1. The **BUILDER** pattern is used to:

Simplify the creation of complex objects.

2. In Java. the difference between **throws** and **throw** is:

throws indicates the type of exception that the method does not handle and throw is used to throw an exception.

throw is used within the method body to indicate that an exception has occurred and needs to be handled

throws is used in the method signature to declare that a method can throw one or more exceptions. It informs the caller of the method about the exceptions that might be thrown during the execution of the method

3. A method is declared to take three arguments. A program calls this method and passes only two arguments. What is the result?

Compilation fails.

4. Which three implementations are valid?


```
class Test implements SampleCloseable{
  public void close() throws java.io.IOException{//do something}
}
```

The exception IOException matches exactly what is declared in the interface.

```
class Test implements SampleCloseable{
  public void close() throws FileNotFoundException {//do something}
}
```

**FileNotFoundException is a subclass of IOException. **

```
class Test implements SampleCloseable{
  public void close(){//do something}
}
```

Omitting the exception is allowed. Declaring no exceptions is narrower than IOException.

5. Which three lines will compile and output “Right on!”?

```
13.  public class Speak {
14.      public static void main(String[] args) {
15.          Speak speakIT = new Tell();
16.          Tell tellIt = new Tell();
17.          speakIT.tellItLikeItIs();
18.          (Truth) speakIT.tellItLikeItIs();
19.          ((Truth) speakIT).tellItLikeItIs();
20.          tellIt.tellItLikeItIs();
21.          (Truth) tellIt.tellItLikeItIs();
22.          ((Truth) tellIt).tellItLikeItIs();
23.      }
24.  }

class Tell extends Speak implements Truth {
    @Override
    public void tellItLikeItIs() {
        System.out.println("Right on!");
    }
}

interface Truth {
    public void tellItLikeItIs();
}
```

```
((Truth)speakIT).tellitLikeItIs();
tellIt.tellitLikeItIs();
((Truth)tellIt).tellitLikeItIs();
```

errores

(speakIT.tellitLikeItIs();): Falla porque el tipo Speak no declara el método tellitLikeItIs.

((Truth)tellIt.tellitLikeItIs();): Falla porque no puedes hacer un cast sobre la invocación de un método.

Las líneas que sí funcionan:

Línea 2: ((Truth)speakIT).tellitLikeItIs();

Funciona porque: Se realiza un **cast explícito** a Truth, lo que indica que el compilador debe tratar a speakIT como un objeto que implementa la interfaz Truth. Luego, el método tellitLikeItIs es accesible porque está declarado en Truth.

Línea 3: tellIt.tellitLikeItIs();

Funciona porque: tellIt es de tipo Tell, y el método tellitLikeItIs está directamente implementado en la clase Tell. No hay necesidad de casting.

Línea 5: ((Truth)tellIt).tellitLikeItIs();

Funciona porque: Aunque tellIt ya implementa Truth, el cast explícito no afecta la funcionalidad. Es redundante pero válida.

6. What changes will make this code compile? SELECCIONA 2

```
class X {
        X() { }
        private void one() { }
}
public class Y extends X {
        Y() { }
        private void two() {
        one();
        }
        public static void main(String[] args) {
        new Y().two();
        }
}
```

Adding the public modifier to the declaration of class X.
Adding the protected modifier to the X() constructor.
Changing the private modifier on the declarationof the one() method to protected.

Removing the Y() constructor.

** 'one()' has private access in 'X' **

7. ¿Qué imprime?

```
public class Main {
  public static void main(string[] args) {
     int x = 2;
     for(;x<5;){
       x=x+1;
       System.out.println(x);
     }
  }
}
```

34…pero tiene error el string?

