. Estas excepciones lanzadas se pueden capturar por medio de bloques catch. Será dentro de este tipo de bloques donde se hará el manejo de las excepciones.

Capturar una excepción.

Su sintaxis es:

```
try {

   código que puede generar excepciones;

} catch (Tipo_excepcion_1 objeto_excepcion) {

   Manejo de excepción de Tipo_excepcion_1;

} catch (Tipo_excepcion_2 objeto_excepcion) {

   Manejo de excepción de Tipo_excepcion_2;

}

...

finally {

   instrucciones que se ejecutan siempre

}
```

Capturar una excepción.

En esta estructura, la parte catch puede repetirse tantas veces como excepciones diferentes se deseen capturar. La parte finally es opcional y, si aparece, solo podrá hacerlo una sola vez.

Cada catch maneja un tipo de excepción. Cuando se produce una excepción, se busca el catch que posea el manejador de excepción adecuado, será el que utilice el mismo tipo de excepción que se ha producido. Esto puede causar problemas si no se tiene cuidado, ya que la clase **Exception** es la superclase de todas las demás. Por lo que si se produjo, por ejemplo, una excepción de tipo **AritmethicException** y el primer catch captura el tipo genérico **Exception**, será ese catch el que se ejecute y no los demás.

Por eso el último catch debe ser el que capture excepciones genéricas y los primeros deben ser los más específicos. Lógicamente si vamos a tratar a todas las excepciones (sean del tipo que sean) igual, entonces basta con un solo catch que capture objetos Exception.

El manejo de excepciones.

Como hemos comentado, siempre debemos controlar las excepciones que se puedan producir o de lo contrario nuestro software quedará expuesto a fallos. Las excepciones pueden tratarse de dos formas:

- **Interrupción.** En este caso se asume que el programa ha encontrado un error irrecuperable. La operación que dio lugar a la excepción se anula y se entiende que no hay manera de regresar al código que provocó la excepción. Es decir, la operación que originó el error, se anula.
- **Reanudación.** Se puede manejar el error y regresar de nuevo al código que provocó el error.

El manejo de excepciones.

Java emplea la primera forma, pero puede simularse la segunda mediante la utilización de un bloque try en el interior de un while, que se repetirá hasta que el error deje de existir. En la siguiente imagen tienes un ejemplo de cómo llevar a cabo esta simulación.

```
/**
 * Esa es la parte de descripción
 * @author ROGERGAMES
 * @etiqueta Uso de la etiqueta y su comentario
 */
public class delegacion_excepciones {
  public static void main(String[] args){
    boolean fueradelimites=true;
    int i; //Entero que tomará valores aleatorios de 0 a 9
    String texto[] = {"uno","dos","tres","cuatro","cinco"}; //String que representa la moneda

    while(fueradelimites){
        try{
            i=  (int) Math.round(Math.random()*9); //Generamos un indice aleatorio
            System.out.println(texto[i]);
            fueradelimites=false;
        }catch(ArrayIndexOutOfBoundsException exc){
            System.out.println("Fallo en el índice");
        }
    }

  }
}
```

El manejo de excepciones.

