Transición a Java

Conceptos generales:Introducción, conceptos generales y entorno Java

Java fue hecho con el objetivo de generar código de tamaño muy reducido y ser una herramienta independiente del CPU. Para eso el código se ejecuta sobre una máquina virtual llamada Java Virtual Machine, donde se interpreta el código neutro y se convierte en el código que entiende la CPU, independientemente cual sea.

La compañía Sun (creadora de Java) describe el lenguaje como “simple, orientado a objetos, distribuido, interpretado, robusto, seguro, de arquitectura neutra, portable, de altas prestaciones, multitarea y dinámico”.

Tipos de ejecuciones de un programa en Java:

- Ejecución como aplicación independiente (Stand-alone Application)
- Applet: se ejecuta en un navegador o browser. Se descarga desde el servidor y no se tiene que instalar en la computadora que tenga el browser.
- Servlet es una aplicación sin interface gráfica que se ejecuta en un servidor de Internet

El JDK (Java Development Kit) es un conjunto de programas y librerías para desarrollar, compilar y ejecutar programas Java. Incluye también un debugger por consola. Los IDE (Integrated Development Environment) son entornos de desarrollo integrados, donde en un mismo programa se puede escribir código Java, compilarlo y ejecutarlo. Los entornos integrados permiten desarrollar de manera más rápida.

El compilador de Java (java**c**.exe) incluido en el JDK analiza la sintaxis del código escrito en archivos de java (.java) y si no encuentra errores genera archivos compilados (.class).

JVM es el intérprete de Java, ejecuta los archivos compilados (creados por el compilador). También tiene la opción de usar Just In Time Compiler.

Programación en Java

Variables

Una **variable** es un nombre que tiene un valor que puede cambiar a lo largo de un programa. Los **tipos de variables** en Java son:

- **Variables de Tipos Primitivos**: Definidos con un unico valor
- **Variables Referencia**: Son referencias o nombres de una informacion mas compleja como arays u objetos.

Segun su papel en el programa, la variables pueden ser:

- **Miembro de una clase**: Son definidas en una clase, fuera de un método.

- **Locales**: Se definen en un método o se crean dentro de un bloque y se destruyen al final del bloque.

Los **tipos primitivos de variables** son las variables que pertenecen a los tipos de información habituales: valores boolean, caracteres y valores numéricos enteros o de punto flotante.

Tipo de variable	Descripción
Boolean	1 byte. Valores true y false
Char	2 bytes. Unicode. Comprende el código ASCII
Byte	1 byte. Valor entero entre -128 y 127
Short	2 bytes. Valor entero entre -32768 y 32767
Int	4 bytes. Valor entero entre -2.147.483.648 y 2.147.483.647
Long	8 bytes. Valor entre -9.223.372.036.854.775.808 y 9.223.372.036.854.775.807
Float	4 bytes (entre 6 y 7 cifras decimales equivalentes). De -3.402823E38 a -1.401298E-45 y de 1.401298E-45 a 3.402823E38
Double	8 bytes (unas 15 cifras decimales equivalentes). De -1.79769313486232E308 a -4.94065645841247E-324 y de 4.94065645841247E-324 a 1.79769313486232E308

Tabla 2.1. Tipos primitivos de variables en Java.

Un **boolean** NO es un valor numerico, solo admite true o false. No se identifica con el igual o distinto de cero.
Un **char** tiene caracteres en código UNICODE (que incluye el código ASCII). Ocupan 16 bits por caracter.
Los **byte, short, int y long** son números enteros que pueden ser positivos o negativos. Varian en los valores maximos y minimos.
Los **float y double** son números reales (de punto flotante). Un float tiene 6-7 cifras decimales y un double tiene 15.
Se usa **void** para la ausencia de un tipo de variable.

Para **definir una variable**, se define el tipo y el nombre. Las variables pueden ser de tipos primitivos o referencias a objetos de alguna clase. Las variables primitivas se inicializan en cero si no se especifica su valor inicial. Un boolean se inicializa en false y un char en '\0' en ese caso. Las variables de referencia si inicializan en null.

Una **referencia** es una variable que indica el lugar de un objeto en memoria. Declarar una referencia no significa que ya se esté apuntando a un objeto en particular salvo que se esté creando un objeto nuevo explícitamente en la declaración (usando el operador new). El operador new reserva en memoria el espacio para el objeto. También se puede igualar la referencia a un objeto existente.

Los arrays son un tipo particular de referencias, tambien llamados vectores. Pueden ser de variables primitivas (vector de enteros) o de objetos.

```
int [] vector;         // Declaración de un array. Se inicializa a null
vector = new int[10]; // Vector de 10 enteros, inicializados a 0
double [] v = {1.0, 2.65, 3.1};// Declaración e inicialización de un vector de 3
                               // elementos con los valores entre llaves
MyClass [] lista=new MyClass[5];// Se crea un vector de 5 referencias a objetos
                               // Las 5 referencias son inicializadas a null
```

El **scope** de una variable es la parte de la aplicación donde la variable es accesible y puede ser utilizada. En Java todas las variables tienen que estar incluidas en una clase. Las variables declaradas adentro de bloques solo son accesibles dentro del bloque.

