

TRABAJO INTEGRADOR

Conceptos y Paradigmas de Lenguajes de Programación

Integrantes del Grupo 36:

Dioguardi Felipe: 16211/4

Lettieri Juan de Dios: 16398/2

Loza Bonora Leonardo Germán: 16181/7

Moschettoni Martín: 15836/0

Introducción

En este informe se explorarán las diferentes características de los lenguajes de programación C y Java, ambos enormemente difundidos en la comunidad informática, uno por un poder de permanencia que asegura su funcionamiento a largo plazo, el otro por su flexibilidad, eficiencia y manejo de instrucciones de bajo nivel.

Lenguaje C

Es un lenguaje de programación de propósito general con el que se han creado programas de diversas disciplinas, como el sistema operativo UNIX y los programas que corren en él.

C presenta una jerarquía de tipos de datos, los fundamentales (caracteres, enteros, números de punto flotante, y "void"), y sus derivados, creados a partir de punteros, arreglos, funciones, uniones y estructuras. Además, los tipos pueden tener calificadores que aporten propiedades especiales.

Lenguaje Java

Es un lenguaje de programación orientado a objetos que se incorporó al ámbito de la informática en los años noventa. La idea del mismo es que permita crear programas que puedan ejecutarse en cualquier contexto y ambiente, siendo así la portabilidad uno de sus principales logros.

Criterios de comparación

Una de las herramientas que se tiene para comparar los distintos lenguajes de programación son los llamados "Criterios de Evaluación", éstos presentan distintas características que poseen los lenguajes, y serán usados en este documento para comparar Java y C.

Simplicidad y Legibilidad

Si un lenguaje tiene demasiados constructos o estos son muy diferentes entre sí, los programadores que pretendan utilizarlo deberán hacer un esfuerzo extra para familiarizarse con todos ellos. Esto suele llevar a un mal uso de los diversos aspectos y funcionalidades del lenguaje, al desuso de otras que suelen ser más elegantes y/o más eficientes que las que finalmente se utilizan, e incluso al uso indebido de mecanismos ya existentes pero desconocidos para el programador.

Se dice que Java es un lenguaje simple ya que no presenta ambigüedad. Se deben declarar los tipos de las variables, los tipos de retorno, y los tipos de los parámetros, además de la visibilidad:

```
public int factorial(int x)
```


Es lo suficientemente expresivo para entender que el método es público, devuelve un entero como resultado, y recibe un entero como parámetro.

C tiene una forma simple de definir nuevas estructuras, que hacen muy fácil el uso de las mismas:

```
struct nombreEstructura{
    tipo variable1;
    tipo variable2;
}
```

El mismo estilo se mantiene para toda la sintaxis de C, de la cual Java hereda. Esto hace de C un lenguaje simple y compacto, pero muy poderoso.

Claridad en los bindings

Binding es un concepto central en la definición de un lenguaje de programación que se refiere a especificar la naturaleza exacta de cada atributo, y se subdivide según cuando lo haga. Java por su parte presenta  binding estático y dinámico:

- 1) El binding estático ocurre en compilación mientras que el dinámico ocurre en ejecución.
- 2) Las variables y métodos private, final, static usan binding estático mientras que los métodos virtuales usan binding dinámico basados.
- 3) El binding estático usa la información de la clase Type mientras que el binding dinámico usa Object para resolverlo.
- 4) Los métodos “overloaded” son resueltos con binding estático mientras que los métodos “overridden” son resueltos con binding dinámico.

En el siguiente es un ejemplo de binding estático porque el compilador lee la llamada a printSomething y solo sabe que “a” es un vehículo:

```
public static void printSomething(Vehicle vehicle) {
    System.out.println("I'm a vehicle");
} public static void printSomething(Vehicle plane) {
    System.out.println("I'm a plane");
} public static void main(String args[]) {
    Vehicle a = new Plane();
    printSomething(a);
}
```

Sin embargo en este caso el binding será dinámico, porque es necesario esperar a la ejecución para saber si “a” entiende el método start y a que método se está invocando:

```
class Vehicle {
    public void start() {
        System.out.println("Inside start method of Vehicle");
    }
} class Car extends Vehicle {
    @Override
    public void start() {
        System.out.println("Inside start method of Car");
    }
} public class DynamicBindingTest {
```

```
        public static void main(String args[]) {
            Vehicle a = new Car();
            a.start();
        }
    }
```

