1. ¿Qué arroja?

```
public class Main {
public static void main(String[] args) {
String[] at = {"FINN", "JAKE"};
for (int x=1; x<4; x++){
for (String s : at){
System.out.println(x + " "+ s);
if(x==1){
break;
}
}
}
}
}
```

Resultado de la ejecución: //1 FINN 2 FINN 2 JAKE 3 FINN 3 JAKE

En la primera (x=1), el bucle interno se ejecuta solo una vez, imprimiendo 1 `FINN` y luego termina debido al `break`. y en la segunda iteración (x=2), el bucle interno imprime 2 `FINN` y 2 `JAKE`. Por último, en la tercera iteración (x=3), el bucle interno imprime 3 `FINN` y 3 `JAKE`.

2. ¿Qué 5 líneas son correctas?

```
class Light{
protected int lightsaber(int x){return 0;}
}
class Saber extends Light{
private int lightsaber (int x){return 0;}
}
```

// Error el modificador de acceso en la clase derivada no puede ser más restrictivo que el modificador de acceso en la clase base

- **protected int lightsaber (long x){return 0;}** // Correcto Sobreescritura del método adecuada, por cambio de parámetro
- **private int lightsaber (long x){return 0;}** // Correcto No se está sobreescribiendo el método, al tener otro parámetro se trata de un método independiente
- **protected long lightsaber (int x){return 0;}** // Error Para que la sobrescritura sea válida, los métodos deben tener la misma firma, incluyendo el tipo de retorno.
- **protected long lightsaber (int x, int y){return 0**;} //Correcto
- **public int lightsaber (int x){return 0;}** // Correcto
- **protected long lightsaber (long x){return 0;}** // Valido por ser sobrecarga de método

- **Private int blipvert(long x) { return 0; }**
- **Protected int blipvert(long x) { return 0; }**
- **Protected long blipvert(int x, int y) { return 0; }**
- **Public int blipvert(int x) { return 0; }**
- Private int blipvert(int x) { return 0; }
- Protected long blipvert(int x) { return 0; }
- **Protected long blipvert(long x) { return 0; }**

3. ¿Qué resultado arroja?

```
class Mouse{
public int numTeeth;
public int numWhiskers;
public int weight;
//Estos son los atributos públicos de la clase Mouse
public Mouse (int weight){
this(weight,16);
} //Este constructor toma solo el weight como parámetro y llama al segundo constructo
public Mouse (int weight, int numTeeth){
this(weight, numTeeth, 6);
} //Este constructor toma dos parámetros: weight y numTeeth. Luego, llama al tercer constructor
public Mouse (int weight, int numTeeth, int numWhiskers){
this.weight = weight;
this.numTeeth= numTeeth;
this.numWhiskers = numWhiskers;
} //Este constructor es el más completo. Asigna directamente los valores
public void print (){
System.out.println(weight + ""+ numTeeth+ ""+ numWhiskers);
}
public static void main (String [] args){
Mouse mouse = new Mouse (15);
mouse.print();
}
}
```

Salida: 15 , 16 , 6

En el método `main`, se crea un objeto `Mouse` llamado `mouse` usando el primer constructor, pasando 15 como `weight`. Dado que el primer constructor llama al segundo, y el segundo llama al tercero, los valores finales que se asignan al objeto `mouse` son:

- `weight` = 15
- `numTeeth` = 16
- `numWhiskers` = 6

4. ¿Cual es la salida?

```
class Arachnid {
public String type = "a";
public Arachnid(){
System.out.println("arachnid");
}
}
class Spider extends Arachnid{
public Spider(){
System.out.println("spider");
}
void run(){
type = "s";
System.out.println(this.type + " " + super.type);
```

```
}
public static void main(String[] args) {
new Spider().run();
}
} // arachnid spider s s
```

La Instancia de `Spider`:Primero, se ejecuta el constructor de `Arachnid`, que imprime `"arachnid"`. Luego, se ejecuta el constructor de `Spider`, que imprime `"spider"`. Método `run`: El valor de `type` en la instancia de `Spider` se cambia de `"a"` a `"s"`. Se imprime el valor de `this.type` (`"s"`) seguido del valor de `super.type` (`"s"`), ya que `super.type` es simplemente `this.type` en este caso.

5. **¿Cual es la salida?**

```
class Test {
public static void main(String[] args) {
int b = 4; Inicialmente, b = 4.
b--; Después de b--, b = 3.
System.out.println(--b); //primera impresión Después de b--, b = 3.
System.out.println(b); //segunda impresión Después de --b, b = 2, y se imprime 2.
}Finalmente, se imprime nuevamente el valor actual de b, que es 2.
} //Resultado: 2 2
```

Inicia con una declaración de una variable, con el valor de 4, b- - es un postdecremento(significa que el valor de b va primero) osea que de 4 pasa a 3, en - - b para esto baja su valor 3 a 2.
Inicialmente, b = 4.

