

Sumario

UT1: Introducción a la programación.....	2
1 Introducción a los lenguajes de programación.....	2
1.1 Los paradigmas de programación.....	2
1.2 Lenguajes de alto y bajo nivel.....	3
2 Datos, algoritmos y programas.....	4
2.1 Tipos de datos.....	4
2.2 El pseudocódigo.....	6
2.3 Los diagramas de flujo.....	6
3 Errores y calidad de los programas.....	8
4 Metodologías en proyectos de diseño de software.....	10



Documentación oficial de Python: <https://docs.python.org/3/>

Referencia **w3schools**: https://www.w3schools.com/python/python_reference.asp



Realizado bajo licencia [Creative Commons Reconocimiento-NoComercial CC-BY-NC 4.0](https://creativecommons.org/licenses/by-nc/4.0/)

UT1: Introducción a la programación

En esta unidad de trabajo veremos qué es un programa y estudiaremos una serie de conceptos generales que nos servirán de base para aprender a diseñar programas.

1 Introducción a los lenguajes de programación.

1.1 Los paradigmas de programación

Los estilos de programación han evolucionado con el tiempo, pasando por diferentes períodos:

- Programación desestructurada (entre 1950 y 1970)
- Programación estructurada clásica (entre 1970 y 1990)
- Programación modular (entre 1970 y 1990)
- Programación orientada a objetos (desde 1990)

Un programa de ordenador se ejecuta secuencialmente, de manera que las instrucciones siguen un orden definido. Generalmente, este orden es el mismo en que se escriben, pero muchas veces es necesario repetir conjuntos de instrucciones (bucles) o realizar saltos de una instrucción a otra. En la programación **desestructurada** clásica se usan bucles y saltos muy frecuentemente, dando lugar a un código farragoso, a veces confuso, que genera gran probabilidad de incurrir en errores y hace muy difícil el mantenimiento o evolución del programa. Los lenguajes más antiguos, como Fortran, Cobol, Basic, etc... se consideran desestructurados.

La programación **estructurada** consiste en la utilización de estructuras que optimizan los recursos lógicos y físicos del ordenador. Los lenguajes estructurados también se conocen como imperativos o de tercera generación. Por ejemplo, C, Pascal o Modula-2 son estructurados. Algunos lenguajes antiguos se adaptaron con el tiempo a este paradigma, aunque permitían seguir haciendo programación desestructurada si el programador lo deseaba.

La programación **modular** convive con la estructurada y con frecuencia se usa conjuntamente. Consiste en dividir el programa complejo en varios programas sencillos que interactúan entre ellos. Cada programa sencillo se llama "módulo". Los módulos deben ser independientes entre sí, no interferir unos con otros. El lenguaje modular clásico es "Modula-2".

La programación **orientada a objetos** (OOP / POO) es una evolución de la anterior. Es programación estructurada y modular al mismo tiempo, en la que las instrucciones y los datos se encapsulan en entidades denominadas "clases", de las que luego se crean los "objetos", que tienen una serie de propiedades y capacidades para realizar acciones o "métodos". Lo veremos más adelante.

1.2 Lenguajes de alto y bajo nivel

El ordenador solo puede manejar números, y más concretamente, unos y ceros, es decir, código o **lenguaje binario**. Los seres humanos manejamos un lenguaje mucho más complejo, con diferentes símbolos. Utilizamos el código decimal (basado en diez cifras diferentes) y utilizamos una serie de reglas sintácticas y semánticas, que denominamos **lenguaje natural**. Entre los dos extremos encontramos los lenguajes de programación. Cuanto más se acerca un lenguaje de programación al lenguaje binario, diremos que es de bajo nivel de abstracción. Los lenguajes más próximos al lenguaje natural se denominan lenguajes de alto nivel de abstracción.

