Parte I

Introducción

Capítulo 1

Lenguajes y Ambientes sugeridos para desarrollo

1.1. Lenguajes de programación

Existe una infinidad de lenguajes de programación. Además de los cubiertos de manera general en este material podemos menciona algunos como:

- **Groovy**. Es un lenguaje orientado a objetos y dinámico, similar a Python, Ruby, Perl y Smalltalk pero que es dinámicamente compilado hacia bytecodes de la máquina virtual de Java.

- **JRuby**. Es una implementación en Java del intérprete de Ruby. Su alta integración con Java permite completo acceso en los dos sentidos entre código Java y Ruby.
- **Jython/JPython**. Una implementación de Python en Java. Programas en Jython pueden importar y usar clases en Java.

- **Kotlin**. Un lenguaje orientado a objetos para JVM para aplicaciónes del lado del servidor, Android y compilación a JavaScript.
- **IronPython**.

- **Dart**
- **Go**
- **Swift**
- **Cobra**. Es un lenguaje de propósito general inspirado en Python y Ruby, pero con tipos estáticos, generación de código para .NET/Mono, ligado dinámico, manejo de contratos (como en Eiffel o D) y soporte de pruebas de unidad en el lenguaje. Hasta marzo de 2009 se encontraba en versión Beta.
- **Fantom** (http://fantom.org/)

1.1.1. IDEs

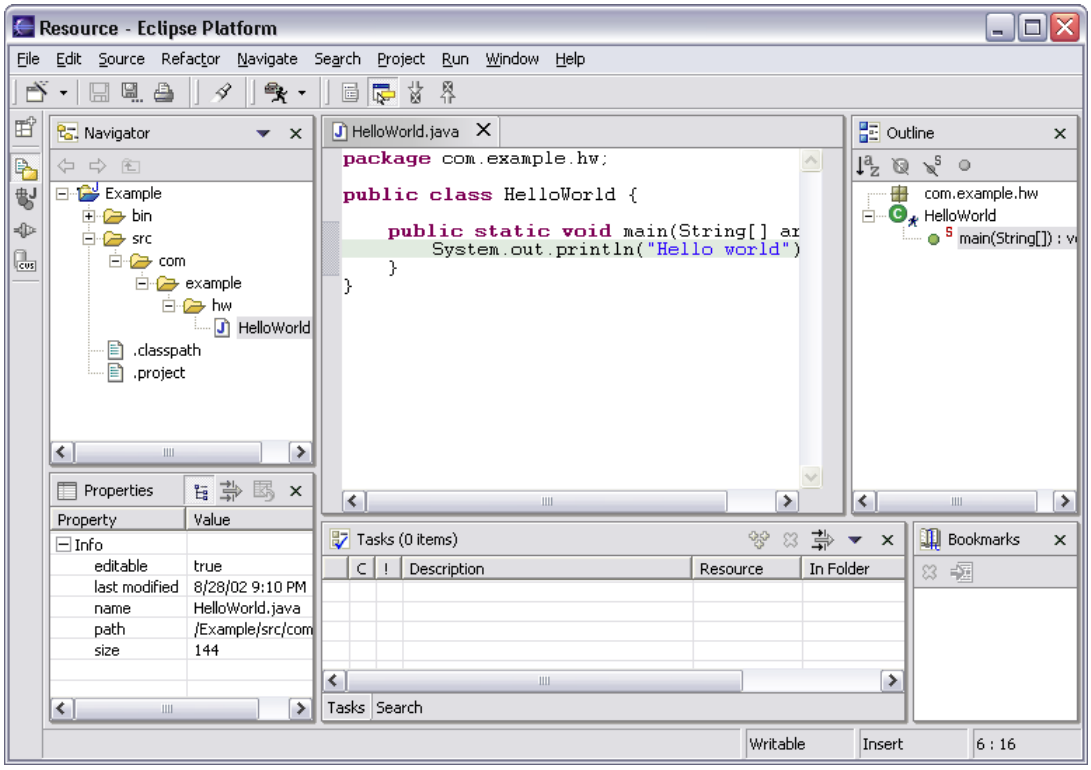
Eclipse

Eclipse es desarrollado como un proyecto de código abierto lanzada en noviembre de 2001 por IBM, Object Technology International y otras compañias. El objetivo era desarrollar una plataforma abierta de desarrollo. Fue planeada para ser extendida mediante plug-ins.

Es desarrollada en Java por lo que puede ejecutarse en un amplio rango de sistemas operativos. Tambien incorpora facilidades para desarrollar en Java aunque es posible instalarle plug-ins para otros lenguajes como C/C++, PHP, Ruby, Haskell, etc. Incluso antiguos lenguajes como Cobol tienen extensiones disponibles para Eclipse [1]:

- Eclipse + JDT = Java IDE
- Eclipse + CDT = C/C++ IDE
- Eclipse + PHP = PHP IDE
- Eclipse + JDT + CDT + PHP = Java, C/C++, PHP IDE
- ...

Trabaja bajo "*workbenchs*" que determinan la interfaz del usuario centrada alrededor del editor, vistas y perspectivas.



Los recursos son almacenados en el espacio de trabajo (workspace) el cual es un folder almacenado normalmente en el directorio de Eclipse. Es posible manejar diferentes áreas de trabajo.

Eclipse, sus componentes y documentación pueden ser obtenidos de: www.eclipse.org

Para trabajar en este curso se requerirá preferentemente al menos:

1. Java 1.5 o 1.6
2. Eclipse SDK 3.x
3. Plug in para C/C++ en Eclipse (CDT) y compilador de ANSI C/C++ si no se tiene uno instalado.
4. Ruby 1.8
5. Plug in para Ruby (RDT – Rubt Development Tools) si se quiere usar Ruby con Eclipse

Tips para instalar CDT (C/C++ Development Toolkit) en Eclipse en Windows:

1. Instalar un compilador de C/C++ a. Si se instala MinGW y se tiene que renombrar el archivo mingw32-make.exe por make.exe. Ver http://www.mingw.org b. Además, añadir el depurador gdb pues no viene incluido en MinGW, aunque existe una copia en el sitio antes mencionado o puede obtenerse de: http://sourceware.org/gdb

2. Instalar el CDT desde el sitio recomendado Usando el Update Manager y obteniendo el plug in del sitio correspondiente[1] Update Site que aparece listado al entrar a Available Software en el Update Manager (Otra opcion es ir a http://www.eclipse.org/cdt/)

3. Agregar \MinGW \bin (o el directorio correspondiente) al la variable PATH del sistema.

Si el CDT esta bien instalado debe ser posible crear proyectos de C/C++, compilarlos, ejecutarlos y depurarlos dentro del ambiente de Eclipse.

En el caso de Ruby, el interprete puede obtenerse de: http://www.ruby-lang.org/en/downloads/

El RDT para eclipse se instala como el CDT y se obtiene de: http://rubyeclipse.sourceforge.net/download.rdt.html

Posteriormente se tiene que especificar en Eclipse la ruta del intérprete.

Mono

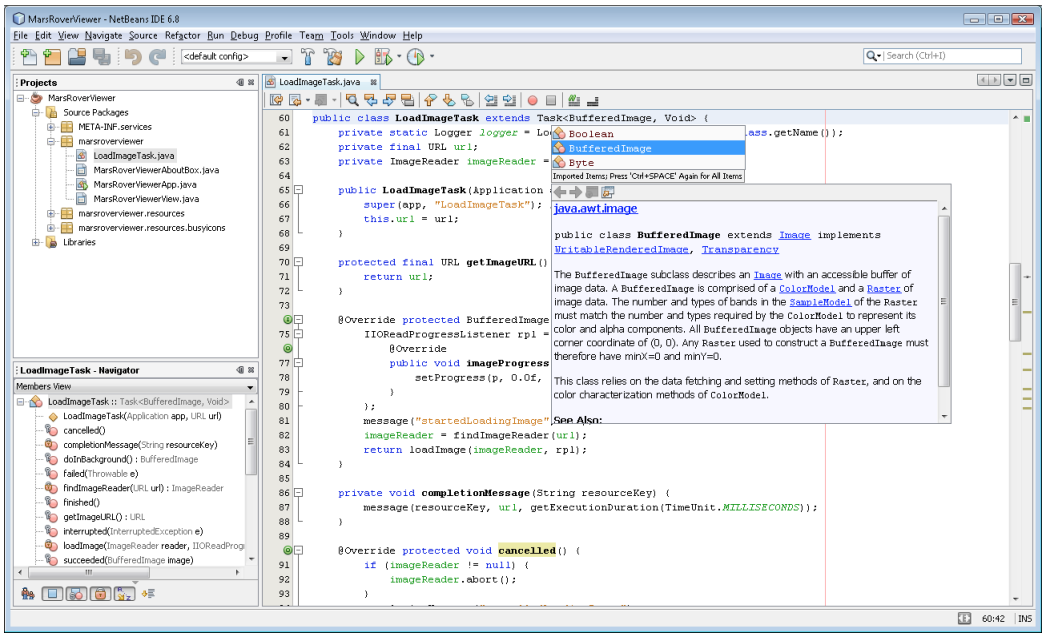
Mono es una alternativa de software libre patrocinada por Novell Implementa principalmente un compilador para C# y el *Common Language Runtime* de .NET. Incluye un IDE y existen versiones para plataformas aparte de Windows. Su IDE es MonoDevelop.

Netbeans

NetBeans es una plataforma de desarrollo y un IDE multilenguaje. Originalmente creado para desarrollo en Java y adquirido por Sun Microsystems. El IDE es actualmente de código abierto y disponible en www.netbeans.org.

Creado originalmente en 1996 para Java, aunque cuenta con extensiones para otros lenguajes. Creado originalmente como un proyecto universitario de la Facultad de Matemáticas y Física en la Universidad *Karlova* (Carolina) en Praga.

[1] El nombre del sitio cambia de acuerdo a la versión de Eclipse



Soporta una variedad de lenguajes y tecnologías como C/C++, Java (SE, EE, ME), Ruby, PHP, Ruby, entre otros. NetBeans tiene incluido el desarrollo visual de aplicaciones, lo cual es bastante conveniente para desarrollo rápido de interfaces de usuario.

Algunos editores ligeros

Si no quieren de momento usar IDEs como Eclipse o Netbeans. Podrían usar algún editor como los siguientes:

- Visual studio code. Un editor gratuito de Microsoft. [2]
- Geany. [3]
- Sublime. [4]
- Atom. [5]
- Brackets. [6]

Hay que tener en cuenta que estos también dependen de que tengan los compiladores del lenguaje instalados.

[2] https://code.visualstudio.com/
[3] https://www.geany.org/
[4] https://www.sublimetext.com/
[5] https://atom.io/
[6] http://brackets.io/

Otros ejemplos de frameworks

Algunos de los principales frameworks usados para desarrollo Web:

- Ruby on Rails. Es un framework gratuito para desarrollo de aplicaciones Web en Ruby. 

- Merb. Un framework para desarrollo web en Ruby. Fue desarrollado después de tratar de manejar hilos seguros en Ruby on Rails.
- Django. Es un framework open source para desarrollo de aplicaciones web usando Python. 

- Grails. Es un framework open source para el lenguaje Groovy.


- SproutCore. Es un framework open source para desarrllo de aplicaciones web con JavaScript con el objetivo de crear aplicaciones web que se comporten y sientan como aplicaciones de escritorio. Usa Ruby para generar HTML estático y archivos JavaScript. Es usado por Apple (anunciado en 2008) para generar aplicaciones multi-plataforma. De hecho, MobileMe está desarrollado con este framework. Es la opción de Apple para competir con Flash y Silverlight.
- Lift. Es un framework para desarrollo web usando el lenguaje Scala. Al usar Scala, es compatible con la JVM y la biblioteca de Java.

Capítulo 2

Introducción a características de C++

Ahora comentaremos algunas características de C++ que no tienen que ver directamente con la programación orientada a objetos.

2.1. Comentarios en C++

Los comentarios en C son:

```
/*  comentario en C     */
```

En C++ los comentarios pueden ser además de una sola línea:

```
// este es un comentario en C++
```

Acabando el comentario al final de la línea, lo que quiere decir que el programador no se preocupa por cerrar el comentario.

2.2. Flujo de entrada/salida

En C, la salida y entrada estándar estaba dada por *printf* y *scanf* principalmente (o funciones similares) para el manejo de los tipos da datos simples y las cadenas. En C++ se proporcionan a través de la biblioteca *iostream*, la cual debe ser insertada a través de un *#include*. Las instrucciones son:

- *cout*. Utiliza el flujo salida estándar. Que se apoya del operador $<<$, el cual se conoce como operador de inserción de flujo *colocar en*.
- *cin*. Utiliza el flujo de entrada estándar. Que se apoya del operador $>>$, conocido como operador de extracción de flujo *obtener de*

Los operadores de inserción y de extracción de flujo no requieren cadenas de formato (%s, %f), ni especificadores de tipo. C++ reconoce de manera automática que tipos de datos son extraídos o introducidos.

En el caso del operador de extracción >> no se requiere el operador de dirección &.

De tal forma un código de desplegado con *printf* y *scanf* de la forma (usando el área de nombres estándar):

```
printf("Número: ");

scanf("\%d", &num);

printf("El valor leído es: " \%d\n", num);
```

No es necesario poner hasta el final de texto la variable

Sería en C++ de la siguiente manera:

```
cout << "Número";

cin >> num;

cout << "El valor leído es: " << num << '\n';
```

2.3. Funciones en línea

No las más recomendables

Las funciones en línea, se refiere a introducir un calificador *inline* a una función de manera que le sugiera al compilador que genere una copia del código de la función en lugar de la llamada.

Ayuda a reducir el número de llamadas a funciones reduciendo el tiempo de ejecución en algunos casos, pero en contraparte puede aumentar el tamaño del programa.

A diferencia de las macros, las funciones *inline* si incluyen verificación de tipos y son reconocidas por el depurador. Un depurador puede ayudar a encontrar un error de lógica resultado de usar una macro, pero no puede atribuir los errores a alguna macro.

Las funciones inline deben usarse sólo para funciones chicas que se usen frecuentemente.

El compilador desecha las solicitudes *inline* para programas que incluyan un ciclo, un *switch* o un *goto*. Tampoco se consideran si no tienen *return* (aunque no regresen valores) o si contienen variables de tipo *static*. Además, lógicamente no genera una función *inline* para funciones recursivas.