6. **¿Cual es la salida?**

```
class Sheep {
public static void main(String[] args) {
int ov = 999;
ov--;
System.out.println(--ov);
System.out.println(ov);
}
}
```

// Respuesta correcta: 997, 997

Se declara e inicializa la variable `ov` con el valor `999`. La instrucción `ov--;` decrementa el valor de `ov` en 1. Ahora, `ov` es igual a `998`.La siguiente línea, `System.out.println(--ov);`, decrementa `ov` en 1 nuevamente (ahora `ov` es `997`) y luego imprime el valor de `ov`, que es `997`.Finalmente, `System.out.println(ov);` imprime el valor actual de `ov`, que sigue siendo `997`.

7. Resultado

```
class Overloading {
public static void main(String[ ] args) {
System.out.println(overload("a"));
System.out.println(overload("a", "b"));
```

```
System.out.println(overload("a", "b", "c"));
}
public static String overload(String s){
return "1";
}
public static String overload(String... s){
return "2";
}
public static String overload(Object o){
return "3";
}
public static String overload(String s, String t){
return "4";
}
}
```

// Salida: 1, 4, 2

1. `overload("a")`:
   - Aquí se pasa un solo argumento de tipo `String`.
   - El método que coincide es `overload(String s)`.
   - Salida: `"1"`
2. `overload("a", "b")`:
   - Aquí se pasan dos argumentos de tipo `String`.
   - El método que coincide es `overload(String s, String t)`.
   - Salida: `"4"`
3. `overload("a", "b", "c")`:
   - Aquí se pasan tres argumentos de tipo `String`.
   - El método que coincide es `overload(String... s)`, porque acepta un número variable de argumentos.
   - Salida: `"2"`

8. Resultado

```
class Base1 extends Base{
public void test(){
System.out.println("Base1");
}
}
class Base2 extends Base{
public void test(){
System.out.println("Base2");
}
}
class Test {
public static void main(String[] args) {
Base obj = new Base1();
((Base2) obj).test();
}
}
```

// ClassCastException:se produce cuando se intenta realizar una conversión de tipos entre clases no relacionadas en una jerarquía de herencia.

Cuando se ejecuta el código, se produce una excepción en tiempo de ejecución:

`ClassCastException`: Esta excepción se lanza porque el programa intenta convertir un objeto de una clase (`Base1`) en una clase no relacionada (`Base2`). Aunque ambas clases comparten la misma superclase (`Base`), no hay una relación de herencia directa entre `Base1` y `Base2`.

9. Resultado

```
public class Fish {
public static void main(String[] args) {
int numFish = 4;
String fishType= "Tuna";
String anotherFish = numFish +1; Esto causará un error de compilación porque Java no puede convertir automáticamente un valor entero en una cadena.
String anotherFish = numFish + 1 + "";  // convierte a cadena,cambiar
System.out.println(anotherFish + " " + fishType);
System.out.println(numFish + " " + 1);
}
}
```

// El código no compila

10. Resultado

```
class MathFun {
public static void main(String[] args) {
int number1 = 0b0111; //Binario
int number2 = 0111_000; //Octal

System.out.println("Number1: "+number1);
System.out.println("Number2: "+number1);
}
}
```

//Salida: 7 7 ojo que imprime dos veces number 1

Number1: 7

Number2: 37560

Este código demuestra cómo se pueden utilizar diferentes sistemas numéricos en Java para inicializar variables enteras, al parecer la salida es diferente a la respuesta marcada.

11. Resultado

```
class Calculator {
int num =100;
public void calc(int num){
this.num =num*10;
}
public void printNum(){
System.out.println(num);
```

```
}
public static void main (String [] args){
Calculator obj = new Calculator ();
obj.calc(2);
obj.printNum();
}
}
```

// Salida: 20

Inicialmente, `num` tiene el valor `100` como variable de instancia.Después de llamar a `calc(2)`, `this.num` se actualiza a `20`. y finalmente, cuando `printNum()` se ejecuta, imprime `20`. Este código ilustra cómo se puede usar `this` para referirse a la variable de instancia de la clase, diferenciándose de los parámetros de un método con el mismo nombre.

12. Que Aseveraciones son correctas

```
class ImportExample {
public static void main (String [] args){
Random r = new Random();
System.out.println(r.nextInt(10));
}
}
```

*** If you omit java.util import statements java compiles gives you an error**
* java.lang and util.random are redundant
* you dont need to import java.lang

Se omite la importación de **`java.util.Random`**, sale un error de compilación. La afirmación de que **`java.lang`** y **`util.random`** son redundantes es incorrecta. Por **default** se importa **`java.lang`**, ya que se importa automáticamente.

13. Resultado

```
public class Main {
public static void main(String[] args) {
int var = 10;
System.out.println(var++);
System.out.println(++var);
}
```

} **//salida: 10, 12**
porque en la primera salida muestra el valor de var (10) y luego le suma 1, dando como resultado 11 y en la segunda salida hace primero la suma dando como resultado 12

14. Resultado

```
class MyTime {
public static void main (String [] args){
short mn =11;
short hr;
short sg =0;
for (hr=mn;hr>6;hr-=1){
```

```
sg++;
}

System.out.println("sg="+sg);
}
}
```

// Salida sg=5;

Respuesta correcta mn = 11

El código resta 1 desde 11 hasta 7 y cuenta cuántas veces lo hace. Al final, imprime 5.