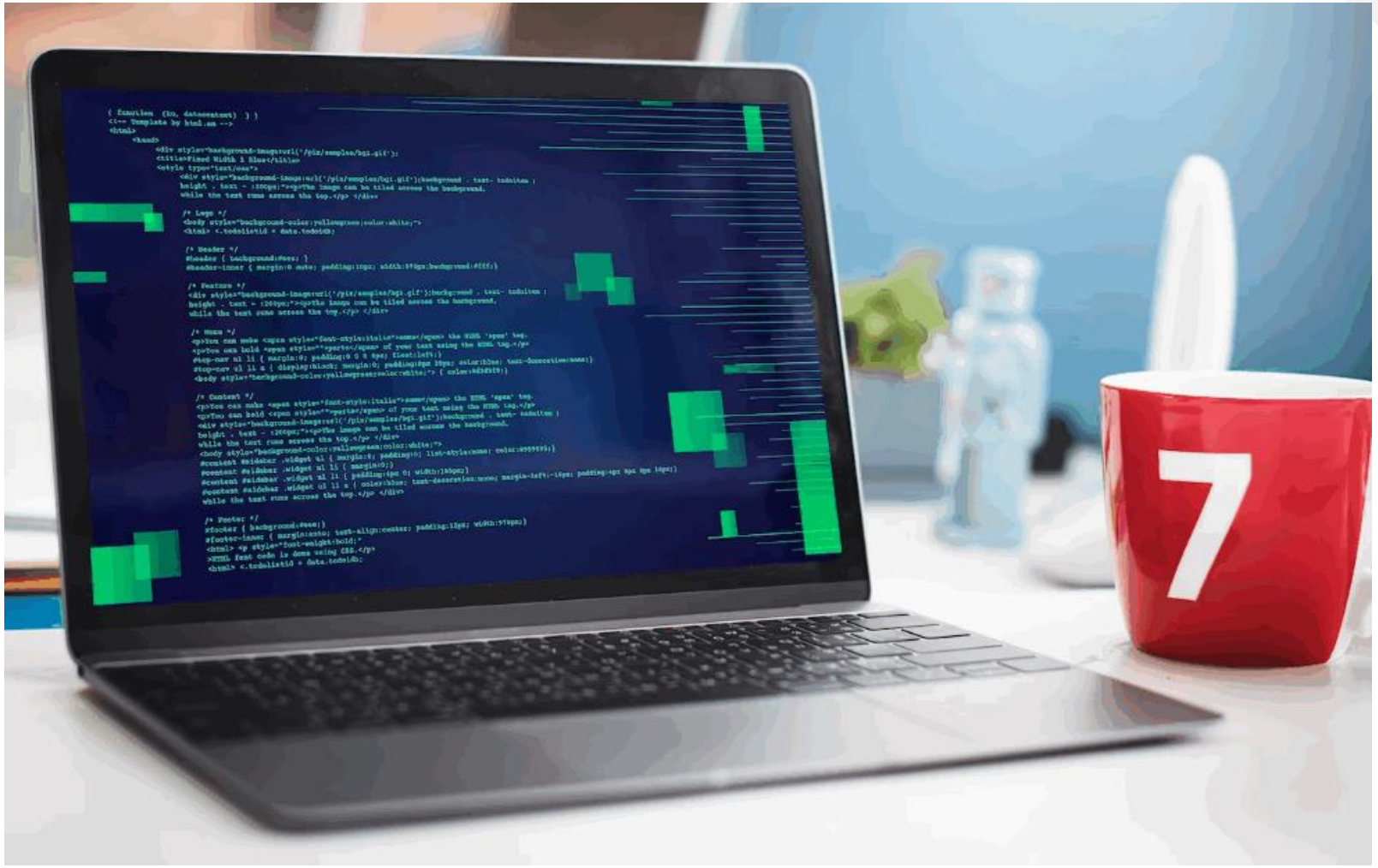
En este ejemplo, a través de la función de generación de números aleatorios se obtiene el valor del índice i. Con dicho valor se accede a una posición del array que contiene cinco cadenas de caracteres. Este acceso, a veces puede generar un error del tipo **ArrayIndexOutOfBoundsException**, que debemos gestionar a través de un catch. Al estar el bloque catch dentro de un while, se seguirá intentando el acceso hasta que no haya error.

Delegación de excepciones con throws.

¿Puede haber problemas con las excepciones al usar llamadas a métodos en nuestros programas? Efectivamente, si se produjese una excepción es necesario saber quién será el encargado de solucionarla. Puede ser que sea el propio método llamado o el código que hizo la llamada a dicho método.

```
public class delegacion_excepciones {

     ...

     public int leeaño(BufferedReader lector) throws IOException, NumberFormatException{

          String linea = teclado.readLine();

          Return Integer.parseInt(linea);

     }

     ...

}
```

Ejercicios



Excepciones.

