➔ 1. What will be the result of attempting to compile and run the following program? (1OP)
¿Cuál será el resultado de intentar compilar y ejecutar el siguiente programa?

```
class TestClass{
  public static void main(String args[]){
    int i = 0;
    loop :       // 1
    {
      System.out.println("Loop Lable line");
      try{
        for ( ; true ; i++ ){
          if( i >5) break loop;      // 2
        }
      }
      catch(Exception e){
        System.out.println("Exception in loop.");
      }
      finally{
        System.out.println("In Finally");     // 3
      }
    }
  }
}
```

No compilation error and line 3 will be executed. (A break without a label breaks the current loop (i.e. no iterations any more) and a break with a label tries to pass the control to the given label. 'Tries to' means that if the break is in a try block and the try block has a finally clause associated with it then it will be executed).

Una instrucción break sin etiqueta interrumpe el bucle actual (es decir, no habrá más iteraciones) y un break con etiqueta intenta pasar el control a la etiqueta dada. 'Intenta' significa que si el break está en un bloque try y el bloque try tiene una cláusula finally asociada, entonces esta se ejecutará.

➔ 2. What will be the result of compiling and running the following program? (1OP)

```
class NewException extends Exception {}
class AnotherException extends Exception {}
public class ExceptionTest{
    public static void main(String [] args) throws Exception{
            try{
                    m2();
            }
            finally{ m3(); }
    }
    public static void m2() throws NewException{  throw new NewException();  }
    public static void m3() throws AnotherException{  throw new
AnotherException();  }
}
```

It will compile but will throw AnotherException when run. m2() throws NewException, which is not caught anywhere. But before exiting out of the main method, finally must be executed. Since finally throw AnotherException (due to a call to m3()), the NewException thrown in the try block (due to call to m2()) is ignored and AnotherException is thrown from the main method.

m2() lanza NewException, que no es capturada en ninguna parte. Pero antes de salir del método principal, se debe ejecutar finally. Dado que finally lanza AnotherException (debido a una llamada a m3()), la NewException lanzada en el bloque try (debido a la llamada a m2()) se ignora y se lanza AnotherException desde el método principal.

➔ 3. What will be the result of attempting to compile and run the following program?(1OP)
¿Cuál será el resultado de intentar compilar y ejecutar el siguiente programa?

```
public class TestClass{
    public static void main(String args[]){
        Exception e = null;
        throw e;
    }
}
```

The code will fail to compile. The main method is throwing a checked exception but there is no try/catch block to handle it and neither is there a throws clause that declares the checked exception. So, it will not compile. If you either put a try catch block or declare a throws clause for the method then it will throw a NullPointerException at run time because e is null. A method that throws a "checked" exception i.e. an exception that is not a subclass of Error or RuntimeException, either must declare it in throws clause or put the code that throws the exception within a try/catch block

El método main está lanzando una excepción comprobada, pero no hay un bloque try/catch para manejarla y tampoco hay una cláusula throws que declare la excepción comprobada. Por lo tanto, no compilará. Si pones un bloque try/catch o declaras una cláusula throws para el método, entonces lanzará un NullPointerException en tiempo de ejecución porque e es nulo. Un método que lanza una excepción "comprobada", es decir, una excepción que no es una subclase de Error o RuntimeException, debe declararla en la cláusula throws o colocar el código que lanza la excepción dentro de un bloque try/catch.

➔ 4. Which statements regarding the following code are correct ? (2OP)
¿Qué afirmaciones sobre el siguiente código son correctas?

```
class Base{
void method1() throws java.io.IOException, NullPointerException{
    someMethod("arguments");
    // some I/O operations
  }
  int someMethod(String str){
    if(str == null) throw new NullPointerException();
    else return str.length();
  }
 }
 public class NewBase extends Base{
```

```
    void method1(){
        someMethod("args");
    }
}
```

- method1 in class NewBase does not need to specify any exceptions.
- There is no problem with the code.

Overriding method only needs to specify a subset of the list of exception classes the overridden method can throw. A set of no classes is a valid subset of that list. Remember that NullPointerException is a subclass of RuntimeException, while IOException is a subclass of Exception.

El método que sobrescribe solo necesita especificar un subconjunto de la lista de clases de excepciones que el método sobrescrito puede lanzar. Un conjunto sin clases es un subconjunto válido de esa lista. Recuerda que NullPointerException es una subclase de RuntimeException, mientras que IOException es una subclase de Exception.

➔ 5. Consider the following code: Which of the following statements are correct?(1OP)

```
class A {
  public void doA(int k) throws Exception {  // 0
    for(int i=0; i< 10; i++) {
      if(i == k) throw new Exception("Index of k is "+i); // 1
    }
  }
  public void doB(boolean f) { // 2
    if(f) {
      doA(15); // 3
    }
    else return;
  }
  public static void main(String[] args) { // 4
    A a = new A();
    a.doB(args.length>0); // 5
  }
}
```

This will compile if throws Exception is added at line //2 as well as //4

Any checked exceptions must either be handled using a try block or the method that generates the exception must declare that it throws that exception. In this case, doA() declares that it throws Exception. doB() is calling doA but it is not handling the exception generated by doA(). So, it must declare that it throws Exception. Now, the main() method is calling doB(), which generates an exception (due to a call to doA()). Therefore, main() must also either wrap the call to doB() in a try block or declare it in its throws clause. The main(String[] args) method is the last point in your program where any unhandled checked exception can bubble up to. After that the exception is thrown to the JVM and the JVM kills the thread.

Cualquier excepción comprobada debe ser manejada utilizando un bloque try o el método que genera la excepción debe declarar que lanza esa excepción. En este caso, doA() declara que lanza Exception. doB() está llamando a doA() pero no está manejando la excepción generada por doA(). Por lo tanto, debe declarar que lanza Exception. Ahora, el método main() está llamando a doB(), que genera una excepción (debido a una llamada a doA()). Por lo tanto, main() también debe envolver la llamada a doB() en un bloque try o declararlo en su cláusula throws. El método main(String[] args) es el último punto en tu programa donde cualquier excepción comprobada no manejada puede burbujear hacia arriba. Después de eso, la excepción se lanza a la JVM y la JVM termina el hilo.

