Capítulo 5

Control de flujo - excepciones - afirmaciones

Revista Digital - Hecha por estudiantes para estudiantes

If y Switch

Las sentencias **if** y **switch** generalmente se refieren a sentencias de decisión. Utilizamos las sentencias de decisión cuando deseamos que nuestro programa ejecute un comportamiento u otro, dependiendo del estado de una o más variables.

If-else

```
if (expresionBooleana) {
  //Código1 ...
} else if (expresionBooleana) {
  //Código2 ...
} else {
  //Código3 ...
}
```

Lo que hace el código es lo siguiente:

1. Evalúa la primer expresión booleana
2. Si el resultado es **true**
   2.1. Se ejecuta **Código1**
3. Si el resultado es **false**
   3.1. Se verifica la segunda expresión booleana
   3.2. Si el resultado es **true**
      3.2.1. Se ejecuta **Código2**
   3.3. Si el resultado es **false**
   3.4. Se ejecuta el **else** (condición por defecto si no se cumplen los **if**´s)
      3.4.1. Se ejecuta **Código3**

Ahora, vamos a aclarar cómo puede llegar a ser la sintaxis:

- Solo debe existir un **if**
- Puede haber uno o más **else if**, siempre que exista un **if** previo
- Puede haber solo un **else**, siempre que esxista un **if** previo
- No es necesario encerrar el contenido de las sentencias entre llaves (esto solo permite escribir código de una línea). No es una práctica recomendada
- Solo se pueden evaluar expresiones booleanas (esto no es **C,** un **boolean true** no es un **int** mayor que 0).
- Los **else if** y **else** deben de estar encadenados al **if** (empiezan a medida que el otro finaliza). Si esta cadena se rompe generará un error de compilación.
- Los **else if** y **else** son encadenados al **if** más próximo.
- Se ejecuta solo la primer condición que califique (en caso de que no califique ninguna, se ejecuta si existe el **else**).

Hay que recordar que una sentencia de condición, en Java, solo evalúa los tipos **boolean**. Es válido asignar un valor a una variable dentro de una condición, pero si el tipo de dato almacenado no es de tipo **boolean**, se generará un error de compilación. De manera que una sentencia como "**if** (**x** = 9)" generará un error de compilación.
Ejemplos de algunos tipos de expresiones válidas, y no válidas:

Expresión 1

```
static public void main(String[] args) {
  boolean expresion = false
  if (expresion)
    System.out.println("Soy el valor 1");
  System.out.println("Soy el valor 2");
  System.out.println("Soy el valor 3");
}
```

```
Soy el valor 2
Soy el valor 3
```

Expresión 2

```
static public void main(String[] args) {
  boolean expresion = false
  if (expresion)
    System.out.println("Soy el valor 1");
  else
    System.out.println("Soy el valor 2");
  System.out.println("Soy el valor 3");
}
```

```
Soy el valor 2
Soy el valor 3
```

Expresión 3

```
static public void ejecutarIf(int valor) {
  if (valor == 1) {
    System.out.println("Soy el valor 1");
  } else if ( valor > 0 ) {
    System.out.println("Soy el valor 2");
  } else {
    System.out.println("Soy el valor 3");
  }
}

static public void main (String[] args) {
  ejecutarIf(1);
  ejecutarIf(0);
  ejecutarIf(87);
}
```

```
Soy el valor 1
Soy el valor 3
Soy el valor 2
```

Expresión 4

```
static public void main (String[] args) {
  boolean expresion = false;
  if (expresion)
    System.out.println("Soy el valor 1");
  System.out.println("Soy el valor 2");
  else
    System.out.println("Soy el valor 3");
}
```

Error de compilación!

Como se puede apreciar en la *expresión 1* solo tenemos el **if**. En la *expresión 2*, le sumamos un **else**. En ambas expresiones, no utilizamos las {} para encerrar el código, pero aun así es válido, dado que ninguno de ellos ocupa más de una línea.

Que se escriba en un renglón no quiere decir que sea código de una sola línea. El `;` delimita el fin de línea por ejemplo, de manera que un código como `int a;a++;` son dos líneas.

En la *expresión 3* agregamos un `else if` a la condición, y las `{}`. Como verás, al llamar a la expresión con el 1, podría calificar para la primera sentencia, o para la segunda, pero como la primera es quien la captura primero, se ejecuta esta.

En la *expresión 4* nos encontramos con un error de ejecución. Este se produce por el precepto de que los `else` – `else if`, deben de estar encadenados unos a otros comenzando por el `if`. Como verán, al no tener `{}`, dentro del `if` solo entra el `print` para "Soy el valor 1". Luego se ejecuta el `print` "Soy el valor 2", y el `else` a continuación. Dado que la sentencia anterior no fue un `if`, o `else if`, se genera un error de compilación.

Veamos un ejemplo un poco más complicado:

```
public class ifTest {
  static public void main(String[] args) {
    if (expresion)
    if (expresion2)
    System.out.println("Me ejecuto 1");
    else
    System.out.println("Me ejecuto 2");
    else
    System.out.println("Me ejecuto 3");
  }
}
```

En realidad, no es más difícil que lo anterior, solamente tiene un `if` anidado, una indentación que no sirve para nada, pero… el examen es generado dinámicamente, por lo cual, los creadores admiten que podemos llegar a encontrarnos con código como este.