Ejercicios

1. Deberán aplicar al menos a 10 ejercicios los try-catch-finally.

1.3 Definir y utilizar funciones: ¿Por qué?

La solución para cuando necesitamos la misma funcionalidad en distintos lugares de nuestro código no es más que etiquetar con un nombre un fragmento de código y sustituir en el programa dicho fragmento, en todos los lugares donde aparezca, por el nombre que le hemos asignado. Esta idea puede verse en el siguiente ejemplo:

```java
/**
 *
 */
package com.cip.pog;

/**
 * @author ROGERGAMES
 *
 */
public class NewFuncion {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // código
        System.out.println("Voy a saludar tres veces:"); //Fragmento repetido
        for (int i = 0; i < 3; i++) {
            System.out.println("Hola");
        }
        //más código
        System.out.println("Voy a saludar tres veces:"); //Fragmento repetido
        for (int i = 0; i < 3; i++) {
            System.out.println("Hola");
        }
        //más código
        System.out.println("Voy a saludar tres veces:"); //Fragmento repetido
        for (int i = 0; i < 3; i++) {
            System.out.println("Hola");
        }
        //resto del código
    }

}
```

1.3 Definir y utilizar funciones: ¿Por qué?

```
package com.cga.pico
public class NewFuncion {
        public static void main(String[ ] args) {
        // código
System.out.println("Voy a saludar tres veces:"); //Fragmento repetido
                for (int i = 0; i < 3; i++) {
                        System.out.println("Hola");
                }
                //más código
System.out.println("Voy a saludar tres veces:"); //Fragmento repetido
        for (int i = 0; i < 3; i++) {
        System.out.println("Hola");
                }
//más código
System.out.println("Voy a saludar tres veces:"); //Fragmento repetido
for (int i = 0; i < 3; i++) {
System.out.println("Hola");
}
//resto del código
}
}
```

1.3 Definir y utilizar funciones.

Una **función** en Java, también conocida como **método**, es un bloque de código que realiza una tarea específica y se ejecuta cuando es llamado. Las funciones pueden recibir datos como **entrada (parámetros), procesar estos datos y devolver un resultado (valor de retorno).**

El uso de funciones nos permite **reutilizar y organizar nuestro código de manera más eficiente, ya que podemos dividir un programa en partes más pequeñas y modulares**.

En Java, las funciones se definen **dentro de una clase y se utilizan para manipular el estado (atributos) y comportamiento (métodos) de los objetos de esa clase.**

Definir y utilizar funciones. Sintaxis

```
modificadorAcceso tipoRetorno nombreFuncion (tipoParametro1 nombreParametro1, tipoParametro2
nombreParametro2, ...) {
    // Cuerpo de la función
    // Procesamiento y cálculos
    return valorRetorno;
}
```

Definir y utilizar funciones. Sintaxis

- **modificadorAcceso:** Controla la visibilidad de la función en relación con otras clases y objetos. Puede ser public, private, protected o sin modificador (por defecto).
- **tipoRetorno:** Indica el tipo de dato que la función devolverá. Puede ser un tipo primitivo (int, float, etc.), una clase, una interfaz o void si la función no devuelve ningún valor.
- **nombreFuncion:** Es el nombre único que identifica a la función y sigue las convenciones de **camelCase.**
- **tipoParametro y nombreParametro:** Son los parámetros que recibe la función como entrada. Pueden ser de cualquier tipo de dato, y su cantidad puede variar. Si la función no recibe parámetros, se dejan los paréntesis vacíos.
- **return valorRetorno;:** Es la instrucción para devolver un valor desde la función. Si la función tiene un tipo de retorno **void, no se utiliza la declaración return.**

Definir y utilizar funciones. Ejemplo

En este ejemplo, hemos definido una función llamada sumar dentro de la clase Calculadora. Esta función toma dos parámetros enteros y devuelve la suma de ambos números como resultado.

```java
public class Calculadora {

    // Función que suma dos números enteros
    public int sumar(int numero1, int numero2) {
        int resultado = numero1 + numero2;
        return resultado;
    }
}
```

Estructura de una función.