➔ 6. What will the following code print when compiled and run? (Assume that MySpecialException is an unchecked exception.) (1OP)
¿Qué imprimirá el siguiente código cuando se compile y ejecute? (Supongamos que MySpecialException es una excepción no comprobada.)

```
1. public class ExceptionTest {
2.   public static void main(String[] args) {
3.     try {
4.       doSomething();
5.     } catch (MySpecialException e) {
6.       System.out.println(e);
7.     }
8.   }
9.
10.  static void doSomething() {
11.    int[] array = new int[4];
12.    array[4] = 4;
13.    doSomethingElse();
14.  }
15.
16.  static void doSomethingElse() {
17.    throw new MySpecialException("Sorry, can't do something else");
18.  }}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
 at ExceptionTest.doSomething(ExceptionTest.java:12)
 at ExceptionTest.main(ExceptionTest.java:4)
```

Since the length of array is only 4, you can't do array[4], because that would access the 5th element. Hence, an ArrayIndexOutOfBoundsException will be thrown at line 12. Line 13 will not even be executed. Since the exception is not caught anywhere, it will be thrown out to the JVM, which will print the stack trace of the exception.

Dado que la longitud del arreglo es solo 4, no puedes hacer array[4], porque eso accedería al quinto elemento. Por lo tanto, se lanzará una ArrayIndexOutOfBoundsException en la línea 12. La línea 13 ni siquiera se ejecutará. Dado que la excepción no se captura en ninguna parte, se lanzará a la JVM, que imprimirá el seguimiento de la pila de la excepción.

➔ 7. What is wrong with the following code?
¿Qué está mal con el siguiente código?

```
class MyException extends Exception {}
public class TestClass{
  public static void main(String[] args){
    TestClass tc = new TestClass();
    try{
      tc.m1();
    }
    catch (MyException e){
      tc.m1();
    }
    finally{
      tc.m2();
    }
  }
  public void m1() throws MyException{
    throw new MyException();
  }
  public void m2() throws RuntimeException{
    throw new NullPointerException();
  }
}
```

It will not compile because of unhandled exception.
The catch block is throwing a checked exception (i.e. non-RuntimeException) which must be handled by either a try catch block or declared in the throws clause of the enclosing method. Note that finally is also throwing an exception here, but it is a RuntimeException so there is no need to handle it or declare it in the throws clause.

El bloque catch está lanzando una excepción comprobada (es decir, no es una RuntimeException), que debe ser manejada ya sea por un bloque try/catch o declarada en la cláusula throws del método que la contiene. Ten en cuenta que finally también está lanzando una excepción aquí, pero es una RuntimeException, por lo que no es necesario manejarla ni declararla en la cláusula throws.

➔ 8. What will the following code snippet print:

```
Float f = null;
try{
   f = Float.valueOf("12.3");
   String s = f.toString();
   int i = Integer.parseInt(s);
   System.out.println(""+i);
 }
 catch(Exception e){
   System.out.println("trouble : "+f);
 }
```

trouble : 12.3

f = Float.valueOf("12.3"); executes without any problem. int i = Integer.parseInt(s); throws a NumberFormatException because 12.3 is not an integer. Thus, the catch block prints trouble : 12.3

f = Float.valueOf("12.3"); se ejecuta sin ningún problema. int i = Integer.parseInt(s); lanza una NumberFormatException porque 12.3 no es un entero. Por lo tanto, el bloque catch imprime "trouble : 12.3".

➔ 9. What will the following code print?

```
public class Test{
  public int luckyNumber(int seed){
    if(seed > 10) return seed%10;
      int x = 0;
        try{
          if(seed%2 == 0) throw new Exception("No Even Number Please.");
          else return x;
        }
        catch(Exception e){
          return 3;
        }
        finally{
          return 7;
        }
      }
      public static void main(String args[]){
       int amount = 100, seed = 6;
       switch( new Test().luckyNumber(6) ){
          case 3: amount = amount * 2;
          case 7: amount = amount * 2;
          case 6: amount = amount + amount;
          default :
       }
       System.out.println(amount);
     }}
```

400

Remember that when a finally clause has a return statement then this return statement supersedes any return statement that the try block or the catch blocks might have. Now, in this case, when you pass 6 to luckyNumber(int seed), if(seed%2 == 0) throw new Exception("No Even Number Please."); is executed and the exception is caught by the catch block where it tries to return 3. But since there is a finally block associated with the try/catch block, it is executed before anything is actually returned to the caller. Now, since the finally clause has a return statement, this return statement supersedes the return statements of the try or catch blocks. Thus, the final return value will be the value returned by the finally block, which is 7. In

fact, irrespective of whether the try block throws an exception or not, this code will always return 7. Now, in the switch there is no break statement. So both - case 7: amount = amount * 2; and case 6: amount = amount + amount; are executed. so the final amount becomes 400.

Recuerda que cuando una cláusula finally tiene una instrucción de retorno, esta instrucción de retorno anula cualquier instrucción de retorno que pueda tener el bloque try o los bloques catch. Ahora, en este caso, cuando pasas 6 a luckyNumber(int seed), se ejecuta if(seed%2 == 0) throw new Exception("No Even Number Please."); y la excepción es capturada por el bloque catch donde intenta devolver 3. Pero dado que hay un bloque finally asociado con el bloque try/catch, se ejecuta antes de que realmente se devuelva algo al llamador. Ahora, dado que la cláusula finally tiene una instrucción de retorno, esta instrucción de retorno anula las instrucciones de retorno de los bloques try o catch. Por lo tanto, el valor de retorno final será el valor devuelto por el bloque finally, que es 7. De hecho, independientemente de si el bloque try lanza una excepción o no, este código siempre devolverá 7. Ahora, en el switch no hay una instrucción break. Así que tanto el case 7: amount = amount * 2; como el case 6: amount = amount + amount; se ejecutan, por lo que el monto final se convierte en 400.

➔ **10. Java's Exception mechanism helps in which of the following ways?**
El mecanismo de excepciones de Java ayuda en las siguientes formas:

It allows creation of new exceptions that are custom to a particular application domain.
Permite la creación de nuevas excepciones personalizadas para un dominio de aplicación particular.

- You can define your own exceptions based on your application business domain. For example, in a banking application, you might want to create a InsufficientFundsException. This increases code clarity as compared to having a single (or a few standard) exception class(es) and looking at the exception code to determine what happened.
- Puedes definir tus propias excepciones basadas en el dominio de negocio de tu aplicación. Por ejemplo, en una aplicación bancaria, podrías querer crear una InsufficientFundsException. Esto aumenta la claridad del código en comparación con tener una sola (o unas pocas) clases de excepción estándar y mirar el código de la excepción para determinar qué ocurrió.