El binding de C, ocurre en muchos momentos. Durante la definición del lenguaje, en la que se especifica y corrige la sintaxis, palabras clave, palabras reservadas, significado de operadores y de objetos primitivos; durante la implementación del lenguaje, cuando se genera una representación interna de los literales; en tiempo de compilación, cuando se especifica el tipo de una variable; en linking, cuando se realizan las llamadas a las librerías; en loading, cuando se asignan las celdas de memoria a las entidades; y en tiempo de ejecución, cuando se guardan los valores en las variables correspondientes. Aquí un ejemplo:

```
int i, x = 0;
void main() {
    for (i = 1; i <= 50; i++)
        x += do_something(x);
}
```

La especificación del lenguaje le da significado a la palabra clave `int` mientras que esta se le asigna a las variables `x` e `y` en tiempo de compilación. La referencia de la función `do_something` puede estar en otro archivo, lo cual se resuelve en link time. En loading time se asignan los espacios de memoria de las entidades `main`, `do_something`, `i` y `x`. Los posibles valores que puedan tener `x` e `y` se enlazan en tiempo de ejecución.

Confiabilidad

La confiabilidad está relacionada con la seguridad del lenguaje a la hora de evitar y manejar errores. Java utiliza “static checking” para analizar el programa durante el tiempo de compilación para demostrar la ausencia de errores de tipo. La idea básica es que nunca ocurran problemas que se podrían haber corregido durante el tiempo de ejecución.

A comparación de otros lenguajes C-Like, Java requiere declaraciones explícitas y no soporta declaraciones implícitas.

El siguiente es un ejemplo de 2 clases en Java:

```
class A {
    A me() { return this; }
    public void doA() { System.out.println("Do A"); }
}
class B extends A {
    public void doB() { System.out.println("Do B"); }
}
```

Si se ejecuta “**new B().me()**”, durante el tiempo de compilación, el compilador solo ve que el método **me()** devuelve una instancia de A. Sin embargo, en realidad el método devuelve un objeto B durante tiempo de ejecución, ya que B hereda los métodos de A y en el “return this” se devuelve a él mismo.

Por lo anteriormente descrito la expresión “**new B().me().doB()**” sería ilegal, mientras que “**new B().me().doA()**” estaría permitida. Sin embargo una solución sería castear el objeto como B de la siguiente forma: “**((B) new B().me()).doB()**”.

De igual manera, en C los compiladores advierten de la mayoría de los errores de tipo. No hay conversión automática de tipos de datos incompatibles, sin embargo, C conserva la filosofía básica que los programadores saben lo que están haciendo.

Para manejar errores que el compilador no pudo corregir, existe en Java el concepto de excepción, un error o situación excepcional que se produce durante la ejecución de un programa. Estas se representan a través de objetos que heredan de la clase “**java.lang.Throwable**”. Existen tres tipos de excepciones:

- Los “errors” son errores graves de la Java Virtual Machine que normalmente no se tratan en los programas de Java.
- Las “checked exceptions” son aquellas que son chequeados por el compilador, son ajenas al código.
- Las “unchecked exceptions” son aquellas que no son chequeadas por el compilador. Representan un error de programación y son subclase directa de RuntimeException.

Las excepciones se manejan de la siguiente forma:

```
try {  
    //código que puede producir excepciones  
} catch(TipoDeExcepcionN eN) {  
    //código que trata excepciones de tipo TipoExcepciónN o subclases  
} finally {  
    //código de finalización (opcional)  
}
```

El lenguaje de programación C no admite el manejo de excepciones ni el manejo de errores. En caso de error la mayoría de las funciones devuelven un valor nulo o -1. La incorporación del archivo de encabezado **<errno.h>** simplifica las cosas.

Básicamente hay 2 tipos de funciones asociadas con errno:

- perror (): es responsable de mostrar la cadena que se le envíe, seguida de dos puntos, un espacio y luego la representación textual del valor actual de errno.
- strerror (): es responsable de devolver un puntero a la representación textual del valor actual de errno.