Sintaxis

```
1 inline <declaración de la función>
```

Ejemplo:

```
inline float suma (float a, float b) {
        return a+b;
}

inline int max( int a, int b) {
                return (a > b) ? a :  b;
}
```

Caso 1 Caso 2

Es una condicional, en caso de que se cumpla retorna caso 1, de lo contrario b caso 2

Nota: Las funciones *inline* tienen conflictos con los prototipos, así que deben declararse completas sin prototipo en el archivo .h. Además, si la función en línea hace uso de otra función, en donde se expanda la función en línea debe tener los *include* correspondientes a esas funciones utilizadas.

2.3.1. Declaraciones de variables

Mientras que en C, las declaraciones deben ir en la función antes de cualquier línea ejecutable, en C++ pueden ser introducidas en cualquier punto, con la condición lógica de que la declaración esté antes de la utilización de lo declarado.

En algunos compiladores podía declararse una variable en la sección de inicialización de la instrucción for, pero es incorrecto declarar una variable en la expresión condicional del *while*, *do-while*, *for*, *if* o *switch*. El actual estándar de C++ no permite la declaración de variables dentro del for.

NOTA;

Ejemplo:

```
#include <iostream>

int main() {
        int i=0;

        for (i=1; i<10; i++){
                int j=10;
                std::cout<<i<<" j: "<<j<<std::endl;
           }

           std::cout<<"\ni al salir del ciclo: "<<i;

        return 0;
}
```

Listing 1. Ejemplo declaración de variables

O usando el área de nombres estándar:

```
#include <iostream>

using namespace std;

int main() {
        int i=0;

        for (i=1; i<10; i++){
                int j=10;
                cout<<i<<" j: "<<j<<endl;
           }

           cout<<"\ni al salir del ciclo: "<<i;

        return 0;
}
```

Listing 2. Ejemplo declaración de variables usando el área de nombres estándar

El alcance de las variables en C++ es por bloques. Una variable es vista a partir de su declaración y hasta la llave "}" del nivel en que se declaró. Lo cual quiere decir que las instrucciones anteriores a su declaración no pueden hacer uso de la variable, ni después de finalizado el bloque.

Importante

2.3.2. Operador de resolución de alcance

Se puede utilizar el operador de resolución de alcance **::** se refiere a una variable (variable, función, tipo, enumerador u objeto), con un alcance de archivo (variable global). No tiene alcance de bloque, aunque esten variables definidas con el mismo nombre en varios niveles de bloques.

Esto le permite al identificador ser visible aún si el identificador se encuentra oculto.

Ejemplo:

```
float h;

void g(int h) {
        float a;
        int b;

        a=::h;        // a se inicializa con la variable global h

```

```
9           b=h;        // b se inicializa con la variable local h
10  }
```

2.3.3. Valores por default

Las funciones en C++ pueden tener valores por default. Estos valores son los que toman los parámetros en caso de que en una llamada a la función no se encuentren especificados.

Los valores por omisión deben encontrarse en los parámetros que estén más a la derecha. Del mismo modo, en la llamada se deben empezar a omitir los valores de la extrema derecha.

Ejemplo:

```
1   #include <iostream>
2
3   using namespace std;
4
5   int punto(int=5, int=4);
6
7   int main () {
8
9           cout<<"valor 1: "<<punto()<<'\n';
10          cout<<"valor 2: "<<punto(1)<<'\n';
11          cout<<"valor 3: "<<punto(1,3)<<'\n';
12          return 0;
13  }
14
15  int punto( int x, int y){
16
17          if(y!=4)
18                  return y;
19          if(x!=5)
20                  return x;
21          return x+y;
22  }
```

Listing 3. Ejemplo de valores por default

C++ no permite la llamada omitiendo un valor antes de la extrema derecha de los argumentos:

```
punto( , 8);
```

Otro ejemplo de valores o argumentos por default:

```
#include <iostream>

using namespace std;

int b=1;
int f(int);
int h(int x=f(b));        // argumento default f(::b)

int main () {
        b=5;
        cout<<b<<endl;
        {
                int b=3;
                cout<<b<<endl;
                cout<<h();                //h(f(::b))
        }

        return 0;
}

int h(int z){
        cout<<"Valor recibido: "<<z<<endl;
        return z*z;
}
int f(int y){
        return y;
}
```

Listing 4. Valores o argumentos por deafult.

2.3.4. Parámetros por referencia

En C todos los pasos de parámetros son por valor, aunque se pueden enviar parámetros ”por referencia” al enviar por valor la dirección de un dato (variable, estructura, objeto), de manera que se pueda acceder directamente el área de memoria del dato del que se recibió su dirección. Aunque se tienen que manejar en algunas ocasiones como apuntadores.

C++ introduce parámetros por referencia **reales**. La manera en que se definen es agregando el símbolo & de la misma manera que se coloca el *: después del tipo de dato en el prototipo y en la declaración de la función.

Ejemplo:

```
1  // Comparando parámetros por valor, por valor con  apuntadores ("referencia"),
2  // y paso por referencia real
3
4  #include <iostream>
5  using namespace std;
6
7  int porValor(int);
8  void porApuntador(int *);
9  void porReferencia( int &);
10
11 int main() {
12         int x=2;
13         cout << "x= " << x << " antes de llamada a porValor \n"
14                 << "Regresado por la función: "<< porValor(x)<<endl
15                 << "x= " << x << " despues de la llamada a porValor\n\n";
16
17         int y=3;
18         cout << "y= " << y << " antes de llamada a porApuntador\n";
19         porApuntador(&y);
20         cout << "y= " << y << " despues de la llamada a porApuntador\n\n";
21
22         int z=4;
23         cout << "z= " << z << " antes de llamada a porReferencia \n";
24         porReferencia(z);
25         cout<< "z= " << z << " despues de la llamada a porReferencia\n\n";
26         return 0;
27 }
28
29 int porValor(int valor){
30         return valor*=valor;     //parámetro no modificado
31 }
32
33 void porApuntador(int *p){
34         *p *= *p;                // parámetro modificado
35 }
36
37 void porReferencia( int &r){
38         r *= r;                //parámetro modificado
39 }
```

Listing 5. Otro ejemplo Parámetros por referencia.

Notar que no hay diferencia en el manejo de un parámetro por referencia y uno por valor, lo que puede ocasionar ciertos errores de programación.

Variables de referencia

También puede declararse una variable por referencia que puede ser utilizada como un seudónimo o alias. Ejemplo:

```
int max=1000, &sMax=max;        //declaro max y sMax es un alias de max
sMax++;                  //incremento en uno max a través de su alias
```

Esta declaración no reserva espacio para el dato, pues es como un apuntador pero se maneja como una variable normal. No se permite reasignarle a la variable por referencia otra variable. La variable por referencia debe ser inicializada en el momento de su declaración.

Ejemplo:

```
// variable por referencia
#include <iostream>

using namespace std;

int main() {
        int x=2, &y=x, z=8;

        cout << "x= "<<x <<endl
                <<"y= "<<y<<endl;

        y=10;
        cout << "x= "<<x <<endl
                <<"y= "<<y<<endl;
// Reasignar no esta permitido
//          \&  y= &z;
//        cout << "z= "<<z <<endl
//                <<"y= "<<y<<endl;

        y++;
        cout << "z= "<<z <<endl
                <<"y= "<<y<<endl;

        return 0;
}
```

Listing 6. Ejemplo de variables por referencia.

2.3.5. Asignación de memoria en C++

En el **ANSI C**, se utilizan *malloc*, *calloc* y *free* para asignar y liberar dinámicamente memoria:

```
1 float *f;
2 f = (float *) malloc(sizeof(float));
3 . . .
4 free(f);
```

Se debe indicar el tamaño a través de *sizeof* y utilizar una máscara (*cast*) para designar el tipo de dato apropiado.

En C++, existen dos operadores para asignación y liberación de memoria dinámica: *new* y *delete*.

```
1 float *f;
2 f= new float;
3 . . .
4 delete f;
```

El operador new crea automáticamente un área de memoria del tamaño adecuado. Si no se pudo asignar la memoria se regresa un apuntador nulo (*NULL* ó 0). Nótese que en C++ se trata de operadores que forman parte del lenguaje, no de funciones de biblioteca.

El operador *delete* libera la memoria asignada previamente por *new*. No se debe tratar de liberar memoria previamente liberada o no asignada con *new*.

Es posible hacer asignaciones de memoria con inicialización:

```
int *max= new int (1000);
```

También es posible crear arreglos dinámicamente:

```
1     char *cad;
2         cad= new char [30];
3         . . .
4         delete [] cad;
```

Usar *delete* **sin** los corchetes para arreglos dinámicos puede no liberar adecuadamente la memoria, sobre todo si son elementos de un tipo definido por el usuario.

Ejemplo 1:

```
1 #include <iostream>
2
3 using namespace std;
4
```

```
5  int main() {
6          int *p,*q;
7
8          p= new int; //asigna memoria
9
10         if(!p) {
11                 cout<<"No se pudo asignar memoria\n";
12                 return 0;
13         }
14         *p=100;
15         cout<<endl<< *p<<endl;
16
17         q= new int (123); //asigna memoria
18         cout<<endl<< *q<<endl;
19
20         delete p; //libera memoria
21 //        *p=20;                Uso indebido de pues ya se liberó
22 //        cout<<endl<< *p<<endl; la memoria
23         delete q;
24         return 0;
25 }
```

Listing 7. Ejemplo 1 de asignación de memoria en C++

Ejemplo 2:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6          float *ap, *p=new float (3) ;
7          const int MAX=5;
8          ap= new float [MAX]; //asigna memoria
9          int i;
10         for(i=0; i<MAX; i++)
11                 ap[i]=i * *p;
12         for(i=0; i<MAX; i++)
13                 cout<<ap[i]<<endl;
14         delete p;
15     delete [] ap;
16         return 0;
```

```
17 }
```

Listing 8. Ejemplo 2 de asignación de memoria en C++

Ejemplo 3:

```
1  #include <iostream>
2
3  using namespace std;
4
5  typedef struct {
6          int n1,
7                  n2,
8                  n3;
9  }cPrueba;
10
11 int main() {
12         cPrueba *pr1, *pr2;
13
14         pr1= new cPrueba;
15         pr1->n1=11;
16         pr1->n2=12;
17         pr1->n3=13;
18
19         pr2= new cPrueba(*pr1);
20         delete pr1;
21
22    cout<< pr2->n1<<" "<<pr2->n2 <<" "<<pr2->n3<<endl;
23
24         delete pr2;
25         return 0;
26 }
```

Listing 9. Ejemplo 3 de asignación de memoria en C++

2.3.6. Plantillas

Cuando las operaciones son idénticas pero requieren de diferentes tipos de datos, podemos usar lo que se conoce como *templates* o plantillas de función.

El término de plantilla es porque el código sirve como base (o plantilla) a diferentes tipos de datos. C++ genera al compilar el código objeto de las funciones para cada tipo de dato involucrado en las llamadas. Esta solución antes se hacia en C usando macros con #define, pero no tenían verificación de tipos.

Las definiciones de plantilla se escriben con la palabra clave *template*, con una lista de parámetros formales entre <> . Cada parámetro formal lleva la palabra clave *class*. La instrucción *type* puede también ser usada.

Cada parámetro formal puede ser usado para sustituir a: tipos de datos básicos, estructurados o definidos por el usuario, tipos de los argumentos, tipo de regreso de la función y para variables dentro de la función.

Ejemplo 1:

```
#include <iostream>

using namespace std;

template <class T>
T mayor(T x, T y)
{
  return (x > y) ? x : y;
};

int main(){
        int a=10, b=20, c=0;
        float x=44.1, y=22.3, z=0 ;

        c=mayor(a, b);
        z=mayor(x, y);
        cout<<c<<" "<<z<<endl;

//        z=mayor(x,b); error no hay mayor( float, int)
//        z=mayor(a, y); ""   ""   "" ""  (int, float)

        return 0;
}
```

Listing 10. Ejemplo.

Consideraciones:

- Cada parámetro formal debe aparecer en la lista de parámetros de la función al menos una vez.
- No puede repetirse en la definición de la plantilla el nombre de un parámetro formal.
- Tener cuidado al manejar mas de un parámetro en los templates.

Ejemplo 2:

```
#include <iostream>

using namespace std;

template <class T>
void desplegaArr(T arr[], const int cont, T pr)
{
        for(int i=0; i<cont; i++)
                cout<< arr[i] << " ";
        cout<<endl;
        cout<<pr<<endl;
}

int main() {
        const int contEnt=4, contFlot=5, contCar=10;
int ent[]={1,2,3,4};
        float flot[]={1.1, 2.2, 3.3, 4.4, 5.5};
        char car[]={"Plantilla"};


        cout<< "Arreglo de flotantes:\n";
        desplegaArr(flot, contFlot,(float)3.33);

        cout<< "Arreglo de caracteres:\n";
        desplegaArr(car, contCar, 'Z');

        cout<< "Arreglo de enteros:\n";
        desplegaArr(ent, contEnt, 99);

        return 0;
}
```

Listing 11. Ejemplo 2 de uso de plantillas en C++.

Ejemplo 3:

```
#include <iostream>

using namespace std;

template <class T, class TT>
T mayor(T x, TT y)
```

```
7  {
8    return (x > y) ? x : y;
9  };
10
11 int main(){
12
13         int a=10, b=20, c=0;
14         float x=44.1, y=22.3, z=0 ;
15
16         c=mayor(a, b);
17         z=mayor(x, y);
18
19         cout<<c<<" "<<z<<endl;
20 //sin error al aumentar un parámetro formal.
21         z=mayor(x,b);
22         cout<<z<<endl;
23         z=mayor(a,y); //regresa entero pues a es entero (tipo T es entero para
24         cout<<z<<endl;                  // este llamado.
25
26         z=mayor(y, a);
27         cout<<z<<endl;
28         c=mayor(y, a);//regresa flotante pero la asignación lo corta en entero.
29         cout<<c<<endl;
30
31         return 0;
32 }
```

Listing 12. Ejemplo 3 de uso de plantillas en C++.

2.3.7. Enumeraciones

Aunque las enumeraciones existen en ANSI C, en ese lenguaje son constantes asociadas al tipo entero; por lo que son una especie de alias hacia estos valores. En C++ una enumeración define realmente un tipo de dato.[1]

Las enumeraciones sirven para agrupar un conjunto de elementos dentro de un tipo definido. El manejo de enumeraciones en C++ es el siguiente:

Sintaxis
`enum <nombreEnum> { <elem 1>, <elem 2>, ..., <elem n> };`

[1]Existe otro tipo de enumeración en C++ llamado *enumclass*

Por lo que para el código anterior, la enumeración sería:

```
enum Temporada { PRIMAVERA, VERANO, OTOÑO, INVIERNO }
```

Ejemplo:

```
#include <iostream>

using namespace std;

enum Temporada { PRIMAVERA, VERANO, OTONO, INVIERNO };

 int main() {
                Temporada tem;
                tem=PRIMAVERA;
                cout<<"Temporada: " << tem<<endl;

        cout<<"\nListado de temporadas:";
                int t;
        for(t=PRIMAVERA; t<=INVIERNO; t++)
                        cout<<"Temporada: "<< t<<endl;
        return 0;
}
```

Listing 13. Ejemplo de enumeración en C++.

2.3.8. Espacios de nombre (*namespaces*)

Los espacios de nombres permiten agrupar entidades que de otro modo tendrían un alcance global. Es preferible tener un alcance de espacio de nombre que un alcance global. Un *namespace* proporciona un un alcance para identificadores tales como tipos, funciones, variables, etc. Algunas características[2]:

- Se pueden tener múltiples bloques con el mismo *namespace*, todas la declaraciones en bloques con el mismo nombre irán al mismo espacio de nombres.
- Es posible tener espacios de nombre anidados.
- Las declaraciones de espacios de nombre no dan acceso público o privado.

[2]Más información: https://en.cppreference.com/w/cpp/language/namespace

Sintaxis

```
namespace <nombre> {
   int x, y; // declaraciones de código donde
             // x , y son declaradas en el alcance de <nombre>
}
```

Ejemplo:

```
1  #include <iostream>
2  using namespace std;
3
4  namespace ns1 {
5      int valor()    { return 5; }
6  }
7  namespace ns2  {
8      const double x = 100;
9      double valor() {  return 2*x; }
10 }
11
12 int main() {
13     cout << ns1::valor() << '\n';
14     cout << ns2::valor() << '\n';
15     cout << ns2::x << '\n';
16 }
```

Listing 14. Ejemplo de uso de *namespace*.

2.3.9. Tipo de dato booleano

Este tipo de dato (*bool*) maneja los valores verdadero o falso (*true* o *false*). Es un tipo de dato y sus literales son además parte de las palabras reservadas de C++. Puede ser declarado como sigue:

```
1 bool b1;
2 b1=true;
3 bool b2 = false;
```

Sin embargo, al final el tipo booleano en C++ sigue siendo equivalente a su uso en el lenguaje C: cero para falso y diferente de cero es verdadero. Por lo que su uso no es obligatorio.

2.3.10. Uso de cadenas con *string*

En C++ podemos hacer uso de cadenas por medio del tipo *string*. En compiladortes actuales es suficiente con incluir la biblioteca *iostream* aunque en realidad se encuentra directamente declarado en *cstring*

Ejemplo:

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5    string nombre="Juan", apellido="Pérez";
6    cout << "Hola, " << nombre << " " << apellido << endl;
7    cout << "¿Cómo estás?" << endl;
8 }
```

Listing 15. Ejemplo de uso de cadenas *string*.

Capítulo 3

Introducción a Java

3.1. Origen

Java es un lenguaje de programación orientada a objetos, diseñado dentro de *Sun Microsystems* por James Gosling. Originalmente, se le asignó el nombre de *Oak* y fue un lenguaje pensado para usarse dentro de dispositivos electrodomésticos, que tuvieran la capacidad de comunicarse entre sí. Posteriormente fue reorientado hacia Internet, aprovechando el auge que estaba teniendo en ese momento la red, y lo rebautizaron con el nombre de Java. Es anunciado al público en mayo de 1995 enfocándolo como la solución para el desarrollo de aplicaciones en *web*. Sin embargo, se trata de un lenguaje de propósito general que puede ser usado para la solución de problemas diversos.

Java es un intento serio de resolver simultáneamente los problemas ocasionados por la diversidad y crecimiento de arquitecturas incompatibles, tanto entre máquinas diferentes como entre los diversos sistemas operativos y sistemas de ventanas que funcionaban sobre una misma máquina, añadiendo la dificultad de crear aplicaciones distribuidas en una red como Internet. El interés que generó Java en la industria fue mucho, tal que nunca un lenguaje de programación había sido adoptado tan rápido por la comunidad de desarrolladores. Las principales razones por las que Java es aceptado tan rápido:

- Aprovecha el inicio del auge de la Internet, específicamente del *World Wide Web*.

- Es orientado a objetos, la cual si bien no es tan reciente, estaba en uno de sus mejores momentos, todo mundo quería programar de acuerdo al modelo de objetos.

- Se trataba de un lenguaje que eliminaba algunas de las principales dificultades del lenguaje C/C++, el cuál era uno de los lenguajes dominantes. Se decía que la ventaja de Java es que es sintácticamente parecido a C++, sin serlo realmente.

- Java era resultado de una investigación con fines comerciales, no era un lenguaje académico como Pascal o creado por un pequeño grupo de personas como C ó C++.

Aunado a esto, las características de diseño de Java, lo hicieron muy atractivo a los programadores.

3.1.1. Características de diseño

Es un lenguaje de programación de alto nivel, de propósito general, y cuyas características son [3]:

- Simple y familiar.
- Orientado a objetos.
- Independiente de la plataforma
- Portable
- Robusto.
- Seguro.
- Multihilos.

Simple y familiar

Es simple, ya que tanto la estructura léxica como sintáctica del lenguaje es muy sencilla. Además, elimina las características complejas e innecesarias de sus predecesores.

Es familiar al incorporar las mejores características de lenguajes tales como: C/C++, Modula, Beta, CLOS, Dylan, Mesa, Lisp, Smalltalk, Objective-C, y Modula 3.

Orientado a objetos

Es realmente un lenguaje orientado a objetos, todo en Java son objetos:

- No es posible que existan funciones que no pertenezcan a una clase.
- La excepción son los tipos de datos primitivos, como números, caracteres y booleanos [1].

Cumple con los 4 requerimientos de Wegner [4]:

OO = abstracción + clasificación + polimorfismo + herencia

[1] Los puristas objetarían que no es totalmente orientado a objetos. En un sentido estricto *Smalltalk* es un lenguaje "más" puro, ya que ahí hasta los tipos de datos básicos son considerados objetos.

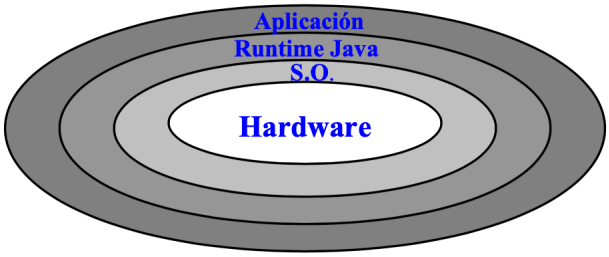


Fig. 3.1. Máquina virtual de Java y su ejecución sobre una plataforma

Independiente de la plataforma

La independencia de la plataforma implica que un programa en Java se ejecute sin importar el sistema operativo que se este ejecutando en una máquina en particular. Por ejemplo un programa en C++ compilado para *Windows*, debe ser al menos vuelto a compilar si se quiere ejecutar en Unix; además, posiblemente habrá que ajustar el código que tenga que ver con alguna característica particular de la plataforma, como la interfaz con el usuario.

Java resuelve el problema de la distribución binaria mediante un formato de código binario (*bytecode*) que es independiente del hardware y del sistema operativo gracias a su máquina virtual.

Si el sistema de *runtime* o máquina virtual está disponible para una plataforma específica, entonces una aplicación puede ejecutarse sin necesidad de un trabajo de programación adicional. Ver figura 3.1.

Portable

Una razón por la que los programas en Java son portables es precisamente que el lenguaje es independiente de la plataforma.

Además, la especificación de sus tipos de datos primitivos y sus tamaños, así como el comportamiento de los operadores aritméticos, son estándares en todas las implementaciones de Java. Por lo que por ejemplo, un entero es definido de un tamaño de 4 bytes, y este espacio ocupará en cualquier plataforma, por lo que no tendrá problemas en el manejo de los tipos de datos. En cambio, un entero en C generalmente ocupa 2 bytes, pero en algunas plataformas el entero ocupa 4 bytes, lo que genera problemas a la hora de adaptar un programa de una plataforma a otra.

Robusto

Java se considera un lenguaje robusto y confiable, gracias a:

- **Validación de tipos.** Los objetos de tipos compatibles pueden ser asignados a otros objetos sin necesidad de modificar sus tipos. Los objetos de tipos potencialmente incompatibles requieren un modificador de tipo (*cast*). Si la modificación de tipo es claramente imposible, el compilador lo rechaza y reporta un **error en tiempo de**

compilación. Si la modificación resulta legal, el compilador lo permite, pero inserta una **validación en tiempo de ejecución.** Cuando el programa se ejecuta se realiza la validación cada vez que se ejecuta una asignación potencialmente inválida.

- **Control de acceso a variables y métodos.** Los miembros de una clase pueden ser privados, públicos o protegidos[2]. En java una variable privada, es realmente privada. Tanto el compilador como la máquina virtual de Java, controlan el acceso a los miembros de una clase, garantizando así su privacidad.

- **Validación del apuntador Null.** Todos los programas en Java usan apuntadores para referenciar a un objeto. Esto no genera inestabilidad pues una validación del apuntador a *Null* ocurre cada vez que un apuntador deja de referencia a un objeto. C y C++ por ejemplo, no tienen esta consideración sobre los apuntadores, por lo que es posible estar referenciando a localidades inválidas de la memoria.

- **Límites de un arreglo.** Java verifica en tiempo de ejecución que un programa no use arreglos para tratar de acceder a áreas de memoria que no le pertenecen. De nuevo, C y C++ no tiene esta verificación, lo que permite que un programa se salga del límite mayor y menor de un arreglo.

- **Aritmética de apuntadores.** Aunque todos los objetos se manejan con apuntadores, Java elimina la mayor parte de los errores de manejo de apuntadores porque no soporta la aritmética de apuntadores:

  - No soporta acceso directo a los apuntadores
  - No permite operaciones sobre apuntadores.

- **Manejo de memoria.** Muchos de los errores de la programación se deben a que el programa no libera la memoria que debería liberar, o se libera la misma memoria más de una vez. Java, hace recolección automática de basura, liberando la memoria y evitando la necesidad de que el programador se preocupe por liberar memoria que ya no utilice.

3.1.2. Diferencias entre Java y C++

Se compara mucho al lenguaje de Java con C++, y esto es lógico debido a la similitud en la sintaxis de ambos lenguajes, por lo que resulta necesario resaltar las diferencias principales que existen entre estos. Java a diferencia de C++:

- **No tiene aritmética de apuntadores.** Java si tiene apuntadores, sin embargo no permite la manipulación directa de las direcciones de memoria. No se proporcionan operadores para el manejo de los apuntadores pues no se considera responsabilidad del programador.

[2]Más adelante en el curso se ahondará en el tema.

- **No permite funciones con ámbito global.** Toda operación debe estar asociada a una clase, por lo que el ámbito de la función esta limitado al ámbito de la clase.
- **Elimina la instrucción *goto*.** La instrucción *goto* se considera obsoleta, ya que es una herencia de la época de la programación no estructurada. Sin embargo, esta definida como palabra reservada para restringir su uso.
- **Las cadenas no terminan con '\0'.** Las cadenas en C y C++ terminan con '\0 ' que corresponde al valor en ASCII 0, ya que en estos lenguajes no existe la cadena como un tipo de dato estándar y se construye a partir de arreglos de caracteres. En Java una cadena es un objeto de la clase *String* y no necesita ese carácter para indicar su finalización.
- **No maneja macros.** Una macro es declarada en C y C++ a través de la instrucción *define*, la cual es tratada por el preprocesador.
- **No soporta un preprocesador.** Una de las razones por las cuales no maneja macros Java es precisamente porque no tiene un preprocesador que prepare el programa previo a la compilación.
- **El tipo *char* contiene 16 bits**. Un carácter en Java utiliza 16 bits en lugar de 8 para poder soportar UNICODE en lugar de ASCII, lo que permite la representación de múltiples símbolos.
- **Java soporta múltiples hilos de ejecución**. Los múltiples hilos de ejecución o multihilos permiten un fácil manejo de programación concurrente. Otros lenguajes dependen de la plataforma para implementar concurrencia.
- **Todas las condiciones en Java deben tener como resultado un tipo booleano.** Debido a que en Java los resultados de las expresiones son dados bajo este tipo de dato. Mientras que en C y C++ se considera a un valor de cero como falso y no cero como verdadero.
- **Java no soporta el operador *sizeof*.** Este operador permite en C y C++ obtener el tamaño de una estructura de datos. En Java esto no es necesario ya que cada objeto "sabe" el espacio que ocupa en memoria.
- **No tiene herencia múltiple.** Java solo cuenta con herencia simple, con lo que pierde ciertas capacidades de generalización que son subsanadas a través del uso de interfaces. El equipo de desarrollo de Java explica que esto simplifica el lenguaje y evita la ambigüedad natural generada por la herencia múltiple.
- **No tiene liberación de memoria explícita (*delete* y *free()*).** En Java no es necesario liberar la memoria ocupada, ya que cuenta con un recolector de basura [3] responsable

[3] *Garbage Collector*

de ir liberando cada determinado tiempo los recursos de memoria que ya no se estén ocupando.

Además:

- No contiene estructuras y uniones (*struct* y *union*).
- No contiene tipos de datos sin signo.
- No permite alias (*typedef*).
- No tiene conversión automática de tipos compatibles.

3.1.3. Archivos .java y .class

En Java el código fuente se almacena en archivos con extensión .java, mientras que el *bytecode* o código compilado se almacena en archivos *.class*. El compilador de Java crea un archivo *.class* por cada declaración de clase que encuentra en el archivo *.java*.

Un archivo de código fuente debe tener solo una clase principal, y ésta debe tener exactamente el mismo nombre que el del archivo *.java*. Por ejemplo, si tengo una clase que se llama *Alumno*, el archivo de código fuente se llamará *Alumno.java*. Al compilar, el archivo resultante será *Alumno.class*.

3.1.4. Programas generados con java

Existen dos tipos principales de programas en Java [4]:

Por un lado están las aplicaciones, las cuales son programas *standalone*, escritos en Java y ejecutados por un intérprete del código de bytes desde la línea de comandos del sistema.