- **Modificadores de acceso (public, private, protected)**

Los modificadores de acceso determinan la visibilidad de una función en relación con otras clases y objetos.

- **public:** La función es accesible desde cualquier clase.
- **private:** La función solo es accesible dentro de la misma clase en la que se define.
- **protected:** La función es accesible dentro de la misma clase y sus subclases.
- Sin modificador (por defecto): La función es accesible dentro del mismo paquete.

Estructura de una función. Ejemplo

```
public class EjemploModificadores {
  public void funcionPublica() {
    // Accesible desde cualquier clase
  }

  private void funcionPrivada() {
    // Accesible solo dentro de la clase EjemploModificadores
  }

  protected void funcionProtegida() {
    // Accesible dentro de la clase EjemploModificadores y sus subclases
  }

  void funcionPorDefecto() {
    // Accesible dentro del mismo paquete
  }
}
```

Estructura de una función.

- **Tipo de retorno**

El tipo de retorno indica el tipo de dato que la función devolverá. Puede ser un tipo primitivo (int, float, etc.), una clase, una interfaz o void si la función no devuelve ningún valor.

Ejemplo:

```
public class EjemploTiposRetorno {
  public int obtenerEntero() {
    return 42;
  }

  public String obtenerCadena() {
    return "Hola, mundo!";
  }

  public List<String> obtenerListaCadenas() {
    return Arrays.asList("uno", "dos", "tres");
  }

  public void funcionSinRetorno() {
    System.out.println("Esta función no devuelve ningún valor.");
  }
}
```

SERVICIO PÚBLICO DE EMPLEO ESTATAL SEPE

Estructura de una función.

- **Nombre de la función**

El nombre de la función debe ser único y seguir las convenciones de camelCase. Por lo general, los nombres de las funciones comienzan con un verbo que indica qué hace la función, como ***calcular, obtener o mostrar***.

Ejemplo:

```
public class EjemploNombresFunciones {
  public void calcularArea() {
    // ...
  }

  public String obtenerNombreCompleto() {
    // ...
  }

  public void mostrarInformacion() {
    // ...
  }
}
```

Estructura de una función.

- **Parámetros y argumentos**

Los parámetros son variables que reciben valores de entrada cuando se llama a una función. Se especifican entre paréntesis después del nombre de la función, y se indica su tipo y nombre. Si la función no recibe parámetros, se dejan los paréntesis vacíos.

Ejemplo:

```
public class EjemploParametros {
  public int sumar(int a, int b) {
    return a + b;
  }

  public void saludar(String nombre) {
    System.out.println("Hola, " + nombre + "!");
  }

  public void mostrarCoordenadas(int x, int y, int z) {
    System.out.println("Coordenadas: (" + x + ", " + y + ", " + z + ")");
  }
}
```

Estructura de una función.

- **Cuerpo de la función**

El cuerpo de la función es el bloque de código delimitado por llaves ({}) que contiene las instrucciones y procesamientos que se ejecutarán cuando la función sea llamada.

Dentro del cuerpo de la función, podemos realizar cálculos, manipular variables, llamar a otras funciones y controlar el flujo del programa mediante estructuras condicionales y bucles.

Estructura de una función.

```
public class EjemploCuerpoFunciones {
    public double calcularAreaCirculo(double radio) {
        double area = Math.PI * Math.pow(radio, 2);
        return area;
    }
    public void imprimirTablaMultiplicar(int numero) {
        for (int i = 1; i <= 10; i++) {
            System.out.println(numero + " x " + i + " = " + (numero * i));
        }
    }
```

Ejemplo

```
public boolean esNumeroPrimo(int numero) {
        if (numero <= 1) {
            return false;
        }
        for (int i = 2; i < numero; i++) {
            if (numero % i == 0) {
                return false;
            }
        }
        return true;
    }
}
```

Estructura de una función.