Este programa que intenta abrir un archivo inexistente da como resultado: “El valor de errno es: 2”:

```
#include<stdio.h>  
#include<errno.h>  
int main()  
{  
    FILE *fpointer;  
    fpointer = fopen("Archivolnexistente.txt", "r"); // Abriendo un archivo  
    printf(" El valor de errno es: %d\n", errno);  
    return 0;  
}
```

Otros casos, como la división por 0, deben ser manejados explícitamente:

```
#include <stdio.h>
#include <stdlib.h>
void main() {
    int dividendo = 60;
    int divisor = 0;
    int resultado;
    if (divisor == 0) {
        fprintf(stderr, "División por 0. Saliendo\n");
        getch();
        exit(-1);
    }
    cociente = dividendo / divisor;
    fprintf(stderr, "Valor del cociente: %d\n", q);
    getch();
    exit(0);
}
```

Soporte

El compilador de C está disponible en todos los sistemas UNIX, y se puede descargar gratuitamente de internet para otros sistemas operativos. También existe una gran cantidad de compiladores online, y algunos de pago.

Es un lenguaje que, además, está muy bien documentado en el libro *The C Programming Language* escrito por Brian Kernighan y Dennis Ritchie (quien diseñó e implementó el lenguaje), y muchos otros manuales y tutoriales creados por su extensa comunidad.

C está estandarizado. El más reciente es el ISO/IEC 9899:2018, comúnmente abreviado como C18 y creado por la Organización Internacional de la Estandarización (ISO). Éste estándar es otra forma de aprender el lenguaje, aunque es un documento pago y puede ser complejo para nuevos usuarios. Una buena alternativa puede ser el Manual de Referencia de GNU, que contiene buenas prácticas (no tutoriales) del lenguaje.

Java es uno de los lenguajes más populares y utilizados. En consecuencia, existen centenas de cursos y tutoriales en internet. Además su documentación oficial de libre uso puede encontrarse en docs.oracle.com/java.

Las licencias de Java SE son manejadas por Oracle. Aunque se permite su uso gratuito con fines personales y para el desarrollo, el soporte y la licencia comercial son de pago. También provee el JDK, bajo la licencia open source GPL, con el nombre de OpenJDK que incluye el compilador y otras herramientas para desarrollar aplicaciones de Java.

Para correr las aplicaciones de Java existe un conjunto de utilidades denominado JRE (Java Runtime Environment) que incluye la Máquina Virtual de Java (JVM), un conjunto de librerías y otros componentes necesarios.

La JVM actúa de intermediario entre el sistema operativo y Java, lo que permite que sus aplicaciones puedan ser usadas en diferentes plataformas en las que se pueda instalar, logrando así que el lenguaje sea independiente de la plataforma.

Abstracción

Las librerías permiten a los programadores reusar código y esconder la implementación de la funcionalidad al programador. Estas se referencian al principio del código con la sentencia “**#include <libreria.h>**”.

Por ejemplo, la librería estándar de C se referencia como “**libc**”. Esta provee una interfaz básica al sistema como las llamadas **read()**, **write()** y **printf()**.

Para hacer una librería de funciones se necesita crear 2 archivos, llamados header file y code file. El header file define el encabezado de las funciones que la librería va a contener, y el/los code file/s las declara. En los code files deben incluirse referencias al header file que complementan.

Un ejemplo de un header file que define la función `add(int a, int b)`, y un code file complementario es:

```
/* File add.h /
#ifndef ADD_H
#define ADD_H
int add(int, int);
#endif / ADD_H */

/* File add.c */
#include "add.h"
int add(int a, int b) { return a + b; };
```

Lo mismo puede hacerse para definir estructuras reutilizables, pudiendo abstraerse del código para el que fueron pensadas en un primer lugar.

```
/* File item.h */
#ifndef ITEM_H
#define ITEM_H
typedef struct ItemStruct *Item;
#endif /* ITEM_H */

/* File item.c */
#include "item.h"
struct ItemStruct {
    char word[MAXSTRING];
    int number;
};
```

Esto es lo más cercano en C a la abstracción alcanzable con un lenguaje orientado a objetos como Java, que tiene la capacidad de definir clases como estructuras genéricas reutilizables de diversas maneras.

Además del acercamiento de C, con la definición de librerías como concepto de modularización entre programas, Java va un paso más allá sumando conceptos típicos de su paradigma; la creación de clases abstractas para la hipergeneralización de código, polimorfismo para que los objetos que utilizan alguna funcionalidad no definida en su clase no tengan que preocuparse de cual la implementa realmente, y principalmente la herencia, de la que derivan los dos mencionados anteriormente.