Por otra parte, los *Applets*, que son pequeñas aplicaciones escritas en Java, las cuales siguen un conjunto de convenciones que les permiten ejecutarse dentro de un navegador. Estos *applets* siempre están incrustados en una página *html*.

En términos del código fuente las diferencias entre un *applet* y una aplicación son:

- Una aplicación debe definir una clase que contenga el método *main()*, que controla su ejecución. Un *applet* no usa el método *main()*; su ejecución es controlado por varios métodos definidos en la clase *applet*.
- Un *applet*, debe definir una clase derivada de la clase *Applet* [5].

[4] Se presenta la división clásica de los programas de Java, aunque existen algunas otras opciones no son relevantes en este curso.

[5] A partir de la versión 1.9 del jdk, los applets ya no son soportados.

3.1.5. El Java Developer's Kit

La herramienta básica para empezar a desarrollar aplicaciones en Java es el JDK (*Java Development Kit*) o Kit de Desarrollo Java, que consiste esencialmente, en un compilador y un intérprete (JVM[6]) para la línea de comandos. No dispone de un entorno de desarrollo integrado (IDE), pero es suficiente para aprender el lenguaje y desarrollar pequeñas aplicaciones[7].

Los principales programas del *Java Development Kit*:

- **javac**. Es el compilador en línea del JDK.
- **java**. Es la máquina virtual para aplicaciones de Java.
- **appletviewer**. Visor de *applets* de java.

Compilación

Utilizando el JDK, los programas se compilan desde el símbolo del sistema con el compilador javac.

Ejemplo:

```
C:\MisProgramas> javac MiClase.java
```

Normalmente se compila como se ha mostrado en el ejemplo anterior. Sin embargo, el compilador proporciona diversas opciones a través de modificadores que se agregan en la línea de comandos.

Sintaxis
`javac [opciones] <archivo1.java>`

donde opciones puede ser:

- $-classpath < ruta >$ Indica donde buscar los archivos de clase de Java
- $-d < directorio >$ Indica el directorio destino para los archivos .class
- $-g$ Habilita la generación de tablas de depuración.
- $-nowarn$ Deshabilita los mensajes del compilador.
- $-O$ Optimiza el código, generando en línea los métodos estáticos, finales y privados.
- $-verbose$ Indica cuál archivo fuente se esta compilando.

[6] *Java Virtual Machine*

[7] Este kit de desarrollo es gratuito y puede obtenerse de la dirección proporcionada al final de este documento. Independientemente del IDE que se use, el jdk debe estar instalado para poder compilar código Java.

3.1.6. "Hola Mundo"

Para no ir en contra de la tradición al comenzar a utilizar un lenguaje, los primeros ejemplos son precisamente dos programas muy simples que lo único que van a hacer es desplegar el mensaje "Hola Mundo".

Hola mundo básico en Java El primero es una aplicación que va a ser interpretado posteriormente por la máquina virtual:

```
1 public class HolaMundo {
2     public static void main(String args[]) {
3         System.out.println("¡Hola, Mundo!");
4     }
5 }
```

Listing 16. Hola, Mundo en Java.

Para los que han programado en C ó C++, notarán ya ciertas similitudes. Lo importante aquí es que una aplicación siempre requiere de un método *main*, este tiene un solo argumento (*String args[ ]*), a través del cual recibe información de los argumentos de la línea de comandos, pero la diferencia con los lenguajes C/C++ es que este método depende de una clase, en este caso la clase *HolaMundo*. Este programa es compilado en al jdk[8]:

```
%javac HolaMundo.java
```

con lo que, si el programa no manda errores, se obtendrá el archivo *HolaMundo.class*.

En Eclipse, al grabar automáticamente el programa se compilará (si la opción *Build Automatically* está activada). De hecho, algunos errores se van notificando, si los hay, conforme se va escribiendo el código en el editor.

Hola mundo básico en C++

En C++ no estamos obligados a usar clases, por lo que un "Hola mundo" en C++ - aunque no en objetos – podría quedar de la siguiente forma:

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main(){
```

[8] Se asume que el jdk se encuentra instalado y que el PATH tiene indicado el directorio bin del jdk para que encuentre el programa javac. También es recomendable añadir nuestro directorio de programas de java a una variable de ambiente llamada CLASSPATH.

```
6       cout << "Hola Mundo!" << endl;
7       return 0;
8   }
```

Listing 17. Hola, Mundo en C++.

Ejecución

Para ejecutar una aplicación usamos la máquina virtual proporcionada por el *jdk*, proporcionando el nombre de la clase:

```
% java HolaMundo
```

A partir de la versión 11 del jdk[9], podemos ejecutar directamente ejemplos pequeños de código java. Se compila y ejecuta, sin generar el código *.class* correspondiente:

```
% java HolaMundo.java
```

3.2. Fundamentos del Lenguaje Java

En esta sección se hablará de cómo está constituido el lenguaje, sus instrucciones, tipos de datos, entre otras características. Antes de comenzar a hacer programación orientada a objetos.

3.2.1. Comentarios

Los comentarios en los programas fuente son muy importantes en cualquier lenguaje. Sirven para aumentar la facilidad de comprensión del código y para recordar ciertas cosas sobre el mismo. Son porciones del programa fuente que el compilador omite, y, por tanto, no ocuparán espacio en el archivo de clase.

Existen tres tipos de comentarios en Java:

- Si el comentario que se desea escribir es de una sola línea, basta con poner dos barras inclinadas //. Por ejemplo:

```
1  for (i=0; i<20;i++) // comentario de ciclo {
2          System.out.println(\Adiós");
3  }
```

No puede ponerse código después de un comentario introducido por // en la misma línea, ya que desde la aparición de las dos barras inclinadas // hasta el final de la línea es considerado como comentario e ignorado por el compilador.

[9] Ver: https://www.codejava.net/java-core/tools/run-a-java-program-directly-from-source-code-file

- Si un comentario debe ocupar más de una línea, hay que anteponerle /* y al final */. Por ejemplo:

```
1 /* Esto es un
2 comentario que
3 ocupa tres líneas */
```

- Existe otro tipo de comentario que sirve para generar documentación automáticamente en formato HTML mediante la herramienta javadoc. Puede ocupar varias líneas y se inicia con /** para terminar con */. Para mas información ver: http://java.sun.com/j2se/javadoc/

3.2.2. Tipos de datos

En Java existen dos tipos principales de datos:

1. Tipos de datos **simples**.
2. **Referencias** a objetos.

Los tipos de datos simples son aquellos que pueden utilizarse directamente en un programa, sin necesidad del uso de clases. Estos tipos son:

- byte
- short
- int
- long
- float
- double
- char
- boolean

El segundo tipo está formado por todos los demás. Se les llama referencias porque en realidad lo que se almacena en los mismos son punteros a áreas de memoria donde se encuentran almacenadas las estructuras de datos que los soportan. Dentro de este grupo se encuentran las clases (objetos) y también se incluyen las interfaces, los vectores y las cadenas o *Strings*.

Pueden realizarse conversiones entre los distintos tipos de datos (incluso entre simples y referenciales), bien de forma implícita o de forma explícita.

Tipo	Descripción	Formato	Longitud	Rango
byte	byte	C-2[10]	1 byte	- 128 ... 127
short	entero corto	C-2	2 bytes	- 32.768 ... 32.767
int	entero	C-2	4 bytes	- 2.147.483.648 ... 2.147.483.647
long	entero largo	C-2	8 bytes	-9.223.372.036.854.775.808 ... 9.223.372.036.854.775.807
float	real en coma flotante de precisión simple	IEEE 754	32 bits	$3,4*10_{-38}$... $3,4*10_{38}$
double	real en coma flotante de precisión doble	IEEE 754	64 bits	$1,7*10_{-308}$... $1,7*10_{308}$
char	Carácter	Unicode	2 bytes	0 ... 65.535
boolean	Lógico		1 bit	true / false

Cuadro 3.1. Tipos de datos simples en Java

Tipos de datos simples

Los tipos de datos simples en Java tienen las siguientes características:

No existen más datos simples en Java. Incluso éstos que se enumeran pueden ser remplazados por clases equivalentes (*Integer*, *Double*, *Byte*, etc.), con la ventaja de que es posible tratarlos como si fueran objetos en lugar de datos simples.

A diferencia de otros lenguajes de programación como C, en Java los tipos de datos simples no dependen de la plataforma ni del sistema operativo. Un entero de tipo *int* siempre tendrá 4 bytes, por lo que no tendremos resultados inesperados al migrar un programa de un sistema operativo a otro.

Eso sí, Java no realiza una comprobación de los rangos. Por ejemplo: si a una variable de tipo *short* con el valor 32.767 se le suma 1, el resultado será -32.768 y no se producirá ningún error de ejecución.

Los valores que pueden asignarse a variables y que pueden ser utilizados en expresiones directamente reciben el nombre de literales.

Referencias a objetos

El resto de tipos de datos que no son simples, son considerados referencias. Estos tipos son básicamente apuntadores a las instancias de las clases, en las que se basa la programación orientada a objetos.

Al declarar una variable de objeto perteneciente a una determinada clase, se indica que ese identificador de referencia tiene la capacidad de apuntar a un objeto del tipo al que pertenece la variable. El momento en el que se realiza la reserva física del espacio de memoria es cuando se instancia el objeto realizando la llamada a su constructor, y no en el momento de la declaración.

Existe un tipo referencial especial nominado por la palabra reservada *null* que puede ser asignado a cualquier variable de cualquier clase y que indica que el puntero no tiene referencia a ninguna zona de memoria (el objeto no está inicializado).

3.2.3. Identificadores

Los identificadores son los nombres que se les da a las variables, clases, interfaces, atributos y métodos de un programa.

Existen algunas reglas básicas para nombrar a los identificadores:

1. Java hace distinción entre mayúsculas y minúsculas, por lo tanto, nombres o identificadores como *var1*, *Var1* y *VAR1* son distintos.

2. Pueden estar formados por cualquiera de los caracteres del código *Unicode*, por lo tanto, se pueden declarar variables con el nombre: *añoDeCreación*, *raïm*, etc.

3. El primer carácter no puede ser un dígito numérico y no pueden utilizarse espacios en blanco ni símbolos coincidentes con operadores.

4. No puede ser una palabra reservada del lenguaje ni los valores lógicos *true* o *false*.

5. No pueden ser iguales a otro identificador declarado en el mismo ámbito.

6. Por convención, los nombres de las variables y los métodos deberían empezar por una letra minúscula y los de las clases por mayúscula.

Además, si el identificador está formado por varias palabras, la primera se escribe en minúsculas (excepto para las clases e interfaces) y el resto de palabras se hace empezar por mayúscula (por ejemplo: *añoDeCreación*). Las constantes se escriben en mayúsculas, por ejemplo *MÁXIMO*.

Esta última regla no es obligatoria, pero es conveniente ya que ayuda al proceso de codificación de un programa, así como a su legibilidad. Es más sencillo distinguir entre clases y métodos, variables o constantes.

3.2.4. Variables

La declaración de una variable se realiza de la misma forma que en C/C++. Siempre contiene el nombre (identificador de la variable) y el tipo de dato al que pertenece. El ámbito de la variable depende de la localización en el programa donde es declarada.

Ejemplo:

```
int x;
```

Las variables pueden ser inicializadas en el momento de su declaración, siempre que el valor que se les asigne coincida con el tipo de dato de la variable.

Ejemplo:

```
int x = 0;
```

Ámbito de una variable

El ámbito de una variable es la porción de programa donde dicha variable es visible para el código del programa y, por tanto, referenciable. El ámbito de una variable depende del lugar del programa donde es declarada, pudiendo pertenecer a cuatro categorías distintas.

- Variable local.
- Atributo.
- Parámetro de un método.
- Parámetro de un manejador de excepciones[11].

Como puede observarse, no existen las variables globales. La utilización de variables globales es considerada peligrosa, ya que podría ser modificada en cualquier parte del programa y por cualquier procedimiento. A la hora de utilizarlas hay que buscar dónde están declaradas para conocerlas y dónde son modificadas para evitar sorpresas en los valores que pueden contener.

Los ámbitos de las variables u objetos en Java siguen los criterios "clásicos", al igual que en la mayoría de los lenguajes de programación como Pascal, C++, etc.

Si una variable que **no es local** no ha sido inicializada, tiene un valor asignado por defecto. Este valor es, para las variables referencias a objetos, el valor *null*. Para las variables de tipo numérico, el valor por defecto es cero, las variables de tipo *char*, el valor ' u0000' y las variables de tipo *boolean*, el valor *false*.

Variables locales Una variable local se declara dentro del cuerpo de un método de una clase y es visible únicamente dentro de dicho método. Se puede declarar en cualquier lugar del cuerpo, incluso después de instrucciones ejecutables, aunque es una buena costumbre declararlas justo al principio.

3.2.5. Operadores

Los operadores son partes indispensables en la construcción de expresiones. Existen muchas definiciones técnicas para el término expresión. Puede decirse que una expresión es una combinación de operandos ligados mediante operadores.

Los operandos pueden ser variables, constantes, funciones, literales, etc. y los operadores se comentarán a continuación.

Operadores aritméticos:

El operador - puede utilizarse en su versión unaria (- op1) y la operación que realiza es la de invertir el signo del operando.

[11] Este se tocará en otra etapa del curso, al hablar de manejo de excepciones.

Operador	Formato	Descripción
+	op1 + op2	Suma aritmética de dos operandos
-	op1 - op2	Resta aritmética de dos operandos
-op1		Cambio de signo
*	op1 * op2	Multiplicación de dos operandos
/	op1 / op2	División entera de dos operandos
%	op1 % op2	Resto de la división entera (o módulo)
++	++op1	Incremento unitario
	op1++	
- -	- -op1	Decremento unitario
	op1- -	

Cuadro 3.2. Operadores aritméticos Java

Como en C/C++, los operadores unarios ++ y – realizan un incremento y un decremento respectivamente. Estos operadores admiten notación prefija y postfija. Ver Cuadro 3.2

- ++op1: En primer lugar realiza un incremento (en una unidad) de *op1* y después ejecuta la instrucción en la cual está inmerso.
- op1++: En primer lugar ejecuta la instrucción en la cual está inmerso y después realiza un incremento (en una unidad) de *op1*.
- –op1: En primer lugar realiza un decremento (en una unidad) de *op1* y después ejecuta la instrucción en la cual está inmerso. Visión General y elementos básicos del lenguaje.
- op1–: En primer lugar ejecuta la instrucción en la cual está inmerso y después realiza un decremento (en una unidad) de *op1*.

La diferencia entre la notación prefija y la postfija no tiene importancia en expresiones en las que únicamente existe dicha operación:

```
1 ++contador; //es equivalente a: contador++;
2 --contador; //contador--;
```

La diferencia es apreciable en instrucciones en las cuáles están incluidas otras operaciones.

Ejemplo:

Operador	Formato	Descripción
>	op1 > op2	Devuelve *true* si op1 es mayor que op2
<	op1 < op2	Devuelve *true* si op1 es menor que op2
>=	op1 >= op2	Devuelve *true* si op1 es mayor o igual que op2
<=	op1 <= op2	Devuelve *true* si op1 es menor o igual que op2
==	op1 == op2	Devuelve *true* si op1 es igual a op2
!=	op1 != op2	Devuelve *true* si op1 es distinto de op2

Cuadro 3.3. Operadores relacionales en Java

```
a = 1; a = 1;
b = 2 + a++; b = 2 + ++a;
```

En el primer caso, después de las operaciones, b tendrá el valor 3 y al valor 2. En el segundo caso, después de las operaciones, b tendrá el valor 4 y al valor 2.

Operadores relacionales:

Los operadores relacionales actúan sobre valores enteros, reales y caracteres; y devuelven un valor del tipo booleano (*true* o *false*). Ver Cuadro 3.3

Ejemplo:

```
public class Relacional {
        public static void main(String arg[]) {
                double op1,op2;
                op1=1.34;
                op2=1.35;
                System.out.println("op1="+op1+" op2="+op2);
                System.out.println("op1>op2 = "+(op1>op2));
                System.out.println("op1<op2 = "+(op1<op2));
                System.out.println("op1==op2 = "+(op1==op2));
                System.out.println("op1!=op2 = "+(op1!=op2));
                char op3,op4;
                op3='a'; op4='b';
                System.out.println("'a'>'b' = "+(op3>op4));
        }
}
```

Listing 18. Ejemplo de operadores relacionales en Java.

Operador	Formato	Descripción
&&	op1 && op2	Y lógico. Devuelve *true* si son ciertos op1 y op2
||	op1 || op2	O lógico. Devuelve *true* si son ciertos op1 o op2
!	!op1	Negación lógica. Devuelve *true* si es falso op1.

Cuadro 3.4. Operadores lógicos en Java

Operadores lógicos:

Estos operadores actúan sobre operadores o expresiones lógicas, es decir, aquellos que se evalúan a cierto o falso. Ver Cuadro 3.4

Ejemplo:

```
1  public class Bool {
2          public static void main ( String argumentos[] ) {
3                  boolean a=true;
4                  boolean b=true;
5                  boolean c=false;
6                  boolean d=false;
7                  System.out.println("true Y true = " + (a && b) );
8                  System.out.println("true Y false = " + (a && c) );
9                  System.out.println("false Y false = " + (c && d) );
10                 System.out.println("true O true = " + (a || b) );
11                 System.out.println("true O false = " + (a || c) );
12                 System.out.println("false O false = " + (c || d) );
13                 System.out.println("NO true = " + !a);
14                 System.out.println("NO false = " + !c);
15                 System.out.println("(3 > 4) Y true = " + ((3 >4) && a) );
16         }
17 }
```

Listing 19. Ejemplo de operadores lógicos en Java.

Operadores de asignación:

El operador de asignación es el símbolo igual (=).

```
op1 = Expresión;
```

Asigna el resultado de evaluar la expresión de la derecha a *op1*.

Además del operador de asignación existen unas abreviaturas, como en C/C++, cuando el operando que aparece a la izquierda del símbolo de asignación también aparece a la derecha del mismo. Ver Cuadro 3.5

Operador	Formato	Equivalencia
$+=$	$op1+=op2$	$op1=op1+op2$
$-=$	$op1-=op2$	$op1=op1-op2$
$*=$	$op1*=op2$	$op1=op1*op2$
$/=$	$op1/=op2$	$op1=op1/op2$
$\%=$	$op1\%=op2$	$op1=op1\%op2$

Cuadro 3.5. Operadores de asignación en Java

Precedencia de operadores en Java

La precedencia (ver Cuadro 3.6)indica el orden en que es resuelta una expresión, la siguiente lista muestra primero los operadores de mayor precedencia.

3.2.6. Valores literales

A la hora de tratar con valores de los tipos de datos simples (y *Strings*) se utiliza lo que se denomina "literales". Los literales son elementos que sirven para representar un valor en el código fuente del programa.

En Java existen literales para los siguientes tipos de datos:

- Lógicos (*boolean*).
- Carácter (*char*).
- Enteros (*byte*, *short*, *int* y *long*).
- Reales (*double* y *float*).
- Cadenas de caracteres (*String*).

Literales lógicos

Son únicamente dos: las palabras reservadas *true* y *false*.

Ejemplo:

```
boolean activado = false;
```

Tipo de operador	Operadores
Operadores postfijos	$[].(paréntesis)$
Operadores unarios	$++expr\ --expr - expr\ !$
Creación o conversión de tipo	$new(tipo)expr$
Multiplicación y división	$*/\%$
Suma y resta	$+-$
Desplazamiento de bits	$<<>>>>>$
Relacionales	$<><=>=$
Igualdad y desigualdad	$==!=$
AND a nivel de bits	&
OR a nivel de bits	$\vert$
AND lógico	&&
OR lógico	$\vert\vert$
Condicional terciaria	? :
Asignación	$=+=-=*=/=\%=\&=\vert=>>=<<=>>>=$

Cuadro 3.6. Precedencia de operadores en Java

Literales de tipo entero

Son *byte*, *short*, *int* y *long* pueden expresarse en decimal (base 10), octal (base 8) o hexadecimal (base 16). Además, puede añadirse al final del mismo la letra L para indicar que el entero es considerado como *long* (64 bits).

Literales de tipo real

Los literales de tipo real sirven para indicar valores float o double. A diferencia de los literales de tipo entero, no pueden expresarse en octal o hexadecimal.

Existen dos formatos de representación: mediante su parte entera, el punto decimal (.) y la parte fraccionaria; o mediante notación exponencial o científica:

Ejemplos equivalentes:

```
3.1415
0.31415e1
.31415e1
0.031415E+2
.031415e2
314.15e-2
31415E-4
```

Al igual que los literales que representan enteros, se puede poner una letra como sufijo. Esta letra puede ser una F o una D (mayúscula o minúscula indistintamente).

Secuencia de escape	Significado
\’	Comilla simple.
\”	Comillas dobles.
\\	Barra invertida.
\b	Backspace (Borrar hacia atrás).
\n	Cambio de línea.
\f	Form feed.
\r	Retorno de carro.
\t	Tabulador.

Cuadro 3.7. Secuencias de escape en Java

- F Trata el literal como de tipo *float*.
- D Trata el literal como de tipo *double*.

Ejemplo:

```
3.1415F
.031415d
```

Literales de tipo carácter

Los literales de tipo carácter se representan siempre entre comillas simples. Entre las comillas simples puede aparecer:

- Un símbolo (letra) siempre que el carácter esté asociado a un código *Unicode*. Ejemplos: ‘a’ , ‘B’ , ‘{‘ , ‘ñ’ , ‘á’ .
- Una “secuencia de escape”. Las secuencias de escape son combinaciones del símbolo seguido de una letra, y sirven para representar caracteres que no tienen una equivalencia en forma de símbolo.

Las posibles secuencias de escape se pueden ver en el Cuadro 3.7

Literales de tipo *String*

Los *Strings* o cadenas de caracteres no forman parte de los tipos de datos elementales en Java, sino que son instanciados a partir de la clase *java.lang.String*, pero aceptan su inicialización a partir de literales de este tipo.

Un literal de tipo *String* va encerrado entre comillas dobles (") y debe estar incluido completamente en una sola línea del programa fuente (no puede dividirse en varias líneas). Entre las comillas dobles puede incluirse cualquier carácter del código *Unicode* (o su código precedido del carácter \) además de las secuencias de escape vistas anteriormente en los literales de tipo carácter. Así, por ejemplo, para incluir un cambio de línea dentro de un literal de tipo *String* deberá hacerse mediante la secuencia de escape \n :

Ejemplo:

```
1 System.out.println("Primera línea \n Segunda línea del string\n");
2 System.out.println("Hol\u0061");
```

La visualización del *String* anterior mediante *println()* produciría la siguiente salida por pantalla:

```
Primera línea
Segunda línea del string
Hola
```

La forma de incluir los caracteres: comillas dobles (") y barra invertida (\) es mediante las secuencias de escape \" y \\ respectivamente (o mediante su código *Unicode* precedido de \).

Si la cadena es demasiado larga y debe dividirse en varias líneas en el código fuente, o simplemente concatenar varias cadenas, puede utilizarse el operador de concatenación de *strings* + .de la siguiente forma:

```
''Este String es demasiado largo para estar en una línea'' +
''del código fuente y se ha dividido en dos.''
```

3.2.7. Estructuras de control

Las estructuras de control son construcciones definidas a partir de palabras reservadas del lenguaje que permiten modificar el flujo de ejecución de un programa. De este modo, pueden crearse construcciones de decisión y ciclos de repetición de bloques de instrucciones.

Hay que señalar que, como en C/C++, un bloque de instrucciones se encontrará encerrado mediante llaves {.....} si existe más de una instrucción.

Estructuras condicionales

Las estructuras condicionales o de decisión son construcciones que permiten alterar el flujo secuencial de un programa, de forma que en función de una condición o el valor de una expresión, el mismo pueda ser desviado en una u otra alternativa de código.

Las estructuras condicionales disponibles en Java son:

- Estructura if-else.
- Estructura switch.

if-else

Sintaxis

Forma simple:

```
if (<expresión>)
    <Bloque instrucciones>
```

El bloque de instrucciones se ejecuta si, y sólo si, la expresión (que debe ser lógica) se evalúa a verdadero, es decir, se cumple una determinada condición.

Ejemplo:

```
1 if (cont == 0)
2         System.out.println("he llegado a cero");
```

La instrucción *System.out.println("he llegado a cero");* sólo se ejecuta en el caso de que *cont* contenga el valor cero.

Sintaxis

Forma bicondicional:

```
if (<expresión>)
    <Bloque instrucciones 1>
else
        <Bloque instrucciones 2>
```

El bloque de instrucciones 1 se ejecuta si, y sólo si, la expresión se evalúa como verdadero. Y en caso contrario, si la expresión se evalúa como falso, se ejecuta el bloque de instrucciones 2.

Ejemplo:

```
if (cont == 0)
        System.out.println("he llegado a cero");
else
        System.out.println("no he llegado a cero");
```

En Java, como en C/C++ y a diferencia de otros lenguajes de programación, en el caso de que el bloque de instrucciones conste de una sola instrucción no necesita ser encerrado en un bloque.

switch

Sintaxis

```
switch (<expresión>) {
case <valor1>: <instrucciones1>;
case <valor2>: <instrucciones2>;
...
case <valorN>: <instruccionesN>;
}
```

En este caso, a diferencia del *if*, si $< instrucciones1 >$, $< instrucciones2 >$ ó $< instruccionesN >$ están formados por un bloque de instrucciones sencillas, no es necesario encerrarlas mediante las llaves (...).

En primer lugar se evalúa la expresión cuyo resultado puede ser un valor de cualquier tipo. El programa comprueba el primer valor (*valor1*). En el caso de que el valor resultado de la expresión coincida con *valor1*, se ejecutará el bloque $< instrucciones1 >$. Pero también se ejecutarían el bloque $< instrucciones2 >$ ... $< instruccionesN >$ hasta encontrarse con la palabra reservada *break*. Por lo que comúnmente se añade una instrucción *break* al final de cada caso del *switch*.

Ejemplo:

```
switch (<expresión>) {

        case <valor1>: <instrucciones1>;
                                break;
        case <valor2>: <instrucciones2>;
```

```
6                                   break;
7           ...
8           case <valorN>: <instruccionesN>;
9   }
```

Si el resultado de la expresión no coincide con $< valor1 >$, evidentemente no se ejecutarían $< instrucciones1 >$, se comprobaría la coincidencia con $< valor2 >$ y así sucesivamente hasta encontrar un valor que coincida o llegar al final de la construcción *switch*. En caso de que no exista ningún valor que coincida con el de la expresión, no se ejecuta ninguna acción.

Ejemplo:

```
1   public class DiaSemana {
2           public static void main(String argumentos[]) {
3                   int dia;
4                   if (argumentos.length<1) {
5                           System.out.println("Uso: DiaSemana num");
6                           System.out.println("Donde num= nº entre 1 y 7");
7                   }
8                   else {
9                           dia=Integer.valueOf(argumentos[0]);
10                          // también: dia=Integer.parseInt(argumentos[0]);
11                          switch (dia) {
12                          case 1: System.out.println("Lunes");
13                          break;
14                          case 2: System.out.println("Martes");
15                          break;
16                          case 3: System.out.println("Miércoles");
17                          break;
18                          case 4: System.out.println("Jueves");
19                          break;
20                          case 5: System.out.println("Viernes");
21                          break;
22                          case 6: System.out.println("Sábado");
23                          break;
24                          case 7: System.out.println("Domingo");
25                          }
26                  }
27          }
28  }
```

Listing 20. Ejemplo de uso de switch en Java.

Nótese que en el caso de que se introduzca un valor no comprendido entre 1 y 7, no se realizará ninguna acción. Esto puede corregirse agregando la opción por omisión:

```
default: instruccionesPorDefecto;
```

donde la palabra reservada *default*, sustituye a *case ¡expr¿* para ejecutar el conjunto de instrucciones definido en caso de que no coincida con ningún otro caso.

Ciclos

Los ciclos o iteraciones son estructuras de repetición. Bloques de instrucciones que se repiten un número de veces **mientras** se cumpla una condición o **hasta** que se cumpla una condición.

Existen tres construcciones para estas estructuras de repetición:

- Ciclo for.
- Ciclo do-while.
- Ciclo while.

Como regla general puede decirse que se utilizará el ciclo for cuando se conozca de antemano el número exacto de veces que ha de repetirse un determinado bloque de instrucciones. Se utilizará el ciclo *do-while* cuando no se conoce exactamente el número de veces que se ejecutará el ciclo pero se sabe que por lo menos se ha de ejecutar una. Se utilizará el ciclo *while* cuando es posible que no deba ejecutarse ninguna vez. Con mayor o menor esfuerzo, puede utilizarse cualquiera de ellas indistintamente.

Ciclo for

Sintaxis
`for (<inicialización> ; <condición> ; <incremento>)` `  <bloque instrucciones>`

- La cláusula inicialización es una instrucción que se ejecuta una sola vez al inicio del ciclo, normalmente para inicializar un contador.
- La cláusula condición es una expresión lógica, que se evalúa al inicio de cada nueva iteración del ciclo. En el momento en que dicha expresión se evalúe a falso, se dejará de ejecutar el ciclo y el control del programa pasará a la siguiente instrucción (a continuación del ciclo *for*).

- La cláusula incremento es una instrucción que se ejecuta en cada iteración del ciclo como si fuera la última instrucción dentro del bloque de instrucciones. Generalmente se trata de una instrucción de incremento o decremento de alguna variable.

Cualquiera de estas tres cláusulas puede estar vacía, aunque siempre hay que poner los puntos y coma (;).

El siguiente programa muestra en pantalla la serie de *Fibonacci* hasta el término que se indique al programa como argumento en la línea de comandos. Siempre se mostrarán, por lo menos, los dos primeros términos

Ejemplo:

```
1  // siempre se mostrarán, por lo menos, los dos primeros //términos
2  public class Fibonacci {
3          public static void main(String argumentos[]) {
4                  int numTerm,v1=1,v2=1,aux,cont;
5                  if (argumentos.length<1) {
6                          System.out.println("Uso: Fibonacci num");
7                          System.out.println("Donde num = nº de términos");
8                  }
9                  else {
10                         numTerm=Integer.valueOf(argumentos[0]);
11                         System.out.print("1,1");
12                         for (cont=2;cont<numTerm;cont++) {
13                                 aux=v2;
14                                 v2+=v1;
15                                 v1=aux;
16                                 System.out.print(","+v2);
17                         }
18                         System.out.println();
19                 }
20         }
21 }
```

Listing 21. Ejemplo Fibonacci con ciclo *for*.

Ciclo do-while

Sintaxis

```
do
        <bloque instrucciones>
while (<Expresión>);
```

En este tipo de ciclo, el bloque instrucciones se ejecuta siempre una vez por lo menos, y el bloque de instrucciones se ejecutará mientras $< Expresión >$ se evalúe como verdadero. Por lo tanto, entre las instrucciones que se repiten deberá existir alguna que, en algún momento, haga que la expresión se evalúe como falso, de lo contrario el ciclo sería infinito.

Ejemplo:

```
1  //El mismo que antes (Fibonacci).
2  public class Fibonacci2 {
3          public static void main(String argumentos[]) {
4                  int numTerm,v1=0,v2=1,aux,cont=1;
5                  if (argumentos.length<1) {
6                          System.out.println("Uso: Fibonacci num");
7                          System.out.println("Donde num = nº de términos");
8                  }
9                  else {
10                         numTerm=Integer.valueOf(argumentos[0]);
11                         System.out.print("1");
12                         do {
13                                 aux=v2;
14                                 v2+=v1;
15                                 v1=aux;
16                                 System.out.print(","+v2);
17                         } while (++cont<numTerm);
18                         System.out.println();
19                 }
20         }
21 }
```

Listing 22. Ejemplo Fibonacci con ciclo *do-while*.

En este caso únicamente se muestra el primer término de la serie antes de iniciar el ciclo, ya que el segundo siempre se mostrará, porque el ciclo do-while siempre se ejecuta una vez por lo menos.

Ciclo while

Sintaxis

```
while (<Expresión>)
    <bloque instrucciones>
```

Al igual que en el ciclo *do-while* del apartado anterior, el bloque de instrucciones se ejecuta mientras se cumple una condición (mientras *Expresión* se evalúe verdadero), pero en este caso, la condición se comprueba antes de empezar a ejecutar por primera vez el ciclo, por lo que si *Expresión* se evalúa como falso en la primera iteración, entonces el bloque de instrucciones no se ejecutará ninguna vez.

Ejemplo:

```
1  //Fibonacci:
2  public class Fibonacci3 {
3          public static void main(String argumentos[]) {
4                  int numTerm,v1=1,v2=1,aux,cont=2;
5                  if (argumentos.length<1) {
6                          System.out.println("Uso: Fibonacci num");
7                          System.out.println("Donde num = nº de términos");
8                  }
9                  else {
10                         numTerm=Integer.valueOf(argumentos[0]);
11                         System.out.print("1,1");
12                         while (cont++<numTerm) {
13                                 aux=v2;
14                                 v2+=v1;
15                                 v1=aux;
16                                 System.out.print(","+v2);
17                         }
18                         System.out.println();
19                 }
20         }
21 }
```

Listing 23. Ejemplo Fibonacci con ciclo *while*.

Como puede comprobarse, las tres construcciones de ciclo (*for*, *do-while* y *while*) pueden utilizarse indistintamente realizando unas pequeñas variaciones en el programa.

Saltos

En Java existen dos formas de realizar un salto incondicional en el flujo normal de un programa: las instrucciones *break* y *continue*.

break La instrucción break sirve para abandonar una estructura de control, tanto de la alternativa (*switch*) como de las repetitivas o ciclos (*for*, *do-while* y *while*). En el momento que se ejecuta la instrucción *break*, el control del programa sale de la estructura en la que se encuentra.

Ejemplo:

```
1  public class Break {
2          public static void main(String argumentos[]) {
3                  int i;
4                  for (i=1; i<=4; i++) {
5                          if (i==3)
6                                  break;
7                          System.out.println("Iteracion: "+i);
8                  }
9          }
10 }
```

Listing 24. Ejemplo de uso de *break*.

Aunque el ciclo, en principio indica que se ejecute 4 veces, en la tercera iteración, i contiene el valor 3, se cumple la condición de *i==3* y por lo tanto se ejecuta el *break* y se sale del ciclo *for*.

continue La instrucción *continue* sirve para transferir el control del programa desde la instrucción *continue* directamente a la cabecera del ciclo (*for*, *do-while* o *while*) donde se encuentra.

Ejemplo:

```
1  public class Continue {
2          public static void main(String argumentos[]) {
3                  int i;
4                  for (i=1; i<=4; i++) {
5                          if (i==3)
6                                  continue;
7                          System.out.println("Itereación: "+i);
8                  }
9          }
10 }
```

Listing 25. Ejemplo de uso de *continue*.

Puede comprobarse la diferencia con respecto al resultado del ejemplo del apartado anterior. En este caso no se abandona el ciclo, sino que se transfiere el control a la cabecera del ciclo donde se continúa con la siguiente iteración.

Tanto el salto *break* como en el salto *continue*, pueden ser evitados mediante distintas construcciones pero en ocasiones esto puede empeorar la legibilidad del código. De todas formas existen programadores que no aceptan este tipo de saltos y no los utilizan en ningún caso; la razón es que - se dice - que atenta contra las normas de las estructuras de control.

3.2.8. Arreglos

Para manejar colecciones de objetos del mismo tipo estructurados en una sola variable se utilizan los arreglos.

En Java, los arreglos son en realidad objetos y por lo tanto se puede llamar a sus métodos. Existen dos formas equivalentes de declarar arreglos en Java:

```
tipo nombreDelArreglo[  ];
```

o

```
tipo[  ] nombreDelArreglo;
```

Ejemplo:

```
1 int arreglo1[], arreglo2[], entero; //entero no es un arreglo
2 int[] otroArreglo;
```

También pueden utilizarse arreglos de más de una dimensión:
Ejemplo:

```
1 int matriz[][];
2 int [][] otraMatriz;
```

Los arreglos, al igual que las demás variables pueden ser inicializados en el momento de su declaración. En este caso, no es necesario especificar el número de elementos máximo reservado. Se reserva el espacio justo para almacenar los elementos añadidos en la declaración.

Ejemplo:

```
1 String Días[]={"Lunes","Martes","Miércoles","Jueves",
2                "Viernes","Sábado","Domingo"};
```

Una simple declaración de un vector no reserva espacio en memoria, a excepción del caso anterior, en el que sus elementos obtienen la memoria necesaria para ser almacenados. Para reservar la memoria hay que llamar explícitamente a *new* de la siguiente forma:

```
new tipoElemento[ <numElementos> ];
```

Ejemplo:

```
1 int matriz[][];
2 matriz = new int[4][7];
```

Dos lineas diferentes

También se puede indicar el número de elementos durante su declaración:
Ejemplo:

```
int vector[] = new int[5];
```

Para hacer referencia a los elementos particulares del arreglo, se utiliza el identificador del arreglo junto con el índice del elemento entre corchetes. El índice del primer elemento es el cero y el del último, el número de elementos menos uno.
Ejemplo:

```
j = vector[0]; vector[4] = matriz[2][3];
```

NOTA: Java va más por seguridad por lo tanto hay una verificación, a diferencia de C donde se va más por eficiencia

El intento de acceder a un elemento fuera del rango del arreglo, a diferencia de lo que ocurre en C, provoca una excepción (error) que, de no ser manejado por el programa, será la máquina virtual quien aborte la operación.

Para obtener el número de elementos de un arreglo en tiempo de ejecución se accede al atributo de la clase llamado *length*. No olvidemos que los arreglos en Java son tratados como un objeto.

Ejemplo:

```
1 public class Array1 {
2         public static void main (String argumentos[]) {
3                 String colores[] = {"Rojo","Verde","Azul",
4                                 "Amarillo","Negro"};
5                 int i;
6                 for (i=0;i<colores.length;i++)
7                         System.out.println(colores[i]);
8         }
9 }
```

Listing 26. Ejemplo de uso de arreglo.

Usando al menos Java 5.0 (jdk 1.5) podemos simplificar el recorrido del arreglo:

```
1  public class Meses {
2
3          public static void main(String[] args) {
4              String meses[] =
5                 {"Enero", "Febrero", "Marzo", "Abril", "Mayo", "Junio", "Julio",
6                 "Agosto", "Septiembre", "Octubre", "Noviembre", "Diciembre"};
7
8              //for(int i = 0; i < meses.length; i++ )
9              //   System.out.println("mes: " + meses[i]);
10
11             // sintaxis para recorrer el arreglo y asignar
12             // el siguiente elemento a la variable mes en cada ciclo
13             // instruccion "for each" a partir de version 5.0 (1.5 del jdk)
14             for(String mes: meses)
15                System.out.println("mes: " + mes);   Se ahorra la variable de iteración
16
17         }
18
19 }
```

Listing 27. Ejemplo de uso de arreglo 2.

3.2.9. Enumeraciones

Java desde la versión 5 incluye el manejo de enumeraciones. Las enumeraciones sirven para agrupar un conjunto de elementos dentro de un tipo definido. Antes, una manera simple de definir un conjunto de elementos como si fuera una enumeración era, por ejemplo:

```
1 public static final int TEMPO_PRIMAVERA = 0;
2 public static final int TEMPO_VERANO = 1;
3 public static final int TEMPO_OTOÑO = 2;
4 public static final int TEMPO_INVIERNO = 3;
```

Lo cual puede ser problemático pues no es realmente un tipo de dato, sino un conjunto de constantes enteras. Tampoco tienen un espacio de nombres definido por lo que tienen que definirse nombre. La impresión de estos datos, puesto que son enteros, despliega solo el valor numérico a menos que sea interpretado explícitamente por código adicional en el programa.

El manejo de enumeraciones en Java tiene la sintaxis de C, C++ y C :

Sintaxis
`enum <nombreEnum> { <elem 1>, <elem 2>, ..., <elem n> }`

Por lo que para el código anterior, la enumeración sería:

```
enum Temporada { PRIMAVERA, VERANO, OTOÑO, INVIERNO }
```

La sintaxis completa de enum es más compleja, ya que una enumeración en Java es realmente una clase, por lo que puede tener métodos en su definición. También es posible declarar la enumeración como pública, en cuyo caso debería ser declarada en su propio archivo.

Ejemplo:

```
1  enum Temporada { PRIMAVERA, VERANO, OTOÑO, INVIERNO }
2
3  public class EnumEj {
4
5
6          public static void main(String[] args) {
7                  Temporada tem;
8                  tem=Temporada.PRIMAVERA;
9                  System.out.println("Temporada: " + tem);
10
11         System.out.println("\nListado de temporadas:");
12
13         for(Temporada t: Temporada.values())
14                         System.out.println("Temporada: " + t);
15
16             }
17 }
```

Listing 28. Ejemplo de enumeración en Java.

3.2.10. Entrada desde consola en Java

La entrada era más complicada en Java

La entrada tradicional en Java desde consola era evitada en libros y cursos en su etapa introductoria, esto debido a que era necesario incluir manejos de *Streams* o flujos, lo que comúnmente requiere mayor experiencia con el lenguaje y la programación orientada a objetos.

Sin embargo, a partir de la versión 1.5 del *jdk* se incluye la clase *Scanner* que proporciona comportamiento de lectura desde consola.

Ejemplo:

```
1  import java.util.Scanner;
2
```

```
3  public class ScannerTest {
4
5     public static void main(String[] args) {
6
7         String nom;
8         int edad;
9         Scanner in = new Scanner(System.in);
10
11        System.out.print("Nombre:");
12        // Lee una linea de consola
13        nom = in.nextLine();
14
15        System.out.print("Edad:");
16        // Lee un entero de consola
17        edad=in.nextInt();      Existe para cualquier dato básico un next
18        in.close();
19
20        System.out.println("Nombre :"+nom);
21        System.out.println("Edad :"+edad);
22
23     }
24 }
```

Listing 29. Ejemplo de entrada de consola en Java con *Scanner*.

Operaciones similares a *nexInt()* y *nextLine()* existen para el resto de los tipos de datos.

La versión 1.6 del *jdk* incluye otra clase: *Console* la cual proporciona el comportamiento de lectura de una linea desde consola y lectura sin eco (tipo *password*) en la consola.

Ejemplo[12]:

```
1  import java.io.Console;
2
3  //no funciona en la consola de Eclipse
4  public class TestConsole {
5
6     public static void main(String... args ) {
7
8             // Obtener un objeto de consola
9             Console console = System.console();
10            if (console == null) {
11            System.err.println("No se obtuvo la consola.");
```

[12]Ejecutar directamente de consola ya que puede no ejecutarse correctamente en el algunos IDEs.