Los lenguajes de bajo nivel son los más cercanos a la máquina. Los programas directamente escritos en código binario se dice que están escritos en **código máquina**. Se basan en instrucciones que ejecutan tareas muy sencillas, cuya combinación permite realizar tareas complejas. La forma de escribir código binario es muy compleja. En los primeros años de la informática se programaba directamente así, pero a medida que la potencia de los ordenadores iba creciendo, la complejidad de estos programas crecía exponencialmente. Surgieron lenguajes más complejos (de alto nivel) en los que cada palabra ya no se correspondía con una instrucción en código máquina, sino con tareas más complejas. Ejemplos de lenguajes de alto nivel: Basic, Cobol, Fortran, Ada, C, PHP, Python, Java, Perl, etc...

La traducción desde un lenguaje de alto nivel al lenguaje máquina es más compleja, y requiere de un proceso previo, que puede estar basado en mecanismos como la compilación o la interpretación, que veremos más adelante.

Un **compilador** permite generar el código binario correspondiente a un programa, a partir del código de alto nivel, de manera que pueda ser ejecutado directamente por la máquina.

Un **intérprete** realiza la traducción a código binario de manera secuencial, a medida que se va ejecutando el programa. La ejecución es más lenta, pero el programa no está vinculado al hardware de una máquina concreta.

En función de esta división, existen lenguajes de programación "compilados" o "interpretados".

En resumen:

Ventajas de los lenguajes de bajo nivel	Ventajas de los lenguajes de alto nivel
Comprensibles directamente por la máquina, sin necesitar usar compiladores o intérpretes.	Son portables, es decir, independientes del hardware.
Los programas se ejecutan rápidamente	Programas más sencillos
Ocupan menos memoria	Programas más fáciles de escribir, depurar y mantener.
Permiten controlar directamente el hardware	Permiten abordar problemas más complejos.

2 Datos, algoritmos y programas

Un **algoritmo** es el conjunto finito de pasos o instrucciones que se emplean para resolver un problema, en este caso un **programa**.

Los programas manejan **datos**. Un tipo de datos es la propiedad de un valor que determina su dominio (qué valores puede tomar), qué operaciones se le pueden aplicar y cómo es representado internamente por el ordenador. Todos los valores que aparecen en un programa tienen un tipo.

2.1 Tipos de datos

Cada lenguaje de programación es capaz de manejar diferentes tipos de datos. Algunos de los más comunes son:

Números enteros

El tipo int (del inglés integer, que significa «entero») permite representar números enteros. Los valores que puede tomar un int son todos los números enteros: ... -3, -2, -1, 0, 1, 2, 3, ...

Los números enteros literales se escriben con un signo opcional seguido por una secuencia de dígitos:

```
1570
+4591
-12
```

Números reales

El tipo float permite representar números reales. El nombre float viene del término "punto flotante", que es la manera en que el computador representa internamente los números reales. Hay que tener mucho cuidado, porque los números reales no se pueden representar de manera exacta en un computador. Por ejemplo, el número decimal 0.7 es representado internamente por el computador mediante la aproximación 0.69999999999999996. Todas las operaciones entre valores float son aproximaciones.

Los números reales literales se escriben separando la parte entera de la decimal con un punto. Las partes entera y decimal pueden ser omitidas si alguna de ellas es cero:

```
>>> 881.9843000
881.9843
>>> -3.14159
-3.14159
>>> 1024.
1024.0
>>> .22
0.22
```

Otra representación es la notación científica, en la que se escribe un factor y una potencia de diez separados por una letra e. Por ejemplo:

```
>>> -2.45E4
-24500.0
>>> 7e-2
0.07
>>> 6.02e23
6.02e+23
>>> 9.1094E-31
9.1094e-31
```

Los dos últimos valores del ejemplo son iguales, respectivamente, a 6.02×10^{23} (la constante de Avogadro) y 9.1094×10^{-31} (la masa del electrón).

Números complejos

El tipo complex permite representar números complejos.

Los números complejos tienen una parte real y una imaginaria. La parte imaginaria es denotada agregando una j inmediatamente después de su valor:

```
3 + 9j
-1.4 + 2.7j
```

Valores lógicos

Los valores lógicos True y False (verdadero y falso) son de tipo bool, que representa valores lógicos. El nombre bool viene del matemático George Boole, quien creó un sistema algebraico para la lógica binaria. Por lo mismo, a True y False también se les llama valores booleanos.