15. Cuales son verdad

* An ArrayList is mutable:
* An Array has a fixed size
* An array is mutable
* An array allows multiple dimensions
* An arrayList is ordered
* An array is ordered

Un `ArrayList` y un `Array` son mutables y ordenados, con `ArrayList` de tamaño variable y `Array` de tamaño fijo, que también puede ser multidimensional.

16. Resultado

```
public class MultiverseLoop {
public static void main (String [] args){
int negotiate = 9;
do{
System.out.println(negotiate);
}while (--negotiate);
}
}
```

//Errores de compilación, necesita un bool el while

El bucle se detiene porque `--negotiate` llega a `0` y la condición `negotiate > 0` ya no se cumple.

17 Resultado

```
class App {
public static void main(String[] args) {
Stream<Integer> nums = Stream.of(1,2,3,4,5);
nums.filter(n -> n % 2 == 1);
nums.forEach(p -> System.out.println(p));
}
}
```

//Exception at runtime, se debe encadernar el stream por que se consume

no produce el resultado esperado porque la operación de filtrado (`filter`) se realiza en un flujo (`Stream`)

18 Pregunta. Suppose the declared type of x is a class, and the declared type of y is an interface. When is the assignment x = y; legal?
* When the type of X is Object
la clase de x debe proporcionar una implementación de todos los métodos de la interfaz y para que la asignación sea válida.

19 Pregunta. When a byte is added to a char, what is the type of the result?
*** int**
Al realizar operaciones aritméticas entre estos tipos, se promueven primero a `int` y luego se realiza la operación. Dado que el resultado de sumar un `byte` a un `char` será de tipo `int`.

20 Pregunta. The standart application programming interface for accessing databases in java?
*** JDBC**
Permite a las aplicaciones Java interactuar con bases de datos utilizando drivers específicos de la base de datos que implementan la interfaz JDBC.

21 Pregunta. Which one of the following statements is true about using packages to organize your code in
Java ?
***** Packages allow you to limit access to classes, methods, or data from classes outside the package.

La declaración **es verdadera**. Los paquetes permiten organizar el código y también limitar el acceso a clases, métodos o datos desde clases fuera del paquete. Esto se logra mediante modificadores de acceso como `public`, `protected`, `private`, y el modificador de acceso por defecto (package-private). El uso de paquetes ayuda a encapsular el código y a controlar la visibilidad entre diferentes partes de un programa.

22 Pregunta. Forma correcta de inicializar un booleano:

* boolean a = (3>6); **// Esto se evalúa a false**
otro ejemplo: boolean b = (5 == 5); **// Esto se evalúa a true**
El resultado de la comparación es `false`, esta es la condición que se asigna a la variable booleana `a`, ya que 3 no es mayor que 6.

23 Pregunta

```
class Y{
public static void main(String[] args) throws IOException {
try {
doSomething();
}catch (RuntimeException exception){
System.out.println(exception);
}
}
static void doSomething() throws IOException {
```

```
if (Math.random() > 0.5){
}// No hace nada si el número es mayor que 0.5
throw new RuntimeException();
}
} * Adding throws IOException to the main() method signature
```

Este código intenta ejecutar una operación que siempre lanza una `RuntimeException`, capturando e imprimiendo esa excepción. La cláusula `throws IOException` en el `main` es redundante, ya que no se está lanzando ni propagando una `IOException` en la lógica actual.

24 Resultado

```
interface Interviewer {
abstract int interviewConducted();
}
public class Manager implements Interviewer{
int interviewConducted() {
return 0;
}
}//Wont compile
```

El error se debe a que la implementación del método `interviewConducted()` en la clase `Manager` no es pública, mientras que en la interfaz `Interviewer`, el método se espera que sea `public`. Al agregar el modificador `public` en la implementación del método en la clase `Manager`, el código compilará correctamente.

25 Pregunta

```
class Arthropod {
public void printName(double Input){
System.out.println("Arth");
}
}
class Spider extends Arthropod {
public void printName(int input) {
System.out.println("Spider");
}
public static void main(String[] args) {
Spider spider = new Spider();
spider.printName(4);
spider.printName(9.0);
}
} // Spider, Arth
```

Este código demuestra cómo Java selecciona el método apropiado para llamar según el tipo de argumento que se pasa, utilizando sobrecarga de métodos (method overloading).

