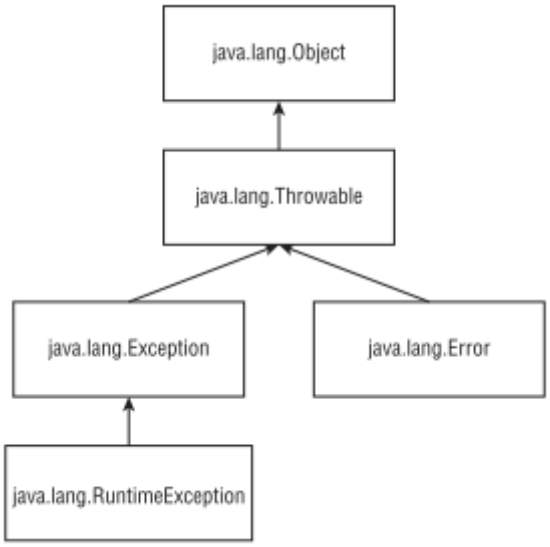
Exceptions

1. Understanding Exceptions

1.1. The Role of Exceptions

Hay dos enfoques que utiliza Java cuando se trata de excepciones. Un método puede manejar el caso de excepción en sí mismo o hacer que sea la responsabilidad del que llama. Las excepciones pueden ocurrir y ocurren todo el tiempo, incluso en un código de programa sólido. Cuando escribe programas más avanzados, tendrá que lidiar con fallas en el acceso a archivos, redes y servicios externos.

1.2. Understanding Exception Types



Una excepción de tiempo de ejecución se define como la clase `RuntimeException` y sus subclases. Las excepciones en tiempo de ejecución tienden a ser inesperadas, pero no necesariamente fatales. Por ejemplo, acceder a un índice de matriz no válido es inesperado. Las excepciones de tiempo de ejecución también se conocen como excepciones sin marcar.

Una excepción marcada incluye `Exception` y todas las subclases que no extienden `RuntimeException`. Las excepciones controladas tienden a ser más anticipadas, por ejemplo, al tratar de leer un archivo que no existe.

Java tiene una regla llamada ***manejar o declarar***. Para las excepciones marcadas, Java requiere que el código las maneje o las declare en la firma del método. Por ejemplo:

```
void fall() throws Exception {
  throw new Exception();
}
```

1.3. Throwing an Exception

En el examen, verá dos tipos de código que dan como resultado una excepción. El primero es el código que está mal. Por ejemplo:

```
String[] animals = new String[0];
System.out.println(animals[0]);
```

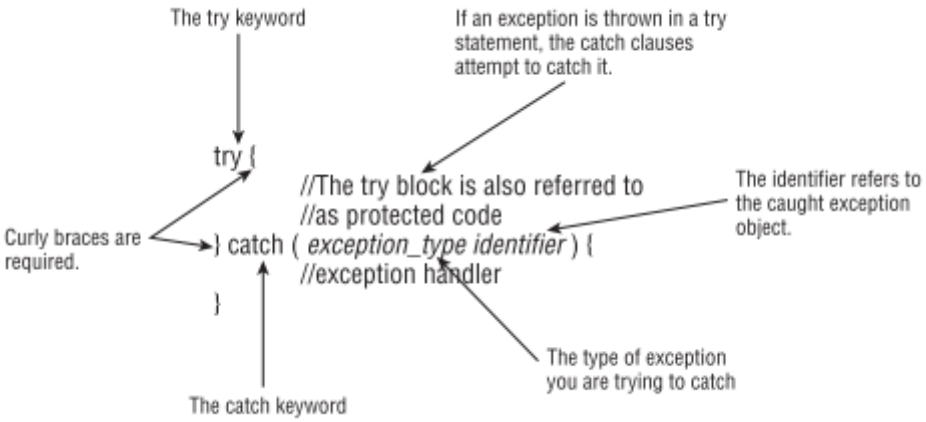
El Código lanza un `ArrayIndexOutOfBoundsException`.