Texto

A los valores que representan texto se les llama strings, y tienen el tipo str. Los strings literales pueden ser representados con texto entre comillas simples o comillas dobles:

```
"ejemplo 1"
```

```
'ejemplo 2'
```

La ventaja de tener dos tipos de comillas es que se puede usar uno de ellos cuando el otro aparece como parte del texto:

```
"Let's go!"
'Ella dijo "hola"'
```

Es importante entender que los strings no son lo mismo que los valores que en él pueden estar representados:

```
>>> 5 == '5'
False
```

```
>>> True == 'True'  
False
```

Los strings que difieren en mayúsculas y minúsculas, o en espacios también son distintos:

```
>>> 'mesa' == 'Mesa'  
False  
>>> ' mesa' == 'mesa '  
False
```

Nulo

Existe un valor llamado None (en inglés, «ninguno») que es utilizado para representar casos en que ningún valor es válido, o para indicar que una variable todavía no tiene un valor que tenga sentido.

2.2 El pseudocódigo

Los programas y algoritmos que forman parte de los mismos se pueden representar mediante gráficos denominados "diagramas de flujo". Para la creación de programas, como paso previo a la utilización de lenguajes de programación, usamos lo que se conoce como "pseudocódigo", es decir, un lenguaje de especificación de algoritmos que usa una notación en lenguaje natural, permitiendo representar la programación estructurada y haciendo que el paso final a la codificación de un programa sea relativamente fácil.

El pseudocódigo debe cumplir estas reglas:

- Debe ser **preciso** e indicar el orden de realización de cada paso.
- Debe estar **definido**, es decir, debe dar un mismo resultado (si ponemos los mismos datos no puede dar un resultado distinto).
- Debe ser **finito**, es decir, debe de acabar en algún punto.
- Debe ser **independiente** de un lenguaje de programación.

La estructura de un algoritmo escrito en pseudocódigo es algo parecido a esto:


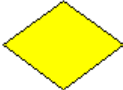








- ➔ Declaración
- ➔ Inicio
- ➔ Secuencia de instrucciones (operaciones de lectura, escritura, estructuras de control, etc.)
- ➔ Final

2.3 Los diagramas de flujo

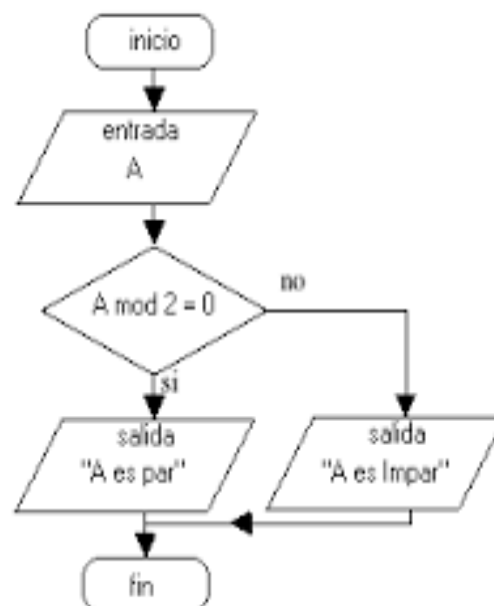
Un diagrama de flujo es la representación gráfica de un algoritmo o proceso. Estos diagramas utilizan símbolos con significados definidos que representan los pasos del algoritmo, y representan el flujo de ejecución mediante flechas que conectan los puntos de inicio y de fin del proceso.

Los diagramas de flujo utilizan una simbología y normas determinadas.