```
12              System.exit(1);
13          }
14
15              String usuario = console.readLine("Usuario:");
16
17              //Lee password y lo recibe en un arreglo de caracteres
18              char[] password = console.readPassword("Password: ");
19
20              if (usuario.equals("admin")
21               && String.valueOf(password).equals("secreto")) {
22                  console.printf("Bienvenido %1$s.\n", usuario);
23
24              } else {
25                  console.printf("Usuario o password inválido.\n");
26              }
27      }
28  }
```

No lo muestra

Otra forma de imprimir

// System.err es para imprimir un error, pero puede que no sea visible en pantalla, recordemos que hay varias salidas

Listing 30. Ejemplo de entrada de consola en Java con *Console*.

Como se pudo apreciar, esta clase también incluye operaciones de salida a consola. También simplifica la salida de caracteres especiales en la consola.

Argumentos de cantidad variable

Ahora, si fueron observadores sabrán que el último ejemplo nos trajo un nuevo tópico: el uso de los ... en la operación *main*. Este operador sirve para definir argumentos de cantidad variable, siendo el resultado almacenado en un arreglo del tipo especificado. **Podemos combinar los argumentos variables con otros argumentos, pero solo podemos meter un argumento variable y debe ir al final.**

Ejemplo:

```
1   public class TestArgs {
2           public static void main(String[] args) {
3                   llamame1(new String[] {"a", "b", "c"});
4                   llamame2("a", "b", "c");
5                   // Otra opción:
6                   // llamame2(new String[] {"a", "b", "c"});
7           }
8
9           public static void llamame1(String[] args) {
10                  for (String s : args)
11                          System.out.println(s);
```

```
12          }
13
14          public static void llamame2(String... args) {
15                  for (String s : args)
16                          System.out.println(s);
17          }
18  }
```

Así es la sintaxis puesto que se desconoce el tamaño concreto del arreglo

Listing 31. Ejemplo de entrada de argumentos de cantidad variable.

Otro ejemplo[13], se presenta a continuación.

Ejemplo:

```
1   public class VarargsTest
2   {
3   // cálculo de promedio
4   public static double average( double... numbers )
5        {
6           double total = 0.0; // inicializar total
7
8           for ( double d : numbers )
9              total += d;
10
11          return total / numbers.length;
12       }
13
14       public static void main( String args[] )
15       {
16          double d1 = 10.0;
17          double d2 = 20.0;
18          double d3 = 30.0;
19          double d4 = 40.0;
20
21          System.out.printf( "d1 = %.1f\nd2 = %.1f\nd3 = %.1f\nd4 = %.1f\n\n",
22             d1, d2, d3, d4 );
23
24          System.out.printf( "Promedio de d1 y d2 es %.1f\n",
25             average( d1, d2 ) );
26          System.out.printf( "Promedio de d1, d2 y d3 es %.1f\n",
27             average( d1, d2, d3 ) );
28          System.out.printf( "Average de d1, d2, d3 y d4 es %.1f\n",
```

[13]Código original de [5]

```
29              average( d1, d2, d3, d4 ) );
30      }
31  }
```

Listing 32. Ejemplo de entrada de argumentos de cantidad variable.

3.2.11. Paquetes

Las clases en Java son organizadas mediante paquetes. Un paquete es entonces el mecanismo para agrupas clases que están relacionadas, ya sea porque sirven a un propósito común o porque dependen unas de otras para realizar sus responsabilidades. Se usan los paquetes cuando importamos clases para ser incorporadas en nuestro código, pero si queremos especificar el paquete al que pertenecen nuestras clases podemos hacerlo.

Sintaxis
`package [<ruta>]<nombre paquete>`

Si no se define el nombre del paquete, por omisión se considera el nombre del directorio donde la clase se encuentra definida.

Los paquetes se usan para organizar, puede incluir una ruta y se mezcla con directorios, con el fin de que en sistemas completos exista cierta organización

Capítulo 4

Introducción a Python

4.1. Introducción

Python es un lenguaje dinámico y con características de orientado a objetos que es muy popular para desarrollo en Web, aunque cuenta también con características de programación funcional. Es similar a lenguajes como Ruby, Perl y Scheme pero también tiene influencias de lenguajes como Java y C.

Fue desarrollado en 1990 por *Guido van Rossum* y es un lenguaje que se ejecuta en las principales plataformas de hardware y sistemas operativos. Actualmente, junto con Ruby, Python es uno de los lenguajes orientados a objetos más usados para desarrollo de web dinámico[1].

Existen tres principales implementaciones de Pyhton:

- *Python / Cpython*. También llamada solamente Python, debido a que es la implementación más popular. La razón es que es la que tiene un desarrollo más completo, actualizado y de rápida ejecución[2].
- *Jython*. Es una implementación de Pyhton para ejecutarse en máquinas virtuales de Java (JVM), de manera similar a Scala. Puede hacer uso de la biblioteca de clases de Java.
- *IronPython*. Es una implementación de Python para la CLR (*Common Language Runtime*) de Microsoft (.NET). Puede usar las bibliotecas de clases de .NET

4.2. Herramientas

El principal programa para usar Python lleva precisamente este nombre. *python* es al mismo tiempo el intérprete y el compilador del lenguaje. El programa genera código de

[1] Un artículo interesante de despedida a Guido por Dropbox donde mencionan su trabajo en Python https://blog.dropbox.com/topics/company/thank-you--guido.html

[2] Ver: https://www.python.org/

bytes que es almacenado en programas *.pyc* o *.pyo*. Estos archivos son generados automáticamente cuando el archivo fuente es actualizado.

Python puede ejecutar código de 2 formas:

1. Interactivamente. Se ejecuta *python* desde el *prompt* de la consola:

```
> python
Python 3.1.1 (r311:74543, Aug 24 2009, 18:44:04)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>>
```

2. Ejecución interpretada de archivos. *python* seguido del nombre del *script* a ejecutar.

```
> python programa.py
```

Además, Python incluye un sencillo ambiente de desarrollo llamado IDLE (*Integrated DeveLopment Environment*), el cual ofrece un *shell* similar al intérprete de Python con ligeras funcionalidades añadidas; además de incluir un editor de texto, visores y un depurador interactivo. Python puede ser usado desde otros IDEs, tales como Eclipse y NetBeans.

4.3. Fundamentos de Python

4.3.1. Convenciones léxicas

Un programa en Python esta formado por una secuencia de líneas lógicas que pueden estar formadas por una o más líneas físicas. Una línea no lleva un delimitador como en otros lenguajes. En cambio, si la línea es muy larga, dos líneas físicas puede unirse con una diagonal ' \' . Aunque Python automáticamente une dos líneas físicas si un paréntesis, corchete o llave no ha sido cerrado.

Identación es importante para Pytthon. A diferencia de muchos lenguajes, Python no usa llaves u otros medios (como begin-end) para delimitar bloques de instrucciones. La identación es la forma en que los bloques son delimitados en Python.

4.3.2. Literales

Python tiene tipos definidos para tipos de datos básicos. Estos son objetos que también pueden ser usados como literales.

Por ejemplo, las literales enteras pueden ser escritas en decimal:

```
>>> 123
123
```

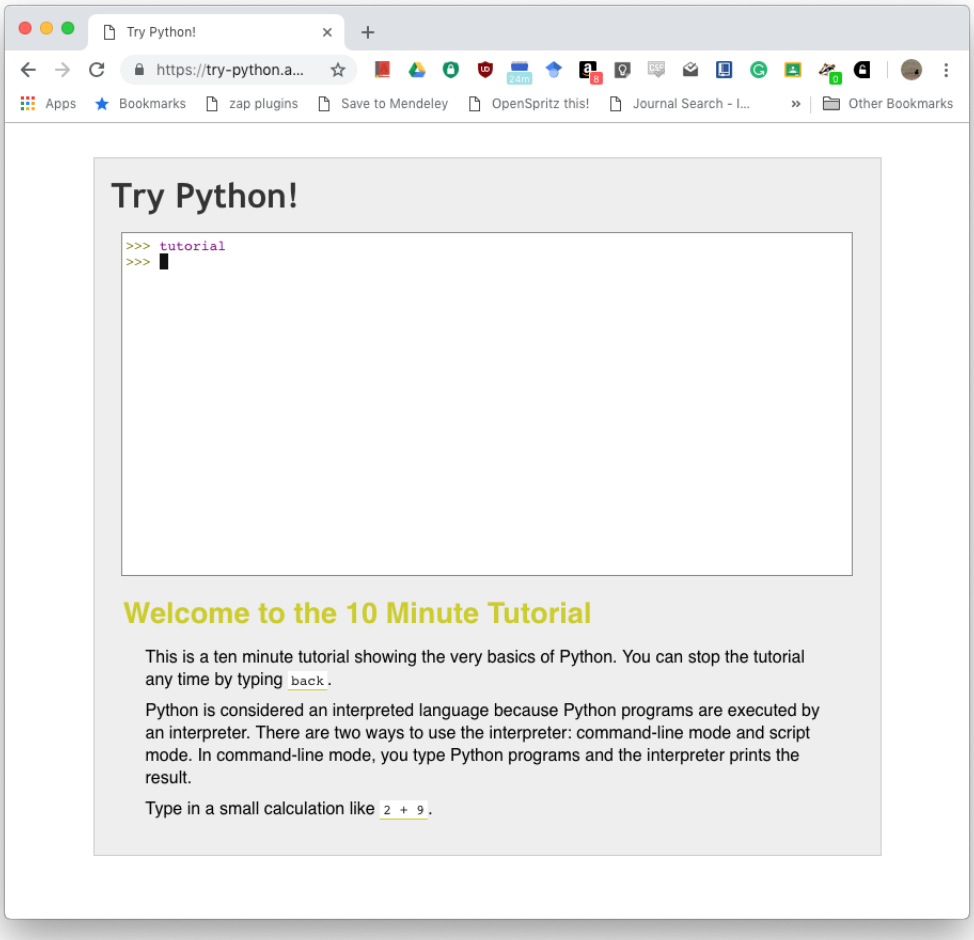
o hexadecimal:

```
>>> 0x17
23
```

Números flotantes se escriben con un punto y tienen el equivalente a un double en C.

4.3.3. Probando Pyhton

¿Por qué no empezar a probar python siguiendo el tutorial el línea? Éste se encuentra disponible en : https://try-python.appspot.com/



4.3.4. Variables

Una variable es un espacio para almacenar datos modificables, en la memoria de una computadora, en Python una variable se define por la sintaxis:

```
nombre_de_la_variable= valor_de_la_variable
```

Cada variable tiene un nombre y un valor, el cual define a la vez el tipo de datos de la variable. Para nombrarlos es necesario un nombre descriptivo y en minúsculas. Pueden utilizarse nombres compuestos pero las palabras se separan por guiones bajos.

Existen otros tipos de datos que no requieren ser modificados a lo largo del programa, y son llamadas constantes. Los nombres de las constantes deben estar escritos con

Símbolo	Significado	Ejemplo	Resultado
+	Suma	10+5	15
-	Resta	12-7	5
-	Negación	-5	-5
*	Multiplicación	7*5	35
**	Exponente	2**3	8
/	División	12.5/2	6.25
//	División entera	12.5//2	6.0
%	Módulo	27%4	3

Cuadro 4.1. Operadores aritméticos en Python

mayúsculas, y se deben separar las palabras por guiones bajos al igual que los nombres de las variables. Para imprimir un valor de pantalla, en Python, se utiliza la palabra clave *print*:

```
1 mi_variable = 15
2 print (mi_variable)
```

Esto imprimirá el valor de la variable *mi_variable* en la pantalla.
Cuando una variable tiene el valor de nulo, se usa la palabra reservada *None*.

4.3.5. Operadores

Operadores aritméticos

Los operadores aritméticos que son utilizados en Python son mostrados en el Cuadro 4.1

El siguiente es un ejemplo donde se utilizan los operadores aritméticos:

```
1 >>>monto_bruto = 175
2 >>>tasa_interes = 12
3 >>> monto_interes = monto_bruto * tasa_interes / 100
4 >>> tasa_bonificacion = 5
5 >>> importe_bonificacion = monto_bruto * tasa_bonificacion / 100
6 >>> monto_neto = (monto_bruto - importe_bonificacion) + monto_interes
7 >>> monto_neto
8 187.25
```

Operadores relacionales

Ver Cuadro 4.2 con el listado de los operadores relacionales.

Curiosamente, mientras muchos lenguajes únicamete permiten usar a los operadores relacionales para comparar pares de elementos. En Python permiten formar expresiones con elementos adicionales.

Símbolo	Significado	Ejemplo	Resultado
==	Igualdad	5==7	Falso
!=	Diferencia	4 != 5	Verdadero
<	Menor que	5 <7	Verdadero
>	Mayor que	4>8	Falso
<=	Menor o igual que	5<=5	Verdadero
>=	Mayor o igual que	4>=5	Falso

Cuadro 4.2. Operadores relacionales en Python

Símbolo	Ejemplo	Resultado
and	5==7 and 7<12	Falso
or	7>5 or 9<12	Verdadero
xor (o excluyente)	4==4 xor 9>3	Falso

Cuadro 4.3. Operadores lógicos en Python

```
>>> 5>3>10
False
>>> 5>10>1
False
>>> 5>3>2
True
>>> 5>2>3
False
>>> 5>2<3
True
```

Operadores lógicos

Y para evaluar más de una condición simultáneamente se utilizan los operadores lógicos presentados en el Cuadro 4.3

4.3.6. Tipos de datos

Una variable puede contener valores de diversos tipos. Entre ellos:

```
Cadena de texto (String)
                mi_cadena = "Hola mundo"
Número entero
edad = 35
Número hexadecimal
edad = 0x23
```

```
Número real
precio = 7435.28
Booleano (verdadero / falso)
verdadero = True
        falso = False
```

Estos son algunos tipos de datos sencillos, además en Python existen otros tipos de datos complejos que admiten una colección de datos, como las **tuplas**, las **listas** y los **diccionarios**.

Listas

Python no tiene el concepto de arreglos. La secuencia de elementos más usada en el lenguaje son las listas. Las listas son colecciones de elementos de tipos arbitrarios y sin un tamaño fijo[6].

Son similares a los arreglos en otros lenguajes, con la diferencia de que son de tamaño dinámico y pueden contener elementos de distinto tipo.

```
mi_lista = ['cadena de texto',15,2.8,'otro dato',25]
```

A los datos de las listas se accede mediante el índice entre corchetes. Sus datos son mutables.