8. ¿Cuál es el resultado?

```
public class Test {
  public static void main(String[] args) {
    int[][] array = { {0}, {0,1}, {0,2,4}, {0,3,6,9}, {0,4,8,12,16} };
    System.out.println(array{4}{1});
    System.out.println(array{1}{4});
  }
}
```

indexoutofbounds exception

9. Which two possible outputs?

```
public class Main {
        public static void main(String[] args) throws Exception {
                doSomething();
        }
        private static void doSomething() throws Exception {
                System.out.println("Before if clause");
                if (Math.random() > 0.5) { throw new Exception();}
                System.out.println("After if clause");
        }
}
```

Before if clause
After if clause

Before if clause
Exception in thread "main" java.lang.Exception

10. ¿Cuál es la salida?

```
public class Main {
  public static void main(String[] args) {
     int x = 2;
     if(x==2) System.out.println("A");
     else System.out.println("B");
     else System.out.println("C");
  }
}
```

Compilation error (missing an if)

11. How many times is 2 printed?

```
class Menu {
public static void main(String[] args) {
String[] breakfast = {"beans", "egg", "ham", "juice"};
for (String rs : breakfast) {
int dish = 1;
while (dish < breakfast.length) {
System.out.println(rs + "," + dish);
++dish;
}
}
}
}
```

It prints each element of the array three times (beans 2, egg 2, ham 2, juice 2)
2 is printed four times

12. What is the result?

```
class Person {
String name = "No name";
public Person (String nm) {name=nm}
}
class Employee extends Person {
String empID = "0000";
public Employee(String id) { empID " //18
}
}
public class EmployeeTest {
public static void main(String[] args) {
Employee e = new Employee("4321");
System.out.println(e.empID);
}
}
```

A. 4321.
B. 0000.
C. An exception is thrown at runtime.
D. Compilation fails because of an error in line 18.
Employee is missing superclass constructor (cannot use the default because it has the String nm parameters), empID is not assigned.

13. What is the result?

```
class Atom {
Atom() {System.out.print("atom ");}
}
class Rock extends Atom {
Rock(String type) {System.out.print(type);}
}
public class Mountain extends Rock {
Mountain(){
```

```
super("granite ");
new Rock("granite ");
}
public static void main(String[] a) {new Mountain();}
} //atom granite atom granite
```
A. Compilation fails.
B. Atom granite.
C. Granite granite.
D. Atom granite granite.
E. An exception is thrown at runtime.
F. Atom granite atom granite.

14. What is the result?

```
import java.text.*;
public class Align {
public static void main(String[] args) throws ParseException {
String[] sa = {"111.234", "222.5678"};
NumberFormat nf = NumberFormat.getInstance();
nf.setMaximumFractionDigits(3); //NO HACE NADA
for (String s : sa) {
System.out.println(nf.parse(s));
}
}
}
```
A. 111.234 222.567
B. 111.234 222.568
C. 111.234 222.5678
D. An exception is thrown at runtime

15. Given:

```
interface Rideable {
String getGait(); //todos los métodos de una interface deben ser public abstract
}

public class Camel implements Rideable {
int weight = 2;
public static void main(String[] args) {
new Camel().go(8);
}

void go (int speed) {
++speed;
weight++;
int walkrate=speed*weight;
System.out.print(walkrate + getGait());
}
String getGait()//error {
```

```
return " mph, lope";
}
}
```

What is the result?

a) 16 mph, lope

b) 18 mph, lope

c) 24 mph, lope

d) 27 mph, lope

e) Compilation fails.

f) An exception is thrown at run time.

16. Tema: Lambdas con predicate
1. Predicate es una interfaz funcional que evalúa una condición y devuelve un booleano.

2. Las lambdas simplifican su implementación.

3. Métodos útiles:

and: Combina dos predicados con un AND lógico.

or: Combina dos predicados con un OR lógico.

negate: Invierte el resultado del predicado.

