# SCJP – Capitulo 5

Control de Flujo, Excepciones y Aserciones

# Agenda

- if – switch
- Iteraciones
- Excepciones
- Aserciones

# Sentencias de Decisión (if)

- Definición Básica

```
if (expresionBooleana)
   // sentencia si expresión es verdadera
else
   // sentencia si expresión es falsa
```

Nota: El *else* es opcional

# Sentencias de Decisión (if)

- Definición Básica (múltiples líneas)

```
if (expresionBooleana) {
    // Muchas sentencias
    // si expresión es verdadera
} else {
    // Muchas sentencias
    // si expresión es falsa
}
```

Nota: El *else* es opcional (también)

# Sentencias de Decisión (if)

- Ojo con la identanción engañosa

```
if (x > 3)
   y = 2;
   z += 8;
a = x + z;
System.out.println("z = " + z);
```

En este caso, la sentencia *if* no tiene definido un bloque de llaves *{ …}*

Por lo tanto la instrucción "z += 8" no es parte del *if* y se ejecutará siempre…

# Sentencias Anidadas

```
if (exam.terminado())
    if (exam.getPuntaje() < 0.61)
        System.out.println("Otra vez…");
else
    System.out.println("Maestro Java!!!");
```

El texto "Maestro Java!!!" de verdad se obtiene si el puntaje es mayor o igual a 61% (0.61)

# Expresiones Válidas

```
int y = 5;
int x = 2;
if (((x > 3) && (y < 2)) | hacerAlgo()) {
   System.out.println(“true”);
}
```

Asumiendo que hacerAlgo() retorna true,
La expresión siempre será verdadera

# Expresiones Válidas

```
int y = 5;
int x = 2;
if ((x > 3) && (y < 2) | hacerAlgo()) {
   System.out.println("true");
}
```

Al no haber paréntesis, el operador && (and) tiene prioridad sobre | (or)

# **if** anidados

```
int x = 3;
if (x == 1) {
    System.out.println(“x es 1”);
} else {
    if (x == 2) {
        System.out.println(“x es 2”);
    } else {
        if (x == 3) {
            System.out.println(“x es 3”);
        } else {
            System.out.println(“x es ????”);
        }
    }
}
```

¿Cual sería una alternativa para *if* anidados?

# if anidados

```
int x = 3;
if (x == 1) {
    System.out.println(“x es 1”);
} else if (x == 2) {
    System.out.println(“x es 2”);
} else if (x == 3) {
    System.out.println(“x es 3”);
} else {
    System.out.println(“x es ????”);
}
```

Ordenarlos…

Pero aún la lectura es algo compleja…

# Switch

```
int x = 3;
switch (x) {
    case 1 :
        System.out.println(“x es 1”);
        break;
    case 2 :
        System.out.println(“x es 2”);
        break;
    case 3 :
        System.out.println(“x es 3”);
        break;
    default  : //opcional
        System.out.println(“x es ????”);
}
```

Nótese el uso de ***break***;

Proporciona la salida de la sentencia ***switch***

El ***default*** es para cuando ***x*** no es igual a ningún valor de los ***case***

# Switch

```
final int uno = 1;
int x = 1;
switch (x) {
    case uno :
        System.out.println("x es 1");
        break;
    default  :
        System.out.println("x es ????");
        break;
    case uno + 1 :
        System.out.println("x es 2");
}
```

Sólo números enteros byte, short, int y char

Si x fuera byte, el valor de los case no pueden ser mayores a 127

Sólo expresiones constantes e inicializadas

Opcionales

También se pueden usar bloques

```
case uno + 2 : {
  System.out.println("")
  break;
}
```

# Swtich sin break

```
final int uno = 1;
int x = 1;
switch (x) {
    case uno     : System.out.println("x es 1");
    case uno+1   : System.out.println("x es 2");
    case uno+2   : System.out.println("x es 3");
}
System.out.println("fuera del switch");
```

Resultado:

```
x es 1
x es 2
x es 3
fuera del switch
```

# Swtich sin break

```
int x = algunValorEntreUnoYDiez();
switch (x) {
    case 2 :
    case 4 :
    case 6 :
    case 8 :
    case 10 : {
        System.out.println("x es par…");
    }
}
```

Vasta que ***x*** tome el valor de : 2, 4, 6, 8 ó 10

Y entra al ***swtich*** y luego recorre secuencialmente hasta encontrar y ***break*** o el fin del ***swtich***

# Swtich y default

```
switch (x) {
   case 2 : System.out.println(“2”);
   default : System.out.println(“default”);
   case 3 : System.out.println(“3”);
   case 4 : System.out.println(“4”);
}
```

Si x = 2

2
default
3
4

Si x = 7

default
3
4

# Iteraciones - while

```
int x = 2
while (x < 10) {
    System.out.println(“dentro del loop”);
    x = 10;
}
System.out.println(“fuera del loop”);
```