Gracias a estas características, es posible obviar detalles sobre implementación de estructuras y funciones y centrarte en la funcionalidad, haciendo más fácil la creación de otras estructuras nuevas y más complejas, que puedan usar las anteriores y ser usadas por otro programa de la misma forma.

```
typedef int vector [10];
```

Tanto Java como C pueden declarar nuevos tipos. Esta declaración establece una ligadura en tiempo de traducción entre el nuevo tipo "vector" y su implementación (el arreglo de 10 números enteros). Como consecuencia de esta ligadura, "vector" hereda todas las operaciones de los arreglos.

Java soporta la implementación de tipos definidos por los usuarios, permitiéndole al nuevo tipo heredar variables y comportamiento de otros, y definir nuevas operaciones que las instancias de ese tipo podrán ejecutar. Estos tipos pueden ser instanciables, o solo agrupar la información que las instancias de clases hijas podrán utilizar. A estas últimas se les llama clases abstractas, y se pueden definir de la siguiente manera:

```
abstract class Nombre_Clase {  
    public void nombreMétodo() {  
        System.out.println("");  
    }  
}
```

Ortogonalidad

Ortogonalidad significa que distintos aspectos del lenguaje pueden ser usados en cualquier combinación, y que esas combinaciones tienen sentido. Además, que el significado de un aspecto determinado es consistente, sin importar con qué otros aspectos es combinado.

Se considera que C tiene inconsistencias en cuanto a los arreglos, ya que los arreglos no pueden ser devueltos por las funciones, pero los structs sí, aunque tengan arreglos dentro.

```
#include <stdio.h>  
struct Arreglo { int arreglo[1]; };  
struct Arreglo prepararArreglo() {  
    struct Arreglo a;  
    a.arreglo[0] = 23;  
    return a;  
};  
int main() {  
    struct Arreglo arr = prepararArreglo();  
    printf("%d",arr.arreglo[0]);  
}
```

En el caso de Java, el cómo funciona la asignación a variables demuestra una falta de ortogonalidad. Por ejemplo, cuando se asigna un objeto a una variable, se le asigna una referencia a ese objeto. Sin embargo, si se le asigna un tipo primitivo, se le asigna el valor.

```
public static void main(String []args) {  
    int var = 3, var2 = var;  
    var = 4;
```

```
System.out.println(var + " " + var2); //Imprime 4 y 3, porque sólo asigna el valor  
Objeto var3,var4;  
var3 = new Objeto();  
var4 = var3;  
var3.valor = 19;  
System.out.println(objeto.valor + " " + var4.valor); //Aquí imprime 19 dos veces  
}
```

Eficiencia

Antiguamente la eficiencia de un programa era el criterio de diseño más importante, debido a la lentitud de las máquinas y la necesidad de una ejecución veloz. Hoy en día a la eficiencia la componen otros aspectos:

- El código máquina: el diseño del lenguaje debe permitir a un traductor generar código ejecutable eficiente. Por ejemplo, las clases de Java, si no son utilizadas aplicando todos los aspectos del paradigma de objetos, tienen un consumo de espacio no tan diferente que un struct de C.
- La traducción: el código de alto nivel debe ser traducido rápidamente y por un traductor de un tamaño razonable. Por ejemplo, en C se puede usar un “One-pass compiler”, un compilador que pasa a través de las unidades de compilación sólo una vez, traduciendo cada parte al código máquina final. Esto es posible gracias a que C obliga a declarar todas las variables antes de usarlas. En el caso de Java, como debe interpretar y compilar a la vez, la velocidad de traducción se ve afectada, siendo más lenta que en C.
- Implementabilidad: la eficiencia con la que un traductor se puede escribir.
- La programación: la rapidez y facilidad con la que un programador puede trabajar en un determinado lenguaje

La expresividad también juega un papel en la eficiencia. Un lenguaje expresivo permite una fácil representación de procesos y estructuras complejas. Esto puede ayudar a que el diseño que un programador tiene pensado sea fácil de escribir en el lenguaje que está usando.

Sintaxis

Es el conjunto de reglas que definen como componer letras, dígitos y otros caracteres para formar los programas. Las reglas sintácticas dictan cómo formar expresiones y programas correctos utilizando sentencias compuestas por words. Con estas reglas se puede determinar si una sentencia de un programa es legal en ese lenguaje.