La segunda forma para que el código resulte en una excepción es solicitar explícitamente a Java que lance una. Por ejemplo:

```
throw new Exception();
throw new Exception("Ow! I fell.");
throw new RuntimeException();
throw new RuntimeException("Ow! I fell.");
```

Tipo	**¿Cómo reconocerlo?**	**¿Está bien para el programa capturarlo?**	**¿El programa requiere manejarlo o capturarlo?**
`Runtime exception`	Subclase de `RuntimeException`	Yes	No
`Checked exception`	Subclase de `Exception` pero no de `RuntimeException`	Yes	Yes
`Error`	Subclase de Error	No	No

2. Using a try Statement



El código en el bloque try se ejecuta normalmente. Si alguna de las sentencias arroja una excepción que puede capturarse mediante el tipo de excepción enumerado en el bloque catch, el bloque try deja de ejecutarse y la ejecución va a la instrucción catch. Si ninguna de las instrucciones en el bloque try arroja una excepción que pueda capturarse, la cláusula catch no se ejecuta. Por ejemplo:

```
3: void explore() {
4:   try {
5:    fall();
6:    System.out.println("never get here");
7:   } catch (RuntimeException e) {
8:     getUp();
9:   }
10:  seeAnimals();
11: }
12: void fall() {  throw new RuntimeException(); }
```

Probablemente haya notado que las palabras "bloque" y "cláusula" se usan indistintamente. El examen también hace esto, así que te estamos acostumbrando. Ambos son correctos.

¿Ves lo que está mal con este?

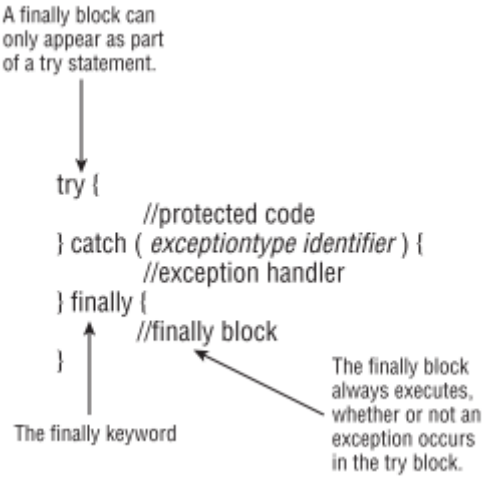
```
try  // DOES NOT COMPILE
  fall();
catch (Exception e)
  System.out.println("get up");
```

El problema es que faltan las llaves. Tiene que verse así:

```
try {
  fall();
} catch (Exception e) {
  System.out.println("get up");
}
```

Las declaraciones try son como métodos en que se requieren llaves, incluso si solo hay una declaración dentro de los bloques de código.

2.1. Adding a finally Block



Hay dos rutas a través del código con un `catch` y un `finally`. Si se lanza una excepción, el bloque `finally` se ejecuta después del bloque catch. Si no se lanza ninguna excepción, el bloque `finally` se ejecuta después de que finalice el bloque try. Por ejemplo:

```
12: void explore() {
13:   try {
14:     System.out.println("animals");
15:     fall();
16:   } catch (Exception e) {
17:     System.out.println("hug");
18:   } finally {
19:     System.out.println("more animals");
20:   }
21:   System.out.println("home");
22: }
```

La salida seria:

```
animals
hug
more animals
home
```

NOTA

En el examen OCA, una instrucción `try` debe tener `catch` y/o `finally`. Tener ambos está bien. Tener ninguno es un problema.

2.2. Catching Various Types of Exceptions

Ahora veamos qué sucede cuando se pueden arrojar diferentes tipos de excepciones desde el mismo método. Primero, debe poder reconocer si la excepción es una excepción marcada o no seleccionada. En segundo lugar, debe determinar si alguna de las excepciones son subclases de las demás.

```
class AnimalsOutForAWalk extends RuntimeException { }
class ExhibitClosed extends RuntimeException { }
class ExhibitClosedForLunch extends ExhibitClosed { }
```

Ahora detectamos ambos tipos de excepciones y las manejamos al imprimir el mensaje apropiado:

```
public void visitPorcupine() {
  try {
    seeAnimal();
  } catch (AnimalsOutForAWalk e) {// first catch block
    System.out.print("try back later");
  } catch (ExhibitClosed e) {// second catch block
    System.out.print("not today");
  }
}
```

Hay tres posibilidades para cuando se ejecuta este código:
Si `seeAnimal()` no arroja una excepción, no se imprime nada.
Si `seeAnimal()` lanza `AnimalsOutForAWalk`, solo se ejecuta el primer bloque de captura.
Si `seeAnimal()` lanza `ExhibitClosed`, solo se ejecuta el segundo bloque catch.