It improves code because error handling code is clearly separated from the main program logic.
Mejora el código porque el manejo de errores está claramente separado de la lógica principal del programa.

- The error handling logic is put in the catch block, which makes the main flow of the program clean and easily understandable.
- La lógica de manejo de errores se coloca en el bloque catch, lo que hace que el flujo principal del programa sea limpio y fácilmente comprensible.

It enhances the security of the application by reporting errors in the logs.
Aumenta la seguridad de la aplicación al informar errores en los registros.

- Exception handling as such has nothing to do with the security of the application but good exception handling in an application can prevent security holes.

- Aunque el manejo de excepciones como tal no tiene que ver con la seguridad de la aplicación, un buen manejo de excepciones puede prevenir agujeros de seguridad.

It improves the code because the exception is handled right at the place where it occured.
Mejora el código porque la excepción se maneja justo en el lugar donde ocurrió.

- Just the opposite is true. It improves the code because the code does not have to include error handling code if it is not capable of handling it. It can propagate the exception up the chain and so that the exception can be handled somewhere at a more appropriate place.
- Justo lo contrario es cierto. Mejora el código porque no tiene que incluir código de manejo de errores si no es capaz de manejarlo. Puede propagar la excepción hacia arriba en la cadena para que se maneje en un lugar más apropiado.

It provides a vast set of standard exceptions that covers all possible exceptions.
Proporciona un vasto conjunto de excepciones estándar que cubren todas las posibles excepciones.

- Although it does provide a vast set of standard exceptions, they cannot cover all scenarios. But you can always create new exceptions tailored for your application.
- Aunque proporciona un vasto conjunto de excepciones estándar, no pueden cubrir todos los escenarios. Sin embargo, siempre puedes crear nuevas excepciones adaptadas a tu aplicación.

→ 1. What will be the result of attempting to compile and run the following program? (1OP) ¿Cuál será el resultado de intentar compilar y ejecutar el siguiente programa?

```
class TestClass{
  public static void main(String args[]){
        int i = 0;
        loop :              // 1
        {
        System.out.println("Loop Lable line");
        try{
        for (  ;  true ;  i++ ){
        if( i >5) break loop;     // 2
        }
        }
        catch(Exception e){
        System.out.println("Exception in loop.");
        }
        finally{
        System.out.println("In Finally");         // 3
        }
```

```
        }
    }
}
```

No compilation error and line 3 will be executed. (A break without a label breaks the current loop (i.e. no iterations any more) and a break with a label tries to pass the control to the given label. 'Tries to' means that if the break is in a try block and the try block has a finally clause associated with it then it will be executed).

Una instrucción break sin etiqueta interrumpe el bucle actual (es decir, no habrá más iteraciones) y un break con etiqueta intenta pasar el control a la etiqueta dada. 'Intenta' significa que si el break está en un bloque try y el bloque try tiene una cláusula finally asociada, entonces esta se ejecutará.

→ 2. What will be the result of compiling and running the following program? (1OP)

```
class NewException extends Exception {}
class AnotherException extends Exception {}
public class ExceptionTest{
        public static void main(String [] args) throws Exception{
                try{
                        m2();
                }
                finally{ m3(); }
        }
        public static void m2() throws NewException{  throw new NewException();  }
        public static void m3() throws AnotherException{  throw new
AnotherException();  }
}
```

It will compile but will throw AnotherException when run. m2() throws NewException, which is not caught anywhere. But before exiting out of the main method, finally must be executed. Since finally throw AnotherException (due to a call to m3()), the NewException thrown in the try block (due to call to m2()) is ignored and AnotherException is thrown from the main method.

m2() lanza NewException, que no es capturada en ninguna parte. Pero antes de salir del método principal, se debe ejecutar finally. Dado que finally lanza AnotherException (debido a una llamada a m3()), la NewException lanzada en el bloque try (debido a la llamada a m2()) se ignora y se lanza AnotherException desde el método principal.

→ 3. What will be the result of attempting to compile and run the following program?(1OP) ¿Cuál será el resultado de intentar compilar y ejecutar el siguiente programa?

```
public class TestClass{
        public static void main(String args[]){
                Exception e = null;
           throw e;
        }
}
```

The code will fail to compile. The main method is throwing a checked exception but there is no try/catch block to handle it and neither is there a throws clause that declares the checked exception. So, it will not compile. If you either put a try catch block or declare a throws clause for the method then it will throw a NullPointerException at run time because e is null. A method that throws a "checked" exception i.e. an exception that is not a subclass of Error or RuntimeException, either must declare it in throws clause or put the code that throws the exception within a try/catch block

El método main está lanzando una excepción comprobada, pero no hay un bloque try/catch para manejarla y tampoco hay una cláusula throws que declare la excepción comprobada. Por lo tanto, no compilará. Si pones un bloque try/catch o declaras una cláusula throws para el método, entonces lanzará un NullPointerException en tiempo de ejecución porque e es nulo. Un método que lanza una excepción "comprobada", es decir, una excepción que no es una subclase de Error o RuntimeException, debe declararla en la cláusula throws o colocar el código que lanza la excepción dentro de un bloque try/catch.

→ 4. Which statements regarding the following code are correct ?
(2OP) ¿Qué afirmaciones sobre el siguiente código son correctas?

```
class Base{
void method1() throws java.io.IOException, NullPointerException{
        someMethod("arguments");
        // some I/O operations
        }
        int someMethod(String str){
        if(str == null) throw new NullPointerException();
        else return str.length();
        }
 }
 public class NewBase extends Base{
        void method1(){
        someMethod("args");
        }
 }
```

- method1 in class NewBase does not need to specify any exceptions.

- There is no problem with the code.

Overriding method only needs to specify a subset of the list of exception classes the overridden method can throw. A set of no classes is a valid subset of that list. Remember that NullPointerException is a subclass of RuntimeException, while IOException is a subclass of Exception.

El método que sobrescribe solo necesita especificar un subconjunto de la lista de clases de excepciones que el método sobrescrito puede lanzar. Un conjunto sin clases es un subconjunto válido de esa lista. Recuerda que NullPointerException es una subclase de RuntimeException, mientras que IOException es una subclase de Exception.