- **Modificadores de acceso (public, private, protected)**

Los modificadores de acceso determinan la visibilidad de una función en relación con otras clases y objetos.

- **public:** La función es accesible desde cualquier clase.
- **private:** La función solo es accesible dentro de la misma clase en la que se define.
- **protected:** La función es accesible dentro de la misma clase y sus subclases.
- Sin modificador (por defecto): La función es accesible dentro del mismo paquete.

Creación de una función.

```
public class CalculadoraIMC {
   // Función que calcula el IMC
   public double calcularIMC(double pesoKg, double alturaMetros){
      double imc = pesoKg / Math.pow(alturaMetros, 2);
      return imc;
   }
}
```

Sintaxis para llamar a una función.

```
NombreClase objeto = new NombreClase();

objeto.nombreFuncion(argumentos);
```

Invocar desde otra clase.

```
public class Main {
        public static void main(String[ ] args) {
                //crear el objeto
                CalculadoraIMC obj = new CalculadoraIMC( );
                //accede a la función o método de otra clase
                double result = obj.calcularIMC(12.5, 20.5);
                // Imprimir
                DecimalFormat df = new DecimalFormat("#.00");
                System.out.println(df.format(result));
                System.out.println(String.format("%.2f", + result));
                System.out.printf("%.2f %n", + result);
                System.out.println((double)Math.round(result * 100d) / 100d);
        }
}
```

Añadir más funciones.

```
public class CalculadoraIMC {
   // Función que calcula el IMC.....
// Añadir la Función que clasifica el IMC
   public String clasificarIMC(double imc) {
      if (imc < 18.5) {
         return "Bajo peso";
      } else if (imc >= 18.5 && imc < 24.9) {
         return "Peso normal";
      } else if (imc >= 24.9 && imc < 29.9) {
         return "Sobrepeso";
      } else {
         return "Obesidad";
      }
}
```

Invocar desde otra clase.

```
public class Main {
        public static void main(String[ ] args) {
                //Acceder a la función
                String clasificacion = obj.clasificarIMC(imc);
                System.out.println("Clasificación: " + clasificacion);        }
}
```

Invocar a un método de clase (estático).

```
package com.cga.pro;

public class Matematicas {

   // Función estática que suma dos números

   public static int sumar(int a, int b){

      return a + b;

   }

}
```

Invocar a un método de clase (estático).

```
package com.cga.pro;

public class Main {

	public static void main(String[ ] args) {

		//Llamar a la función estática sumar

		int suma = Matematicas.sumar(5, 7);

		System.out.println(suma);

	}


}
```

Sobrecarga de funciones.

Java permite que dos o más funciones compartan el mismo identificador en un mismo programa. La forma de distinguir entre las distintas funciones sobrecargadas es mediante su lista de parámetros, que deben ser distintas, ya sean en número o en tipo.

Supongamos que queremos diseñar una función para calcular la suma de dos enteros, pero también es útil hacer una suma ponderada, donde cada sumando tenga un peso distinto.

Sobrecarga de funciones. Ejemplo

```
// Función sobrecargada
static int suma(int a, int b){
        int suma;
        suma = a + b;
        return(suma)
}
//Función sobrecargada
static double suma(int a, double pesoA, int b, double pesoB){
        double suma;
        suma = a * pesoA / (pesoA + pesoB) + b * pesoB / (pesoA +pesoB);
        return (suma);
}
```

suma(2,3)

suma(2, 0.25, 3, 0.75)

Recursividad

Una función puede ser invocada desde cualquier lugar: desde el programa principal, desde otra función e incluso dentro de su propio cuerpo de instrucciones. En este último caso, cuando una función se invoca así misma, diremos que es una función recursiva.

Una función recursiva es una función que se llama a sí misma durante su ejecución. La recursión es una técnica poderosa en la programación que puede simplificar la solución de ciertos problemas, como el cálculo de factoriales, la secuencia de Fibonacci y el recorrido de estructuras de datos jerárquicas.