Algunos de los símbolos más utilizados en los diagramas de flujo son:

	Inicio/Final Se utiliza para indicar el inicio y el final de un diagrama; de inicio sólo puede salir una línea de flujo y al final sólo debe llegar una línea		Decisión Indica la comparación de dos datos y dependiendo del resultado lógico (falso o verdadero) se toma la decisión de seguir un camino del diagrama u otro
	Entrada/Salida Entrada/Salida de datos por cualquier dispositivo (scanner, lector de código de barras, micrófono, parlantes, etc.)		Impresora/Documento. Indica la presentación de uno o varios resultados en forma impresa
	Entrada por teclado. Entrada de datos por teclado. Indica que el computador debe esperar a que el usuario teclee un dato que se guardará en una variable o constante		Pantalla Instrucción de presentación de mensajes o resultados en pantalla
	Acción/Proceso Indica una acción o instrucción general que debe realizarse (operaciones aritméticas, asignaciones, etc.)		Conector Interno Indica el enlace de dos partes de un diagrama dentro de la misma página
	Flujo/Flecas de Dirección Indica el seguimiento lógico del diagrama. También indica el sentido de ejecución de las operaciones		Conector Externo Indica el enlace de dos partes de un diagrama en páginas diferentes

- Óvalo o Elipse: Inicio y Final (Abre y cierra el diagrama).
- Rectángulo: Actividad (Representa la ejecución de una o más actividades o procedimientos).
- Rombo: Decisión (Formula una pregunta o cuestión).
- Círculo: Conector (Representa el enlace de actividades con otra dentro de un procedimiento).
- Triángulo boca abajo: Archivo definitivo (Guarda un documento en forma permanente).
- Triángulo boca arriba: Archivo temporal (Proporciona un tiempo para el almacenamiento del documento).



3 Errores y calidad de los programas

La calidad del software es una preocupación a la que se dedican muchos esfuerzos. Sin embargo, el software casi nunca es perfecto. Todo proyecto tiene como objetivo producir software de la mejor calidad posible, que cumpla, y si puede supere las expectativas de los usuarios.

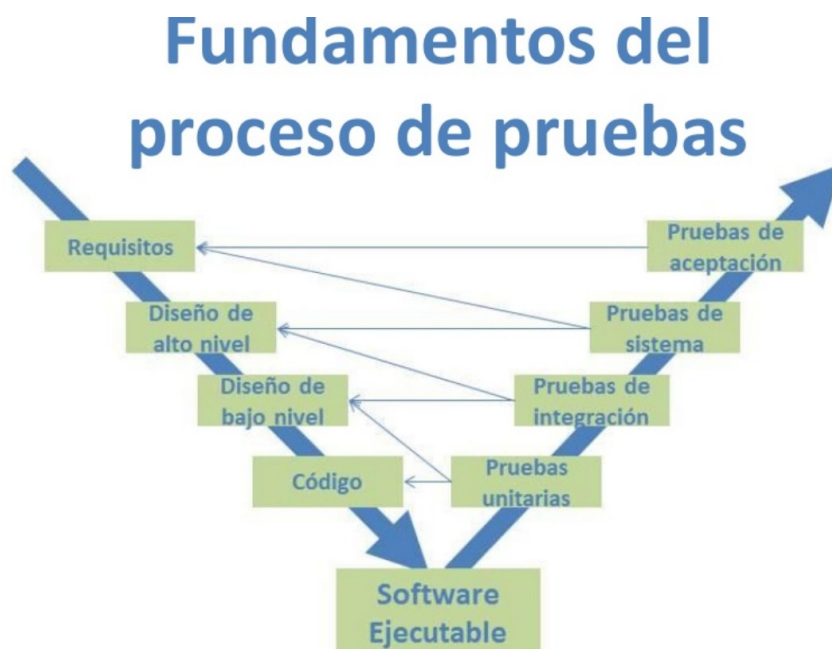
El proceso básico de corrección de errores de código es la "depuración". En el caso de Java, en un primer nivel (sintaxis, etc...) se realiza dentro del JDK con la herramienta **jdb**.

Otro aspecto que define la calidad del software es la DOCUMENTACIÓN. Es muy importante documentar los proyectos de una manera clara y organizada, lo que repercute en una mejor mantenibilidad y desarrollo futuro. La herramienta de documentación incluida en el JDK es **JavaDoc**

A la hora de escribir código, es necesario prever posibles errores en el código y tratarlos mediante el **Control de Excepciones**.