→ 5. Consider the following code: Which of the following statements are correct?(1OP)

```
class A {
        public void doA(int k) throws Exception {  // 0
        for(int i=0; i< 10; i++) {
        if(i == k) throw new Exception("Index of k is "+i); // 1
        }
        }
        public void doB(boolean f) { // 2
        if(f) {
        doA(15); // 3
        }
        else return;
        }
        public static void main(String[] args) { // 4
        A a = new A();
        a.doB(args.length>0); // 5
        }
}
```

This will compile if throws Exception is added at line //2 as well as //4

Any checked exceptions must either be handled using a try block or the method that generates the exception must declare that it throws that exception. In this case, doA() declares that it throws Exception. doB() is calling doA but it is not handling the exception generated by doA(). So, it must declare that it throws Exception. Now, the main() method is calling doB(), which generates an exception (due to a call to doA()). Therefore, main() must also either wrap the call to doB() in a try block or declare it in its throws clause. The main(String[] args) method is the last point in your program where any unhandled checked exception can bubble up to. After that the exception is thrown to the JVM and the JVM kills the thread.

Cualquier excepción comprobada debe ser manejada utilizando un bloque try o el método que genera la excepción debe declarar que lanza esa excepción. En este caso, doA() declara que lanza Exception. doB() está llamando a doA() pero no está manejando la excepción generada por doA(). Por lo tanto, debe declarar que lanza Exception. Ahora, el método main() está llamando a doB(), que genera una excepción (debido a una llamada a doA()). Por lo tanto, main() también debe envolver la llamada a doB() en un bloque try o declararlo en su cláusula throws. El método main(String[] args) es el último punto en tu programa donde cualquier excepción comprobada no manejada puede burbujear hacia arriba. Después de eso, la excepción se lanza a la JVM y la JVM termina el hilo.

→ **6.What will the following code print when compiled and run? (Assume that MySpecialException is an unchecked exception.) (1OP)**

```
1. public class ExceptionTest {
2.     public static void main(String[] args) {
3.     try {
4.     doSomething();
5.     } catch (MySpecialException e) {
6.     System.out.println(e);
7.     }
8.     }
9.
10.    static void doSomething() {
11.    int[] array = new int[4];
12.    array[4] = 4;
13.    doSomethingElse();
14.    }
15.
16.    static void doSomethingElse() {
17.    throw new MySpecialException("Sorry, can't do something else");
18. } }
```

¿Qué imprimirá el siguiente código cuando se compile y ejecute? (Supongamos que MySpecialException es una excepción no comprobada.)

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
at ExceptionTest.doSomething(ExceptionTest.java:12)
at ExceptionTest.main(ExceptionTest.java:4)

Since the length of array is only 4, you can't do array[4], because that would access the 5th element. Hence, an ArrayIndexOutOfBoundsException will be thrown at line 12. Line

13 will not even be executed. Since the exception is not caught anywhere, it will be thrown out to the JVM, which will print the stack trace of the exception.

Dado que la longitud del arreglo es solo 4, no puedes hacer array[4], porque eso accedería al quinto elemento. Por lo tanto, se lanzará una ArrayIndexOutOfBoundsException en la línea 12. La línea 13 ni siquiera se ejecutará. Dado que la excepción no se captura en ninguna parte, se lanzará a la JVM, que imprimirá el seguimiento de la pila de la excepción.

→ 7. What is wrong with the following code? ¿Qué está mal con el siguiente código?

```
class MyException extends Exception {}
public class TestClass{
  public static void main(String[] args){
        TestClass tc = new TestClass();
        try{
        tc.m1();
        }
        catch (MyException e){
        tc.m1();
        }
        finally{
        tc.m2();
        }
  }
  public void m1() throws MyException{
        throw new MyException();
  }
  public void m2() throws RuntimeException{
        throw new NullPointerException();
  }
}
```

It will not compile because of unhandled exception.

The catch block is throwing a checked exception (i.e. non-RuntimeException) which must be handled by either a try catch block or declared in the throws clause of the enclosing method. Note that finally is also throwing an exception here, but it is a RuntimeException so there is no need to handle it or declare it in the throws clause.

El bloque catch está lanzando una excepción comprobada (es decir, no es una RuntimeException), que debe ser manejada ya sea por un bloque try/catch o declarada en

la cláusula throws del método que la contiene. Ten en cuenta que finally también está lanzando una excepción aquí, pero es una RuntimeException, por lo que no es necesario manejarla ni declararla en la cláusula throws.

→ **8. What will the following code snippet print:**

```
Float f = null;
try{
        f = Float.valueOf("12.3");
        String s = f.toString();
        int i = Integer.parseInt(s);
        System.out.println(""+i);
 }
 catch(Exception e){
        System.out.println("trouble : "+f);
 }
```

trouble : 12.3

f = Float.valueOf("12.3"); executes without any problem. int i = Integer.parseInt(s); throws a NumberFormatException because 12.3 is not an integer. Thus, the catch block prints trouble : 12.3

f = Float.valueOf("12.3"); se ejecuta sin ningún problema. int i = Integer.parseInt(s); lanza una NumberFormatException porque 12.3 no es un entero. Por lo tanto, el bloque catch imprime "trouble : 12.3".

→ **9. What will the following code print?**

```
public class Test{
        public int luckyNumber(int seed){
        if(seed > 10) return seed%10;
        int x = 0;
         try{
                if(seed%2 == 0) throw new Exception("No Even Number Please.");
                else return x;
        }
        catch(Exception e){
                return 3;
        }
        finally{
                return 7;
        }
```

```
}
public static void main(String args[]){
int amount = 100, seed = 6;
switch( new Test().luckyNumber(6) ){
        case 3: amount = amount * 2;
        case 7: amount = amount * 2;
        case 6: amount = amount + amount;
        default :
}
System.out.println(amount);
} }
```

400

Remember that when a finally clause has a return statement then this return statement supersedes any return statement that the try block or the catch blocks might have. Now, in this case, when you pass 6 to luckyNumber(int seed), if(seed%2 == 0) throw new Exception("No Even Number Please."); is executed and the exception is caught by the catch block where it tries to return 3. But since there is a finally block associated with the try/catch block, it is executed before anything is actually returned to the caller. Now, since the finally clause has a return statement, this return statement supersedes the return statements of the try or catch blocks. Thus, the final return value will be the value returned by the finally block, which is 7. In fact, irrespective of whether the try block throws an exception or not, this code will always return 7. Now, in the switch there is no break statement. So both - case 7: amount = amount * 2; and case 6: amount = amount + amount; are executed. so the final amount becomes 400.