```
mi_lista[2] = 3.5
print (mi_lista)  # Devuelve ['cadena de texto', 15, 3.5, 'otro dato', 25]
```

También pueden agregarse nuevos datos a la lista,

```
mi_lista.append('Nuevo dato')
print (mi_lista)
        #Devuelve ['cadena de texto', 15, 3.5, 'otro dato', 25, 'Nuevo dato']
```

Ejemplos:

```
>>> lista=[123, 'xxx', 3.14]
>>> len(lista)
3
>>> lista[0]
123
>>> lista + [4, 5, 6]
[123, 'xxx', 3.14, 4, 5, 6]
>>> lista * 2
[123, 'xxx', 3.14, 123, 'xxx', 3.14]
>>> lista
```

```
[123, 'xxx', 3.14]

>>> lista=lista+[4,5,6]
>>> lista
[123, 'xxx', 3.14, 4, 5, 6]
>>> lista.append('zzz')
>>> lista
[123, 'xxx', 3.14, 4, 5, 6, 'zzz']
>>> lista.pop(2)
3.14
>>> lista
[123, 'xxx', 4, 5, 6, 'zzz']
>>> orden=['c', 'a', 'b']
>>> orden.sort()
>>> orden
['a', 'b', 'c']
>>> orden.reverse()
>>> orden
['c', 'b', 'a']
>>> lista
[123, 'xxx', 4, 5, 6, 'zzz']
>>> lista[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> lista[100]=1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range


>>> matriz=[[1, 2, 3],
... [4, 5, 6],
... [7, 8, 9]]
>>> matriz
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> matriz[1]
[4, 5, 6]
>>> matriz[1][2]
6
```

Tuplas

Una tupla es una variable que permite almacenar varios datos inmutables de tipos diferentes.

```
mi_tupla = ('Cadena de texto', 15 , 2.8 ,'otro dato', 25)
```

Se puede acceder a cada uno de estos datos mediante su índice correspondiente, siendo el 0 el índice del primer elemento.

```
print (mi_tupla[1])    #Devuelve 15
print (mi_tupla)    # Devuelve  ('Cadena de texto', 15, 2.8, 'otro dato', 25)
```

Conjuntos

Podemos también crear un conjunto

```
>>> s={1, 2, 3}
>>> print(s)
{1, 2, 3}
>>> s.add(4)
>>> s
{1, 2, 3, 4}
>>> s.add(2)
>>> s
{1, 2, 3, 4}
>>> s.discard(2)
>>> s
{1, 3, 4}
>>> s.remove(4)
>>> s
{1, 3}
>>> s.discard(4)
>>> s.remove(4)
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    s.remove(4)
KeyError: 4
```

Diccionarios

A diferencia de las listas y las tuplas, los diccionarios permiten utilizar una clave para declarar y acceder a un valor.

```
mi_diccionario = {'clave_1': 12, 'clave_2': 10}
print (mi_diccionario['clave_1'])        # Devuelve   12
```

Un diccionario permite eliminar cualquier elemento,

```
del(mi_diccionario['clave_2'])
print (mi_diccionario)      # Devuelve {'clave_1': 12}
```

Al igual que las listas permite modificar los elementos,

```
mi_diccionario['clave_1'] = 24
print (mi_diccionario)    #Devuelve    {'clave_1': 24}
```

¿Es Python tipado dinámica o estáticamente?

Mientras En el paso de parámetros y el retorno de una función no es necesario declarar el tipo de dato, Python 3 permite *sugerir* el tipo de dato esperado. Esta sugerencia no es usada directamente por Python pero sirve como información para el programador y de ayuda para herramientas externas al revisar el código. Por ejemplo:

```
def f(n: int = 10) -> int:
    return n*n
print("Buen uso", f(8))
print("Mal uso", f(.9))
print(f())
```

El código anterior corre sin problema a pesar de pasar un valor flotante donde se espera un entero.

4.3.7. Estructuras de control

Una estructura de control es un bloque de código que permite agrupar instrucciones de manera controlada. Las principales estructuras de control son de dos tipos:

- Estructuras de control condicionales.
- Estructuras de control iterativas.

Estructuras de control condicionales

Para evaluar más de una condición simultáneamente se utilizan los operadores lógicos presentados anteriormente.

Las estructuras de control condicionales, se definen mediante el uso de tres palabras reservadas *if* , *elif* y *else* .

Ejemplos:

```
>>> def cruzar(semaforo):
...             if semaforo == "verde":
...                             print ("Cruzar la calle")
...             else:
...                     print ("No cruzar la calle")

>>>def decidir_pago (compra)
...         if compra <= 100:
...                          print ("Pago en efectivo")
...          elif compra > 100 and compra < 300:
...                          print ("Pago con tarjeta de débito")
...          else:
...                          print ("Pago con tarjeta de crédito")
```

Estructuras de control iterativas

En Python se dispone de dos estructuras cíclicas:

while Ejecuta una misma acción mientras una determinada condición se cumpla. Ejemplo:

```
>>> def imp_anios(anio):
...     while (anio <= 2012):
...             print "Informes del año", str (anio)
...             anio += 1
...
```

Al probar este programa teniendo como dato de entrada $anio = 2009$, se obtiene:

```
>>> imp_anios(2009)
Informes del año 2009
Informes del año 2010
Informes del año 2011
Informes del año 2012
```

Con la última línea del programa $anio+ = 1$, estamos incrementando en uno la variable *anio*. Esto hace que el ciclo en algún momento termine. Si ocurriera que el valor que se evalúa para el ciclo no es un valor numérico que no puede incrementarse; en ese caso, podremos utilizar una estructura de control condicional, anidada dentro del ciclo, y frenar la ejecución cuando el condicional deje de cumplirse, con la palabra clave reservada *break*:

Ejemplo:

```
>>> def edad():
        while True:
```

```
                edad = input("Edad: ")
                if int(edad)>100:
                        break

>>> edad()
Edad: 4
Edad: 5
Edad: 101
>>>
```

Este programa continuará hasta que el usuario introduzca su nombre.

for El ciclo *for*, en Python, es aquel que nos permitirá iterar sobre una variable compleja, del tipo lista o tupla.

Ejemplo:

```
>>> mi_lista = ['Juan', 'Antonio', 'Pedro', 'Herminio']
        >>> for nombre in mi_lista:
        ...     print nombre
        ...
```

Este programa devuelve como resultado:

```
  Juan
        Antonio
        Pedro
        Herminio
```

Otra forma de iterar con el *for* es la siguiente:

```
>>> for anio in range(2001, 2009):
        ...             print "Informes del Año", str(anio)
        ...
```

Instrucciones de control de ciclos Como en otros lenguajes, se tienen instrucciones para modificar el comportamiento del flujo de ejecución, ya sea para saltar o detener la ejecución de un ciclo:

- *break*. Dada una condición, la instrucción *break* detiene la ejecución y salta el flujo fuera del ciclo.
- *continue*. Dada una condición, el flujo salta al inicio del ciclo y permite la siguiente iteración, si la condición del ciclo sigue siendo verdadera.

4.3.8. Entrada y Salida básica en Python

Las funciones principales de entrada y salida de datos son:

- La función print() es interna del lenguaje Python y se utiliza para imprimir en la pantalla. Para concatenar varios datos a imprimir en pantalla se utilizan comas, e.g. *print*(”*Hola*”, *alguien*, ”!”).
- La función *a= input (”Introduzca un valor”)* [3] despliega un mensaje en pantalla para solicitar un dato que será almacenado en la variable a como texto. La función *input*, devuelve el valor ingresado por teclado tal como se escribe.

```
>>> variable=input("Edad:")
Edad:45
>>> variable
'45'
>>> type(variable)
<class 'str'>
>>> int(variable)
45

>>> nombre=input("Nombre:")
Nombre:carlos a
>>> nombre
'carlos a'
```

Ejemplo:

```
s=" ¡Hola, Mundo! "
print(s)
print(s[1])
print(s[7:12])

# elimina caracteres en blanco del lado izquierdo y derecho de la cadena
print(s.strip())
print(len(s))
print(s.lower())
print(s.upper())

#sustituye "l" por "j"
print(s.replace("l", "j"))

```

[3] Antes de Python 3 era llamada raw_input

```
15 #separa una cadena y regresa una lista de cadenas
16 str = "Un ejemplo de string....!"
17 print (str.split( ))
18 print (str.split('e',1))
19 print (str.split('e'))
20
21 print("Nombre:")
22 n=input()
23 print("Hola, "+n)
```

Listing 33. Ejemplo de entrada en Python.

4.3.9. Funciones

Aunque ya hemos usado funciones en Python no se han explicado formalmente.

Sintaxis

```
def <nombre_función> ( [<parametros>] ) :
    <cuerpo de la función>
    [return <expresión>]
```

Como puede verse la instrucción de retorno es opcional. En caso de no regresarse nada explícitamente, se regresa *None*.

Valores por omisión en parámetros

Un argumento por omisión es un parámetro que asume un valor si el valor no es proporcionado en la el argumento de la llamada de la función.

```
1 # Ejemplo de valores por omisión
2
3 def fun(x, y=50):
4         print("x: ", x)
5         print("y: ", y)
6
7 fun(10)
```

Listing 34. Ejemplo valores por omisión en Python.

Al igual que en C++, los valores por omisión deben estar a la extrema derecha en la lista de argumentos.

Importante
Hay que tener en cuenta que el ligado a los valores por omisión ocurre en la definición de la función.

Usualmente este comportamiento no es el deseado:

```
1  def f(x = None):
2      if x is None:
3          x = []
4      x.append(1)
5      return x
6
7  print(f())
8  print(f())
9  print(f())
10 print(f(x = [9,9,9]))
11 print(f())
12 print(f())
```

Listing 35. Ejemplo en Python.

Palabras clave de argumentos intercambiables

Este concepto implica poner el nombre del argumento en la llamada con sus valores, de manera que no se tenga que recordar el orden de los parámetros.

```
1  def estudiante(nombre, apellido):
2          print(nombre, apellido)
3
4  # Argumentos clave
5  estudiante(nombre='Ada', apellido='Lovelace')
6  estudiante(apellido='Turing', nombre='Alan')
```

Listing 36. Ejemplo palabras clave de argumentos en Python.

Argumentos de cantidad variable

En Python, podemos tener argumentos de cantidad variable usando $*$ o $**$. En el primer caso para argumentos tradicionales sin palabra clave y el segundo caso para argumentos con palabras clave (ver ejemplo anterior).

Argumentos de longitud variable sin palabras clave

```
# *argvar  como variable para los argumentos

def fun(*argvar):
  for arg in argvar:
    print(arg)
  print(type(argvar)) # argvar es una tupla

fun('Hola', 'Bienvenido', 'a', 'POO')
```

Listing 37. Ejemplo Argumentos de longitud variable sin palabras clave en Python.

```
# *kwargs como variable para los argumentos con palabras clave

def fun(**kwargs):
  for clave, val in kwargs.items():
    print("%s == %s" % (clave, val))
  print(type(kwargs))  #kwargs es un diccionario

fun(uno='Prueba', dos='de', tres='argumentos')
```

Listing 38. Ejemplo Argumentos de longitud variable con palabras clave en Python.

Funciones de primera clase in Python

Los **objetos de primera clase** en un lenguaje son manejados uniformemente. Pueden ser usados dentro de una estructura de datos, pasados como argumentos, o usados en estructuras de control. Un lenguaje como Python que soporta **funciones de primera clase** es aquel que trata a sus funciones como objetos de primera clase. Las funciones de primera clase en Python:

- Una función es una instancia de la clase $function$ y $object$.
- Una función puede asignarse a una variable.
- Una función puede pasarse como parámetro a otra función.

- Se puede regresar una función como valor de retorno de una función.
- Se puede almacenar una función en estructuras de datos, por ejemplo en una lista.

Funciones como objetos

```
#Funciones como objetos

def saludar(texto):
        return texto.upper()

print (saludar('Hola'))

gritar = saludar

print (gritar('¡Hola!'))
```

Listing 39. Ejemplo funciones como objetos en Python.

Funciones como argumentos de otras funciones También conocidas como funciones de orden superior.

```
def gritar(text):
        return text.upper()

def susurrar(text):
        return text.lower()

def saludar(func):
        # almacenando la función en una variable
        saludar = func("""Hola, hemos creado una función
        pasada como un argumento.""")
        print (saludar)

saludar(gritar)
saludar(susurrar)
```

Listing 40. Ejemplo de función como argumento de otra función en Python.

Función regresando a otra función

```
def crea_sumar(x):
        def sumar(y):
                return x+y
```

```
4
5           return sumar
6
7   suma_15 = crea_sumar(15)
8
9   print (suma_15(10))
10
11  print (suma_15(15))
```

Listing 41. Ejemplo de función regresando a otra función en Python.

4.3.10. Módulos en Python

En Python cada uno de los archivos *.py* se denominan módulos.

El contenido de cada módulo, podrá ser utilizado a la vez, por otros módulos. Para ello, es necesario importar los módulos que se quieran utilizar. Para importar un módulo, se utiliza la instrucción *import*, seguida del nombre del paquete (si aplica) más el nombre del módulo (sin el *.py*) que se desee importar.

También es posible usar $from < modulo > import < elem1, elem2, ... >$. Esto implica que del módulo en el archivo del mismo nombre, únicamente se crean en el espacio de nombres actual los elementos listados. En este caso la ventaja sería además que los elementos importados pueden usarse sin necesidad de especificar el módulo.

Ejemplo:

```
1  import modulo   # importar un módulo que no pertenece a un paquete
2  import paquete.modulo1  # importar un módulo que está dentro de un paquete
3  import paquete.subpaquete.modulo1
4
5  from math import sqrt, sin, cos
```

Además, Python tiene sus propios módulos, los cuales forman parte de su biblioteca de módulos estándar, que también pueden ser importados.

A su vez, los módulos pueden agruparse formando paquetes.

4.3.11. Paquetes en python

Un paquete es una carpeta que contiene archivos *.py*. Pero para que una carpeta pueda ser considerada un paquete, esta debe contener una archivo de inicio *__init__.py*. Este archivo no necesita contener ninguna instrucción, de hecho puede estar completamente vacío. Sin embargo, considerar que este archivo es invocado cuando el paquete es importado por lo que puede contener código necesario para la inicialización del paquete.

Además dentro de los paquetes pueden estar contenidos otros subpaquetes y los módulos no necesariamente pueden estar en un paquete.