UNIDAD DE TRABAJO 1

INTRODUCCIÓN A LA PROGRAMACIÓN

1. ORÍGENES DE LA PROGRAMACIÓN I

- La programación **comenzó en la década de 1940** con los *lenguajes de máquina*, que eran difíciles de utilizar y propensos a errores.
- Charles Babbage y Ada Lovelace son pioneros destacados en la historia de la programación, sentando las bases teóricas y conceptuales.
- Con el tiempo, se desarrollaron *lenguajes de ensamblador* y *lenguajes de alto nivel*, que facilitaron la programación y permitieron expresar ideas y lógica de manera más clara y concisa.
- Estos avances impulsaron un rápido crecimiento y evolución en la programación, abriendo el camino hacia el desarrollo de software más accesible y productivo en las décadas siguientes.

1. ORÍGENES DE LA PROGRAMACIÓN II

1.1. GENERACIONES DE LOS LENGUAJES DE PROGRAMACIÓN I

- **Primera generación (lenguaje de máquina):** Los primeros lenguajes de programación, *basados en código de máquina*, eran de bajo nivel y *difíciles de entender y programar.*

- **Segunda generación (lenguajes ensambladores):** Los lenguajes ensambladores mejoraron la programación al utilizar *mnemónicos para representar las instrucciones del procesador*, ofreciendo mayor comprensión.

- **Tercera generación (lenguajes de alto nivel):** Los lenguajes de alto nivel, como FORTRAN, COBOL y ALGOL, hicieron la programación más accesible al introducir *estructuras de control avanzadas y abstracciones de nivel más alto*.

1. ORÍGENES DE LA PROGRAMACIÓN III

1.1. GENERACIONES DE LOS LENGUAJES DE PROGRAMACIÓN II

- **Cuarta generación (lenguajes de propósito específico):** Los lenguajes de cuarta generación *se especializaron en aplicaciones específicas, priorizando la productividad del programador*, como SQL para bases de datos y MATLAB para computación numérica.

- **Quinta generación (lenguajes de programación declarativos):** Los lenguajes de quinta generación adoptaron un enfoque declarativo, como Prolog, permitiendo *especificar qué se desea obtener en lugar de cómo hacerlo.*

2. DATOS, ALGORITMOS Y PROGRAMAS

- **Datos:** Los datos son la *información con la que trabajamos en un programa*. Pueden ser números, texto, imágenes, sonidos, entre otros. Los datos se almacenan en variables y se manipulan mediante operaciones específicas.

- **Algoritmos:** Un algoritmo es una *secuencia de pasos o instrucciones que se siguen para resolver un problema*. Un algoritmo toma los datos de entrada, realiza operaciones y produce una salida.

- **Programas:** Un programa es la *implementación de un algoritmo en un lenguaje de programación específico*. Consiste en un conjunto de instrucciones que se ejecutan secuencialmente para lograr un objetivo determinado.

3. PARADIGMAS DE PROGRAMACIÓN I

3.1. PROGRAMACIÓN ESTRUCTURADA I

- Se basa en la división del código en estructuras de control como secuencias, bucles y decisiones. Este paradigma busca la claridad y la simplicidad del código.

- Usa tres tipos de ***estructuras de control***:
  - **ESTRUCTURA SECUENCIAL:** es la forma más básica de control de flujo y simplemente indica que las instrucciones se ejecutan en orden, una después de la otra, de arriba hacia abajo. Cada instrucción se ejecuta una vez y luego se pasa a la siguiente.
    - **Ejemplo:**

```
Instrucción 1
Instrucción 2
Instrucción 3
```

3. PARADIGMAS DE PROGRAMACIÓN II

3.1. PROGRAMACIÓN ESTRUCTURADA II

✧ Usa tres tipos de ***estructuras de control***: (continuación)