Ahora pensemos, que valor veríamos si, `expresion` es `true` y `expresion2` es `false`?

```
Me ejecuto 2
```

Y si `expresion` es `false` y `expresion2` es `true`?

```
Me ejecuto 3
```

Y si `expresion` es `false` y `expresion2` es `false`?

```
Me ejecuto 3
```

Para que vean más claramente, el código indentado y con llaves sería el siguiente:

```
public class ifTest {
  static public void main(String[] args) {
  if (expresion) {
    if (expresion2) {
    System.out.println("Me ejecuto 1");
    } else {
    System.out.println("Me ejecuto 2");
    }
  } else {
    System.out.println("Me ejecuto 3");
  }
  }
}
```

Si quieren hacer la prueba se darán cuenta de que el código resultante es el mismo.

Switch

Una manera de simular el uso de múltiples `else if` es la sentencia `switch`.

Básicamente, cuando tengan que verificar una variable por algunos valores en concreto, lo mejor es utilizar `switch`. Cuando el conjunto de valores a verificar es muy grande (ejemplo, cualquier valor menor que 100), se utiliza `if`.

Estructura de la sentencia `switch`:

```
static public class switchtest {
  static public void main(String[] args) {
  int x = 45;
  switch (x) {
    case 1:
    System.out.println("valor: 1");
    break;
    case 2:
    System.out.println("valor: 2");
    break;
    case 3:
    case 4:
    System.out.println("valor: 3 o 4");
    break;
    case 5:
    System.out.println("valor: 5");
    break;
    default:
    System.out.println("Ningún valor fue tenido en cuenta");
  }
  }
}
```

Iremos explicando el código anterior por pasos.

Primero, tengamos en cuenta las condiciones que debe cumplir una sentencia switch para ser válida:

- Cada valor especificado en el case debe de ser un literal o una constante final cargada en tiempo de compilación (cuando se declara se inicializa. `final int` a = 1;)
- Solo se puede comparar contra un valor, no contra un rango (`>`, `<`, `>=`, `<=` no están permitidos).
- La sentencia **`break`** es opcional, pero si no se especifica, se seguirán ejecutando las líneas inferiores hasta toparse con otro **`break`** o terminar la sentencia **`switch`**.
- Los valores tanto para el **`case`**, como para la entrada del **`switch`**, solo pueden ser aquellos que se puedan tipificar como **`int`** (no importa que el casteo sea implícito). Tampoco califica un String. Puede llegar calificar un **`enum`**.
- No pueden haber dos **`case`** que comparen el mismo valor.
- Si se utilizan tipos de dato que no sean **`int`**, tener mucho cuidado, ya que si alguno de los valores de un case produce un overflow, se genera un error de compilación.
- Se pueden combinar varios case sucesivamente (ver ejemplo **`case 3: case 4:`**) puede calificar con cualquiera de los valores, y solo es necesario que califique para 1.
- La sentencia **`default`** se ejecuta si no calificó ningún **`case`**. Esta no necesita estar al final.
- Por más que se ejecute **`default`**, si no se especifica un break, y hay más sentencias debajo de esta, se ejecutarán como con un **`case`** normal.

Bucles e iteraciones

En java existen tres sentencias para bucles: **while**, **do** y **for** (este último tiene dos alternativas).

While

La sentencia **while** es adecuada cuando no tenemos idea de cuantas veces se ejecutará un bloque de código, pero si queremos que en algún momento, este finalice.
La sintaxis del **while**:

```
while (condicion) {
  //Bloque de código ...
}
```

- Siempre que la condición sea verdadera se ejecutará el bloque de código.
- Puede que el bloque de código no se ejecute nunca, si la condición inicialmente es **false**.
- La condición tiene que ser una expresión booleana.
- Cualquier variable utilizada dentro del **while** debe de ser declarada previamente (es ilegal declarar una variable dentro de la condición).

Do

La sentencia **do** es semejante al **while**, son la diferencia de que primero ejecuta el bloque de código, y luego evalúa la condición (al revés del **while**).
La sintaxis del do:

```
do {
  //Bloque de código ...
} while (condición);
```

A diferencia del while:

- La condición se ejecutara al menos una vez.
- El **while** se finaliza con **;**.

For

El for es un poco más complejo, pero simplemente porque contiene más elementos en su invocación que la condición. Estos se pueden resumir de la siguiente manera:

```
for ( inicializacion ; condicion ; incremento variables contador ) {
  //Bloque de código ...
}
```

Vamos a describir que hace, y que se puede hacer en cada sección:

- Inicialización

  En esta parte se pueden asignar valores a variables ya declaradas, así como declarar una variable (esta solo será visible para la sentencia **for**) e inicializarla.
- Condición

  Al igual que en las sentencias de bucle anteriores, la condición indica si se vuelve a ejecutar el bloque de código o no.
- Incremento variables contador

  Aquí se incrementan las variables que son utilizadas para verificar la condición (en realidad, se pueden incrementar o decrementar con cualquier operación matemática). A su vez, esta sección permite realizar cualquier acción, hasta puedes realizar un **print** dentro de la misma.