Existe una regla para el orden de los bloques `catch`. Java los mira en el orden en que aparecen. Si es imposible ejecutar uno de los bloques `catch`, se produce un error de compilación sobre código inalcanzable. Esto sucede cuando se captura una superclase antes de una subclase.

El siguiente ejemplo muestra los tipos de excepción que heredan el uno del otro:

```
public void visitMonkeys() {
  try {
    seeAnimal();
  } catch (ExhibitClosedForLunch e) {// subclass exception
    System.out.print("try back later");
  } catch (ExhibitClosed e) {// superclass exception
    System.out.print("not today");
  }
}
```

Otro ejemplo:

```
public void visitMonkeys() {
  try {
    seeAnimal();
  } catch (ExhibitClosed e) {
    System.out.print("not today");
  } catch (ExhibitClosedForLunch e) {// DOES NOT COMPILE
    System.out.print("try back later");
  }
}
```

2.3. Throwing a Second Exception

La mayoría de los ejemplos que ve con el manejo de excepciones en el examen son abstractos. Esta muestra que solo importa la última excepción que se arroje:

```
26: try {
27:    throw new RuntimeException();
28: } catch (RuntimeException e) {
29:    throw new RuntimeException();
30: } finally {
31:    throw new Exception();
32: }
```

El bloque `finally` arroja una excepción propia en la línea 31, esta es lanzada. La excepción del bloque catch se olvida.

¿Qué crees que devuelve este método?

```
30: public String exceptions() {
31:    String result = "";
32:    String v = null;
33:    try {
34:       try {
35:         result += "before";
36:         v.length();
37:         result += "after";
38:       } catch (NullPointerException e) {
39:        result += "catch";
40:        throw new RuntimeException();
41:       } finally {
42:        result += "finally";
43:        throw new Exception();
44:       }
45:    } catch (Exception e) {
46:      result += "done";
47:    }
48:    return result;
49: }
```

La respuesta correcta es:
`before catch finally done`

3. Recognizing Common Exception Types

Debe reconocer tres tipos de excepciones para el examen OCA: excepciones de tiempo de ejecución, excepciones comprobadas y errores.

3.1. Runtime Exceptions

`ArithmeticException`: Lanzado por la JVM cuando el código intenta dividir por cero. Por ejemplo:
`int answer = 11 / 0;`

`ArrayIndexOutOfBoundsException`: Lanzado por la JVM cuando el código usa un índice ilegal para acceder a una matriz. Por ejemplo:

```
int[] countsOfMoose = new int[3];
System.out.println(countsOfMoose[-1]);
```

`ClassCastException`: Lanzado por la JVM cuando se intenta lanzar una excepción a una subclase de la cual no es una instancia. Por ejemplo:

```
String type = "moose";
Integer number = (Integer) type;  // DOES NOT COMPILE
```

`IllegalArgumentException`: Lanzado por el programador para indicar que un método ha pasado un argumento ilegal o inapropiado. Por ejemplo:

```
public static void setNumberEggs(int numberEggs) {
  if (numberEggs < 0)
    throw new IllegalArgumentException(
        "# eggs must not be negative");
  this.numberEggs = numberEggs;
}
```

`NullPointerException`: Lanzado por la JVM cuando hay una referencia nula donde se requiere un objeto. Por ejemplo:

```
String name;
public void printLength() throws NullPointerException {
  System.out.println(name.length());
}
```

`NumberFormatException`: Lanzado por el programador cuando se intenta convertir una cadena a un tipo numérico, pero la cadena no tiene un formato apropiado. Por ejemplo:

```
Integer.parseInt("abc");
```

3.2. Checked Exceptions

`FileNotFoundException`: Lanzado programáticamente cuando el código intenta hacer referencia a un archivo que no existe.

`IOException`: Lanzado programáticamente cuando hay un problema al leer o escribir un archivo.