- **ESTRUCTURA ALTERNATIVA:** permite tomar decisiones en el programa y ejecutar diferentes bloques de código según una condición o expresión booleana. Generalmente se utiliza la estructura "if-else" para implementar la lógica de la estructura alternativa.
  - **Ejemplo:**

```
if (condición) {
    // Bloque de código que se ejecuta si la condición es verdadera
} else {
    // Bloque de código que se ejecuta si la condición es falsa
}
```

  - *Si la condición especificada es verdadera, se ejecuta el primer bloque de código. Si la condición es falsa, se ejecuta el segundo bloque de código.*

3. PARADIGMAS DE PROGRAMACIÓN III

3.1. PROGRAMACIÓN ESTRUCTURADA III

✧ Usa tres tipos de ***estructuras de control***: (continuación)

- **ESTRUCTURA ITERATIVA:** permite repetir un bloque de código múltiples veces mientras que se cumpla una condición específica.
  - **Ejemplo** de estructura iterativa (bucle "while"):

```
while (condición) {
    // Bloque de código que se repite mientras la condición sea verdadera
}
```

  - **Ejemplo** de estructura iterativa (bucle "do-while"):

```
do {
    // Bloque de código que se repite mientras la condición sea verdadera
} while (condición);
```

- **PREGUNTA:** ¿Cuál es la diferencia entre las estructuras *do-while* y *while*?

3. PARADIGMAS DE PROGRAMACIÓN IV

3.2. PROGRAMACIÓN MODULAR I

- La programación modular es un *enfoque de diseño de software que busca dividir un programa en módulos más pequeños y autónomos*. Cada módulo tiene una función específica y se puede desarrollar, mantener y probar de manera independiente, aunque el programa final estará compuesto por una serie de módulos que colaborarán entre ellos.

- El **objetivo** en la programación modular es *lograr un equilibrio entre el acoplamiento y la cohesión*. Un sistema modular ideal tendrá un bajo acoplamiento y una alta cohesión, lo que implica que los módulos son independientes pero están bien organizados y enfocados en tareas específicas.

- Veamos qué son el **acoplamiento** y la **cohesión**…

3. PARADIGMAS DE PROGRAMACIÓN V

3.2. PROGRAMACIÓN MODULAR II

3.2.1. ACOPLAMIENTO I

- El **acoplamiento** se refiere al *grado de interdependencia entre los módulos de un sistema*. Indica cómo de fuertemente están conectados los módulos entre sí.

- Un **bajo acoplamiento** es deseable en la programación modular, ya que *indica que los módulos son independientes y se pueden modificar o reutilizar sin afectar a otros módulos*.

3. PARADIGMAS DE PROGRAMACIÓN VI

3.2. PROGRAMACIÓN MODULAR III
3.2.1. ACOPLAMIENTO II

✧ **Características** del *bajo acoplamiento*:

- **Independencia:** Los módulos no dependen excesivamente de otros módulos y pueden funcionar de forma autónoma.
- **Flexibilidad:** Los cambios en un módulo tienen un impacto mínimo en otros módulos, lo que facilita la modificación y evolución del sistema.
- **Reutilización:** Los módulos bien acoplados son más fáciles de reutilizar en otros proyectos, ya que no están estrechamente vinculados a un contexto específico.

3. PARADIGMAS DE PROGRAMACIÓN VII

3.2. PROGRAMACIÓN MODULAR IV
3.2.2. COHESIÓN I

✧ La **cohesión** se refiere a la relación interna y la unidad funcional de un módulo. *Indica cómo de estrechamente están relacionadas las partes del módulo y si se centran en una única tarea o responsabilidad*.

✧ Una **alta cohesión** es deseable en la programación modular, ya que *permite que los módulos sean más claros, comprensibles y mantenibles.*

3. PARADIGMAS DE PROGRAMACIÓN VIII

3.2. PROGRAMACIÓN MODULAR V
3.2.2. COHESIÓN II

✧ **Características** de *alta cohesión:*

- **Responsabilidad única:** Cada módulo tiene una tarea o función específica claramente definida.
- **Legibilidad:** La estructura interna de un módulo bien cohesionado es fácil de entender y seguir.
- **Mantenibilidad:** Los cambios y actualizaciones en un módulo afectan solo a su ámbito específico y no tienen un impacto generalizado en todo el sistema.
- **Pruebas:** Los módulos bien cohesionados son más fáciles de probar, ya que se pueden evaluar y verificar de manera aislada.