Aparte de los errores de código, es necesaria la realización de una serie de pruebas funcionales, para determinar si un proyecto cumple con las especificaciones y se ha realizado correctamente el proceso de diseño e implementación de software. Estas pruebas se realizan a diferentes niveles, tal como se representa en la figura.

El proceso del desarrollo y pruebas de un proyecto de software suele representarse mediante un diagrama de ciclo como este:



Tests unitarios

Los desarrolladores implementan baterías de pruebas para los elementos que forman parte de un proyecto de software. Estos tests unitarios son la primera barrera de control contra posibles bugs. Son la base del "test driven development", si se quisiera seguir ese proceso. Agilizan el trabajo al poder cambiar partes del código y comprobar los fallos rápidamente. Son ejemplos para los nuevos desarrolladores que se añadan al proyecto.

Para el diseño y realización de pruebas unitarias en Java se utiliza la librería **Junit**.

Algunas ideas contrarias a los tests unitarios se basan en que un buen diseño, o una buena planificación, arregla muchos más problemas que los detectados por este tipo de tests. Sin embargo, aunque el diseño sea fantástico, nunca se puede asegurar que el código funciona sin estos tests.

Tests de integración

Los tests de integración, son los que verdaderamente comprueban que el sistema está funcionando. Unen partes del sistema y comprueban que encajan sin problemas. Son la base del "behaviour driven development", junto con los tests funcionales. Los tests unitarios no tienen en cuenta elementos tan importantes como los accesos, la base de datos, o peticiones de red. No son suficiente para comprobar que el comportamiento es correcto. Hacen que el sistema sea fiable, pudiendo confiar en las partes protegidas por estos tests, ya que tendrán el comportamiento deseado al ser llamados por otros componentes.

Tests funcionales

Son un paso más allá de los tests de integración, y tratan de probar el sistema como lo haría un usuario. Aquí entra especialmente la automatización de interfaces gráficas. Son las pruebas funcionales que más mantenimiento necesitan, y las más lentas. Sin embargo, los beneficios son similares a los anteriores tipos de pruebas. Protegen contra interfaces fallidas y quitan trabajo a la hora de hacer pruebas manuales. Son especialmente útiles al configurarlos y prepararlos, para ser ejecutados en todos los entornos en los que se distribuye o se ejecuta la aplicación. Por ejemplo, en los diferentes navegadores soportados, si la interfaz a probar fuera un entorno web.

¿Qué son las pruebas de software?



4 Metodologías en proyectos de diseño de software

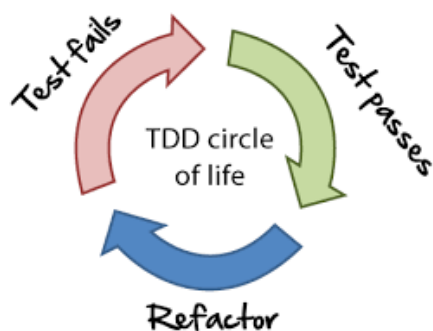
Los equipos de desarrollo de software necesitan utilizar técnicas y herramientas de diseño que ayuden a conseguir los objetivos establecidos.



La metodología **Agile** de desarrollo está basada en el desarrollo iterativo e incremental, donde los requisitos y soluciones evolucionan con el tiempo según la necesidad del proyecto.



Cada iteración del ciclo de vida incluye: planificación, análisis de requisitos, diseño, codificación, pruebas y documentación. Teniendo gran importancia el concepto de "Finalizado" (Done), ya que el objetivo de cada iteración no es agregar toda la funcionalidad para justificar el lanzamiento del producto al mercado, sino incrementar el valor por medio de "software que funciona" (sin errores).



El **diseño guiado por pruebas (TDD – Test Driven Development)** es una práctica basada en escribir las pruebas en primer lugar y conseguir que las pruebas den los resultados especificados mediante la refactorización del código en un proceso cíclico. Si las pruebas están bien diseñadas, se consigue un código limpio y escalable.