Los programas en C están conformados por un conjunto de funciones que debe contener a la función **main()**, descrita como:

```
main () {  
    Lista_de_sentencias;  
}
```

El resto de funciones, deberían ser declaradas antes de su utilización. En el caso contrario el compilador asumiría que la función tiene la estructura

```
int nombreDeFuncion();
```

Si efectivamente se llamase a la función sin declararla antes y esta no devolviera un entero, el compilador generaría un error.

En cuanto a los parámetros de las funciones, si la lista de parámetros está vacía (como en el ejemplo anterior) significa que la función puede ser llamada con cualquier cantidad y tipo de parámetros. Para asegurarse de que la función solo puede ser llamada si no recibe ningún parámetro, se declara:

```
tipoDeRetorno nombreDeFuncion(void);
```

Las funciones en Java tienen una estructura distinta a las de C

```
modDeAcceso [static] tipoRetorno nomFuncion(tipoDeParametro nomParametro) {  
    ListaDeSentencias;  
}
```

Java, al ser un lenguaje orientado a objetos, necesita indicar un modificador de acceso del que C carece. Esto indica si una función puede o no ser accedida desde otras clases. El miembro static indica si la función es ejecutable por la clase o una instancia de esa clase.

El resto de la declaración funciona igual que C, excepto en la lista de parámetros. Cuando esta está vacía significa que esa función no puede recibir parámetros.

Los arreglos en C se pueden declarar de varias maneras:

```
int arreglo[10]; (Definiendo el tamaño)  
int arreglo[] = {5, 3, 9, 12}; (Inicializando elementos)  
int arreglo[3] = {5, 3, 9}; (Ambos)
```

Como Java deriva de C, adoptó gran parte de su sintaxis. La definición e instanciación de arreglos, por ejemplo, es muy parecida en ambos:

```
tipo[] nombreVariable;  
nombreVariable = new tipo[tamaño];
```

Y ambas sentencias pueden combinarse en una de varias formas:

```
tipo[] nombreVariable = new tipo[tamaño];  
tipo[] nombreVariable = {valor1, valor2, valor3};
```

Semántica

La semántica hace referencia al estudio y comprobación del significado de sentencias sintácticamente válidas. No es fácil describirla ya que esto se puede hacer de diversas maneras. A grandes rasgos se puede dividir la semántica en dos tipos:


Semántica Estática:

Toma protagonismo antes de la ejecución del programa, durante la compilación. Su principal función es la de chequear sentencias contextualmente ambiguas. Se procesan las

declaraciones, tipos, números de parámetros para una función determinada, etc. Se crea una Tabla de Símbolos conteniendo toda la información de cada token identificado en el análisis sintáctico (identificador, tipo, operaciones permitidas, palabras clave, etc).

C	Java
<pre>int main() { int x; x++; printf("%d",x); return 0; }</pre>	<pre>public class Main { public static void main(String[] args) { int x; x++; System.out.println(x); } }</pre>

En esta comparación se puede observar cómo impacta la semántica estática en la compilación del código fuente. En ambos casos se observa código sintácticamente correcto que difiere en su semántica debido a cómo se tratan las declaraciones de variables durante el análisis de semántica estática y la formación de la Tabla de Símbolos.

En C, al momento de la declaración de una variable integer, se la inicializa en  mientras que en Java las variables locales no se inicializan nunca. Por lo tanto el código en C es semánticamente válido, y la sentencia `printf("%d",x)` hará que se imprima un 1 en ejecución, mientras que en Java se obtendrá un error de compilación puesto que la sentencia `x++` causa un conflicto de valor al no estar inicializada, es decir, no se puede incrementar un valor que no existe.

Semántica Dinámica:

Durante la ejecución de un programa es probable que algunas de las variables que hayan sido declaradas cambien de valor. Es en estos casos que se recurre a la semántica dinámica para hacer todos los chequeos necesarios a medida que son requeridos en tiempo de ejecución, para asegurar la integridad del código.

<pre>for (expr1; expr2; expr3) { ... }</pre>	<pre> expr1 loop: if expr2 == 0 goto out ... expr3; goto loop out: ...</pre>
--	---

Semánticamente, tanto en C como en Java la instrucción `for` representa el `loop` de la derecha; lo que ilustra cómo maneja valores de variables durante la ejecución del código que se modifican de manera dinámica.