3. PARADIGMAS DE PROGRAMACIÓN IX

3.3. PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

✧ Se centra en la creación de objetos que encapsulan datos y funcionalidad. Los objetos interactúan entre sí a través de mensajes, lo que promueve la reutilización y la modularidad del código.

3.4. PROGRAMACIÓN FUNCIONAL

✧ Se basa en el uso de funciones como elementos fundamentales del programa. Las funciones se tratan como valores y se enfatiza la inmutabilidad de los datos.

3.5. OTROS PARADIGMAS

✧ Además de los mencionados, existen otros paradigmas de programación como la programación lógica, la programación basada en eventos, entre otros.

4. LENGUAJES DE PROGRAMACIÓN I

4.1. LENGUAJES DE BAJO NIVEL I

- Estos lenguajes *se acercan más al lenguaje de la máquina* y *requieren un mayor nivel de conocimiento técnico*.

- Son lenguajes que están más cercanos al hardware de la computadora y que *brindan un mayor control sobre los recursos y las instrucciones de la máquina*.

- **Ejemplo:** El **lenguaje de ensamblador x86** para procesadores de esa misma arquitectura, tanto de Intel como de AMD, es un lenguaje de bajo nivel, donde cada instrucción corresponde directamente a una operación realizada por el procesador.

4. LENGUAJES DE PROGRAMACIÓN II

4.1. LENGUAJES DE BAJO NIVEL II

4.1.1. LENGUAJE DE MÁQUINA

- El lenguaje de máquina es el más *básico y directamente entendido por la computadora*.

- *Consiste en instrucciones binarias que representan operaciones fundamentales del procesador*, como sumar y mover datos en memoria.

- *Estos programas son difíciles de leer y entender para los humanos* al expresarse en códigos numéricos y binarios.

- **EJEMPLO:** 01000011 10010001 00000000 00000001 (suma de dos valores)

4. LENGUAJES DE PROGRAMACIÓN III

4.1. LENGUAJES DE BAJO NIVEL III
4.1.2. LENGUAJE DE ENSAMBLADOR

- El lenguaje de ensamblador *proporciona una representación simbólica de las instrucciones de máquina*.

- *Cada instrucción de máquina se reemplaza por un mnemónico que es más legible para los programadores*, permitiendo utilizar símbolos, nombres y comentarios para facilitar la comprensión del código y su mantenimiento.

- Sin embargo, el lenguaje de ensamblador sigue estando directamente relacionado con las instrucciones de máquina y la estructura del hardware subyacente.

- **EJEMPLO:** ADD AX, BX (Hace la operación AX+BX y guarda el resultado en AX)

4. LENGUAJES DE PROGRAMACIÓN IV

4.1. LENGUAJES DE BAJO NIVEL IV
4.1.3. LENGUAJE C

- Aunque el lenguaje C es técnicamente un lenguaje de alto nivel, *se considera de bajo nivel debido a su estrecha relación con el hardware y su capacidad para manipular directamente la memoria y los registros del procesador*.

- El lenguaje C proporciona un mayor nivel de abstracción y portabilidad que los lenguajes de máquina y ensamblador, pero aún *requiere un conocimiento detallado del hardware y permite un control más directo sobre los recursos de la computadora*.

- **EJEMPLO:** a = a + b; (Guarda en la variable *a* el resultado de *a* + *b*)

4. LENGUAJES DE PROGRAMACIÓN V

4.2. LENGUAJES DE ALTO NIVEL I

- *Los lenguajes de alto nivel* se diseñan con una sintaxis cercana al lenguaje humano, lo que *facilita la escritura y comprensión del código*.