Las variables declaradas como **public** en un objeto se pueden acceder a través de una referencia a ese objeto usando el operador punto (.). Las variables declaradas como **private** no son accesibles directamente desde otras clases.

Las funciones que pertenecen a una clase tienen acceso a todas las variables de esa clase. Pero las funciones que pertenecen a una clase B derivada de una clase A, solo tienen acceso a las variables de A que fueron declaradas como public o protected.

Si declaramos una variable dentro de un bloque que tiene el mismo nombre de una variable miembro, la variable del bloque oculta a la variable del miembro dentro de ese bloque. Para acceder a la variable miembro dentro del bloque, se usa el operador this.

La eliminación de los objetos la hace el **garbage collector**. Este automáticamente libera la memoria ocupada por un objeto cuando no existe ninguna referencia apuntando a ese objeto. Aunque una variable de tipo referencia deje de existir, el objeto al cual apunta no se elimina si hay otras referencias apuntando a ese mismo objeto.

Los objetos de tipo **BigInteger** son capaces de almacenar cualquier número entero (cualquier número de cifras) sin perder información durante las operaciones. Los objetos de tipo **BigDecimal** permiten trabajar con el número de decimales deseado. BigInteger y BigDecimal son clases incorporadas en Java 1.1.

Operadores

Operadores Aritméticos: Son operadores binarios (requieren dos operandos) y hacen operaciones aritméticas: suma (+), resta (-), multiplicación (*), división (/) y resto de la división (%).

Operadores de Asignación: Asigna un valor a una variable. El operador de asignación es el operador igual (=).

Operador	Utilización	Expresión equivalente
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2

Tabla 2.2. Otros operadores de asignación.

Operadores Unarios: Mantienen o cambian el signo de una variable, constante o expresión numérica. Se usan los operadores + y -.

Operador *instanceof*: Permite saber si un objeto pertenece a una clase o no. Devuelve true o false.

Operador Condicional *?:* Es un operador ternario, que permite hacer bifurcaciones condicionales.

Operadores Incrementales: El operador ++ incrementa en una unidad la variable a la que se le aplica. El operador - - reduce una unidad. Si el operador precede a la variable (++i), primero se incrementa la variable y después se usa. Si el operador está después de la variable (i++), se usa la variable en su valor anterior y luego se incrementa.

Operadores relacionales: Se usan para comparaciones de igualdad, desigualdad y relación de menor o mayor.

Operador	Utilización	El resultado es true
>	op1 > op2	si op1 es mayor que op2
>=	op1 >= op2	si op1 es mayor o igual que op2
<	op1 < op2	si op1 es menor que op2
<=	op1 <= op2	si op1 es menor o igual que op2
==	op1 == op2	si op1 y op2 son iguales
!=	op1 != op2	si op1 y op2 son diferentes

Tabla 2.3. Operadores relacionales.

Operadores lógicos: Se usan para hacer expresiones lógicas

Operador	Nombre	Utilización	Resultado
&&	AND	op1 && op2	true si op1 y op2 son true. Si op1 es false ya no se evalúa op2
||	OR	op1 || op2	true si op1 u op2 son true. Si op1 es true ya no se evalúa op2
!	negación	! op	true si op es false y false si op es true
&	AND	op1 & op2	true si op1 y op2 son true. Siempre se evalúa op2
|	OR	op1 | op2	true si op1 u op2 son true. Siempre se evalúa op2

Tabla 2.4. Operadores lógicos.

Operador de concatenación de cadenas de caracteres: Se usa el operador + para concatenar cadenas de caracteres.

Operadores que actúan a nivel de bits: Se usan para definir **flags**, es decir, variables de tipo entero en la que cada uno de sus bits indican si una opción está activada o no.

Operador	Utilización	Resultado
>>	op1 >> op2	Desplaza los bits de op1 a la derecha una distancia op2
<<	op1 << op2	Desplaza los bits de op1 a la izquierda una distancia op2
>>>	op1 >>> op2	Desplaza los bits de op1 a la derecha una distancia op2 (positiva)
&	op1 & op2	Operador AND a nivel de bits
|	op1 | op2	Operador OR a nivel de bits
^	op1 ^ op2	Operador XOR a nivel de bits
~	~op2	Operador complemento

Tabla 2.5. Operadores a nivel de bits.