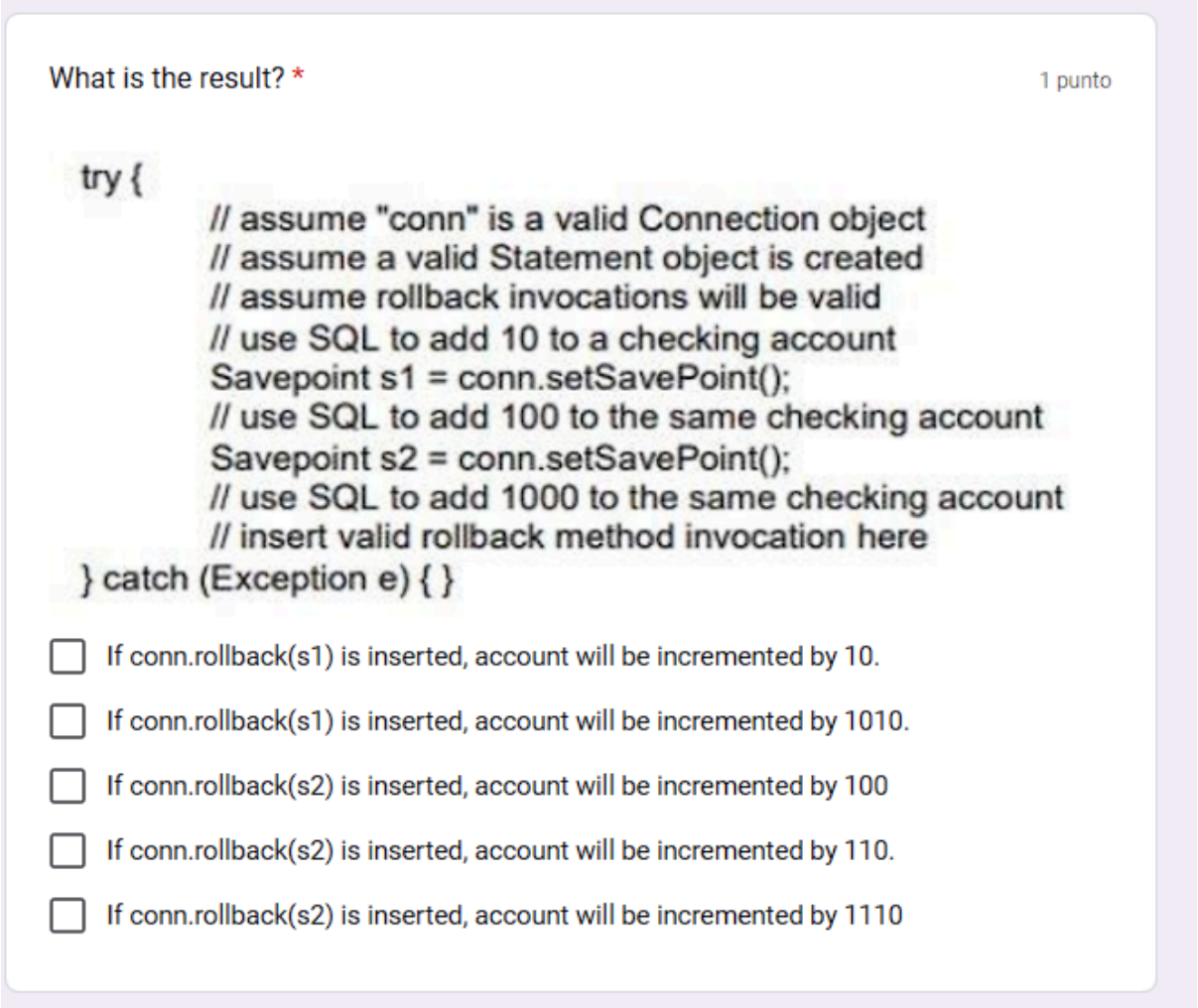
4. Es muy útil para filtrar, validar o aplicar condiciones.

17. Which two actions, used independently, will permit this class to compile?

```
import java.io.IOException;
public class Y {
public static void main(String[] args) {
try {
doSomething();
} catch (RuntimeException e) {
System.out.println(e);
}
}
static void doSomething() {
if (Math.random() > 0.5) {
throw new IOException();
}
```

throw new RuntimeException();
}
}
A. Adding throws IOException to the main() method signature and to the doSomething() method.
B. Adding throws IOException to the main() method signature and changing the catch argument to IOException.
C. Adding throws IOException to the doSomething() method signature.
D. Adding throws IOException to the main() method signature.
E. Adding throws IOException to the doSomething() method signature and changing the catch argument to IOException.