- **Ejemplos:** Python, **Java**, C++, entre otros.

- En nuestro caso, a lo largo de este módulo trabajaremos con **Java** ya que es ampliamente reconocido como un **lenguaje de programación multiplataforma**, lo que significa que *los programas escritos en Java pueden ejecutarse en diferentes sistemas operativos y arquitecturas sin la necesidad de modificar o reescribir el código fuente*. Esta capacidad de portabilidad es una de las ventajas clave de Java en comparación con otros lenguajes de programación.

4. LENGUAJES DE PROGRAMACIÓN VI

4.2. LENGUAJES DE ALTO NIVEL II

- Algunas de las ventajas de usar Java como lenguaje multiplataforma son:

1. **Portabilidad:** Java se ejecuta en diversas plataformas sin necesidad de modificar el código, gracias a la JVM (Java Virtual Machine, Máquina Virtual de Java).

2. **Independencia de arquitectura:** Funciona en sistemas con distintas arquitecturas de hardware.

3. **Amplia disponibilidad:** Es ampliamente adoptado en computadoras, dispositivos móviles y sistemas integrados.

4. LENGUAJES DE PROGRAMACIÓN VII

4.2. LENGUAJES DE ALTO NIVEL III

✧ Algunas de las ventajas de usar JAVA como lenguaje multiplataforma son:

4. **Bibliotecas y frameworks:** Cuenta con numerosas bibliotecas y frameworks de código abierto para acelerar el desarrollo de aplicaciones. Un **framework** es una *infraestructura que ofrece un marco de trabajo para facilitar la creación, organización y mantenimiento de software.*

5. **Seguridad:** La JVM implementa mecanismos de seguridad que lo hacen popular para aplicaciones seguras.

6. **Madurez y estabilidad:** Ha sido ampliamente utilizado y mejorado a lo largo del tiempo, brindando estabilidad para el desarrollo de aplicaciones.

4. LENGUAJES DE PROGRAMACIÓN VIII

4.3. LENGUAJES DE SCRIPTING

✧ Los lenguajes de scripting son *diseñados para escribir **scripts**, secuencias de instrucciones que se ejecutan sin compilar previamente*.

✧ *Se utilizan para automatizar tareas, controlar programas y sistemas, y proveer una interfaz en aplicaciones específicas*.

✧ *No necesitan una compilación previa*, ya que utilizan un intérprete para ejecutar las instrucciones en tiempo real, lo que permite cambios rápidos y pruebas sin recompilar el código.

✧ **EJEMPLOS:** Bash, Python, Perl, Ruby.

5. HERRAMIENTAS Y ENTORNOS PARA EL DESARROLLO DE PROGRAMAS I

5.1. EDITORES DE CÓDIGO

- Son herramientas básicas que permiten escribir y editar código fuente. Ejemplos populares incluyen Visual Studio Code, Atom y Sublime Text.

5.2. ENTORNOS DE DESARROLLO INTEGRADO (IDE) I

- Son herramientas más completas que ofrecen características como resaltado de sintaxis, depuración, autocompletado y gestión de proyectos.

- **Ejemplos:** Eclipse, PyCharm, Visual Studio.

- En nuestro caso usaremos **Eclipse** porque es un entorno de desarrollo integrado (IDE) ampliamente utilizado para el desarrollo de aplicaciones Java, aunque es importante tener en cuenta que la elección de un IDE depende en gran medida de las preferencias personales y los requisitos específicos del proyecto.

5. HERRAMIENTAS Y ENTORNOS PARA EL DESARROLLO DE PROGRAMAS II

5.2. ENTORNOS DE DESARROLLO INTEGRADO (IDE) II

- Aquí se enumeran algunas de sus ventajas:

1. **Amplia comunidad y soporte:** Cuenta con una gran comunidad y desarrolladores que brindan ayuda y recursos a través de foros y grupos de discusión.