Recursividad

Ejemplo

- **Calcular el factorial de un número utilizando recursión**

```
public class EjemploRecursion {
    public long factorial(long numero) {
        // Condición base: si el número es 0 o 1, el factorial es 1
        if (numero <= 1) {
            return 1;
        }
        // Llamada recursiva: multiplicar el número por el factorial de (número - 1)
        return numero * factorial(numero - 1);
    }
}
```

Recursividad

Ejemplo

- **Calcular el factorial de un número utilizando recursión**

```
public class Main {
    public static void main(String[ ] args) {
        EjemploRecursion ejemplo = new EjemploRecursion();
        long factorial5 = ejemplo.factorial(5); // 5! = 5 * 4 * 3 * 2 * 1 = 120
        System.out.println("Factorial de 5: " + factorial5);

        long factorial10 = ejemplo.factorial(10); // 10! = 3628800
        System.out.println("Factorial de 10: " + factorial10);
    }
}
```

Ejercicios


Gobierno de Canarias
Servicio Canario de Empleo
Centro GASTRONÓMICO del Atlántico
GOBIERNO DE ESPAÑA
MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL
MINISTERIO DE TRABAJO Y ECONOMÍA SOCIAL
SERVICIO PÚBLICO DE EMPLEO ESTATAL
SEPE

Funciones. Ejercicios

1. Diseñar la función eco( ) a la que se le pasa como parámetro un número n, y muestra por pantalla n veces el mensaje «Eco...».
2. Escribir una función a la que se le pasen dos enteros y muestre todos los números comprendidos entre ellos.
3. Realizar una función que calcule y muestre el área o el volumen de un cilindro, según se especifique. Para distinguir un caso de otro se le pasará como opción un número: 1 (para el área) o 2 (para el volumen). Además, hay que pasarle a la función el radio de la base y la altura.

Área = $2\pi \cdot radio \cdot (altura + radio)$

Volumen = $\pi \cdot radio^2 \cdot altura$

Funciones. Ejercicios

4. Diseñar una función que recibe como parámetros dos números enteros y devuelve el máximo de ambos.
5. Crea una función que, mediante un booleano, indique si el carácter que se pasa como parámetro de entrada corresponde con una vocal. (Automático)
6. Crea una función que, mediante un booleano, indique si el carácter que se pasa como parámetro de entrada corresponde con una vocal. (Que lo solicite el usuario)
7. Diseñar una función con el siguiente prototipo:

   boolean esPrimo (int n)

   que devolverá true si n es primo y false en caso contrario.
8. Escribir una función a la que se le pase un número entero y devuelva el número de divisores primos que tiene.

Funciones. Ejercicios

9. **Diseñar la función calculadora ( ), a la que se le pasan dos números reales (operandos) y qué operación se desea realizar con ellos. Las operaciones disponibles son: sumar, restar, multiplicar o dividir. Estas se especifican mediante un número: 1 para la suma, 2 para la resta, 3 para la multiplicación y 4 para la división. La función devolverá el resultado de la operación mediante un número real.**
10. **Repetir la actividad resuelta 4 con una versión que calcule el máximo de tres números. (Sobrecarga)**
11. **Diseñar una función que calcule an, donde n es entero no negativo. Realizar una versión recursiva. (Recursiva)**

Funciones. Ejercicios

12. **Escribir una función que calcule de forma recursiva el máximo común divisor de dos números. Para ello sabemos:**

$$mcd(a, b) = \begin{cases} mcd(a - b, b) & \text{si } a \geq b \\ mcd(a, b - a) & \text{si } b > a \\ a & \text{si } b = 0 \\ b & \text{si } a = 0 \end{cases}$$

13. **Diseñar una función recursiva que calcule el enésimo término de la serie de Fibonacci. En esta serie el enésimo valor se calcula sumando los dos valores anteriores de la serie. Es decir:**

$$fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)$$

$$fibonacci(0) = 1$$

$$fibonacci(1) = 1$$