Resultado:

dentro del loop
fuera del loop

# Iteraciones – do while

```
do {
   System.out.println(“dentro del loop”);
} while (false);
```

Siempre se ejecuta por lo menos una vez

Ojo con el punto y coma al final del loop

# Iteraciones - for

Opcional
Ocurre antes que cualquier cosa
Se puede declara varias variables (del mismo tipo)
No se pueden acceder las variables fuera del *for*

Opcional
Ocurre justo después de la inicialización
Se debe evaluar como expresión boolean
Puede ser una expresión muy compleja

```
for (/*inicializacion*/ ; /*condicion*/ ; /*iteracion*/){
    /*dentro del loop*/
}
```

Opcional
Ocurre después de la ejecución del cuerpo del *loop*
Se pueden colocar múltiples sentencias separadas por coma
No se pueden acceder las variables

# Iteraciones - for

```
for ( ; ; ) {
    /* dentro de un loop sin fin */
}
```

| Código en loop | Que pasa !!!!!! |
|---|---|
| break | Salta a la primera sentencia después del *for* |
| return<br>throw new Exception() | Salta in mediatamente al método que lo llamo |
| System.out.exit() | Finaliza toda la VM |

# Iteraciones - for

```
int z = 5;
for (int x = 0, y = 10; x < 10 | y > 0; x++, --y , z+=2){
    System.out.println(x);
    System.out.println(y);
    System.out.println(z);
    break;
}
```

Resultado:

```
0
10
5
```

# iteraciones con ***continue***

- continue: provoca que la iteración actual finalice (pero sale del bucle) y se vuelve a evaluar la condición

```
while (!finDeArchivo){
    Datos d = leerLinea();
    if (lineaEnBlanco) continue;
    procesarLinea(d);
}
```

# Iteraciones y etiquetas

```
fuera:
   for (int x = 0; x < 10; x++){
        for (int z = 0; z < 5; z++){
                System.out.println("Hola");
                break fuera;
        } // fin for interno
        System.out.println("fuera");
   }
System.out.println(“Adios...");
```

Salida:

Hola
Adios…

# Iteraciones y etiquetas

```
fuera:
   for (int x = 0; x < 10; x++){
        for (int z = 0; z < 5; z++){
                System.out.println("Hola");
                continue fuera;
         } // fin for interno
         System.out.println("fuera");
   }
System.out.println(“Adios...");
```

Salida:

Hola
Hola
Hola
Hola
Hola
Hola
Hola
Hola
Hola
Hola
Adios…

# Excepciones - Capturar

```
try {
    /* código potencialmente peligroso */
} catch (PrimeraExcepcion ex) {
    /* código para manejar la esta excepcion */
} catch (SegundaExcepcion ex){
    /* código para manejar la segunda excepcion */
} finally {
    /* código que siempre se ejecuta,
    con o sin excepcion (es opcional) */
}
```

# Excepciones - Capturar

```
try {
   getArchivoDesdeLaRed();
   leerArchivoYPoblarTabla();
} catch (NoSePuedeObtenerArchivo ex) {
   usarArchivoLocal();
}

// nótese que no se usa el finally
```

# Excepciones - Capturar

```
try {
    /* procesar archivo */
} finally {
    /* siempre cerrar archivo */
}
```

Nótese que no se usa el *catch* esta correcto, pero es útil????

(si, siempre y cuando dejemos propagar la excepción)

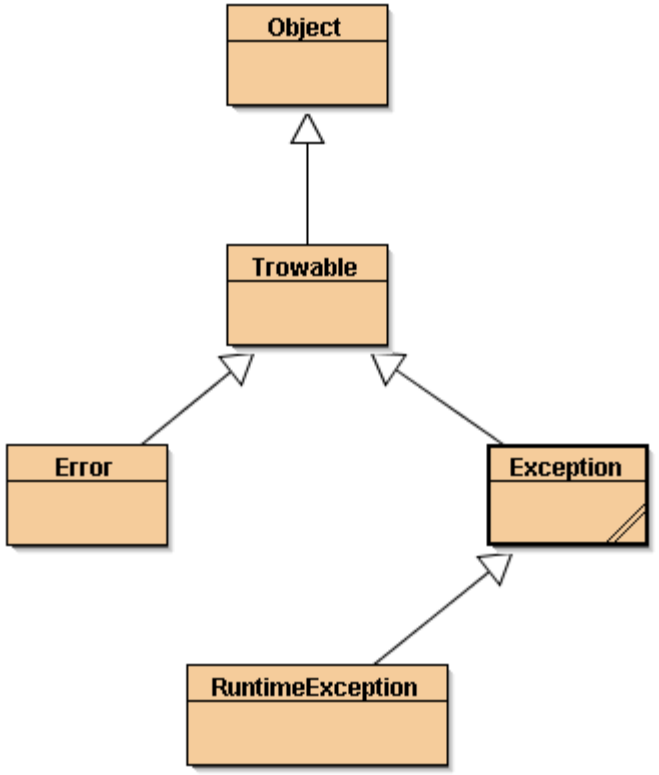
# Excepciones - propagación

```
class TestEx {
    public static void main (String[ ] args){
        hacerAlgo();
    }
    public static hacerAlgo(){
        hacerMas();
    }
    public static hacerMas(){
        int x = 5 / 0; // no se puede dividir por cero
    }
}
```

```
Exception in thread “main” java.lang.AritmeticExcepcion : / by Zero
at TestEx.hacerMas(TestEx.java:10)
at TestEx.hacerAlgo(TestEx.java:7)
at TestEx.main(TestEx.java:3)
```