2. **Versatilidad y extensibilidad:** Su arquitectura modular permite instalar una variedad de complementos y adaptar el IDE a diferentes necesidades de desarrollo.

3. **Depuración y herramientas de desarrollo:** Ofrece herramientas potentes de depuración, autocompletado, refactorización y análisis estático para escribir y depurar código eficientemente.

5. HERRAMIENTAS Y ENTORNOS PARA EL DESARROLLO DE PROGRAMAS III

5.2. ENTORNOS DE DESARROLLO INTEGRADO (IDE) III

- Aquí se enumeran algunas de sus ventajas: (continuación …)

4. Integración de control de versiones: Tiene una sólida integración con sistemas de control de versiones, como Git y Subversion, facilitando la gestión de cambios en el código fuente.

5. Interfaz de usuario intuitiva: Proporciona una interfaz personalizable con atajos de teclado y navegación para mejorar la productividad del desarrollo.

6. Soporte multiplataforma: Es un IDE multiplataforma que se ejecuta en Windows, macOS y Linux, permitiendo a los desarrolladores trabajar en su sistema operativo preferido.

5. HERRAMIENTAS Y ENTORNOS PARA EL DESARROLLO DE PROGRAMAS IV

5.3. COMPILADORES E INTÉRPRETES

- *Traducen el código fuente a un código ejecutable*. Los **compiladores** *traducen el código completo de una vez*, mientras que los **intérpretes** lo *traducen línea por línea durante la ejecución*.

- En el caso de **Java** estamos ante un *lenguaje de programación híbrido*, lo que significa que combina características de lenguajes compilados e interpretados. *Aunque Java se compila en bytecode*, que es un código intermedio ejecutable, *su ejecución final es interpretada por la máquina virtual de Java (JVM)*.

- La combinación de compilación a bytecode y ejecución interpretada en la JVM brinda a Java la capacidad de ofrecer portabilidad, seguridad y flexibilidad, mientras que también permite optimizaciones de rendimiento.

6. ERRORES Y CALIDAD DE LOS PROGRAMAS I

6.1. ERRORES DE LOS PROGRAMAS I

- Los errores son *inevitables en la programación*.

- Es importante tener en cuenta que la **detección y corrección de errores** es una *parte esencial del proceso de desarrollo de software*.

- Utilizando **técnicas de depuración y pruebas,** los programadores pueden *identificar y corregir errores en sus programas para garantizar que funcionen correctamente*.

6. ERRORES Y CALIDAD DE LOS PROGRAMAS II

6.1. ERRORES DE LOS PROGRAMAS II

- Los errores con los que nos podemos encontrar al crear o ejecutar un programa son:

  1. **Errores de sintaxis:** *Incumplimiento de reglas del lenguaje*, como falta de paréntesis o llaves, errores de palabras clave, entre otros, detectados por el compilador o entorno de desarrollo durante la compilación.

  2. **Errores de tiempo de ejecución:** Ocurren durante la ejecución del programa debido a situaciones excepcionales, como división por cero o desbordamiento de memoria, provocando cierres inesperados o resultados incorrectos.

  3. **Errores lógicos:** Relacionados con *razonamientos incorrectos en el diseño o implementación del programa*, sin generar mensajes de error, pero causando resultados no deseados.

6. ERRORES Y CALIDAD DE LOS PROGRAMAS III

6.2. CALIDAD DE LOS PROGRAMAS I

- La calidad de un programa *se mejora mediante pruebas exhaustivas y la depuración de errores*. Las **pruebas** pueden ser manuales o automatizadas, y *ayudan a identificar problemas y verificar el correcto funcionamiento del programa.*

- Cabe mencionar que la **calidad del software** es un *concepto multidimensional y complejo*, y la combinación de diferentes estándares y metodologías puede ser utilizada para evaluar diferentes aspectos de la calidad en función de las necesidades y requisitos específicos del proyecto o la organización.