26 Pregunta

```
public class Main {
public enum Days{Mon,Tue, Wed}
public static void main(String[] args) {
for (Days d:Days.values()
) {
Days[] d2 = Days.values();
```

```
System.out.println(d2[2]);
}
}
```
} **// wed**

El bucle `for-each` se ejecuta una vez por cada valor en el enum `Days` (es decir, 3 veces: para `Mon`, `Tue` y `Wed`), pero en cada iteración se imprime siempre `d2[2]`, que es `"Wed"`.
wed wed wed

27 Pregunta

```
public class Main{
public enum Days {MON, TUE, WED};
public static void main(String[] args) {
boolean x= true, z = true;
int y = 20;
x = (y!=10)^(z=false);
System.out.println(x + " " + y + " "+ z);
}
}
```
// true 20 false

`(y != 10)`: Se evalúa si y es diferente de `10`. Como y es `20`, la expresión es `true`.
`(z = false)`: Aquí, la variable z se asigna el valor `false`. En Java, la asignación en una expresión devuelve el valor asignado, por lo que esta parte de la expresión también se evalúa como `false`.
En este caso, `true ^ false` es `true`, por lo que x se asigna a `true`.
Esto refleja:
x = true después de la operación XOR.
y permanece sin cambios como 20.
z es false después de la asignación.

28 Pregunta

```
class InitializacionOrder {
static {add(2);}
static void add(int num){
System.out.println(num+"");
}
InitializacionOrder(){add(5);}
static {add(4);}
{add(6);}
static {new InitializacionOrder();}
{add(8);}
public static void main(String[] args) {}
}
```
//2 4 6 8 5

Los bloques estáticos se ejecutan en el orden en que aparecen en la clase, primero se ejecutan los bloques y por último el constructor que en este caso es el 5.

29 Pregunta

```
public class Main {
public static void main(String[] args) {
String message1 = "Wham bam";
String message2 = new String("Wham bam");
```

```
if (message1!=message2){
System.out.println("They dont match");
}else {
System.out.println("They match");
}
}
}
// They dont match
```

Se comparan dos cadenas de texto (String). Dado que message1 y message2 son referencias a diferentes objetos en memoria, la comparación message1 != message2 es true.

30 Pregunta

```
class Mouse{
public String name;
public void run(){
System.out.println("1");
try{
System.out.println("2");
name.toString();
System.out.println("3");
}catch(NullPointerException e){
System.out.println("4");
throw e;
}
System.out.println("5");
}
public static void main(String[] args) {
Mouse jerry = new Mouse();
jerry.run();
System.out.println("6");
}
} // Salida 1 2 4 NullPointerException
```

La primera impresión es 1, al continuar pasa por un TRY e imprime un 2, seguiría la continuación con el 3, pero se cancela por el CATCH. Aquí, name es null, por lo que intentar llamar a toString() en un valor null genera una excepción NullPointerException e imprime 4.

```
31 pregunta
public class Main {
public static void main(String[] args) {
try (Connection con = DriverManager.getConnection(url, uname,
pwd)){
Statement stmt =con.createStatement();
System.out.print(stmt.exeuteUpdate("INSERT INTO User
VALUES (500, 'Ramesh')"));
}
}
} // Salida: arroja 1Si la conexión a la base de datos y la consulta se ejecutan correctamente, el programa imprimirá 1, lo que indica que una fila fue insertada en la tabla User. Si ocurre un error, se mostrará una traza de la excepción en la consola.
```

32 pregunta

```
class MarvelClass{
public static void main (String [] args){

MarvelClass ab1, ab2, ab3;
ab1 =new MarvelClass();
ab2 = new MarvelMovieA();
ab3 = new MarvelMovieB();
System.out.println ("the profits are " + ab1.getHash()+ "," +
ab2.getHash()+","+ab3.getHash());
}
public int getHash(){
return 676000;
}
}
class MarvelMovieA extends MarvelClass{
public int getHash (){
return 18330000;
}
}
class MarvelMovieB extends MarvelClass {
public int getHash(){
return 27980000;
}
}
// the profits are 676000, 18330000, 27980000
```

Se visualiza el polimorfismo, en el método getHash() es sobrescrito en las clases derivadas (MarvelMovieA y MarvelMovieB). Se declaran 3 referencias como una instancia, e inicializa como una instancia de 3 veces. El método getHash() es sobrescrito en las subclases MarvelMovieA y MarvelMovieB. Dependiendo de la instancia real a la que hace referencia cada variable (ab1, ab2, ab3), se ejecuta la versión correspondiente del método getHash(). Esto demuestra cómo el polimorfismo permite que un método de una clase base puede ser sobrescrito en una subclase, y la versión sobrescrita es la que se ejecuta en tiempo de ejecución.

33 pregunta

```
class Song{
public static void main (String [] args){
String[] arr = {"DUHAST","FEEL","YELLOW","FIX YOU"};
for (int i =0; i <= arr.length; i++){
System.out.println(arr[i]);
}
}
}
//4 An arrayindexoutofbondsexception
```

Se intenta imprimir un array de cadenas (String[]). Sin embargo, hay un problema que provocará una excepción en el tiempo de ejecución. se encuentra en la condición del bucle for. Sin embargo, los índices de un array en Java comienzan en 0 y terminan en arr.length - 1.
ArrayIndexOutOfBoundsException: Este es un tipo de excepción que ocurre cuando se intenta acceder a un índice que está fuera del rango válido de un array. En este caso, ocurre cuando i es igual a 4.