También tenga en cuenta que `FileNotFoundException` es una subclase de `IOException`, aunque el examen le recordará ese hecho si aparece.

3.3. Errors

`ExceptionInInitializerError`: Lanzado por la JVM cuando un inicializador estático arroja una excepción y no lo maneja. Por ejemplo:

```
static {
  int[] countsOfMoose = new int[3];
  int num = countsOfMoose[-1];
}
```

`StackOverflowError`: Lanzado por la JVM cuando un método se llama a sí mismo demasiadas veces (esto se llama recurrencia infinita porque el método típicamente se llama a sí mismo sin fin). Por ejemplo:

```
public static void doNotCodeThis(int num) {
```

```
  doNotCodeThis(1);
}
```

`NoClassDefFoundError`: Lanzado por la JVM cuando una clase que utiliza el código está disponible en tiempo de compilación pero no en tiempo de ejecución.

4. Calling Methods That Throw Exceptions

Veamos el siguiente código que no compila:

```
class NoMoreCarrotsException extends Exception {}
public class Bunny {
  public static void main(String[] args) {
    eatCarrot();// DOES NOT COMPILE
  }
  private static void eatCarrot() throws NoMoreCarrotsException {
  }
}
```

El problema es que `NoMoreCarrotsException` es una excepción marcada. Las excepciones controladas deben ser manejadas o declaradas. El código se compilaría si cambiamos el método `main()` a cualquiera de estos:

```
public static void main(String[] args)
  throws NoMoreCarrotsException {// declare exception
  eatCarrot();
}
public static void main(String[] args) {
  try {
  eatCarrot();
  } catch (NoMoreCarrotsException e ) {// handle exception
    System.out.print("sad rabbit");
  }
}
```

El compilador aún está buscando código inalcanzable. Declarar una excepción no utilizada no se considera código inalcanzable. ¿Ves el problema aquí?

```
public void bad() {
  try {
    eatCarrot();
  } catch (NoMoreCarrotsException e ) {// DOES NOT COMPILE
    System.out.print("sad rabbit");
  }
}
public void good() throws NoMoreCarrotsException {
  eatCarrot();
}
private static void eatCarrot() { }
```

4.1. Subclasses

Cuando una clase anula un método de una superclase o implementa un método desde una interfaz, no está permitido agregar nuevas excepciones comprobadas a la firma del método. Por ejemplo:

```
class CanNotHopException extends Exception { }
class Hopper {
  public void hop() { }
```

```
}
class Bunny extends Hopper {
  public void hop() throws CanNotHopException { } // DOES NOT COMPILE
}
```

Una subclase puede declarar menos excepciones que la superclase o la interfaz. Esto es legal porque las personas que llaman ya los están manejando. Por ejemplo:

```
class Hopper {
  public void hop() throws CanNotHopException { }
}
class Bunny extends Hopper {
  public void hop()  { }
}
```

...otro más:

```
class Hopper {
  public void hop() throws Exception { }
}
class Bunny extends Hopper {
  public void hop() throws CanNotHopException { }
}
```

Esta regla se aplica solo a las excepciones marcadas. El siguiente código es legal porque tiene una excepción de tiempo de ejecución en la versión de la subclase:

```
class Hopper {
  public void hop() { }
}
class Bunny extends Hopper {
  public void hop() throws IllegalStateException { }
}
```

Los métodos son libres de arrojar cualquier excepción de tiempo de ejecución que deseen sin mencionarlos en la declaración del método.

4.2. Printing an Exception

Hay tres formas de imprimir una excepción. Puede dejar que Java lo imprima, imprima solo el mensaje o imprima de dónde proviene el seguimiento de la pila. Este ejemplo muestra los tres enfoques:

```
5: public static void main(String[] args) {
6:    try {
7:     hop();
8:    } catch (Exception e) {
9:      System.out.println(e);
10:     System.out.println(e.getMessage());
11:     e.printStackTrace();
12:   }
13: }
14: private static void hop() {
15:   throw new RuntimeException("cannot hop");
16: }
```

Este código da como resultado el siguiente resultado:

```
java.lang.RuntimeException: cannot hop
cannot hop
```

```
java.lang.RuntimeException: cannot hop
at trycatch.Handling.hop(Handling.java:15)
at trycatch.Handling.main(Handling.java:7)
```

La primera línea muestra lo que Java imprime de manera predeterminada: el tipo de excepción y el mensaje. La segunda línea muestra solo el mensaje. El resto muestra un seguimiento de la pila.