6. ERRORES Y CALIDAD DE LOS PROGRAMAS IV

6.2. CALIDAD DE LOS PROGRAMAS II

- Existen varios estándares y metodologías ampliamente utilizadas para medir la calidad de un programa o software. A continuación, veamos algunos de los estándares más comunes:

1. **ISO/IEC 9126:** Estándar internacional que define características y subcaracterísticas para evaluar la calidad del software, abarcando funcionalidad, confiabilidad, usabilidad, eficiencia, mantenibilidad y portabilidad.

2. **Modelo de madurez y capacidad (CMMI):** Marco de mejora de procesos con múltiples disciplinas, incluido el desarrollo de software, proporcionando criterios para evaluar y mejorar la calidad de procesos y productos, cuyo nivel más alto es el 5.

6. ERRORES Y CALIDAD DE LOS PROGRAMAS V

6.2. CALIDAD DE LOS PROGRAMAS III

3. **Métricas de mantenibilidad del software:** Métricas específicas como índice de mantenibilidad, complejidad ciclomática, duplicación de código, acoplamiento y cohesión evalúan la facilidad de mantener, modificar y evolucionar el software a lo largo del tiempo.

4. **Pruebas y control de calidad:** Esenciales para evaluar y asegurar la calidad del software, involucran pruebas de unidad, integración, regresión, rendimiento, aceptación y enfoques como TDD y BDD para garantizar calidad mediante pruebas automatizadas.

5. **Estándares de codificación y buenas prácticas:** Definen guías para el código limpio, legible y mantenible, como el estilo de codificación de Google, el estilo de codificación de Java de Oracle y el PEP 8 de Python.

7. FASES EN LA CREACIÓN Y EJECUCIÓN DE UN PROGRAMA I

7.1. FASES EN LA CREACIÓN DE UN PROGRAMA I

1. **Análisis y especificación de requisitos:** Se identifican y comprenden los requisitos del software en colaboración con los clientes, usuarios finales y partes interesadas, documentando las funcionalidades necesarias.

2. **Diseño:** Los requisitos se traducen en un diseño técnico, definiendo estructuras de datos, algoritmos y diagramas para la arquitectura del programa.

3. **Codificación:** Se implementa el diseño en un lenguaje de programación específico, siguiendo las mejores prácticas y estándares de programación.

7. FASES EN LA CREACIÓN Y EJECUCIÓN DE UN PROGRAMA II

7.1. FASES EN LA CREACIÓN DE UN PROGRAMA II

4. Pruebas: Se realizan pruebas exhaustivas para verificar el correcto funcionamiento del software, identificando y corrigiendo errores y comportamientos inesperados.

5. Integración y despliegue: Se integra el software con otros componentes y se prepara para su despliegue en el entorno de producción o hardware objetivo.

6. Mantenimiento: Se inicia la fase de mantenimiento con actualizaciones, correcciones de errores y mejoras continuas para adaptarse a las necesidades cambiantes de los usuarios y requisitos del sistema.

7. FASES EN LA CREACIÓN Y EJECUCIÓN DE UN PROGRAMA III

7.1. FASES EN LA CREACIÓN DE UN PROGRAMA III

- Es **importante** destacar que *estas fases no son estrictamente secuenciales y pueden solaparse en ciertos momentos del desarrollo*.
- Además, **el ciclo de desarrollo de software es iterativo,** lo que significa que *las fases se repiten a medida que se obtiene retroalimentación y se realizan mejoras*.
- Los profesionales de la programación siguen estas fases para asegurar un proceso organizado y controlado en la creación de software de calidad.

7. FASES EN LA CREACIÓN Y EJECUCIÓN DE UN PROGRAMA IV

7.1. FASES EN LA CREACIÓN DE UN PROGRAMA IV

7.1.1. MÉTRICA 3 I

- En el ámbito de la ingeniería del software en España, Métrica 3 es una *metodología de desarrollo de software que fue desarrollada por el Ministerio de Hacienda y Administraciones Públicas de España*.