34 pregunta

```
class Menu {
public static void main(String[] args) {
String[] breakfast = {"beans", "egg", "ham", "juice"};
for (String rs : breakfast) {
int dish = 2;
while (dish < breakfast.length) {
System.out.println(rs + "," + dish);
dish++;
}
}
}
}
/*
beans,2
beans,3
egg,2
egg,3
ham,2
ham,3
juice,2
juice,3*/
```

Se genera una salida que muestra cada elemento del array `breakfast` seguido de números consecutivos empezando desde 2 hasta el tamaño del array menos 1.

35 Pregunta. Which of the following statement are true:

* string builder es generalmente más rápido qué string buffer
porque no está sincronizado, lo que significa que no tiene la sobrecarga adicional de la sincronización para garantizar la seguridad en el contexto de múltiples hilos.
* string buffer is threadsafe; stringbuildder is not
Está sincronizado, lo que permite su uso seguro en entornos multihilo, mientras que StringBuilder no está sincronizado y, por lo tanto, no es seguro para el uso en múltiples hilos simultáneamente.

36 pregunta

```
class CustomKeys{
Integer key;
CustomKeys(Integer k){
key = k;
```

```
}
public boolean equals(Object o){
return ((CustomKeys)o).key==this.key;
}
}
```

// Salida: compilation fail

Este enfoque asegura que la comparación en el método equals sea correcta, evitando problemas relacionados con la comparación de objetos Integer usando ==.

37 Pregunta. The catch clause is of the type:

- Throwable:Captura todo, incluyendo excepciones y errores, pero generalmente no es recomendable.
- Exception but NOT including RuntimeException:Captura todas las excepciones comprobadas, que deben ser manejadas.
- Checked Exception:Excepciones que el compilador requiere que se manejen.
- RunTimeException: Excepciones de tiempo de ejecución que no requieren manejo explícito.
- Error: Problemas graves en la JVM que generalmente no se deben capturar.

38 Pregunta. An enhanced for loop

* also called for each, offers simple syntax to iterate through a collection but it can´t be used to delete elements of a collection
también conocido como "for-each loop") en Java es verdadera.
Ofrece una sintaxis simple y limpia para iterar sobre colecciones, pero no puede ser utilizado para eliminar elementos de la colección durante la iteración. Para esto, es necesario usar un iterador explícito.

39 Pregunta. Which of the following methods may appear in class Y, which extends x ?

public void doSomething(int a, int b){…}
Es completamente válido y puede aparecer en la clase Y sin problemas.

40 pregunta

```
public class Main {
public static void main(String[] args) {
String s1= "Java";
String s2 = "java";
if (s1.equalsIgnoreCase(s2)){
System.out.println ("Equal");
} else {
System.out.println ("Not equal");
}
}

}
```

// Salida: Equal; respuesta: s1.equalsIgnoreCase(s2)

Este método compara s1 y s2 sin considerar las diferencias de mayúsculas y minúsculas. Si las cadenas son iguales al ignorar las diferencias de caso, el método devuelve true, de lo contrario, devuelve false.

41 pregunta

```
class App {
public static void main(String[] args) {
String[] fruits = {"banana", "apple", "pears", "grapes"};
// Ordenar el arreglo de frutas utilizando compareTo
Arrays.sort(fruits, (a, b) -> a.compareTo(b));
// Imprimir el arreglo de frutas ordenado
for (String s : fruits) {
System.out.println(""+s);
}
}
}
/* apple
banana
grapes
pears */
```

Se crea un arreglo de cadenas, que contiene cuatro elementos.La función lambda (a, b) -> a.compareTo(b) compara dos cadenas a y b utilizando el método compareTo. Este método ordena las cadenas en orden lexicográfico (alfabético).

42 pregunta

```
public class Main {
public static void main(String[] args) {
int[]countsofMoose = new int [3];
System.out.println(countsofMoose[-1]);
}
}
//this code wull trow an arrayindexoutofboundsexpression
```

Se lanzará una excepción ArrayIndexOutOfBoundsException porque se está intentando acceder a un índice que no existe en el arreglo, específicamente el índice -1. En Java, los índices de los arreglos deben estar dentro de los límites válidos, que en este caso serían 0, 1, 2 y así sucesivamente, dependiendo del tamaño del arreglo. Intentar acceder a un índice fuera de estos límites, como -1, provoca que se lance esta excepción, ya que el arreglo no tiene ningún elemento en esa posición.

43 Pregunta

```
class Salmon{
int count;
public void Salmon (){
count =4;
}
public static void main(String[] args) {
Salmon s = new Salmon();
System.out.println(s.count);
```

```
}
}
```
// Salida: 0 -> cero

La clase `Salmon`, es una instancia de tipo `int`, la cual, si no es explícitamente inicializada, tendrá el valor predeterminado de 0.El código `Salmon s = new Salmon();` está diseñado para crear una nueva instancia de la clase `Salmon`. Sin embargo, si `Salmon` no tiene un constructor definido y el método `Salmon()` es tratado como un método normal, el campo `count` de la instancia no será inicializado a 4 (o cualquier otro valor), sino que conservará el valor predeterminado de 0. Un constructor no debe especificar un tipo de retorno, y si se define un método con el mismo nombre que la clase pero con un tipo de retorno, este no será tratado como un constructor, lo que podría causar que los campos no se inicializan como se esperaba.