# Excepciones – Jerarquía

- Error : es cuando ocurre un problema prácticamente irrecuperable
- Exception: los subtipos de Exception son excepciones chequeadas (se deben controlar)
- RuntimeException : este subtipo de Exception es especia, los subtipos de ReuntimeException no son excepciones no chequeas



# Excepciones - propagación

```
class MiException extends Exception{ }
class TestEx {
      public static void main (String[ ] args){
          try {
              hacerAlgo();
          catch (MiException ex) {
                ex.printStackTrace()
          }
      }
      public static hacerAlgo() throws MiException {
                hacerMas();
      }
      public static hacerMas() throws MiException {
                throw new MiException();
      }
}
```

Si la Excepción no es Capturada, debe ser anunciada

# Excepciones - propagación

```
class TestEx {
    public static void main (String[ ] args){
        try {
            hacerAlgo();
        } catch (Error ex) {
            ex.printStackTrace()
        }
    }
    public static hacerAlgo() {
            hacerMas();
    }
    public static hacerMas() {
            throw new Error();
    }
}
```

Si la Excepción es no chequeda no es necesario anunciarla o capturarla (esta correcto si se hace)

# Excepciones – Jerarquía

```
class GeneralException extends Exception { }
class ParticularException extends GeneralException { }
class Test {
    void metodo(){
        try{
            throw new ParticularException();
        } cacth (GeneralException ex){
            /* control de la exception */
        } cacth (ParticularException ex) {
            /* control de la exception */
        }
    }
}
```

Al colocar GeneralException en primer lugar, ésta capturará todas las instancias de GeneralException y ParticularException…

… y nunca se ejecutará el *cacth* de ParticularException

# Aserciones – depuración

```
// versión del a “vieja escuela”
private void metodo(int num){
      if (num >= 0){
            calculos(num + x);
      } else {
            System.out.println (“Pánico: num no puede ser negativo ”);
      }
}
//versión con aserciones (para ambiente de testing)
private void metodo(int num){
      assert (num >= 0); //lanza un AssertException si no es verdad
      calculos(num + x);
}
//versión sin aserciones (para ambiente de producción)
private void metodo(int num){
      calculos(num + x);
}
```

# Aserciones – Definición

Versión muy simple

```
assert (false);
```

Versión simple

```
assert (x < z) : “x es ” + x + “ y es ” + y;
```

Nota: después de los dos puntos puede ir cualquier expresión que se pueda transformar a String (incluso llamada a métodos o creación de instancias)

# Aserciones – Activar / Desactivar

| Command - Line | Que sucede ! |
|---|---|
| java –ea<br>java –enableassertions | habilita las aserciones |
| java –da<br>java –disableassertions | deshabilita las aserciones (esto es por defecto) |
| java –ea:cl.duoc.Profe | habilita aserciones en clase Profe del paquete cl.duoc |
| java –ea:cl.duoc… | habilita aserciones en el paquete cl.duoc y en todos los sub-paquetes |
| java –ea –dsa | habilita las aserciones en general, pero deshabilita las aserciones de las clases del sistema |
| java –ea –da:cl.duoc… | habilita las aserciones en general, pero deshabilita las aserciones del paquete cl.duoc y todos los sub-paquetes |

# Aserciones – Buenas Prácticas

- NO usar para validar argumentos de métodos públicos
- Usar para validar argumentos en métodos privados
- NO usar para validar parámetros de la línea de comando
- Usar (incluso en métodos públicos) para validar condiciones que **nunca, pero nunca deben ocurrir**

# Aserciones – Buenas Prácticas

- NO usar Aserciones que causen efectos colaterales.
  - La regla es : *Una expresión de assert debe dejar el programa en el mismo estado que tenia antes de ejecutar dicha expresión*

```
public void hacerAlgo(){
    assert(modificar());
    // mas codigo…
}
public boolean modificar(){
    y = x++;
    return true;
}
```