Veamos una sentencia real:

```
int y = 0;
int x = 0;
for ( x = 0, y = 3 ; x == 0 && y < 10 ; y++, x-- ) {
  //Bloque de código ...
}
```

Algunas cuestiones sobre la sentencia **for**:

- A diferencia de los bucles anteriores, el **for** es utilizado cuando conocemos la cantidad de veces que queremos que se ejecute un bloque de código.
- En la inicialización podemos especificar varias variables, separadas cada una por una coma, pero solo una acción
  - Podemos cambiar el valor de variables ya declaradas previamente al for
  - Podemos declarar e inicializar variables solo para el for, que no se encuentren definidas previamente en el mismo scope.
- Solo podemos evaluar una condición (esto no quiere decir que no podamos combinar sentencias con los operadores **&&**, **&**, **||** y **|**).
- Podemos realizar cualquier operación matemática sobre las variables en el incremento de variables contador.
- La sección de incremento variables contador se ejecuta cada vez que se completa el bloque de código.
- Para separar elementos en la misma sección se utiliza la **,**.

En un bucle **for**, no es necesario especificar ninguna de las tres secciones. Se las puede dejar en blanco. Un ejemplo válido sería for(; ;){}.
En la sección incremento variables contador, en realidad podemos realizar cualquier operación, indistintamente de que la misma sea incremento, o una salida por pantalla.

For-each

Es una nueva funcionalidad que se encuentra disponible a partir de java 6, la cual permite realizar una iteración de un array o colección.
En vez del tradicional **for** el cual contiene tres secciones, el **for-each** contiene dos secciones.
Veamos un ejemplo con los dos tipos de sintaxis:

```
//Sintaxis for
int[] arr = {1, 2, 3, 4, 5};
for (int x = 0 ; x < arr.length ; x++ ){
   System.out.println(a[x]);
}

//Sintaxis for-each
int[] arr = {1, 2, 3, 4, 5};
for (int x : arr) {
   System.out.println(x);
}
```

La nueva sintaxis está formada por:

```
for ( declaración : expresión ) {
   //Bloque de código ...
}
```

Vamos a describir que hace, y que se puede hacer en cada sección:

- Declaración
  - Permite declarar una variable, la cual se irá completando con cada elemento del array/colección.
  - La variable declarada debe ser del mismo tipo que cada elemento dentro del array, o un tipo compatible (casteo implícito, o dentro del mismo árbol de herencia siempre que sea downcasting, no se permite upcasting).

- La variable no debe de existir en el mismo nivel de visión (scope).

- Expresión
  - Especifica el array o colección sobre el cual se realiza la iteración.
  - Se puede especificar la llamada a un método, siempre que este devuelva un array o colección.
  - El tipo del array puede ser cualquiera: primitivas, objetos, incluso array de arrays.

Algunos ejemplos de for-eachs válidos:

```
class Vehiculo {
  public String toString(){
    return "Soy un vehiculo";
  }
}
class Auto extends Vehiculo {
  public String toString() {
    return "Soy un auto";
  }
}
class Moto extends Vehiculo {
  public String toString() {
    return "Soy una moto";
  }
}

public class forEachTest {
  static public void main(String[] args) {
    int[] intArray = {1, 2, 3};
    int[][] intIntArray = {{1, 2}, {3, 4}, {5, 6}};
    Vehiculo[] vehiculoArray = {new Auto(), new Moto()};

    System.out.println("intArray:");
    for (int x : intArray) {
      System.out.println(x);
    }

    System.out.println("\nintIntArray:");
    for (int[] x : intIntArray) {
      System.out.println(x);
    }

    System.out.println("\nvehiculoArray:");
    for (Vehiculo x : vehiculoArray) {
      System.out.println(x);
    }
  }
}
```

Break y continue

Estos operadores pueden ser utilizados para salir del bucle, o para saltar a la siguiente iteración.

continue debe de declararse dentro de un bucle, dentro de una sentencia **switch** generará un error de compilación. **Break** puede ser utilizado dentro de un bucle o sentencia **switch**.

Pero que significa esto, bueno, vamos a ver en profundidad cada una de las sentencias dentro de un bucle **if**.
Ejemplo con **continue**:

```
if ( int x = 0 ; x < 10 ; x++ ) {
  if ( x == 3 )
    continue;
  System.out.print(x);
}
System.out.println("\nSe terminó el if");
```

```
12456789
Se terminó el if
```

Podemos ver como **continue** hace que se pase a la siguiente iteración.

Ejemplo con **break**:

```
if ( int x = 0 ; x < 10 ; x++ ) {
  if ( x == 3 )
    break;
  System.out.print(x);
}
System.out.println("\nSe terminó el if");
```

```
12
Se terminó el if
```

En este caso, al ejecutarse el **break** se sale del bucle, lo que produce la salida anterior.

Los **continue** y **break** sin etiqueta (**label**) afectan al bucle más profundo sobre el que se encuentren.