18. ¿Cuál es el resultado?



by 10
by 110

19. Tema: API java.time - Uso de LocalDateTime y Period para manejar fechas y periodos.

Uso de `LocalDateTime` y `Period` en la API `java.time`

---

1. ¿Qué es `LocalDateTime`?

`LocalDateTime` combina una fecha (`LocalDate`) y una hora (`LocalTime`) en un solo objeto.

Creación:

LocalDateTime dateTime = LocalDateTime.of(2024, 11, 21, 10, 30);
System.out.println(dateTime); // 2024-11-21T10:30
También puedes obtener la fecha y hora actual:

LocalDateTime now = LocalDateTime.now();

- **Métodos útiles:**
    - `plusDays(long days)`, `minusMonths(long months)` → Modifican la fecha.
    - `getDayOfWeek()` → Obtiene el día de la semana.
    - `toLocalDate()` o `toLocalTime()` → Extraen la parte de fecha o hora.

---

2. ¿Qué es `Period`?

`Period` representa un intervalo de tiempo en términos de **años**, **meses** y **días**.

Creación:

Period period = Period.of(1, 2, 15); // 1 año, 2 meses, 15 días
O usar métodos como:

Period oneMonth = Period.ofMonths(1);

-
- **Métodos útiles:**
    - `plusYears(int years)` → Añade años al periodo.
    - `getDays()`, `getMonths()` → Obtienen valores específicos del periodo.

---

3. Uso de `LocalDateTime` y `Period` juntos

Puedes sumar o restar un `Period` a un `LocalDateTime` para calcular nuevas fechas:

Ejemplo práctico:
```
 LocalDateTime dateTime = LocalDateTime.of(2024, 11, 21, 10, 30);
Period period = Period.of(0, 1, 10); // 1 mes y 10 días

LocalDateTime newDateTime = dateTime.plus(period);
System.out.println(newDateTime); // 2024-12-31T10:30
```

-

---

4. Comparación de fechas con `LocalDateTime`

Puedes comparar fechas para saber si una es anterior o posterior:

Ejemplo:
```
 LocalDateTime start = LocalDateTime.of(2024, 1, 1, 12, 0);
LocalDateTime end = LocalDateTime.of(2024, 12, 31, 12, 0);

System.out.println(start.isBefore(end)); // true
System.out.println(start.isAfter(end)); // false
```

-

---

5. Resumen:

- `LocalDateTime` → Maneja fechas y horas combinadas.
- `Period` → Representa períodos de años, meses y días.
- **Uso combinado** → Permite realizar cálculos y operaciones entre fechas.

20. ¿Cuáles de las siguientes opciones son válidas?**
A. El constructor predeterminado proporcionado por el compilador puede ser llamado utilizando this().
B. Un constructor puede ser invocado desde un método de instancia.
C. Una variable de instancia puede ser accedida dentro de un método de clase (static).
D. Un constructor puede ser llamado dentro de otro constructor utilizando la palabra clave this().

21. Cual es el resultado

What is the result? * 1 punto

```
class X {
        static void m(int i) {
                i += 7;
        }
        public static void main(String[] args) {
                int i = 12;
                m(i);
                System.out.println(i);
        }
}
```

- ○ 7
- ◉ 12
- ○ 19
- ○ Compilation fails.
- ○ An exception is thrown at run time

12

22.