Recuerda que cuando una cláusula finally tiene una instrucción de retorno, esta instrucción de retorno anula cualquier instrucción de retorno que pueda tener el bloque try o los bloques catch. Ahora, en este caso, cuando pasas 6 a luckyNumber(int seed), se ejecuta if(seed%2 == 0) throw new Exception("No Even Number Please."); y la excepción es capturada por el bloque catch donde intenta devolver 3. Pero dado que hay un bloque finally asociado con el bloque try/catch, se ejecuta antes de que realmente se devuelva algo al llamador. Ahora, dado que la cláusula finally tiene una instrucción de retorno, esta instrucción de retorno anula las instrucciones de retorno de los bloques try o catch. Por lo tanto, el valor de retorno final será el valor devuelto por el bloque finally, que es 7. De hecho, independientemente de si el bloque try lanza una excepción o no, este código siempre devolverá 7. Ahora, en el switch no hay una instrucción break. Así que tanto el case 7: amount = amount * 2; como el case 6: amount = amount + amount; se ejecutan, por lo que el monto final se convierte en 400.

→ 10. Java's Exception mechanism helps in which of the following ways?

El mecanismo de excepciones de Java ayuda en las siguientes formas:

- **It allows creation of new exceptions that are custom to a particular application domain.** Permite la creación de nuevas excepciones personalizadas para un dominio de aplicación particular.

You can define your own exceptions based on your application business domain. For example, in a banking application, you might want to create a InsufficientFundsException. This increases code clarity as compared to having a single (or a few standard) exception class(es) and looking at the exception code to determine what happened.

Puedes definir tus propias excepciones basadas en el dominio de negocio de tu aplicación. Por ejemplo, en una aplicación bancaria, podrías querer crear una InsufficientFundsException. Esto aumenta la claridad del código en comparación con tener una sola (o unas pocas) clases de excepción estándar y mirar el código de la excepción para determinar qué ocurrió.

- **It improves code because error handling code is clearly separated from the main program logic.** Mejora el código porque el manejo de errores está claramente separado de la lógica principal del programa.

The error handling logic is put in the catch block, which makes the main flow of the program clean and easily understandable.

La lógica de manejo de errores se coloca en el bloque catch, lo que hace que el flujo principal del programa sea limpio y fácilmente comprensible.

- **It enhances the security of the application by reporting errors in the logs.** Aumenta la seguridad de la aplicación al informar errores en los registros.

Exception handling as such has nothing to do with the security of the application but good exception handling in an application can prevent security holes.

Aunque el manejo de excepciones como tal no tiene que ver con la seguridad de la aplicación, un buen manejo de excepciones puede prevenir agujeros de seguridad.

- **It improves the code because the exception is handled right at the place where it occured.** Mejora el código porque la excepción se maneja justo en el lugar donde ocurrió.

Just the opposite is true. It improves the code because the code does not have to include error handling code if it is not capable of handling it. It can propagate the exception up the chain and so that the exception can be handled somewhere at a more appropriate place.

Justo lo contrario es cierto. Mejora el código porque no tiene que incluir código de manejo de errores si no es capaz de manejarlo. Puede propagar la excepción hacia arriba en la cadena para que se maneje en un lugar más apropiado.

- **It provides a vast set of standard exceptions that covers all possible exceptions.** Proporciona un vasto conjunto de excepciones estándar que cubren todas las posibles excepciones.

Although it does provide a vast set of standard exceptions, they cannot cover all scenarios. But you can always create new exceptions tailored for your application.

Aunque proporciona un vasto conjunto de excepciones estándar, no pueden cubrir todos los escenarios. Sin embargo, siempre puedes crear nuevas excepciones adaptadas a tu aplicación.

→ 11. What will be the output when the following program is run? (Assume that there is no error in the line numbers given in the options.)

```
package exceptions;
public class TestClass {
    public static void main(String[] args) {
        try{
            doTest();
        }
        catch(MyException me){
           System.out.println(me);
        }
    }
         static void doTest() throws MyException{
        int[] array = new int[10];
        array[10] = 1000;
        doAnotherTest();
    }
         static void doAnotherTest() throws MyException{
               throw new MyException("Exception from doAnotherTest");
    }
} class MyException extends Exception {
    public MyException(String msg){
     super(msg);
   }
}
```

OPCIONES:

- Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
  at exceptions.TestClass.doTest(TestClass.java:14)
  at exceptions.TestClass.main(TestClass.java:5)
- Error in thread "main" java.lang.ArrayIndexOutOfBoundsException
- exceptions.MyException: Exception from doAnotherTest
- exceptions.MyException: Exception from doAnotherTest
  at exceptions.TestClass.doAnotherTest(TestClass.java:29)
  at exceptions.TestClass.doTest(TestClass.java:25)
  at exceptions.TestClass.main(TestClass.java:14)

EXPLICACION:

You are creating an array of length 10. Since array numbering starts with 0, the last element would be array[9]. array[10] would be outside the range of the array and therefore an ArrayIndexOutOfBoundsException will be thrown, which cannot be caught by catch(MyException) clause. The exception is thus thrown out of the main method and is handled by the JVM's uncaught exception handling mechanism, which prints the stack trace.

Note that there are a few questions in the exam that test your knowledge about how exception messages are printed. When you use System.out.println(exception), a stack trace is not printed. Just the name of the exception class and the message is printed. When you use exception.printStackTrace(), a complete chain of the names of the methods called, along with the line numbers, is printed. It contains the names of the methods in the chain of method calls that led to the place where the exception was created going back up to the point where the thread, in which the exception was created, was started.

Estás creando un arreglo de longitud 10. Dado que la numeración de arreglos comienza en 0, el último elemento sería array[9]. array[10] estaría fuera del rango del arreglo y, por lo tanto, se lanzará una ArrayIndexOutOfBoundsException, que no puede ser capturada por la cláusula catch(MyException). La excepción, por lo tanto, se lanza fuera del método main y es manejada por el mecanismo de manejo de excepciones no capturadas de la JVM, que imprime el rastreo de la pila.