44 pregunta

```
class Circuit {
public static void main(String[] args) {
runlap();
int c1=c2;
int c2 = v;
}
static void runlap(){
System.out.println(v);
}
static int v;
}
```
// corregir linea 6; c1 se le asigna c2 pero c2 aun no se declara

En int c1 = c2;, la variable c2 se usa antes de haber sido declarada, lo que provoca un error.

45 pregunta

```
class Foo {
public static void main(String[] args) {
int a=10;
long b=20;
short c=30;
System.out.println(++a + b++ *c);
}
} // salida: 611 (11+20*30)
```
Este resultado proviene de la evaluación de la expresión teniendo en cuenta el incremento de a y b, así como la multiplicación de b por c.

46 pregunta

```
public class Shop{
public static void main(String[] args) {
new Shop().go("welcome",1);
new Shop().go("welcome", "to", 2);
}
public void go (String... y, int x){
System.out.print(y[y.length-1]+"");
}
```

```
}
// Compilation fails
```

El error principal es la forma en que está definido el método go.

47 pregunta

```
class Plant {
Plant() {
System.out.println("plant");
}
}
class Tree extends Plant {
Tree(String type) {
System.out.println(type);
}
}
class Forest extends Tree {
Forest() {
super("leaves");
new Tree("leaves");
}
public static void main(String[] args) {
new Forest();
}
}
/*plant
leaves
plant
leaves*/
```

El orden de impresión es importante aquí. Primero se construyó Plant, luego Tree, y finalmente Forest. Dentro del constructor de Forest, se imprimen "leaves" dos veces: una vez al crear la instancia con super("leaves") y otra vez al crear una nueva instancia de Tree. Este orden refleja el flujo de ejecución al crear instancias de las clases en la jerarquía de herencia.

48 Pregunta

```
class Test {
public static void main(String[] args) {
String s1 = "hello";
String s2 = new String ("hello");
s2=s2.intern(); // el intern() asigna el mismo hash conforme a
la cadena
System.out.println(s1==s2);
}
} // Salida: true
```

Las cadenas literales se almacenan en el pool de cadenas (String pool). Si una cadena literal ya existe en el pool, se reutiliza. Aunque el contenido de esta nueva instancia es "hello", esta instancia es diferente de la cadena literal almacenada en el pool de

cadenas.Esto indica que ambas variables s1 y s2 apuntan al mismo objeto en el pool de cadenas.

49 Pregunta. Cuál de las siguientes construcciones es un ciclo infinito while
Ambas construcciones crean un ciclo infinito, pero de maneras ligeramente diferentes:
* while(true);
Tiene un cuerpo vacío representado por un punto y coma (;), lo que significa que no realiza ninguna acción dentro del ciclo, pero sigue ejecutándose indefinidamente.
* while(1==1){}
Tiene un cuerpo vacío representado por {}. Aunque el cuerpo está vacío, el ciclo sigue ejecutándose indefinidamente debido a que la condición siempre es true.

50 pregunta

```
class SampleClass{
public static void main(String[] args) {
AnotherSampleClass asc =new AnotherSampleClass ();
SampleClass sc = new SampleClass();
//sc = asc;
}
}
class AnotherSampleClass extends SampleClass {}
```

// Respuesta: sc = asc;
Es una asignación válida en Java debido a la herencia y el polimorfismo. La variable sc de tipo SampleClass puede referirse a una instancia de AnotherSampleClass, ya que AnotherSampleClass es una subclase de SampleClass.

51 pregunta

```
public class Main {
public static void main(String[] args) {
int a= 10;
int b =37;
int z= 0;
int w= 0;
if (a==b){
z=3;
}else if(a>b){
z=6;
}
w=10*z;
System.out.println(z);
}
}
```

// Salida: 0 -> cero
La creación infinita de instancias de course en el constructor causa que la pila de llamadas se llene. Cada instancia de course intenta crear otra instancia, y este proceso no termina nunca.Dado que ninguna de las condiciones es verdadera, el valor de z permanece 0.

52 Pregunta

```
public class Main{
public static void main(String[] args) {
course c = new course();
c.name="java";

System.out.println(c.name);
}
}
class course {
String name;
course(){
course c = new course();
c.name="Oracle";
}
} // Exception StackOverflowError
```

La creación infinita de instancias de course en el constructor causa que la pila de llamadas se llene. Cada instancia de course intenta crear otra instancia, y este proceso no termina nunca.

52 Pregunta

```
public class Main{
public static void main(String[] args) {
String a;
System.out.println(a.toString());
}
} // builder fails
```

El método toString() en una variable a que no ha sido inicializada.