5. Review Questions

5.1.

Which of the following statements are true? (Choose all that apply)

A. Runtime exceptions are the same thing as checked exceptions.
B. Runtime exceptions are the same thing as unchecked exceptions.
C. You can declare only checked exceptions.
D. You can declare only unchecked exceptions.
E. You can handle only Exception subclasses.

5.2.

What is printed besides the stack trace caused by the `NullPointerException` from line 16?

```
1: public class DoSomething {
2:   public void go() {
3:     System.out.print("A");
4:     try {
5:         stop();
6:     } catch (ArithmeticException e) {
7:         System.out.print("B");
8:     } finally {
9:         System.out.print("C");
10:    }
11:    System.out.print("D");
12:  }
13:  public void stop() {
14:    System.out.print("E");
15:    Object x = null;
16:    x.toString();
17:    System.out.print("F");
18:  }
19:  public static void main(String[] args) {
20:    new DoSomething().go();
21:  }
22: }
```

A. `AE`
B. `AEBCD`
C. `AEC`
D. `AECD`
E. No output appears other than the stack trace.

5.3.

What is the output of the following snippet, assuming a and b are both 0?

```
3:     try {
4:       return a / b;
5:     } catch (RuntimeException e) {
6:       return -1;
```

```
7:      } catch (ArithmeticException e) {
8:        return 0;
9:      } finally {
10:       System.out.print("done");
11:     }
```

A. `-1`
B. `0`
C. `done-1`
D. `done0`
E. The code does not compile.
F. An uncaught exception is thrown.

5.4.

What is printed by the following? (Choose all that apply)

```
1:  public class Mouse {
2:    public String name;
3:    public void run() {
4:      System.out.print("1");
5:      try {
6:        System.out.print("2");
7:        name.toString();
8:        System.out.print("3");
9:      } catch (NullPointerException e) {
10:       System.out.print("4");
11:       throw e;
12:     }
13:     System.out.print("5");
14:   }
15:   public static void main(String[] args) {
16:     Mouse jerry = new Mouse();
17:     jerry.run();
18:     System.out.print("6");
19:   } }
```

A. `1`
B. `2`
C. `3`
D. `4`
E. `5`
F. `6`
G. The stack trace for a `NullPointerException`

5.5.

Which of the following are unchecked exceptions? (Choose all that apply)

A. `ArrayIndexOutOfBoundsException`
B. `IllegalArgumentException`
C. `IOException`
D. `NumberFormatException`
E. Any exception that extends `RuntimeException`
F. Any exception that extends `Exception`

5.6.

Which of the following can be inserted into Lion to make this code compile? (Choose all that apply)

```
class HasSoreThroatException extends Exception {}
class TiredException extends RuntimeException {}
interface Roar {
  void roar() throws HasSoreThroatException;
}
class Lion implements Roar {// INSERT CODE HERE
}
```

A. `public void roar(){}`
B. `public void roar() throws Exception{}`
C. `public void roar() throws HasSoreThroatException{}`
D. `public void roar() throws IllegalArgumentException{}`
E. `public void roar() throws TiredException{}`

5.7.

What does the output of the following contain? (Choose all that apply)

```
12: public static void main(String[] args) {
13:   System.out.print("a");
14:   try {
15:     System.out.print("b");
16:     throw new IllegalArgumentException();
17:   } catch (IllegalArgumentException e) {
18:     System.out.print("c");
19:     throw new RuntimeException("1");
20:   } catch (RuntimeException e) {
21:     System.out.print("d");
22:     throw new RuntimeException("2");
23:   } finally {
24:     System.out.print("e");
25:     throw new RuntimeException("3");
26:   }
27: }
```

A. `abce`
B. `abde`
C. An exception with the message set to "1"
D. An exception with the message set to "2"
E. An exception with the message set to "3"
F. Nothing; the code does not compile.