En el siguiente código podemos apreciar el nivel sobre el que afecta el **break**:

```
if ( int x = 0 ; x < 3 ; x++ ) {
  if ( int y = 0 ; y < 2 ; y++ ) {
    if ( x > 0 )
      break;
    System.out.print(" y:" + y);
  }
  System.out.print(" x:" + x);
}
```

Cuando se ejecuta el **break**, solo sale del bucle de mayor anidación, que en el caso anterior, es el **if** (**int y** ...
Veamos una anidación de **if**´s con **breaks** y **continues** más compleja para entender cómo funciona:

```
public class breakTest {
  static public void main(String[] args) {
    for ( int x = 0 ; x < 5 ; x++ ) {
    if ( x == 4 )  //El bucle correspondiente a X=4 no se ejecuta
    break;
      for ( int y = 0 ; y < 3 ; y++ ) {
        System.out.print(" y:" + y);
        if ( y == 1 ) { //y=1 hace que se pase a la siguiente iteración
          continue;
        } else if ( y == 2 ) {
          break;  //y=2 hace que se corte el bucle for( int y…
        } else {
          System.out.print(" z:" + y); //Si y=0 se muestra z
        }
        if ( x == 2 )
          continue;
        System.out.println(" x:" + x); //Solo cuando y=0 && x!=2 se llega a este punto
      }
    }
  }
}
```

Es tortuoso el solo mirar este código, pero si puedes obtener la salida por pantalla, significa que eres uno con los bucles.
La salida por pantalla sería:

```
y:0 z:0 x:0
y:1 y:2 y:0 z:0 x:1
y:1 y:2 y:0 z:0 y:1 y:2 y:0 z:0 x:3
y:1 y:2
```

Sentencias de ruptura con etiquetas

La etiqueta permite identificar a un bucle mediante un nombre, de manera que cuando ejecutamos un **break** o **continue**, se le puede especificar a qué bucle deseamos que afecte, sin importar donde se encuentre.
Las etiquetas deben de cumplir con las reglas de identificadores legales y convenciones de Java.

La sentencia de ruptura que llame a una etiqueta debe de estar contenida dentro del bucle con dicha etiqueta, de lo contrario generará un error.

Ejemplo:

```
exterior:
  for(int a = 0 ; a < 10 ; a++ ) {
   for(b = 0 ; b < 3 ; b++ ) {
    if (a == 3)
      break exterior;
    System.out.println("B:" + b);
    continue exterior;
   }
   System.out.println("A:" + a);
  }
```

```
B:0
B:1
B:2
```

Como verán, nunca se ejecuta la salida por pantalla de A, ni ninguna de B para valores de b>0. Esto es porque las sentencias de ruptura no afectan al elemento más próximo, sino al que contiene la etiqueta, que en este caso es el más externo.

Manejo de excepciones

Una excepción significa una "condición excepcional", la cual altera el flujo normal de ejecución del programa.
El manejo de excepciones en Java permite una manera limpia de manejar las mismas aislándolas del código que las puede lanzar, sin tener que escribir código que verifique los valores de retorno. Además, permite utilizar la misma forma de tratar una excepción para un rango de estas.

Capturando una excepción utilizando try y catch

Cuando se genera una excepción se dice que esta se lanza (**throw**). El código encargado de manejar la excepción es el manejador de excepciones (**exception** handler), y este atrapa (**catch**) la excepción lanzada.
Para poder especificar que código ejecutar si es que una excepción es lanzada, utilizamos la sintaxis **try** – **catch**.

```
try {
  //Bloque de código a verificar ...
} catch(excepcion) {
  //Excepción con que se verifica ...
} finally {
  //Bloque de código que se ejecutará siempre.
}
```

Dentro del **try** se coloca todo el bloque de código que pueda llegar a lanzar una excepción y quisiéramos verificar y/o, que dependa de algún otro bloque que puede llegar a lanzar la excepción.
Dentro del **catch** se especifica el código a ejecutar en caso de identificarse la excepción declarada.
Dentro del **finally** se pone el bloque de código que se ejecutará siempre, sin importar si se lanzó o no alguna excepción, sea cual sea.
Algunas cuestiones

- **Try**
  - Una vez que se lance una excepción, no se continúa ejecutando el bloque de código del **try**.
  - Solo se detecta la primera excepción lanzada.
  - Si se especifica un try, se debe especificar un **catch** y/o un **finally**.
- **Catch**
  - Se pueden especificar la cantidad de **catch** que se desee, con las siguientes condiciones
    - Estos se deben de ubicar inmediatamente después del **try** y antes del **finally** si es que se especifica.
    - Si hay más de un **catch**, estos se deben de especificar uno a continuación del otro.
    - No es posible especificar dos **catch** que manejen la misma excepción
    - No es posible especificar un **catch** que maneja una excepción que un **catch** anterior también lo hace (el **catch** anterior es la superclase, y el **catch** siguiente una subclase. Genera error de compilación).
    - Un **catch** puede manejar un tipo de excepción que admita un grupo de estas
  - Una vez que se detecta una excepción, se entra en el primer **catch** que califique con el tipo de excepción lanzada
  - No hay forma de que una vez cargado el catch se continúe con la ejecución del **try**.
  - Es posible no especificar un **catch** y que la compilación sea válida (siempre que se especifique un **throws** en la firma).
- **Finally**
  - Este bloque es opcional.
  - Se ejecutará siempre, por más que se lance una excepción o no, sea cual sea (en realidad es un 90% cierto). Es posible que no se ejecute si se lanza un **return** en alguna de las sentencias **try** o **catch**.
  - Se ejecuta luego del **try** y **catch**.
  - Debe especificarse inmediatamente luego de un bloque **catch** (si es que se lo especificó), en su defecto, del **try**.