53 Pregunta

```
public class Main{
public static void main(String[] args) {
System.out.println(2+3+5);
System.out.println("+"+2+3+5);
}
} // salida 10 + 235
```

Se están sumando los números 2, 3, y 5.el operador + se comporta de manera diferente debido a la concatenación de cadenas.

54 Pregunta

```
public class Main {
public static void main(String[] args) {
int a = 2;
int b = 2;
if (a==b)
System.out.println("Here1");
```

```
if (a!=b)
System.out.println("here2");
if (a>=b)
System.out.println("Here3");
}
} // salida: Here1 , here 3
```

Dado que las condiciones de impresión:
a == b es verdadero, imprime Here1.
a != b es falso, no imprime nada.
a >= b es verdadero, imprime Here3.

55 Pregunta

```
public class Main extends count {
public static void main(String[] args) {
int a = 7;
System.out.println(count(a,6));
}
}
class count {
int count(int x, int y){return x+y;}
}// builder fails
```

Se intenta llamar al método “count” de la clase “count” como si fuera un método estático desde la clase Main, pero count no es un método estático ni es accesible de esa forma.

56 Pregunta

```
class trips{
void main(){
System.out.println("Mountain");
}
static void main (String args){
System.out.println("BEACH");
}
public static void main (String [] args){
System.out.println("magic town");
}
void mina(Object[] args){
System.out.println("city");
}
} // Salida: magic town
```

Tiene tres métodos llamados main y un método adicional llamado mina. Este método se ejecuta al iniciar el programa, por lo que el programa imprimirá "magic town" en la consola.

57 Pregunta

```
public class Main{
public static void main(String[] args) {
int a=0;
System.out.println(a++ +2);
```

```
System.out.println(a);
}
} // salida: 2,1
```

La primera línea imprime 2, que es el resultado de 0 + 2 usando el valor original de a antes del incremento.
La segunda línea imprime 1, que es el valor actualizado de a después del incremento.

58 Pregunta

```
public class Main{
public static void main(String[] args) {
List<E> p =new ArrayList<>();
p.add(2);
p.add(1);
p.add(7);
p.add(4);
}
} // builder fails
```

No compilara debido a problemas con el uso de tipos genéricos en la lista List<E>.

59 Pregunta

```
public class Car{
private void accelerate(){
System.out.println("car acelerating");
}
private void break(){
System.out.println("car breaking");
}
public void control (boolean faster){
if(faster==true)
accelerate();
else
break();
}
public static void main (String [] args){
Car car = new Car();

car.control(false);
}
} //break es una palabra reservada
```

Utilizada para salir de bucles o bloques switch. Usar break como nombre de un método generará un error de compilación.

60 Pregunta

```
class App {
App() {
System.out.println("1");
}
```

```
App(Integer num) {
System.out.println("3");
}
App(Object num) {
System.out.println("4");
}
App(int num1, int num2, int num3) {
System.out.println("5");
}
public static void main(String[] args) {
new App(100);
new App(100L);
}
} // Salida: 3, 4 …
```

3 corresponde al constructor App(Integer num) llamado con el valor 100.
4 corresponde al constructor App(Object num) llamado con el valor 100L.

61 Pregunta

```
class App {
public static void main(String[] args) {
int i=42;
String s = (i<40)?"life":(i>50)?"universe":"everething";
System.out.println(s);
}
} // Salida: everything
```

Utilizando el operador ternario anidado para asignar un valor a la variable s. Asigna correctamente "everything" a la variable s y lo imprime.

62 Pregunta

```
class App {
App(){
System.out.println("1");
}
App(int num){
System.out.println("2");
}
App(Integer num){
System.out.println("3");
}
App(Object num){
System.out.println("4");
}
public static void main(String[] args) {
String[]sa = {"333.6789","234.111"};
NumberFormat inf= NumberFormat.getInstance();
inf.setMaximumFractionDigits(2);
for(String s:sa){
```

```
System.out.println(inf.parse(s));
}
}
} // java: unreported exception java.text.ParseException; must be caught or declared to be thrown
```

Ocurre porque el método inf.parse(s) puede lanzar una excepción ParseException, y debes manejar esta excepción. Las excepciones verificadas deben ser capturadas con un bloque try-catch o deben ser declaradas en la firma del método usando throws.

63 Pregunta

```
class Y{
public static void main(String[] args) {
String s1 = "OCAJP";
String s2 = "OCAJP" + "";
System.out.println(s1 == s2);
}
} // salida: true
```

Las cadenas literales y la concatenación de cadenas en tiempo de compilación. Dado que tanto s1 como s2 apuntan a la misma instancia de la cadena en el pool de cadenas, la comparación con == devuelve true.

64 Pregunta

```
class Y{
public static void main(String[] args) {
int score = 60;
switch (score) {
default:
System.out.println("Not a valid score");
case score < 70:
System.out.println("Failed");
break;
case score >= 70:
System.out.println("Passed");
break;
}
} // salida: Error de compilación - java: reached end of file while parsing
```

Generará un error de compilación debido a la forma en que se han definido los casos en la instrucción switch.