- Esta metodología se utilizó ampliamente en proyectos de tecnología de la información del sector público en España, aunque también se ha adoptado en ciertos ámbitos del sector privado.

- Es importante destacar que **Métrica 3** es una *metodología específica de España y su uso ha sido más prevalente en el sector público*.

7. FASES EN LA CREACIÓN Y EJECUCIÓN DE UN PROGRAMA V

7.1. FASES EN LA CREACIÓN DE UN PROGRAMA V

7.1.1. MÉTRICA 3 II

- Métrica 3 define un conjunto de fases, actividades y productos que guían el proceso de desarrollo de software. Estas fases son:

1. **Inicio del proyecto:** Evaluación inicial, definición del alcance, objetivos y plan preliminar.

2. **Estudio de viabilidad:** Análisis detallado de viabilidad técnica, económica y organizativa, identificación de requisitos y evaluación de riesgos.

3. **Análisis del sistema:** Definición de requisitos, funciones y especificación de requisitos.

7. FASES EN LA CREACIÓN Y EJECUCIÓN DE UN PROGRAMA VI

7.1. FASES EN LA CREACIÓN DE UN PROGRAMA VI

7.1.1. MÉTRICA 3 III

4. Diseño del sistema: Diseño detallado con estructuras de datos, algoritmos y arquitectura, especificación de diseño.

5. Construcción del sistema: Implementación del software, codificación, pruebas unitarias y documentación.

6. Implantación y aceptación del sistema: Integración, pruebas de aceptación por usuarios finales y puesta en producción.

7. Mantenimiento del sistema: Actualizaciones, correcciones de errores y mejoras para cumplir con requisitos cambiantes.

7. FASES EN LA CREACIÓN Y EJECUCIÓN DE UN PROGRAMA VII

7.1. FASES EN LA EJECUCIÓN DE UN PROGRAMA I

- Conocer las fases de ejecución de un programa es esencial para comprender cómo se ejecuta un programa y cómo interactúa con su entorno de ejecución.

- Es importante destacar que estas fases de ejecución *pueden variar según el entorno de desarrollo y el lenguaje de programación utilizado*. Por ejemplo, algunos lenguajes, como los interpretados, pueden tener una fase de interpretación en lugar de la compilación y el enlazado.

- Además, los sistemas operativos y las plataformas específicas pueden agregar etapas adicionales o variaciones en el proceso de ejecución.

7. FASES EN LA CREACIÓN Y EJECUCIÓN DE UN PROGRAMA VIII

7.1. FASES EN LA EJECUCIÓN DE UN PROGRAMA II

✧ Veamos cuáles son esas fases:

1. **Compilación:** Traducción del código fuente a lenguaje de bajo nivel mediante un compilador, generando código ejecutable.

2. **Enlazado (linking):** Combinación de código compilado con bibliotecas externas para crear el archivo ejecutable final y resolver referencias a símbolos externos.

3. **Carga (loading):** El programa compilado y enlazado se carga en la memoria principal del sistema, asignando espacio y resolviendo referencias simbólicas.

7. FASES EN LA CREACIÓN Y EJECUCIÓN DE UN PROGRAMA IX

7.1. FASES EN LA EJECUCIÓN DE UN PROGRAMA III

✧ Veamos cuáles son esas fases: (continuación …)

4. **Ejecución:** Las instrucciones del programa se interpretan o ejecutan directamente por el procesador, realizando operaciones y cálculos según la lógica programada.

5. **Finalización:** Al terminar la ejecución o alcanzar un punto de salida definido, se liberan recursos, cierran archivos y se realiza limpieza antes de que el programa termine su ejecución.

8. DESPEDIDA

- **Estos son solo algunos aspectos básicos sobre la introducción a la programación.** A medida que avancemos en el módulo de *Programación*, exploraremos en mayor detalle cada uno de estos temas y profundizaremos en conceptos más avanzados.

¡Este es el camino!