Propagación de excepciones

En realidad no es necesario capturar una excepción inmediatamente. A lo que me refiero es que cada código que pueda lanzar una excepción no necesariamente debe de estar rodeado por un **try**.
Cuando un método no captura la excepción de esta manera, se dice que la está esquivando (ducking).

Pero hay una regla fundamental, si se produce una excepción, alguien la tiene que capturar, o lo hará la JVM y créeme, no te agradará que lo haga (si lo hace, detendrá la ejecución del programa, y mostrará por pantalla la excepción lanzada, junto con el stack de la misma).

Veamos un ejemplo simple:

```
public class exceptionDividoPorCero {
  static public int tercerMetodo() {
    return 4/0;  //Ouch!, dividido por 0...
  }

  static public int segundoMetodo() {
    return exceptionDividoPorCero.tercerMetodo();
  }

  static public int primerMetodo() {
    return exceptionDividoPorCero.segundoMetodo();
  }
  static public void main(String[] args) {
    System.out.println("Valor:" + exceptionDividoPorCero.primerMetodo());
  }
}
```

Como podemos apreciar, una división por 0 lanza una excepción en tiempo de ejecución. En la salida por pantalla (producida porque

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
  at exceptionDividoPorCero.tercerMetodo(exceptionDividoPorCero.java:3)
  at exceptionDividoPorCero.segundoMetodo(exceptionDividoPorCero.java:7)
  at exceptionDividoPorCero.primerMetodo(exceptionDividoPorCero.java:11)
  at exceptionDividoPorCero.main(exceptionDividoPorCero.java:14)
```

la excepción llego hasta la JVM, o dicho de otra forma, la excepción fue esquivada por el método main), vemos dos cosas:

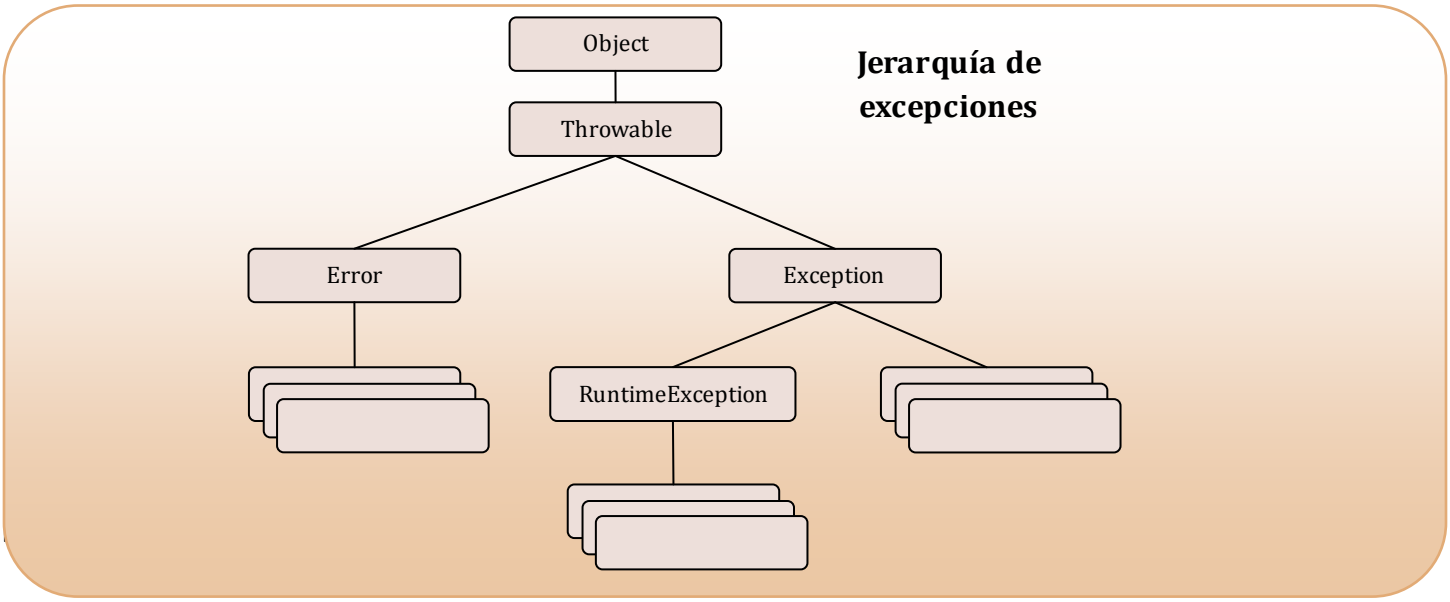
- El tipo de excepción lanzada: java.lang.ArithmeticException con el texto "/ by zero" (dividido por cero) que en realidad es una excepción de tipo **RuntimeException**.
- El stack (pila) de llamadas, desde donde se lanzo la excepción, hasta el main. También informa las líneas de las llamadas (esto último es muy útil para rastrear los errores).

Definiendo que es una excepcion

En Java, o es una primitiva o es un Objeto, y dado que ya cubrimos todas las primitivas, nos queda… así es, una Excepción es un Objeto. Con la salvedad de que todas las excepciones tienen en su árbol de herencia a la clase java.lang.Exception.

toda excepción contiene en su árbol de herencia java.lang.Exception. Pero no es necesario que herede directamente, puede ser una subclase, o más niveles de herencia como se desee. Pero siempre, al final del camino habrá una clase Exception esperando.

Jerarquía de excepciones



Exception — De esta se desprenden dos categorías de excepciones: Excepciones que son posibles de interceptar, y excepciones en tiempo de ejecución, que son muy difíciles de detectar, y que pueden derivarse de errores del programa.

Para el examen solo debes de saber que cualquier tipo de excepción derivable de **Throwable**, **Error**, **Exception** y **RuntimeException** puede ser lanzada con un **throw**, y que todos los tipos derivados de estos pueden ser capturados (aunque es muy raro capturar otra cosa que no sea una **Exception**).

Atrapando excepciones

Cuando decimos que se puede atrapar una excepción por su tipo, o por un tipo de una de sus superclases, nos referimos al comportamiento habitual de un objeto con respecto a los parámetros de un método.

Solo hay que tener en cuenta una cuestión, cuando en una sentencia **try-catch** se lanza una excepción, se verifican los **catch** desde arriba hacia abajo, y se ejecuta el que primero califique para la excepción.

Veamos un ejemplo: (No se preocupen en entender ahora el por qué de las clases que cree, solo sepan que podemos crear nuestras propias excepciones)