65 Pregunta

```
class Y{
public static void main(String[] args) {
int a = 100;
System.out.println(-a++);
}
} // salida -100
```

La variable a inicialmente tiene el valor 100.

-a++ primero niega 100, dando -100, y luego incrementa a a 101.
Por lo tanto, la salida es -100.

66 Pregunta

```
class Y{
public static void main(String[] args) {
byte var = 100;
switch(var) {
case 100:
System.out.println("var is 100");
break;
case 200:
System.out.println("var is 200");
break;
default:
System.out.println("In default");
}
}
} // salida: Error de compilación - java: incompatible types: possible
lossy conversion from int to byte
```

Los valores en los case de una instrucción switch deben coincidir con el tipo de la variable que se está evaluando.

67 Pregunta

```
class Y{
public static void main(String[] args) {
A obj1 = new A();
B obj2 = (B)obj1;
obj2.print();
}
}
class A {
public void print(){
System.out.println("A");
}
}
class B extends A {
public void print(){
System.out.println("B");
}
}
// ClassCastException
```

lanzará una ClassCastException en tiempo de ejecución debido a un problema de conversión de tipos (casting) entre clases que no están relacionadas por herencia en la forma esperada.

68 Pregunta

```
class Y{
public static void main(String[] args) {
String fruit = "mango";
switch (fruit) {
default:
System.out.println("ANY FRUIT WILL DO");
case "Apple":
System.out.println("APPLE");
case "Mango":
System.out.println("MANGO");
case "Banana":
System.out.println("BANANA");
break;
}
}
}
```

```
//ANY FRUIT WILL DO
APPLE
MANGO
BANANA
```

- default: Se ejecuta primero, imprime "ANY FRUIT WILL DO".
- case "Apple": No coincide, pero se ejecuta debido al fall-through, imprime "APPLE".
- case "Mango": No coincide (recuerda que "mango" no es lo mismo que "Mango"), pero se ejecuta debido al fall-through, imprime "MANGO".
- case "Banana": No coincide, pero se ejecuta debido al fall-through, imprime "BANANA", y luego se encuentra un break, terminando el switch.

69 Pregunta

```
abstract class Animal {
private String name;
Animal(String name) {
this.name = name;
}
public String getName() {
return name;
}
}
class Dog extends Animal {
private String breed;
Dog(String breed) {
this.breed = breed;
}
Dog(String name, String breed) {
super(name);
this.breed = breed;
}
```

```
public String getBreed() {
return breed;
}
}
class Test {
public static void main(String[] args) {
Dog dog1 = new Dog("Beagle");
Dog dog2 = new Dog("Bubbly", "Poodle");
System.out.println(dog1.getName() + ":" + dog1.getBreed() +
":" +
dog2.getName() + ":" + dog2.getBreed());
}
} // compilation fails
```

fallará al compilar debido a la ausencia de una llamada explícita al constructor de la clase base (Animal) en el constructor de la clase derivada (Dog) que toma solo un parámetro breed. Esto es necesario porque la clase Animal no tiene un constructor sin argumentos (constructor por defecto), y el compilador espera que se invoque un constructor de la clase base.

70 Pregunta

```
public class Main {
public static void main(String[] args) throws ParseException {
String[]sa = {"333.6789","234.111"};
NumberFormat nf = NumberFormat.getInstance();
nf.setMaximumFractionDigits(2);
for (String s: sa
) {

System.out.println(nf.parse(s));
}
}
}/*Salida
333.6789
234.111
*/
```

El método parse devuelve un Number que representa el valor exacto de la cadena de texto, sin modificar el número de dígitos fraccionarios.

71 Pregunta

```
public class Main {
public static void main(String[] args) throws ParseException {
Queue<String> products = new ArrayDeque<String>();
products.add("p1");
products.add("p2");
products.add("p3");
System.out.println(products.peek());
System.out.println(products.poll());
```

```
System.out.println("");
products.forEach(s -> {
System.out.println(s);
});
}
}/**
*p1
* p1
*
* p2
* p3
*/
```

Utiliza una cola (Queue) de tipo ArrayDeque para almacenar y manipular una lista de productos.
Primera impresión (peek()):
Se imprime "p1" porque es el primer elemento de la cola, pero no se remueve.
Segunda impresión (poll()):
Se imprime "p1" nuevamente, pero esta vez se remueve de la cola.
Iteración:
Quedan "p2" y "p3" en la cola, y ambos se imprimen en líneas separadas.

72 Pregunta

```
public class Main {
public static void main(String[] args) throws ParseException {
System.out.println(2+3+5);
System.out.println("+"+2+3*5);
}
}// Salida: 10 + 215
```

La combinación de operadores aritméticos y la concatenación de cadenas
Concatenación de Cadenas:
"+" + 2 se evalúa como la concatenación de la cadena "+" y el número 2, resultando en la cadena "+2".
Luego, "+2" + 15 se evalúa como la concatenación de la cadena "+2" y el número 15, resultando en "+215".
Salida: +215.