Programación 2

Introducción al Lenguaje Java
Programación Estructurada

Profesores:
Ismael Figueroa - ifigueroap@gmail.com
Eduardo Godoy - eduardo.gl@gmail.com

Programación Estructurada

La Programación Estructurada es un paradigma de programación que busca mejorar la claridad, calidad, y mantenibilidad de los programas, mediante el uso exclusivo de *estructuras de control*

Las estructuras de control permiten manipular el *flujo de ejecución del programa*, fomentando la reutilización de código y minimizando el código duplicado.

Estructuras de Control

- **Estructuras Secuenciales**: en Java por defecto los programas son secuenciales
- **Estructuras de Selección**: permiten ejecutar segmentos de código en base al cumplimiento (o no) de condiciones que dependen del estado del programa
- **Estructuras de Iteración**: permiten ejecutar repetidamente un segmento de código, en base a condiciones lógicas, o recorriendo los elementos de una colección
- **Estructuras de Salto**: permiten romper de—forma controlada—con el flujo dentro de otras estructuras, como las de iteración
- **Excepciones**: permiten declarar un flujo alternativo que se ejecuta solo cuando el programa encuentra un error

Estructura `if`

- Permite ejecutar condicionalmente un bloque de código
- Un `if` sencillo solo tiene un bloque que se ejecuta si la condición es `true`
- Un `if` compuesto tiene una cláusula `else` con un bloque que se ejecuta solo si la condición es `false`
- Pueden anidarse arbitrariamente varios `if` en las cláusulas `else`. Esto se conoce como `else-if`

Estructura if

```
void compararN(int n) {
  // if sencillo
  if(n < 45) {
    System.out.println("n: " + n + " es menor que 45");
  }

  // if con alternativa
  if(n < 3) {
    System.out.println("n: " + n + " es menor que 3");
  } else {
    System.out.println("n: " + n + " NO es menor que 3");
  }

  // if anidados
  if(n < 3) {
    System.out.println("n: " + n + " es menor que 3");
  } else if (n < 5) {
    System.out.println("n: " + n + " es menor que 5");
  } else {
    System.out.println("n: " + n + " es mayor que 3 y que 5");
  }
}
```

Otro Ejemplo de `if`

Se desea escribir el método `clasificar` que retorna una letra según el valor numérico del parámetro `valor`, con el siguiente criterio:

- `A`, para un valor entre 100 y 91 (inclusive).
- `B`, para un valor entre 90 y 81 (inclusive).
- `C`, para un valor entre 80 y 71 (inclusive).
- `F`, si no es ninguno de los anteriores.

Método clasificar

```
char clasificar(int valor) {
  char clase;

  if(valor >= 91 && valor <= 100) {
    clase = 'A';
  } else if (valor >= 81 && valor <= 90) {
    clase = 'B';
  } else if (valor >= 71 && valor <= 80) {
    clase = 'C';
  } else {
    clase = 'F';
  }

  return clase;
}
```

Estructura `switch`

Cuando se tienen muchos casos distintos el uso de `if-else-if` puede volverse difícil de leer. Java provee la estructura `switch`, diseñada para manejar muchos casos posibles:

- Permite seleccionar en base a distintos *casos*, donde cada caso es un valor constante
- Cada caso debe ser único, y el valor asociado debe ser del mismo tipo que el valor analizado por `switch`
- `switch` tiene un comportamiento en cascada, que debe romperse usando la instrucción `break`

Ejemplo de `switch`

Ahora se desea escribir el método `desclasificar` que toma como parámetro una calificación en forma de letra, y que retorna el valor mínimo que generó esa clasificación. Si la letra es 'F', debe retornar 0.