Ten en cuenta que hay algunas preguntas en el examen que ponen a prueba tu conocimiento sobre cómo se imprimen los mensajes de excepción. Cuando usas System.out.println(exception), no se imprime un rastreo de la pila. Solo se imprime el nombre de la clase de excepción y el mensaje. Cuando usas exception.printStackTrace(), se imprime una cadena completa de los nombres de los métodos llamados, junto con los números de línea. Contiene los nombres de los métodos en la cadena de llamadas que llevaron al lugar donde se creó la excepción, retrocediendo hasta el punto donde se inició el hilo en el que se creó la excepción.

→ 12. What will be the output of the following program?

```
class TestClass{
   public static void main(String[] args) throws Exception{
     try{
       amethod();
       System.out.println("try ");
     }
     catch(Exception e){
       System.out.print("catch ");
     }
     finally   {
       System.out.print("finally ");
     }
     System.out.print("out ");
   }
   public static void amethod(){ }
}
```

OPCIONES

- try finally
- try finally out
- try out
- catch finally out
- It will not compile because amethod() does not throw any exception.

EXPLICACION:
Since the method amethod() does not throw any exception, try is printed and the control goes to finally which prints finally. After that out is printed.

→13. Which of the following standard java exception classes extend java.lang.RuntimeException?

OPCIONES

- java.lang.SecurityException

SecurityException extends RuntimeException: It is thrown by the security manager upon security violation. For example, when a java program runs in a sandbox (such as an applet) and it tries to use prohibited APIs such as File I/O, the security manager throws this exception. Since this exception is explicitly thrown using the new keyword by a security manager class, it can be considered to be thrown by the application programmer.
(SecurityException extiende RuntimeException: Se lanza por el administrador de seguridad en caso de una violación de seguridad. Por ejemplo, cuando un programa Java se ejecuta en un sandbox (como un applet) y intenta utilizar APIs prohibidas, como la entrada/salida de archivos, el administrador de seguridad lanza esta excepción. Dado que esta excepción se lanza explícitamente utilizando la palabra clave new por una clase del administrador de seguridad, se puede considerar que es lanzada por el programador de la aplicación.)

- java.lang.ClassCastException

ClassCastException extends RuntimeException: Usually thrown by the JVM. Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance. For example, the following code generates a ClassCastException: Object x = new Integer(0); System.out.println((String)x);
(ClassCastException extiende RuntimeException: Normalmente es lanzada por la JVM. Se lanza para indicar que el código ha intentado convertir un objeto a una subclase de la cual no es una instancia. Por ejemplo, el siguiente código genera una ClassCastException:
Object x = new Integer(0);
System.out.println((String)x);)

- java.lang.NullPointerException

NullPointerException extends RuntimeException: Usually thrown by the JVM. Thrown when an application attempts to use null in a case where an object is required. These include: Calling the instance method of a null object. Accessing or modifying the field of a null object. Taking the length of null as if it were an array. Accessing or modifying the slots of null as if it were an array. Throwing null as if it were a Throwable value.

Applications should throw instances of this class to indicate other illegal uses of the null object.

NullPointerException extiende RuntimeException: Normalmente es lanzada por la JVM. Se lanza cuando una aplicación intenta usar null en un caso donde se requiere un objeto. Estos incluyen:

- Llamar al método de instancia de un objeto null.
- Acceder o modificar el campo de un objeto null.
- Tomar la longitud de null como si fuera un arreglo.
- Acceder o modificar los elementos de null como si fuera un arreglo.
- Lanzar null como si fuera un valor Throwable.

Las aplicaciones deben lanzar instancias de esta clase para indicar otros usos ilegales del objeto null.

- java.lang.CloneNotSupportedException

public class CloneNotSupportedException extends Exception Thrown to indicate that the clone method in class Object has been called to clone an object, but that the object's class does not implement the Cloneable interface. Applications that override the clone method can also throw this exception to indicate that an object could not or should not be cloned.

public class CloneNotSupportedException extends Exception

Lanzada para indicar que se ha llamado al método clone en la clase Object para clonar un objeto, pero que la clase del objeto no implementa la interfaz Cloneable. Las aplicaciones que sobrescriben el método clone también pueden lanzar esta excepción para indicar que un objeto no pudo o no debería ser clonado.

- java.lang.IndexOutOfBoundsException

IndexOutOfBoundsException extends RuntimeException: Usually thrown by the JVM. Thrown to indicate that an index of some sort (such as to an array, to a string, or to a vector) is out of range. Applications can subclass this class to indicate similar exceptions. ArrayIndexOutOfBoundsException and StringIndexOutOfBoundsException both extend IndexOutOfBoundsException.
IndexOutOfBoundsException extiende RuntimeException: Normalmente es lanzada por la JVM. Se lanza para indicar que un índice de algún tipo (como para un arreglo, una cadena o un vector) está fuera de rango. Las aplicaciones pueden subclasificar esta clase para indicar excepciones similares. ArrayIndexOutOfBoundsException y StringIndexOutOfBoundsException ambas extienden IndexOutOfBoundsException.

EXPLICACION:
The other two exceptions you should know about are: IllegalArgumentException extends RuntimeException: If a parameter passed to a method is not valid. Usually thrown by the application. IllegalStateException extends RuntimeException: Signals that a method has been invoked at an illegal or inappropriate time. In other words, the Java environment or Java application is not in an appropriate state for the requested operation. Usually thrown by the application.

- **IllegalArgumentException** extiende RuntimeException: Si un parámetro pasado a un método no es válido. Normalmente lanzada por la aplicación.

- **IllegalStateException** extiende RuntimeException: Señala que un método ha sido invocado en un momento ilegal o inapropiado. En otras palabras, el entorno Java o la aplicación Java no están en un estado apropiado para la operación solicitada. Normalmente lanzada por la aplicación.

→ **14. Which statements regarding the following code are correct ?**

```
class Base{
   void method1() throws java.io.IOException, NullPointerException{
      someMethod("arguments");
      // some I/O operations
   }
   int someMethod(String str){
      if(str == null) throw new NullPointerException();
      else return str.length();
   }
}
 public class NewBase extends Base{
      void method1(){
          someMethod("args");
      }
}
```

OPCIONES:
-method1 in class NewBase does not need to specify any exceptions.
-The code will not compile because RuntimeExceptions cannot be specified in the throws clause.
(Any Exception can be specified in the throws clause.)
-method1 in class NewBase must at least specify IOException in its throws clause.
-method1 in class NewBase must at least specify NullPointerException in its throws clause.
(This is not needed because NullPointerException is a RuntimeException.)
-There is no problem with the code.