```
class PadreException extends Exception{
  public PadreException(){}
  public PadreException(String s){ super(s); }
}

class HijoException extends PadreException{
  public HijoException(){}
  public HijoException(String s){ super(s); }
}

class NietoException extends HijoException{
  public NietoException(){}
  public NietoException(String s){ super(s); }
}

public class pruebaExcepciones_1 {
  static public void metodoLanzaException() throws NietoException{
    throw new NietoException();
  }
  static public void main(String[] args) {
    try {
       pruebaExcepciones_1.metodoLanzaException();
    } catch(PadreException e) {
       System.out.println(e);
    }

    try {
       pruebaExcepciones_1.metodoLanzaException();
    } catch(HijoException e) {
       System.out.println(e);
    }

    try {
       pruebaExcepciones_1.metodoLanzaException();
    } catch(NietoException e) {
       System.out.println(e);
    }

    try {
       pruebaExcepciones_1.metodoLanzaException();
    } catch(NietoException e){
       System.out.println("Nieto");
    } catch(HijoException e){
       System.out.println("Hijo");
    } catch(PadreException e){
       System.out.println("Padre");
    }
  }
}
```

```
NietoException
NietoException
NietoException
Nieto
```

Como verán, la excepción puede ser captura por el tipo de la clase o de alguna de sus superclases.
También se puede apreciar como en la colección de **catch**s califica en el primer intento.

Si se especifican varios **catch**, y alguno de estos es una subclase de otro, la misma debe posicionarse siempre por encima de sus clases padres. Si por ejemplo en el código anterior, NietoException se hubiera especificado luego de alguna de las otras dos, hubiera generado un error de compilación, dado que esa excepción ya se está capturando.

Declaracion de excepciones e interface pública

Como verán en el ejemplo anterior, el método que lanza la excepción tiene en su firma un **throws**, esto es así porque las excepciones que puede llegar a generar un método o clase son parte de la firma.

Todo método que lance (**throw**) o esquive (duck) una excepción, a menos que sea una **RuntimeException** o subclase de esta, debe declararla (**throws**) o capturarla (**try-catch**).

Relanzando excepciones

Si utilizas un **try-catch**, al capturar la excepción puedes relanzarla, o lanzar otro tipo.

Veamos un ejemplo:

```
class MyException extends Exception{
  public MyException(){}
  public MyException(String s){ super(s); }
}

public class ejemploRelanzarException{

  static public void metodoTercero() throws Exception{
    try{
      throw new Exception();  //Lanza una excepcion
    }catch(Exception ex){    //La atrapa (catch)
      throw ex;         //La vuelve a relanzar y se produce un duck (sale del método)
    }
  }

  static public void metodoSegundo() throws MyException{
    try{
      ejemploRelanzarException.metodoTercero(); //Llama a un método que puede devolver
una excepcion de tipo Exception
    }catch(Exception ex){            //Verifica por la excepcion de ese tipo
      throw new MyException(ex.toString()); //Relanza una nueva excepcion con el
informe de la primera como mensaje
    }
  }