Método desclasificar

```
int desclasificar(char clase) {
  int valor;
  switch(valor) {
    case 'A': valor = 91;
    case 'B': valor = 81;
    case 'C': valor = 71;
    case 'F': valor = 0;
  }

  return valor;
}
```

Método `desclasificar`

¿Qué se imprime por pantalla en el siguiente programa?

```
System.out.println(desclasificar('A'));
```

Método `desclasificar`

¿Qué se imprime por pantalla en el siguiente programa?

```
System.out.println(desclasificar('A'));
```

Sorprendentemente imprime 0! Esto se debe al comportamiento en cascada de `switch`

Método `desclasificar` correcto

```
int desclasificar(char clase) {
  int valor;
  switch(valor) {
    case 'A': valor = 91; break;
    case 'B': valor = 81; break;
    case 'C': valor = 71; break;
    case 'F': valor = 0;  break;
  }

  return valor;
}
```

Método `desclasificar`

¿Qué se imprime por pantalla en el siguiente programa?

```
System.out.println(desclasificar('Z'));
```

Método `desclasificar`

¿Qué se imprime por pantalla en el siguiente programa?

```
System.out.println(desclasificar('Z'));
```

Esto es problemático pues se retorna una variable no inicializada

Método `desclasificar` más correcto

La cláusula `default` permite manejar los valores que no caen en ninguno de los casos del `switch`

```
int desclasificar(char clase) {
  int valor;
  switch(valor) {
    case 'A': valor = 91; break;
    case 'B': valor = 81; break;
    case 'C': valor = 71; break;
    case 'F': valor = 0;  break;
    default : valor = 0;
  }

  return valor;
}
```

Estructuras de Iteración

Java provee tres estructuras de iteración:

- `while`
- `do-while`
- `for`

Si bien las 3 son equivalentes, pueden hacer lo mismo, dependiendo del problema algunas pueden ser más cómodas de usar que otras.

Ejemplo para iteración

Se desea escribir el método `sumaIntervalo`, que dados los valores `n` y `m` de tipo `int`, y asumiendo que siempre `n <= m`, se quiere calcular la suma de todos los enteros comprendidos entre `n` y `m`, ambos inclusive.

Por ejemplo, `sumaIntervalo(3,5)` debe retornar `12`

Estructura while

- Repite el bloque mientras la condición lógica, un booleano, sea verdadera
- Si la condición es falsa la primera vez, nunca se ejecuta el bloque
- Dentro del bloque, debemos eventualmente hacer cambios para que la condición se vuelva falsa

```
int sumaIntervalo(int n, int m) {
  int suma = 0;
  int i = n;

  while(i <= m) {
    suma += i;
    i = i + 1;
  }

  return suma;
}
```

Estructura do-while

- Repite el bloque mientras la condición lógica, un booleano, sea verdadera
- El bloque siempre se evalúa la menos una vez, porque se revisa la condición luego de ejecutar el bloque
- También debemos preocuparnos de hacer falsa la condición para salir de la iteración

```
int sumaIntervalo(int n, int m) {
  int suma = 0;
  int i = n;

  do {
    suma += i;
    i = i + 1;
  } while (i <= m);

  return suma;
}
```

Estructura `for`

- Combina los pasos de inicialización de variables de iteración, condición lógica, y actualización de las variables de iteración
- Es una forma muy compacta de escribir iteraciones, aunque es equivalente en su poder a `while` y `do-while`

```
int sumaIntervalo(int n, int m) {
  int suma = 0;

  for(int i = n; i <= m; i = i + 1) {
    suma += i;
  }

  return suma;
}
```

Comportamiento de `for`

- La **inicialización** es una sentencia que se ejecuta *solo una vez*, justo antes de evaluar la condición de término
- La **condición de término** es una expresión booleana, que determina si se ejecuta el bloque o no. Se ejecuta justo antes de ejecutar el bloque. El bloque del `for` se ejecuta solo cuando la condición de término es verdadera.
- La **actualización de variables de iteración** se ejecuta justo después de la ejecución del bloque del `for`, si es que fue ejecutado. Se debe usar para, eventualmente, hacer falsa la condición de término es una expresión invocada en cada iteración del bucle.
- Algunas de estos elementos pueden ser expresiones vacías, pero el `for` debe tener como mínimo la forma `for( ; ; ) ...`

Preguntas

Preguntas ?