Para construir una variable flag que sea 00010010 bastaría hacer flags=2+16 (00000010 + 00010000). Para saber si el segundo bit por la derecha está o no activado:

```
if (flags & 2 == 2) {...}
```

Operador	Utilización	Equivalente a
&=	op1 &= op2	op1 = op1 & op2
|=	op1 |= op2	op1 = op1 | op2
^=	op1 ^= op2	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

Tabla 2.6. Operadores de asignación a nivel de bits.

Precedencia de operadores: Es el orden en que se ejecutan los distintos operadores en un sentencia. De mayor a menor precedencia:

postfix operators	`[] . (params) expr++ expr--`
unary operators	`++expr --expr +expr -expr ~ !`
creation or cast	`new (type)expr`
multiplicative	`* / %`
additive	`+ -`
shift	`<< >> >>>`
relational	`< > <= >= instanceof`
equality	`== !=`
bitwise AND	`&`
bitwise exclusive OR	`^`
bitwise inclusive OR	`|`
logical AND	`&&`
logical OR	`||`
conditional	`? :`
assignment	`= += -= *= /= %= &= ^= |= <<= >>= >>>=`

Estructuras de Programación

Una **expresión** es un conjunto de variables unidos por operadores (también puede haber llamados a funciones).
Una **sentencia** es una expresión que acaba en punto y coma (;).

Una **Bifurcación if** ejecuta un conjunto de sentencias en función del valor de la expresión de comparación.

```
if (booleanExpression) {
      statements;
} else if (booleanExpression2) {
      statements2;
} else {
      statements3;
}
```

Una Sentencia Switch compara la misma expresión con distintos valores.

```
switch (expression) {
       case value1: statements1; break;
       case value2: statements2; break;
       case value3: statements3; break;
       case value4: statements4; break;
      [default: statements5;]
}
```

Cada sentencia case es un valor único de expression. Se compara con valores concretos. Si no se pone un break, cuando se ejecuta una sentencia case se ejecutan también todas las que van a continuación hasta llegar a un break o hasta que termine el switch.

Un **bucle** se utiliza para realizar un proceso repetidas veces.

```
while (booleanExpression) {
      statements;
}
```

```
for (initialization; booleanExpression; increment) {
      statements;
}
```

```
initialization;
while (booleanExpression) {
 statements;
 increment;
}
```

```
do {
      statements
} while (booleanExpression);
```

Clases y Objetos

Ejemplo de como declarar una clase:

```
public class Bicycle {

    // the Bicycle class has
    // three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has
    // one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has
    // four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }
```

```
    public void speedUp(int increment) {
        speed += increment;
    }

}
```

Pueden existir clases que son subclases de otras y que heredan todos los campos y métodos. Incluso pudiendo agregar métodos y campos propios de la subclase.

```
public class MountainBike extends Bicycle {

    // the MountainBike subclass has
    // one field
    public int seatHeight;

    // the MountainBike subclass has
    // one constructor
    public MountainBike(int startHeight, int startCadence,
                        int startSpeed, int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    // the MountainBike subclass has
    // one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }

}
```

Los constructores inicializan nuevos objetos. Las declaraciones de los campos dan el estado de la clase y sus objetos. Los métodos implementan los comportamientos de la clase y sus objetos.

Hacer **overloading** de métodos significa crear métodos con el mismo nombre, dentro de la misma clase, pero se diferencian en los tipos y cantidad de parámetros que aceptan.

Objetos

Ejemplo de como crear objetos a partir de clases

```
public class CreateObjectDemo {

    public static void main(String[] args) {

        // Declare and create a point object and two rectangle objects.
        Point originOne = new Point(23, 94);
        Rectangle rectOne = new Rectangle(originOne, 100, 200);
        Rectangle rectTwo = new Rectangle(50, 100);

        // display rectOne's width, height, and area
        System.out.println("Width of rectOne: " + rectOne.width);
        System.out.println("Height of rectOne: " + rectOne.height);
        System.out.println("Area of rectOne: " + rectOne.getArea());
```

```
        // set rectTwo's position
        rectTwo.origin = originOne;

        // display rectTwo's position
        System.out.println("X Position of rectTwo: " + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: " + rectTwo.origin.y);
    }
}
```

Una clase es el blueprint de un objeto. Para crear un objeto, se siguen los siguientes pasos:

- Declaration: declaramos una variable que asocia un nombre de variable y un tipo de objeto.
- Instantiation (Instanciar): se usa el operador new para crear un objeto.
- Initialization (Inicialización): seguido del operador new, se llama al constructor para inicializar al objeto.

Referencias

Cuando declaramos una variable, avisamos al compilador que vamos a usar la variable para referirnos a data de cierto tipo (*type name*;). Solamente declarar referencias/variables no crea un objeto. Cuando asignamos la variable sin darle un valor, tenemos una referencia que todavía no apunta a nada. Cuando instanciamos una clase (con el operador new), estamos asignando memoria a un objeto y retornando la referencia a ese espacio en memoria. Un objeto puede tener varias referencias.