  static public void main(String[] args){
    try{
      ejemploRelanzarException.metodoSegundo(); //Verifica un método que puede devolver
una excepcion de tipo MyException
    }catch(MyException ex){  //Atrapa la excepcion
      System.out.println(ex);  //Muestra la información de la excepcion
    }
  }
}
```

¡Demasiadas excepciones! Ahhh… ERROR!!!

Vamos, que no es tan difícil. Primero prestemos atención a las firmas de cada método:

- metodoSegundo dice lanzar una excepción de tipo myException
- metodoTercero dice lanzar una excepción de tipo Excepcion

En el caso del metodoTercero, este lanza la excepción, la captura, y relanza la misma (cuando hace el segundo **throw**, no tiene el **new**).

segundoMetodo captura la excepción, y en respuesta lanza una nueva excepción de tipo MyException.

Dado que el método main solo interactua con metodoSegundo, cuando se realiza el try, solo buscamos por la excepción lanzada por este (de todas maneras, si hubiéramos especificado como tipo Exception hubiera sido válido, dado que MyException hereda de esta).

Cuando una excepción es atrapada (try-catch), y como consecuencia, se relanza otra excepción, la información de la anterior se pierde. Salvo que se pase dicha información como descripción a la nueva excepción.

Errores y excepciones comunes

Dado que en la última versión del examen de Java se agregaron a varias preguntas las opciones de "error de compilación", "error de ejecución", y "se lanza una excepción", es imperativo conocer cómo funcionan estas, y saber determinar cuándo serán lanzadas.

De donde provienen las excepciones

Es importante conocer que causa los errores y excepciones, y de donde provienen. Para propósitos del examen, dividimos los orígenes en dos categorías:

- Excepciones de la JVM — Aquellas que son lanzadas por la JVM.
- Excepciones de programación — Aquellas que son lanzadas explícitamente mediante un `throw`.

Veamos una tabla que resume las excepciones que necesitamos conocer para el examen:

Excepción	Descripción	Lanzado por
ArrayIndexOutOfBoundsException	Lanzada cuando se intenta acceder a un elemento del array con un índice negativo o mayor a la cantidad de elementos contenidos (length – 1).	JVM
ClassCastException	Lanzado cuando se intenta castear una clase que falla a la verificación Is-A.	JVM
IllegalArgumentException	Lanzada cuando un método recibe un argumento formateado de forma diferente a la que lo espera.	Programáticamente
IllegalStateException	Lanzada cuando el estado del ambiente no coincide con la operación solicitada.	Programáticamente
NullPointerException	Lanzado cuando se intenta acceder a un objeto a través de una referencia que contiene un valor nulo (null).	JVM
NumberFormatException	Lanzada cuando un método que convierte un `String` en un número recibe una cadena que no puede ser formateada como tal.	Programáticamente
AssertionError	Lanzada cuando una sentencia de verificación booleana detecta un `false`.	Programáticamente
ExceptionInInitializerError	Lanzado cuando intentas inicializar una variable estática o un bloque de inicialización.	JVM
StackOverflowError	Lanzada generalmente cuando un método se llama recursivamente muchas veces (cada invocación se agrega al stack).	JVM
NoClassDefFoundError	Lanzada cuando la JVM no puede encontrar una clase que necesita, debido a un error en la línea de comandos, un problema con los classpaths, o un .class que falta.	JVM

Afirmaciones

Las afirmaciones (assertions), son una herramienta para poder verificar aquellas partes del código que creemos no fallarán nunca en nuestro código, pero que si las tuviéramos que verificar, tendríamos que empezar a generar más excepciones.

Un vistazo a las afirmaciones

Supongamos que tienes un código como el siguiente:

```
private float hazAlgo(int valor1, int valor2) {
  float resultado = 0.0f;

  if ( valor 2 != 0 ) {
    resultado = (float)(valor1 / valor2);
  }  else {
    //Nunca debería de suceder este caso
    System.out.println("Error crítico. Tenemos una división por 0!!!! Ahhhhhh!!!!");
    //El mensaje lo exageré un poco para acentuar el problema de mostrar este mensaje
al usuario
  }

  return resultado;
}
```

Si no quisieras realizar excepciones para verificar esta condición, y no deseas hacer un **if**/**else**, ya que si no se da la condición, en realidad el programa "it´s toast".
Las afirmaciones nos permiten realizar esta verificación de manera que la verificación que realicemos tiene dos posibilidades:

- Si es **true** El código continúa ejecutándose con naturalidad
- Si es **false** Se detiene la ejecución del programa y se lanza una **AssertionException**.

Jamás debes atrapar esta excepción. Debes esperar a que tú programa literalmente "explote" y rastrear el error utilizando como ayuda el stack de la excepción.

Otro punto interesante de las afirmaciones, es que estas solo se ejecutan cuando las mismas se encuentran habilitadas, caso contrario es como si no existieran. Esto es muy práctico cuando se trabaja en el ambiente de desarrollo, pero no querrás que el programa que publicaste en el ambiente de producción se detenga totalmente por haberse lanzado una afirmación.
De todas formas, las afirmaciones siguen existiendo por más que estén deshabilitadas, es solo que la JVM las ignora. Esto facilita que si tenemos algún error en producción que era verificado por las afirmaciones, estas pueden ser habilitadas para detectar el problema más rápido.

Reglas de sintaxis de las afirmaciones

Una afirmación tiene dos tipos de sintaxis, la simple y la detallada.

Afirmación simple