What is the result if the integer value is 33? *

```
public static void main(String[] args) {
        if (value >= 0) {
                if (value != 0) {
                        System.out.print("the ");
                } else {
                        System.out.print("quick ");
                }
                if (value < 10) {
                        System.out.print("brown ");
                }
                if (value > 30) {
                        System.out.print("fox ");
                } else if (value < 50) {
                        System.out.print("jumps ");
                } else if (value < 10) {
                        System.out.print("over ");
                } else {
                        System.out.print("the ");
                }
                if (value > 10) {
                        System.out.print("lazy ");
                } else {
                        System.out.print("dog ");
                }
                System.out.print("... ");
        }
}
```

- ○ The fox jump lazy ?
- ◉ The fox lazy ?
- ○ Quick fox over lazy ?

The fox lazy …

23. What is the result if you try to compile Truthy.java and then run it with assertions enabled?

```
public class Truthy{
public static void main(String[] args){
int x = 7;
```

assert (x == 6) ? "x == 6" : "x != 6";
}
}

A. Truthy.java compiles and the output is x != 6
B. Truthy.java compiles and an AssertionError isthrown with x != 6 as additionaloutput.
C. Truthy.java does NOT compile. ←
D. Truthy.java compiles and an AssertionError isthrown with no additional output
Incorrect assert syntax

24. Definición del singleton:
The SINGLETON pattern allows: Having a single instance of a class, while allowing all classes to have access to that instance.

25. ¿Cuál de las siguientes afirmaciones sobre la herencia y las interfaces en Java es correcta?

Si una clase puede heredar de una interfaz (error)
Si una clase puede heredar de múltiples clases
Si en una interfaz puede haber métodos con comportamiento y default

26. Multihilos

---

1. Conceptos Básicos

- **Thread:** Representa un hilo de ejecución independiente.
- **Estado de un hilo:**
  - `NEW`: No ha iniciado.
  - `RUNNABLE`: Listo para ejecutarse.
  - `BLOCKED`: Esperando acceso a un recurso.
  - `WAITING` / `TIMED_WAITING`: Esperando indefinidamente o con tiempo límite.
  - `TERMINATED`: Ha finalizado.

---

2. Crear un Hilo

Extender `Thread`:

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Hilo en ejecución");
    }
}
new MyThread().start();
```

-

Implementar `Runnable`:

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Hilo en ejecución");
    }
}
new Thread(new MyRunnable()).start();
```

-

Lambdas para `Runnable`:

```
new Thread(() -> System.out.println("Hilo en ejecución")).start();
```

-

---

3. Métodos Importantes de `Thread`

- **Inicio y finalización:**
  - `start()`: Inicia el hilo.
  - `run()`: Contiene el código que se ejecuta en el hilo.
  - `join()`: Espera a que el hilo termine.
- **Control de suspensión:**
  - `sleep(milliseconds)`: Pausa temporalmente el hilo actual.
- **Interrupciones:**
  - `interrupt()`: Señala que el hilo debe interrumpirse.
  - `isInterrupted()`: Verifica si el hilo ha sido interrumpido.

---

4. Sincronización

Bloques sincronizados:

```
synchronized (objeto) {
    // Código sincronizado
}
```

-

Métodos sincronizados:

```
synchronized void metodo() {
    // Código sincronizado
}
```

-

- **Evitar condiciones de carrera:** Usa sincronización para acceso seguro a recursos compartidos.

---

5. API Avanzada de Concurrencia

ExecutorService:
```
 ExecutorService executor = Executors.newFixedThreadPool(3);
executor.submit(() -> System.out.println("Tarea ejecutada"));
executor.shutdown();
```

-

`ForkJoinPool`**:** Ideal para dividir tareas en subtareas.
```
 ForkJoinPool pool = new ForkJoinPool();
pool.invoke(new RecursiveTask<>() { ... });
```

-
- **Colecciones Concurrentes:**
    - `ConcurrentHashMap`, `CopyOnWriteArrayList`, etc.

---

6. Problemas Comunes

- **Deadlock:** Dos o más hilos esperando indefinidamente por recursos bloqueados.
- **Livelock:** Hilos activos pero incapaces de progresar.
- **Starvation:** Un hilo no puede acceder a recursos debido a la prioridad de otros hilos.