EXPLICACION:
Overriding method only needs to specify a subset of the list of exception classes the overridden method can throw. A set of no classes is a valid subset of that list. Remember that NullPointerException is a subclass of RuntimeException, while IOException is a subclass of Exception.

Aquí tienes la traducción al español:

El método que sobrescribe solo necesita especificar un subconjunto de la lista de clases de excepción que el método sobreescrito puede lanzar. Un conjunto sin clases es un

subconjunto válido de esa lista. Recuerda que NullPointerException es una subclase de RuntimeException, mientras que IOException es una subclase de Exception.

→15. Following is a supposedly robust method to parse an input for a float :

```
public float parseFloat(String s){
  float f = 0.0f;
  try{
    f = Float.valueOf(s).floatValue();
    return f ;
  }
  catch(NumberFormatException nfe){
    System.out.println("Invalid input " + s);
    f = Float.NaN ;
    return f;
  }
  finally { System.out.println("finally");  }
  return f ;
}
```

Which of the following statements about the above method is/are true?

OPCIONES:
-If input is "0.1" then it will return 0.1 and print finally.
-If input is "0x.1" then it will return Float.NaN and print Invalid input 0x.1 and finally.
-If input is "1" then it will return 1.0 and print finally.
-If input is "0x1" then it will return 0.0 and print Invalid input 0x1 and finally.
-The code will not compile.

EXPLICACION:
Note that the return statement after finally block is unreachable. Otherwise, if this line were not there, choices 1, 2, 3 are valid.

Ten en cuenta que la declaración de retorno después del bloque finally es inalcanzable. De lo contrario, si esta línea no estuviera allí, las opciones 1, 2 y 3 serían válidas.

→16. What is the result of compiling and running this code?

```
class MyException extends Throwable{}
class MyException1 extends MyException{}
class MyException2 extends MyException{}
class MyException3 extends MyException2{}
public class ExceptionTest{
   void myMethod() throws MyException{
     throw new MyException3();
   }
   public static void main(String[] args){
     ExceptionTest et = new ExceptionTest();
     try{
       et.myMethod();
     }
```

```
      catch(MyException me){
        System.out.println("MyException thrown");
      }
      catch(MyException3 me3){
        System.out.println("MyException3 thrown");
      }
      finally{
        System.out.println(" Done");
      }
    }
}
```

OPCIONES:
-MyException thrown
-MyException3 thrown
-MyException thrown Done
-MyException3 thrown Done
-It fails to compile

EXPLICACION:
You can have multiple catch blocks to catch different kinds of exceptions, including exceptions that are subclasses of other exceptions. However, the catch clause for more specific exceptions (i.e. a SubClassException) should come before the catch clause for more general exceptions (i.e. a SuperClassException). Failure to do so results in a compiler error as the more specific exception is unreachable. In this case, catch for MyException3 cannot follow catch for MyException because if MyException3 is thrown, it will be caught by the catch clause for MyException. And so, there is no way the catch clause for MyException3 can ever execute. And so it becomes an unreachable statement.

Puedes tener múltiples bloques catch para capturar diferentes tipos de excepciones, incluidas las excepciones que son subclases de otras excepciones. Sin embargo, la cláusula catch para excepciones más específicas (es decir, una SubClassException) debe venir antes de la cláusula catch para excepciones más generales (es decir, una SuperClassException). No hacerlo resulta en un error de compilación, ya que la excepción más específica es inalcanzable. En este caso, el catch para MyException3 no puede seguir al catch para MyException, porque si se lanza MyException3, será capturada por la cláusula catch para MyException. Por lo tanto, no hay forma de que la cláusula catch para MyException3 pueda ejecutarse. Así que se convierte en una declaración inalcanzable.

→ 17. What will be the result of compiling and running the following program ?

```
class NewException extends Exception {}

class AnotherException extends Exception {}

public class ExceptionTest{
   public static void main(String[] args) throws Exception{
      try{
```

```
            m2();
        }
        finally{
            m3();
        }
        catch (NewException e){}
    }

    public static void m2() throws NewException { throw new NewException(); }

    public static void m3() throws AnotherException{ throw new AnotherException(); }

}
```

OPCIONES:

-It will compile but will throw AnotherException when run.
-It will compile but will throw NewException when run.
-It will compile and run without throwing any exceptions.
-It will not compile. (Because a catch block cannot follow a finally block!)
-None of the above.

EXPLICACION:
La sintaxis de try/catch/finally es:

```java
try {
    // Código que puede lanzar excepciones
} catch (Exception1 e) {
    // Manejo de Exception1
} catch (Exception2 e) {
    // Manejo de Exception2
}
...
catch (ExceptionN e) {
    // Manejo de ExceptionN
} finally {
    // Código que se ejecuta siempre
}
```

Con un try, puede ocurrir uno o más catch y/o finally. Un try DEBE ser seguido por al menos un catch o finally. (A menos que sea una declaración try con recursos, que no está en el alcance de este examen).

En Java 7, puedes colapsar los bloques catch en uno solo:

```java
try {
    // Código que puede lanzar excepciones
```

```
} catch (SQLException | IOException | RuntimeException e) {
    /* En este bloque, la clase del objeto de excepción real será la excepción que se lance en tiempo de ejecución. Pero la clase de la referencia e será la superclase común más cercana de todas las excepciones en el bloque catch. En este caso, será java.lang.Exception porque es la clase más específica que es una superclase para las tres excepciones. */

    e.printStackTrace();
}
```

→18. Identify the exceptions that will be received when the following code snippets are executed.

```
1. int factorial(int n){
       if(n==1) return 1;
     else return n*factorial(n-1);
   }
```
Assume that it is called with a very large integer.

```
 2. void printMe(Object[] oa){
      for(int i=0; i<=oa.length; i++)
      System.out.println(oa[i]);
   }
```
Assume that it is called as such: printMe(null);

```
 3. Object m1(){
      return new Object();
   }
   void m2(){
      String s = (String) m1();
   }
```
Assume that method m2 is invoked.