```
private float hazAlgo(int valor1, int valor2) {
  assert (valor2 != 0);

  return (float)(valor1 / valor2);
}
```

Verifica una condición. Si esta se cumple la ejecución continúa, caso contrario, lanza la excepción **AssertionExcepcion**.

Afirmación detallada

```
private float hazAlgo(int valor1, int valor2) {
  assert (valor2 != 0) : "valor1:" + valor1 + " - valor2:" + valor2;

  return (float)(valor1 / valor2);
}
```

Se comporta de la misma manera que la afirmación simple, con la diferencia de que añade una expresión. La función del segundo parámetro es simplemente la de añadirse al stack para brindar un detalle del error ocurrido.

La expresión debe de resultar en un valor, no importa si se utiliza un enumerador, se llama a un método, o se pasa un String.

El prototipo de la sintaxis sería el siguiente:

```
assert (condicion) : expresion;
```

Condición: debe resultar siempre en un valor de tipo **boolean**.

Expresión: puede resultar en un valor de cualquier tipo, pero debe de especificarse un valor.

Habilitando las afirmaciones

Es importante saber cómo ejecutar y cómo compilar código con afirmaciones.

Un problema que nos podemos llegar a encontrar es que en versiones anteriores de Java, **assert** se podía utilizar como identificador o como palabra reservada. A partir de Java 6, **assert** es una palabra reservada, cualquier intento de utilizarla como variable derivará en un error de compilación.

Por defecto las afirmaciones se encuentran desactivadas, pero podemos habilitarlas en tiempo de ejecución.

Al invocar nuestro programa desde la consola de comandos, le indicamos si se encuentran activadas o no.

Ejemplo de llamada con afirmaciones habilitadas:

java -ea MiClase
java -enableassertions MiClase

Ejemplo de llamada con afirmaciones deshabilitadas:

java -da MiClase
java -disableassertions MiClase

Utilizando la sintaxis para afirmaciones deshabilitadas obtenemos el mismo comportamiento que la invocación por defecto, pero, cambia cuando lo utilizamos de manera selectiva.

Habilitando afirmaciones de manera selectiva

Podemos aplicar la habilitación o des habilitación sobre una clase o **package** (y sus subpaquetes) específico.

Ejemplo de llamada con afirmaciones selectivas habilitadas:

java -ea:MiOtraClase MiClase
java -ea:com.utils... MiClase

En este ejemplo, solo habilitamos las afirmaciones para la clase MiOtraClase, y en la segunda llamada, para el paquete com.utils y todos sus subpaquetes.

Ejemplo de llamada con afirmaciones selectivas deshabilitadas:

java -da:MiOtraClase MiClase

Como mencionamos anteriormente, tampoco es de mucha ayuda este código, dado que por defecto, las afirmaciones se encuentran deshabilitadas.

¿Cuál es el secreto entonces?

Se pueden combinar un comando general y uno selectivo, siempre que sean opuestos (uno habilita y el otro deshabilita).

Ejemplos de combinaciones:

java -da -ea:MiOtraClase MiClase
java -ea -da:com.utils... MiClase

En el primero, se encuentran deshabilitadas, salvo en la clase MiOtraClase.

En el segundo, se encuentran habilitadas, salvo dentro del paquete com.utils y sus subpaquetes.

Utilizar las afirmaciones apropiadamente

- **No utilices** nunca las afirmaciones para validar los parámetros de un método **public**, si deseas validar la entrada de un parámetro, puedes utilizar la excepción **IllegalArgumentException**.
- **Utiliza** las afirmaciones para verificar los parámetros de un método privado. Cuando asumes que los parámetros que le envías son correctos, verifica esto con una afirmación.

Lo que se viene

En la próxima entrega estaremos adentrándonos en la API de Java para Entrada/Salida.

Veremos como escribir en archivos de texto, y como obtener la información almacenada.

Aprenderemos como funcionan a bajo nivel los Strings, tanto los literales como la instanciación mediante new.

Descubriremos la serialización de objetos.

Y aprenderemos a manejar fechas en Java.

- **No utilices** nunca las afirmaciones para validar los argumentos de entrada de la línea de comandos. En todo caso, utiliza una excepción.
- **Utiliza** las afirmaciones para verificar código que nunca debería de ejecutarse.
- **No utilices** nunca las afirmaciones cuando estas puedan causar efectos secundarios. El programa no debe de modificarse porque una afirmación se encuentra habilitada.

```
private x;
private void verificaAsertion (int valor) {
      assert(validacionAsertion());
      //Se llama a un método que modifica un
atributo de la clase
}

private boolean validacionAsertion() {
      x++;
      return true;
}
```

¿Tenés lo que se necesita para ser un SCJP?

Autores
Gustavo Alberola
Matias Álvarez

Diseño
Leonardo Blanco