La explicación de la sentencia `for` mediante una serie de instrucciones de bajo nivel se corresponde con la **semántica operacional**, que es un método informal muy usado para describir la semántica. Suponiendo que la expresión 1 sea una variable de tipo integer (podría ser de otro tipo, como `char`), se chequea si se cumple la expresión 2, esto es, que devuelva un 0; si esta condición no se cumple se ejecutarán una serie de sentencias que incluyen la expresión 3, que involucra la actualización del valor de la expresión 1; la siguiente instrucción

es retornar al punto dónde se hace nuevamente el chequeo con la expresión 2. Esto sucederá hasta que se cumpla el chequeo de la etiqueta loop que permitirá salir de esta estructura y continuar con la ejecución del resto del código.

Variables

Java, C, y la mayoría de los lenguajes, utilizan el concepto de variable como abstracción de las celdas de memoria que almacenan los datos. Las variables, al ser entidades, tienen atributos que permiten definir las e identificarlas. Están formadas por la quintupla <nombre, alcance, tipo, l_value, r_value>.

```
1.  #include <stdio.h>
2.  int num = 2;
3.  int main() {
4.      int suma = 5 + num;
5.      printf("El valor de suma es %d\n", suma);
6.      return 0;
7.  }
```

Este programa escrito en C muestra una variable local, declarada dentro de la función main y una variable global declarada en la línea 2 del código.

La variable global tiene por **nombre** "num", su **alcance** va desde la línea de su declaración hasta el final de programa, dado que es de tipo estático/léxico (al igual que Java), lo que significa que cada referencia a ella puede ligarse a su declaración sin necesidad de ejecutar el programa.

Su **tipo** es int, que indica el conjunto de valores (los números enteros) al que puede asociarse, y las operaciones legales para crear, acceder y modificarlos. El tipado de esta y de todas las variables en C es estático y explícito.

El **l_value** es la posición de memoria asociada a la variable. Las variables globales y estáticas en C definen su **l_value** con asignación estática, lo que implica que el **tiempo de vida** de la variable num será desde su declaración hasta después de la finalización de la ejecución del programa.

El **r_value** es el valor encriptado que se guarda en la celda de memoria variable, por tanto en la línea 2, éste se hace 2. De no haberse inicializado, C le hubiese asignado un 0 automáticamente.

El nombre de la variable local es "suma", y su **alcance** abarca desde la línea 4 (su declaración) hasta la línea 7 (el final del bloque en el que se define).

Es de **tipo** int, implicando las mismas observaciones que en el caso de la variable global.

El **l_value** de las variables locales en C es definido con asignación automática, por ende, el tiempo de vida de "suma" es desde la línea 4 hasta la 7, igual que su alcance.

El **r_value** de "suma" será 7 desde la línea 4, gracias a la asignación hecha luego de la declaración, que implica una operación de suma entre el valor 5 y el valor que representa la variable global num (cuyo alcance admite que sea referenciado desde el bloque main).

Referencias

- Louden, Kenneth y Lambert, Kenneth. (2011). *Programming Languages: Principles and Practices (Advanced Topics) 3era Edición*. Cengage Learning.
- Kernighan, Brian y Ritchie, Dennis. (1988). *The C programming language Second Edition*. Prentice Hall.
- Weiss, Mark Allen. (2010). *Data Structures & Problem Solving Using Java Fourth Edition*. Pearson Addison-Wesley.
- Ghezzi, Carlo y Jazayeri, Mehdi. (2008). *Programming language concepts - Third edition*. Wiley.
- Gabrielli, Maurizio y Martini, Simone. (2010). *Principles and Paradigms (Undergraduate Topics in Computer Science)*. Springer.
- Sebesta, Robert W. (2012). *Concepts of Programming Languages*. Pearson Addison-Wesley.
- Rothwell, Trevis y Youngman, James. (2015). *The GNU C Reference Manual*. Free Software Foundation, inc.
- Gurevich, Yuri & Huggins, James. (1996). *The Semantics of the C Programming Language*. Oracle. (2020). *JDK 14 Documentation*. <https://docs.oracle.com/en/java/javase/14/>
- Data Flair. (2020). *Learn C - C Tutorials*. <https://data-flair.training/blogs/c-tutorials-home/>
- Data Flair. (2020). *A Complete Guide to Mastering Java*. <https://data-flair.training/blogs/java-tutorials-home/>
- W3 Schools. (2020). *C Tutorial*. <https://www.w3schools.in/c-tutorial/>
- W3 Schools. (2020). *Java Tutorial*. <https://www.w3schools.in/java-tutorial/>