OPCIONES:
- ClassCastException
  ArrayIndexOutOfBoundsException
  StackOverflowError

- ClassCastException
  ArrayIndexOutOfBoundsException
  SecurityException

- No Exception Will Be Thrown
  SecurityException
  Will Not Compile

- StackOverflowError
  NullPointerException

No Exception Will Be Thrown

- StackOverflowError
  ArrayIndexOutOfBoundsException
  ClassCastException

- StackOverflowError
  NullPointerException
  NullPointerException

- SecurityException
  NullPointerException
  No Exception Will Be Thrown

- StackOverflowError
  NullPointerException
  ClassCastException

EXPLICACION:
Please read ExceptionClassSummary document in the "Study References" section.
Por favor, lee el documento ExceptionClassSummary en la sección "Referencias de Estudio".

19. What will be the output of the following class.

```
class Test{
    public static void main(String[] args){
       int j = 1;
       try{
          int i = doIt() / (j = 2);
       } catch (Exception e){
          System.out.println(" j = " + j);
       }
    }
    public static int doIt() throws Exception {  throw new Exception("FORGET IT");  }
 }
```

OPCIONES:
-It will print j = 1;
-It will print j = 2;
-The value of j cannot be determined.
-It will not compile.
-None of the above.

EXPLICACION
If evaluation of the left-hand operand of a binary operator completes abruptly, no part of the right-hand operand appears to have been evaluated. So, as doIt() throws exception, j = 2 never gets executed.

Si la evaluación del operando izquierdo de un operador binario se completa de manera abrupta, ninguna parte del operando derecho parece haber sido evaluada. Por lo tanto, como doIt() lanza una excepción, j = 2 nunca se ejecuta.

20. What will be the output when the following program is run?

```
package exceptions;
public class TestClass{
    public static void main(String[] args) {
        try{
            hello();
        }
        catch(MyException me){
            System.out.println(me);
        }
    }

    static void hello() throws MyException{
        int[] dear = new int[7];
        dear[0] = 747;
        foo();
    }

    static void foo() throws MyException{
        throw new MyException("Exception from foo");
    }
}

class MyException extends Exception {
    public MyException(String msg){
        super(msg);
    }
}
```

(Assume that line numbers printed in the messages given below are correct.)

OPCIONES:

- Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
  at exceptions.TestClass.doTest(TestClass.java:24)
  at exceptions.TestClass.main(TestClass.java:14)

You are creating an array of length 7. Since array numbering starts with 0, the first element would be array[0]. So ArrayIndexOutOfBoundsException will NOT be thrown. (Estás creando un arreglo de longitud 7. Dado que la numeración de los arreglos comienza en 0, el primer elemento sería array[0]. Por lo tanto, no se lanzará ArrayIndexOutOfBoundsException.)

- Error in thread "main" java.lang.ArrayIndexOutOfBoundsException

java.lang.ArrayIndexOutOfBoundsException extends java.lang.RuntimeException, which in turn extends java.lang.Exception. Therefore, ArrayIndexOutOfBoundsException is an Exception and not an Error. (java.lang.ArrayIndexOutOfBoundsException extiende java.lang.RuntimeException, que a su vez extiende java.lang.Exception. Por lo tanto, ArrayIndexOutOfBoundsException es una excepción y no un error.)

- exceptions.MyException: Exception from foo

- exceptions.MyException: Exception from foo
    at exceptions.TestClass.foo(TestClass.java:29)
    at exceptions.TestClass.hello(TestClass.java:25)
    at exceptions.TestClass.main(TestClass.java:14)
me.printStackTrace() would have produced this output.

EXPLICACION:
Note that there are a few questions in the exam that test your knowledge about how exception messages are printed. When you use System.out.println(exception), a stack trace is not printed. Just the name of the exception class and the message is printed. When you use exception.printStackTrace(), a complete chain of the names of the methods called, along with the line numbers, is printed. It contains the names of the methods in the chain of method calls that led to the place where the exception was created going back up to the point where the thread, in which the exception was created, was started.

Ten en cuenta que hay algunas preguntas en el examen que evalúan tu conocimiento sobre cómo se imprimen los mensajes de excepción. Cuando usas System.out.println(exception), no se imprime un seguimiento de la pila. Solo se imprime el nombre de la clase de excepción y el mensaje. Cuando usas exception.printStackTrace(), se imprime una cadena completa de los nombres de los métodos llamados, junto con los números de línea. Contiene los nombres de los métodos en la cadena de llamadas que llevaron al lugar donde se creó la excepción, volviendo hasta el punto donde se inició el hilo en el que se creó la excepción.

21. What will be the result of attempting to compile and run the following program?

```
class TestClass{
  public static void main(String args[]){
    int i = 0;
    loop :        // 1
    {
      System.out.println("Loop Lable line");
      try{
        for (  ;  true ;  i++ ){
          if( i >5) break loop;       // 2
        }
      }
      catch(Exception e){
        System.out.println("Exception in loop.");
      }
      finally{
        System.out.println("In Finally");      // 3
      }
    }
  }
}
```

OPCIONES:

- Compilation error at line 1 as this is an invalid syntax for defining a label.
You can apply a label to any code block or a block level statement (such as a for statement) but not to declarations. For example: loopX : int i = 10; (Puedes aplicar una etiqueta a cualquier bloque de código o a una declaración a nivel de bloque (como una declaración for), pero no a declaraciones. Por ejemplo: loopX: int i = 10;)

- Compilation error at line 2 as 'loop' is not visible here.

- No compilation error and line 3 will be executed.
Even if the break takes the control out of the block, the finally clause will be executed. (Incluso si el break saca el control del bloque, la cláusula finally se ejecutará.)

- No compilation error and line 3 will NOT be executed.

- Only the line with the label loop will be printed.

EXPLICACION:
A break without a label breaks the current loop (i.e. no iterations any more) and a break with a label tries to pass the control to the given label. 'Tries to' means that if the break is in a try block and the try block has a finally clause associated with it then it will be executed.
Un break sin una etiqueta rompe el bucle actual (es decir, no habrá más iteraciones), y un break con una etiqueta intenta pasar el control a la etiqueta dada. "Intenta" significa que si el break está en un bloque try y el bloque try tiene una cláusula finally asociada, entonces esta se ejecutará.